

Verification Continuum™

# VCS® User Guide

---

P-2019.06-SP1, September 2019

**SYNOPSYS®**

# **Copyright Notice and Proprietary Information**

© 2019 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

## **Third-Party Software Notices**

VCS® and configurations of VCS includes or is bundled with software licensed to Synopsys under free or open-source licenses. For additional information regarding Synopsys's use of free and open-source software, refer to the third\_party\_notices.txt file included within the <install\_path>/doc directory of the installed VCS software.

## **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.

## **Trademarks**

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <https://www.synopsys.com/company/legal/trademarks-brands.html>. All other product or company names may be trademarks of their respective owners.

## **Free and Open-Source Software Licensing Notices**

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

## **Third-Party Links**

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

[www.synopsys.com](http://www.synopsys.com)

# Contents

---

1. Getting Started . . . . .	1-2
Simulator Support with Technologies . . . . .	1-2
Simulation Preemption Support. . . . .	1-4
Setting Up the Simulator . . . . .	1-4
Verifying Your System Configuration . . . . .	1-5
Obtaining a License . . . . .	1-6
Setting Up Your Environment. . . . .	1-7
Setting Up Your C Compiler. . . . .	1-8
Creating a synopsys_sim.setup File . . . . .	1-9
The Concept of a Library In VCS . . . . .	1-11
Library Name Mapping . . . . .	1-11
Including Other Setup Files . . . . .	1-12
Using the SYNOPSYS_SIM_SETUP Environment Variable	1-13
Displaying Setup Information. . . . .	1-13
Displaying Design Information Analyzed Into a Library . . . . .	1-14
Using the Simulator. . . . .	1-16
Two-step Flow . . . . .	1-16

Compiling the Design . . . . .	1-17
Simulating the Design . . . . .	1-17
Three-step Flow . . . . .	1-17
Basic Usage Model . . . . .	1-18
Two-step Flow . . . . .	1-18
Three-step Flow . . . . .	1-19
Default Time Unit and Time Precision . . . . .	1-20
Searching Identifiers in the Design Using UNIX Commands . . . . .	1-21
Examples . . . . .	1-22
<b>2. VCS Flow</b>	
Three-step Flow . . . . .	2-2
Analysis . . . . .	2-2
Using vhdlan. . . . .	2-4
Using vlogan. . . . .	2-7
Analyzing the Design to Different Libraries . . . . .	2-15
Elaboration. . . . .	2-15
Using VCS . . . . .	2-16
Simulation . . . . .	2-19
Interactive Mode. . . . .	2-19
Batch Mode . . . . .	2-20
Commonly Used Runtime Options. . . . .	2-20
Two-step Flow . . . . .	2-22
Compilation . . . . .	2-22
Using vcs . . . . .	2-23
Simulation . . . . .	2-28
Interactive Mode. . . . .	2-29

Batch Mode .....	2-29
Commonly Used Runtime Options.....	2-29
<b>3. Modeling Your Design</b>	
<b>Avoiding Race Conditions .....</b>	<b>3-2</b>
Using and Setting a Value at the Same Time .....	3-3
Setting a Value Twice at the Same Time .....	3-3
Flip-Flop Race Condition .....	3-4
Continuous Assignment Evaluation .....	3-5
Counting Events .....	3-7
Time Zero Race Conditions .....	3-7
<b>Race Detection in Verilog Code .....</b>	<b>3-8</b>
The Dynamic Race Detection Tool.....	3-8
Introduction to the Dynamic Race Detection Tool .....	3-9
Enabling Race Detection .....	3-12
The Race Detection Report .....	3-12
Post-Processing the Report .....	3-16
Debugging Simulation Mismatches .....	3-18
The Static Race Detection Tool .....	3-20
Race Detection Tool to Identify Race between Clock and Data ..	3-23
Use Model .....	3-23
Examples .....	3-24
Limitations .....	3-26
<b>Optimizing Testbenches for Debugging.....</b>	<b>3-26</b>
Conditional Compilation.....	3-27
Enabling Debugging Features at Runtime.....	3-29

Combining the Techniques . . . . .	3-32
Creating Models That Simulate Faster . . . . .	3-33
Unaccelerated Data Types, Primitives, and Statements . . . . .	3-34
Inferring Faster Simulating Sequential Devices . . . . .	3-35
Modeling Faster always Blocks . . . . .	3-39
Using Verilog 2001 Constructs . . . . .	3-40
Case Statement Behavior . . . . .	3-42
Precedence in Text Macro Definitions . . . . .	3-43
Memory Size Limits in the Simulator . . . . .	3-43
Using Sparse Memory Models . . . . .	3-45
Obtaining Scope Information . . . . .	3-47
Scope Format Specifications . . . . .	3-47
Returning Information About the Scope . . . . .	3-51
Avoiding Circular Dependency . . . . .	3-54
Designing With \$lsi_dumpports for Simulation and Test . . . . .	3-55
Dealing With Unassigned Nets . . . . .	3-56
Code Values at Time 0 . . . . .	3-57
Cross Module Forces and No Instance Instantiation . . . . .	3-57
Signal Value/Strength Codes . . . . .	3-59
<b>4. Compiling/Elaborating the Design</b>	
Compiling/Elaborating the Design in the Debug Mode . . . . .	4-2
Optimizing Simulation Performance for Desired Debug Visibility With the -debug_access Option . . . . .	4-3

Use Model . . . . .	4-3
Incrementally Removing Debug Capabilities . . . . .	4-7
Assertion Debug Support . . . . .	4-8
Verdi One Search Support . . . . .	4-9
Reporting Global Debug Capability Diagnostics . . . . .	4-9
Specifying Design Regions for -debug_access Capabilities .	4-13
Enabling Additional Debug Capabilities . . . . .	4-15
Reduction in the Objects Being Dumped . . . . .	4-17
Using -debug_access With Tab Files . . . . .	4-18
Unused Tab File Calls . . . . .	4-18
Including Tab Files . . . . .	4-18
Interaction With Other Debug Options . . . . .	4-19
Dynamic Loading of DPI Libraries at Runtime . . . . .	4-20
Use Model . . . . .	4-20
Dynamic Loading of PLI Libraries at Runtime . . . . .	4-22
Key Compilation or Elaboration Features . . . . .	4-23
Initializing Verilog Variables, Registers, and Memories . . . . .	4-23
Initializing Verilog Variables, Registers, and Memories in an entire Design . . . . .	4-24
Initializing Verilog Variables, Registers, and Memories in Selective Parts of a Design . . . . .	4-26
Selections for Initialization of Registers or Memories . . . . .	4-30
Reporting the Initialized Values of Variables, Registers, and Memories . . . . .	4-31
Overriding Generics and Parameters . . . . .	4-31
Usage Model . . . . .	4-34
Checking for x and z Values In Conditional Expressions . . . . .	4-34

Enabling the Checking . . . . .	4-36
Filtering Out False Negatives . . . . .	4-36
Cross Module References (XMRs) . . . . .	4-38
The <code>hdl_xmr</code> Procedure and the <code>\$hdl_xmr</code> System Task .	4-39
Data Types Supported . . . . .	4-40
Using the <code>hdl_xmr</code> Procedure . . . . .	4-41
Using the <code>\$hdl_xmr</code> Task . . . . .	4-43
Use Model . . . . .	4-46
Examples . . . . .	4-46
\$ <code>hdl_xmr</code> Support for VHDL Variables . . . . .	4-52
Data Type Support and Usage Examples . . . . .	4-53
Verilog Configurations and Libmaps . . . . .	4-57
Library Mapping Files . . . . .	4-58
Configurations . . . . .	4-59
Hierarchical Configurations . . . . .	4-63
The <code>-top</code> Compile-Time Option . . . . .	4-64
Limitations of Configurations . . . . .	4-64
Use Model . . . . .	4-64
Example . . . . .	4-65
Using the <code>-liblist</code> Option . . . . .	4-70
Design Cells and Library Cells . . . . .	4-73
Library Search Order Rules . . . . .	4-75
Example Testcase Files . . . . .	4-84
Usage Examples for Library Search Order Rules for Verilog or SystemVerilog Designs . . . . .	4-86
Usage Examples for Library Search Order Rules for Verilog or SystemVerilog Designs Without a Configuration File . . . . .	4-104
Lint Warning Message for Missing ‘ <code>endcelldefine</code> ’ . . . . .	4-107
Error/Warning/Lint Message Control . . . . .	4-111

Controlling Error/Warning/Lint Messages Using Compile-Time Options .....	4-112
Controlling Error/Warning/Lint Messages Using a Configuration File .....	4-128
Extracting the Files Used in Elaboration/Compilation .....	4-137
XML File Format .....	4-138
Limitations .....	4-144
<b>5. Simulating the Design</b>	
Using Verdi .....	5-2
Using UCLI .....	5-3
-ucli2Proc Option .....	5-5
Options for Debugging Using Verdi and UCLI .....	5-6
Reporting Forces/Injections in a Simulation .....	5-8
Use Model .....	5-9
Reporting Force/Deposit/Release Information .....	5-11
Handling Forces on Bit/Part Select and MDA Word .....	5-12
Handling Forces on Concatenated Codes .....	5-13
Output Format .....	5-13
Usage Examples .....	5-16
Generating Force List Report for Desired Instance Hierarchies Examples .....	5-24
Displaying Different ID for Each Verilog Force .....	5-28
Enhanced Force List Report .....	5-28
Use Model .....	5-29
Displaying Source Information for VHDL Forces .....	5-30
Use Model .....	5-30

Displaying Source Information for External Forces .....	5-31
Use Model .....	5-33
Viewing Force Information in Interactive Debug Mode .....	5-34
Use model .....	5-34
Example .....	5-36
Reporting \$deposit Value Changes .....	5-38
Limitations .....	5-38
Key Runtime Features.....	5-41
Overriding Generics at Runtime.....	5-41
Use Model .....	5-42
Passing Values from the Runtime Command Line .....	5-46
Using -f Runtime Option.....	5-47
Limitations .....	5-47
Saving and Restarting the Simulation .....	5-48
Save and Restart Example.....	5-49
Save and Restart File I/O.....	5-50
Save and Restart With Runtime Options .....	5-51
Specifying Long Time Before Stopping the Simulation .....	5-52
Preventing Time 0 Race Conditions.....	5-54
Resolving RTL Simulation Races in Mixed HDL Designs .....	5-55
Recommended Approach to Resolve Race Conditions ..	5-55
Resolving RTL Simulation Races in Verilog Designs.....	5-57
Recommended Approach to Resolve Race Conditions ..	5-58
Supporting Simulation Executable to Return Non-Zero Value on Error Results .....	5-62
Use Model .....	5-64
Limitations .....	5-64

Supporting Memory Load and Dump Task Verbosity.....	5-64
Use Model .....	5-66
<b>6. The Unified Simulation Profiler</b>	
The Use Model .....	6-2
Omitting Profiling at Runtime.....	6-4
Omitting the -simprofile Runtime Option .....	6-5
Omitting Profile Report Writing after Runtime .....	6-6
Specifying a Directory for the Profile Database.....	6-7
Post Simulation Profile Information .....	6-7
Specifying the Name of the Profile Report.....	6-8
Running the profrpt Profile Report Generator .....	6-8
Specifying Views .....	6-11
The Snapshot Mechanism .....	6-14
Specifying Timeline Reports .....	6-15
Recording and Viewing Memory Stack Traces .....	6-16
Reporting PLI, DPI, and DirectC Function Call Information..	6-16
Compiling and Running the Profiler Example.....	6-17
Profiling Time Used by Various Parts of the Design.....	6-19
Profiling Memory Used by Various Parts of the Design ..	6-20
The Output Directories and Files .....	6-21
The Enhanced Accumulative Views.....	6-22
The Comparative View.....	6-29
The Caller-Callee Views .....	6-31
HTML Profiler Reports.....	6-37

Display of Parameterized Class Functions and Tasks in Profiling Reports . . . . .	6-66
Hypertext Links to the Source Files . . . . .	6-68
Single Text Format Report . . . . .	6-71
Stack Trace Report Example . . . . .	6-72
SystemC Views . . . . .	6-74
Constraint Profiling Integrated in the Unified Profiler . . . . .	6-81
Changes to the Use Model for Constraint Profiling . . . . .	6-82
The Time Constraint Solver View . . . . .	6-83
The Memory Constraint Solver View . . . . .	6-92
Performance/Memory Profiling for Coverage Covergroups . . . . .	6-96
Use Model . . . . .	6-97
Example . . . . .	6-97
HTML Profiler Reports . . . . .	6-97
Default Summary View . . . . .	6-98
Time/Memory Summary View . . . . .	6-98
Time/Memory Module View . . . . .	6-99
Time/Memory Construct View . . . . .	6-100
Time/Memory Covergroup View . . . . .	6-101
Limitations . . . . .	6-102
Reporting Debug Capabilities for Each Module . . . . .	6-103
Use Model . . . . .	6-103
HTML Reports . . . . .	6-104
Text Reports . . . . .	6-107
Limitations . . . . .	6-107
Supporting Line-Based CPU Time Profiler . . . . .	6-108

Use Model . . . . .	6-108
Limitations . . . . .	6-110
Supporting Simulation Time Slice Based Profiler . . . . .	6-110
Use Model . . . . .	6-111
Diagnostics . . . . .	6-113
Limitations . . . . .	6-114
Isolating the Cost of Garbage Collection . . . . .	6-114
Use Model . . . . .	6-115
Isolating the Cost of Loading Design Database . . . . .	6-115
Use Model . . . . .	6-115
Support for Third-Party Shared Library Profiler Report . . . . .	6-116
Use Model . . . . .	6-117
Enhancements to VHDL Unified Simulation Profiler Report . . . . .	6-117
Limitations . . . . .	6-119
Enhancing the HSIM View With Simulation Profiler Report . . . . .	6-119
Benefits . . . . .	6-120
<b>7. Diagnostics</b>	
Using Diagnostics . . . . .	7-2
Using <code>-diag</code> Option . . . . .	7-2
Compile-time Diagnostics . . . . .	7-4
Libconfig Diagnostics . . . . .	7-4
Example . . . . .	7-4
Timescale Diagnostics . . . . .	7-7
Example . . . . .	7-7

Generating Information on Unused Libraries at vlogan . . . . .	7-10
Use Model . . . . .	7-10
Usage Example . . . . .	7-10
Generating Information on Unused Libraries at VCS. . . . .	7-12
Use Model . . . . .	7-12
Obtaining Statistics on Package Utilization . . . . .	7-14
Use Model . . . . .	7-14
 Runtime Diagnostics . . . . .	7-16
Diagnostics for VPI/VHPI PLI Applications . . . . .	7-16
Keeping the UCLI/Verdi Prompt Active After a Runtime Error	7-21
UCLI Use Model. . . . .	7-21
Verdi Use Model. . . . .	7-23
UCLI Usage Example. . . . .	7-24
Limitations . . . . .	7-26
Diagnosing Quickthread Issues . . . . .	7-26
Diagnosing Quickthread Issues in DPI. . . . .	7-27
Diagnosing Quickthread Issues in SystemC . . . . .	7-28
 Post-Processing Diagnostics. . . . .	7-32
Using the vpduilt Utility to Generate Statistics . . . . .	7-32
The vpduilt Utility Syntax . . . . .	7-32
Options . . . . .	7-33
 Sparse Memory Diagnostics . . . . .	7-34
Compile Time Options . . . . .	7-35
Example . . . . .	7-37
Runtime Options . . . . .	7-38
Sparse Disable Options . . . . .	7-40

## 8. VPD, VCD, and EVCD Utilities

Advantages of VPD . . . . .	8-2
Dumping a VPD File . . . . .	8-3
Using System Tasks . . . . .	8-3
Enable and Disable Dumping . . . . .	8-3
Override the VPD Filename . . . . .	8-7
Dump Multi-Dimensional Arrays and Memories . . . . .	8-8
Using \$vcdblplusmemorydump System Task . . . . .	8-10
Capture Delta Cycle Information . . . . .	8-11
Dumping an EVCD File . . . . .	8-12
Using \$dumports System Task . . . . .	8-12
Dumping EVCD File for Mixed Designs Using UCLI dump Command	
8-13	
Use Model . . . . .	8-13
Use Model for Dumping CCN Driver Through INOUT . . . . .	8-14
Limitations . . . . .	8-16
Post-processing Utilities . . . . .	8-17
The vcdpost Utility . . . . .	8-18
Scalarizing the Vector Signals . . . . .	8-18
Uniquifying the Identifier Codes . . . . .	8-19
The vcdpost Utility Syntax . . . . .	8-20
The vcdiff Utility . . . . .	8-21
Syntax . . . . .	8-21
The vcdiff Utility Output Example . . . . .	8-29
The vcat Utility . . . . .	8-31
The vcat Utility Syntax . . . . .	8-32
Generating Source Files From VCD Files . . . . .	8-36

Writing the Configuration File . . . . .	8-37
The vcsplit Utility . . . . .	8-41
The vcsplit Utility Syntax . . . . .	8-42
The vcd2vpd Utility . . . . .	8-45
Options for Specifying EVCD Options . . . . .	8-47
The vpd2vcd Utility . . . . .	8-47
The Command File Syntax . . . . .	8-53
The vpdmmerge Utility . . . . .	8-56
The vpduutil Utility . . . . .	8-59
<b>9. Performance Tuning</b>	
<b>Compile-time Performance</b> . . . . .	9-3
Incremental Compilation . . . . .	9-3
Compile Once and Run Many Times . . . . .	9-4
Parallel Compilation . . . . .	9-5
Improving VCS Compile Performance and Capacity . . . . .	9-5
Use Model . . . . .	9-6
<b>Runtime Performance</b> . . . . .	9-6
Using Radiant Technology . . . . .	9-7
Compiling With Radiant Technology . . . . .	9-8
Applying Radiant Technology to Parts of the Design . . . . .	9-8
Improving Performance When Using PLIs . . . . .	9-17
Use Model . . . . .	9-18
Enabling TAB File Capabilities in UCLI Using -debug_access	9-21
Use Model . . . . .	9-21
Example . . . . .	9-22
Impact on Performance . . . . .	9-23

Obtaining VCS Consumption of CPU Resources . . . . .	9-24
Use Model . . . . .	9-25
Compile Time . . . . .	9-25
Simulation Time . . . . .	9-26
Dumping Design Statistics . . . . .	9-27
Use Model . . . . .	9-27
Usage Example . . . . .	9-28
Limitations . . . . .	9-30
<b>10. Using X-Propagation</b>	
Introduction to X-Propagation . . . . .	10-2
Guidelines for Running X-Propagation Simulations. . . . .	10-4
Using the X-Propagation Simulator . . . . .	10-6
Specifying X-Propagation Merge Mode . . . . .	10-9
Querying X-Propagation at Runtime . . . . .	10-14
X-Propagation Instrumentation Report. . . . .	10-15
Automatic Hardware Inference of Flip-Flops Enabled by Default	
10-17	
X-Propagation Configuration File. . . . .	10-18
X-Propagation Configuration File Syntax . . . . .	10-19
Xprop Instrumentation Control. . . . .	10-23
Limitation . . . . .	10-26
Process Based X-Propagation Exclusion . . . . .	10-27
Support for XIndex Element Merging. . . . .	10-28
X-Index . . . . .	10-29
Index BSpace. . . . .	10-29
Addressing Models. . . . .	10-31

Merge Modes . . . . .	10-32
Index Selection Methods . . . . .	10-33
Disabling XIndex Merging for Read or Write Operations . . . . .	10-35
Use Model . . . . .	10-36
Examples . . . . .	10-37
Limitations . . . . .	10-41
Bounds Checking. . . . .	10-41
Detecting Unknown Values in Type Conversion Functions . . . . .	10-43
Time Zero Initialization. . . . .	10-43
Handling Non-pure Functions Due to Static Lifetime. . . . .	10-44
Supporting UCLI Commands for X-Propagation Control Tasks	10-45
Use Model . . . . .	10-46
UCLI Command to Specify the Merge Mode . . . . .	10-46
UCLI Command to Control Error Messages or Warning Messages	
10-47	
VHDL Two-State Objects in X-Propagation. . . . .	10-48
Supported XValues . . . . .	10-49
Supported Objects . . . . .	10-49
Supported Expressions . . . . .	10-49
Supported Attributes . . . . .	10-55
Limitations . . . . .	10-56
X-Propagation Code Examples . . . . .	10-57
If Statement . . . . .	10-58
Verilog Example . . . . .	10-58
VHDL Example . . . . .	10-59
Case Statement. . . . .	10-61
Verilog Example . . . . .	10-61

VHDL Example . . . . .	10-62
Unique/Priority Case Variants . . . . .	10-64
Edge Sensitive Expression . . . . .	10-65
Verilog Example . . . . .	10-65
VHDL Example . . . . .	10-67
Latch . . . . .	10-69
Verilog Example . . . . .	10-69
VHDL Example . . . . .	10-70
Support for Active Drivers in X-Propagation . . . . .	10-71
Combinational Logic. . . . .	10-72
Latches. . . . .	10-75
Flip-flops. . . . .	10-77
Key points to Note . . . . .	10-79
Support for Ternary Operator. . . . .	10-80
Limitations . . . . .	10-80
<b>11. Gate-Level Simulation</b>	
SDF Annotation . . . . .	11-2
Using the Unified SDF Feature . . . . .	11-3
Using the \$sdf_annotation System Task. . . . .	11-4
Using the -xlrn Option for SDF Retain, Gate Pulse Propagation, and Gate Pulse Detection Warning . . . . .	11-6
Using the Optimistic Mode in SDF . . . . .	11-7
Using Gate Pulse Propagation . . . . .	11-8
Generating Warnings During Gate Pulses . . . . .	11-9
Enhancing SDF Annotation to Support Nets Through SPICE	11-9
Precompiling an SDF File . . . . .	11-12

## Feedback

Creating the Precompiled Version of the SDF File . . . . .	11-13
<b>SDF Configuration File . . . . .</b>	<b>11-15</b>
Delay Objects and Constructs . . . . .	11-15
SDF Configuration File Commands . . . . .	11-16
The INTERCONNECT_MIPD Command . . . . .	11-17
The MTM Command . . . . .	11-18
The SCALE Commands . . . . .	11-18
An SDF Example With Configuration File . . . . .	11-20
<b>Delays and Timing . . . . .</b>	<b>11-22</b>
Transport and Inertial Delays . . . . .	11-23
The Inertial Delay Implementation . . . . .	11-25
Enabling Transport Delays . . . . .	11-26
Pulse Control . . . . .	11-27
Pulse Control With Transport Delays . . . . .	11-29
Pulse Control With Inertial Delays . . . . .	11-31
Specifying Pulse on Event or Detect Behavior . . . . .	11-35
Specifying the Delay Mode . . . . .	11-40
Support for Delayed Annotation During Simultaneous Switching on Inputs . . . . .	11-42
Usage Example . . . . .	11-42
<b>Using the Configuration File to Disable Timing . . . . .</b>	<b>11-43</b>
<b>Using the timopt Timing Optimizer . . . . .</b>	<b>11-44</b>
Editing the timopt.cfg File . . . . .	11-46
Editing Potential Sequential Device Entries . . . . .	11-46
Editing Clock Signal Entries . . . . .	11-47
<b>Using Scan Simulation Optimizer . . . . .</b>	<b>11-48</b>

ScanOpt Configuration File Format . . . . .	11-49
ScanOpt Assumptions . . . . .	11-50
Improving the ScanOpt for Debug Support . . . . .	11-51
Use Model . . . . .	11-51
Usage Example . . . . .	11-52
Negative Timing Checks . . . . .	11-54
The Need for Negative Value Timing Checks . . . . .	11-55
The \$setuphold Timing Check Extended Syntax . . . . .	11-60
Negative Timing Checks for Asynchronous Controls . . . . .	11-63
The \$recrm Timing Check Syntax . . . . .	11-64
Enabling Negative Timing Checks . . . . .	11-66
Other Timing Checks Using the Delayed Signals . . . . .	11-67
Checking Conditions . . . . .	11-71
Toggling the Notifier Register . . . . .	11-72
SDF Back-Annotation to Negative Timing Checks . . . . .	11-73
How VCS Calculates Delays . . . . .	11-74
Using VITAL Models and Netlists . . . . .	11-76
Validating and Optimizing a VITAL Model . . . . .	11-77
Validating the Model for VITAL Conformance . . . . .	11-77
Verifying the Model for Functionality . . . . .	11-78
Optimizing the Model for Performance and Capacity . . . . .	11-78
Re-Verifying the Model for Functionality . . . . .	11-79
Understanding Error and Warning Messages . . . . .	11-80
Distributing a VITAL Model . . . . .	11-81
Simulating a VITAL Netlist . . . . .	11-82
Applying Stimulus . . . . .	11-82
Overriding Generic Parameter Values . . . . .	11-82

Understanding VCS Error Messages . . . . .	11-84
Viewing VITAL Subprograms . . . . .	11-85
Timing Back-annotation . . . . .	11-85
VCS Naming Styles . . . . .	11-85
Negative Constraints Calculation (NCC) . . . . .	11-86
Simulating in Functional Mode . . . . .	11-87
Understanding VITAL Timing Delays and Error Messages . . . . .	11-88
Negative Constraint Calculation (NCC) . . . . .	11-89
Conformance Checks . . . . .	11-89
Error Messages . . . . .	11-91
Support for Identifying Non-Annotated Timing Arc and Timing Check Statements . . . . .	11-101
Usage Example . . . . .	11-102
<b>12. Coverage</b>	
Code Coverage . . . . .	12-2
Functional Coverage . . . . .	12-3
Options For Coverage Metrics . . . . .	12-3
<b>13. Using OpenVera Native Testbench</b>	
Usage Model . . . . .	13-3
Example . . . . .	13-4
Usage Model . . . . .	13-8
Importing VHDL Procedures . . . . .	13-8
Exporting OpenVera Tasks . . . . .	13-10
Using Template Generator . . . . .	13-11
Example . . . . .	13-12

<b>Key Features . . . . .</b>	<b>13-24</b>
Multiple Program Support . . . . .	13-24
Configuration File Model . . . . .	13-25
Configuration File . . . . .	13-25
Usage Model for Multiple Programs . . . . .	13-26
NTB Options and the Configuration File . . . . .	13-28
Class Dependency Source File Reordering . . . . .	13-29
Circular Dependencies . . . . .	13-31
Dependency-based Ordering in Encrypted Files . . . . .	13-32
Using Encrypted Files . . . . .	13-32
Functional Coverage . . . . .	13-33
Using Reference Verification Methodology . . . . .	13-33
Limitations . . . . .	13-35

## 14. Using SystemVerilog

<b>Use Model . . . . .</b>	<b>14-2</b>
Using UVM With VCS . . . . .	14-3
Update on UVM-1.2 . . . . .	14-4
Natively Compiling and Elaborating UVM-1.1d . . . . .	14-4
Natively Compiling and Elaborating UVM-1.2 . . . . .	14-5
Compiling the External UVM Library . . . . .	14-6
Using the -ntb_opts uvm Option . . . . .	14-6
Explicitly Specifying UVM Files and Arguments . . . . .	14-7
Accessing HDL Registers Through UVM Backdoor . . . . .	14-8
Generating UVM Register Abstraction Layer Code . . . . .	14-9
Recording UVM Transactions . . . . .	14-9
Recording UVM Phases . . . . .	14-10

UVM Template Generator . . . . .	14-11
Using Mixed VMM/UVM Libraries . . . . .	14-12
Migrating from OVM to UVM . . . . .	14-14
Where to Find UVM Examples. . . . .	14-15
Where to Find UVM Documentation . . . . .	14-15
UVM-1.1d Documentation . . . . .	14-15
UVM-VMM Interop Documentation . . . . .	14-16
Using VMM with VCS . . . . .	14-16
Using OVM with VCS . . . . .	14-17
Native Compilation and Elaboration of OVM 2.1.2 . . . . .	14-17
Compiling the External OVM Library . . . . .	14-19
Using the -ntb_opts ovm Option. . . . .	14-19
Explicitly Specifying OVM Files and Arguments. . . . .	14-20
Recording OVM Transactions . . . . .	14-20
Debugging SystemVerilog Designs . . . . .	14-22
Functional Coverage . . . . .	14-23
SystemVerilog Constructs . . . . .	14-23
Extern Task and Function Calls through Virtual Interfaces . . . . .	14-25
Modport Expressions in an Interface . . . . .	14-28
Limitations . . . . .	14-30
Interface Classes . . . . .	14-30
Difference Between Extends and Implements . . . . .	14-33
Cast and Interface Class . . . . .	14-36
Name Conflicts and Resolution . . . . .	14-37
Interface Class and Randomization . . . . .	14-41
Package Exports . . . . .	14-42

Severity System Tasks as Procedural Statements .....	14-43
Width Casting Using Parameters.....	14-44
The std::randomize() Function.....	14-47
SystemVerilog Bounded Queues.....	14-50
wait() Statement with a Static Class Member Variable .....	14-51
Support for Consistent Behavior of Class Static Properties..	14-52
Parameters and Local Parameters in Classes.....	14-53
SystemVerilog Math Functions .....	14-54
Streaming Operators .....	14-55
Packing (Used on RHS).....	14-55
Unpacking (Used on LHS) .....	14-56
Packing and Unpacking .....	14-56
Propagation and force Statement.....	14-56
Error Conditions .....	14-56
Structures with Streaming Operators .....	14-57
Support for with Expression .....	14-57
Constant Functions in Generate Blocks.....	14-60
Support for Aggregate Methods in Constraints Using the “with” Construct .....	14-62
Debugging During Initialization SystemVerilog Static Functions and Tasks in Module Definitions .....	14-63
Explicit External Constraint Blocks .....	14-67
Generate Constructs in Program Blocks .....	14-70
Error Condition for Using a Genvar Variable Outside of its Generate Block.....	14-72
Randomizing Unpacked Structs.....	14-73
Using the Scope Randomize Method std::randomize() ..	14-73
Using the Class Randomize Method <code>randomize()</code> ..	14-77

Disabling and Re-enabling Randomization . . . . .	14-79
Using In-Line Random Variable Control . . . . .	14-83
Limitation . . . . .	14-87
Making wait fork Statements Compliant with the SV LRM . . . . .	14-88
Making disable fork Statements Compliant with the SV LRM . . . . .	14-90
Using a Package in a SystemVerilog Module, Program, and Interface Header . . . . .	14-92
Disabling DPI Tasks and Functions . . . . .	14-94
Use Model . . . . .	14-95
Support for Overriding Parameter Values through Configuration . . . . .	14-95
Example . . . . .	14-95
Precedence Override Rules . . . . .	14-96
Limitations . . . . .	14-97
Support for Inclusion of Dynamic Types in Sensitivity List . . . . .	14-97
Support for Assignment Pattern Expression in Non-Assignment Like Context . . . . .	14-99
Usage Example . . . . .	14-99
Limitations . . . . .	14-100
User-Defined Nettypes . . . . .	14-101
The Resolution Function . . . . .	14-101
Limitations . . . . .	14-102
Example of User-Defined Nettype . . . . .	14-102
Example of User-Defined Nettype in Arrays . . . . .	14-103
Example of Nettype MDAs of Type Real . . . . .	14-105
Example of Nettype MDAs of Type Unpacked Struct . . . . .	14-105
Support for Connecting Nettypes through Transif Gates . . . . .	14-106
Limitations . . . . .	14-107

Generic Interconnect Nets . . . . .	14-107
Converting Wire Nets and Logic Variables to Interconnect Nets	
14-108	
Limitations . . . . .	14-108
Support for Associative Array With Unpacked Structure as Key	
14-109	
Specifying a SystemVerilog Keyword Set by LRM Version at Command Line . . . . .	14-110
The -sv Compile-Time Option . . . . .	14-111
The `begin_keywords and `end_keywords Compiler Directives	
14-112	
Support for .triggered Property with Clocking Block Name . .	14-113
Usage Examples . . . . .	14-114
Support for Intra Assignment Delay With Non-Blocking Assignments in Program Block . . . . .	14-115
Limitations . . . . .	14-115
Extensions to SystemVerilog . . . . .	14-115
Unique/Priority Case/IF Final Semantic Enhancements (-xlrn uniq_prior_final Compile-Time Option) . . . . .	14-116
Using Unique/Priority Case/If with Always Block or Continuous Assign . . . . .	14-117
Using Unique/Priority Inside a Function . . . . .	14-120
System Tasks to Control Warning Messages . . . . .	14-123
Controlling Runtime Warning Messages Generated Using Unique/ Priority If Constructs . . . . .	14-124
Support for Unique0 in Conditional Statements . . . . .	14-126
Usage Example . . . . .	14-127
Enhancements to the -xlrn uniq_prior_final Compile-Time Option	
14-128	

Single-Sized Packed Dimension Extension . . . . .	14-134
Covariant Virtual Function Return Types . . . . .	14-136
Self Instance of a Virtual Interface . . . . .	14-138
UVM Example . . . . .	14-140
Support for Shuffle Method for MultiDimensional Arrays . . . . .	14-141
Use Model . . . . .	14-141
Usage Example . . . . .	14-141
Enhanced Clocking Block Behavior When Skew is negedge/posedge 14-144	
Usage Example . . . . .	14-144
Support for Slice of String Variable . . . . .	14-146

## 15. Aspect Oriented Extensions

Aspect-Oriented Extensions in SystemVerilog . . . . .	15-3
Processing of Aspect-Oriented Extensions as a Precompilation Expansion. . . . .	15-5
Weaving Advice Into the Target Method . . . . .	15-10
Precompilation Expansion Details . . . . .	15-15
Precedence . . . . .	15-16

## 16. Using Constraints

Support for Array Slice in Unique Constraints . . . . .	16-2
Support for Object Handle Comparison in Constraint Guards . . . . .	16-4
Support for Pure Constraint Block . . . . .	16-8
Support for SystemVerilog Bit Vector Functions in Constraints . . . . .	16-14
\$countones Function . . . . .	16-16

\$onehot Function . . . . .	16-17
\$onehot0 Function . . . . .	16-18
\$countbits Function . . . . .	16-20
\$bits Function . . . . .	16-21
Inconsistent Constraints . . . . .	16-23
Constraint Debug . . . . .	16-25
Partition . . . . .	16-26
Randomize Serial Number. . . . .	16-28
Solver Trace. . . . .	16-29
Constraint Profiler . . . . .	16-34
Test Case Extraction . . . . .	16-35
Using multiple +ntb_solver_debug arguments . . . . .	16-37
Summary for the +ntb_solver_debug Option . . . . .	16-37
+ntb_solver_debug=serial . . . . .	16-37
+ntb_solver_debug=trace . . . . .	16-37
+ntb_solver_debug=profile . . . . .	16-38
+ntb_solver_debug=extract . . . . .	16-38
+ntb_solver_debug=verbose . . . . .	16-38
Support for Save and Restore Stimulus. . . . .	16-39
Use Model . . . . .	16-39
Limitations . . . . .	16-40
Constraint Guard Error Suppression . . . . .	16-40
Error Message Suppression Limitations . . . . .	16-42
Flattening Nested Guard Expressions . . . . .	16-42
Pushing Guard Expressions into Foreach Loops . . . . .	16-43

Support for Array and Cross-Module References in <code>std::randomize()</code>	
16-44	
Error Conditions . . . . .	16-46
Support for Cross-Module References in Constraints . . . . .	16-46
XMR Function Calls in Constraints . . . . .	16-48
State Variable Index in Constraints . . . . .	16-49
Runtime Check for State Versus Random Variables . . . . .	16-49
Array Index . . . . .	16-50
Using DPI Function Calls in Constraints . . . . .	16-50
Invoking Non-pure DPI Functions from Constraints. . . . .	16-51
Using Foreach Loops Over Packed Dimensions in Constraints .	16-55
Memories with Packed Dimensions. . . . .	16-55
Single Packed Dimension . . . . .	16-55
Multiple Packed Dimensions . . . . .	16-56
MDAs with Packed Dimensions. . . . .	16-56
Single Packed Dimension . . . . .	16-56
Multiple Packed Dimensions . . . . .	16-56
Just Packed Dimensions . . . . .	16-57
The foreach Iterative Constraint for Packed Arrays . . . . .	16-58
Randomized Objects in a Structure. . . . .	16-59
Support for Typecast in Constraints . . . . .	16-61
Syntax . . . . .	16-61
Description . . . . .	16-61
Strings in Constraints . . . . .	16-64
SystemVerilog LRM 1800™-2012 Update . . . . .	16-65

Using Soft Constraints in SystemVerilog . . . . .	16-65
Using Soft Constraints . . . . .	16-66
Soft Constraint Prioritization. . . . .	16-67
Soft Constraints Defined in Classes Instantiated as rand Members in Another Class . . . . .	16-68
Soft Constraints Inheritance Between Classes . . . . .	16-70
Soft Constraints in AOP Extensions to a Class . . . . .	16-71
Soft Constraints in View Constraints Blocks . . . . .	16-74
Discarding Lower-Priority Soft Constraints. . . . .	16-78
Unique Constraints . . . . .	16-80
Enhancement to the Randomization of Multidimensional Array Functionality. . . . .	16-82
Limitation . . . . .	16-84
Supporting Random Array Index . . . . .	16-84
Limitation . . . . .	16-85
Supporting System Function Calls . . . . .	16-85
\$size() System Function Call . . . . .	16-86
\$clog2() System Function Call. . . . .	16-87
Usage Example . . . . .	16-87
Supporting Foreach Loop Iteration over Array Select . . . . .	16-88
Support for Enumerated Type Methods in a Constraint Expression	16-89
Usage Example . . . . .	16-89
Limitation . . . . .	16-90
<b>17. Extensions for SystemVerilog Coverage</b>	
Support for Reference Arguments in get_coverage() and get_inst_coverage() . . . . .	17-1

get_coverage() method .....	17-2
get_inst_coverage() method .....	17-3
Functional Coverage Methodology Using the SystemVerilog C/C++ Interface.....	17-3
SystemVerilog Functional Coverage Flow .....	17-5
Covergroup Definition .....	17-6
SystemVerilog (Covergroup for C/C++): covg.sv .....	17-7
C Testbench: test.c.....	17-7
Approach #1: Passing Arguments by Reference .....	17-8
Approach #2: Passing Arguments by Value .....	17-8
Compile Flow .....	17-8
Runtime .....	17-9
C/C++ Functional Coverage API Specification .....	17-9
<b>18. OpenVera-SystemVerilog Testbench Interoperability</b>	
Scope of Interoperability .....	18-2
Importing OpenVera Types Into SystemVerilog.....	18-3
Data Type Mapping .....	18-6
Mailboxes and Semaphores .....	18-7
Events .....	18-9
Strings .....	18-9
Enumerated Types .....	18-10
Integers and Bit-Vectors .....	18-12
Arrays .....	18-13
Structs and Unions .....	18-15
Connecting to the Design .....	18-15

Mapping Modports to Virtual Ports . . . . .	18-15
Virtual Modports . . . . .	18-16
Importing Clocking Block Members Into a Modport . . . . .	18-16
Semantic Issues With Samples, Drives, and Expects . . . . .	18-22
Notes to Remember . . . . .	18-22
Blocking Functions in OpenVera . . . . .	18-22
Constraints and Randomization . . . . .	18-23
Functional Coverage . . . . .	18-23
Usage Model . . . . .	18-25
Limitations . . . . .	18-26
<b>19. Using SystemVerilog Assertions</b>	
Using SVAs in the HDL Design . . . . .	19-3
Using VCS Checker Library . . . . .	19-3
Instantiating SVA Checkers in Verilog . . . . .	19-3
Instantiating SVA Checkers in VHDL . . . . .	19-5
Binding SVA to a Design . . . . .	19-7
Inlining SVAs in the Verilog Design . . . . .	19-8
Use Model . . . . .	19-9
Inlining SVA in the VHDL design . . . . .	19-10
Use Model . . . . .	19-11
Number of SystemVerilog Assertions Supported in a Module	19-13
Controlling SystemVerilog Assertions . . . . .	19-13
Compilation/Elaboration and Runtime Options . . . . .	19-13
Concatenating Assertion Options . . . . .	19-17
Assertion Monitoring System Tasks . . . . .	19-17

Using Assertion Categories . . . . .	19-21
Using System Tasks . . . . .	19-21
Using Attributes . . . . .	19-22
Starting and Stopping Assertions Using Assertion System Tasks 19-24	
Viewing Results . . . . .	19-29
Using a Report File . . . . .	19-30
Enhanced Reporting for SystemVerilog Assertions in Functions	19-30
Introduction . . . . .	19-31
Use Model . . . . .	19-32
Name Conflict Resolution . . . . .	19-33
Checker and Generate Blocks . . . . .	19-33
Controlling Assertion Failure Messages . . . . .	19-33
Introduction . . . . .	19-34
Options for Controlling Default Assertion Failure Messages .	19-34
Options to Control Termination of Simulation . . . . .	19-36
Option to Enable Compilation of OVA Case Pragmas . . . . .	19-39
Reporting Values of Variables in the Assertion Failure Messages	19-40
Limitations . . . . .	19-41
Reporting Messages When \$uniq_prior_checkon/\$uniq_prior_checkoff System Tasks are Called . . . . .	19-42
Assertion and Unique/Priority Re-Trigger Feature . . . . .	19-44
Flushing Off the Assertion Re-Trigger Feature . . . . .	19-46
Enabling Lint Messages for Assertions . . . . .	19-47
Fail-Only Assertion Evaluation Mode . . . . .	19-50

Key Points to Note . . . . .	19-51
Limitations . . . . .	19-53
Treating x as true on an Assertion Precondition . . . . .	19-54
Use Model . . . . .	19-55
Usage Example . . . . .	19-55
Using SystemVerilog Constructs Inside vunits . . . . .	19-56
Limitations . . . . .	19-57
Calling \$error Task When Else Block is Not Present. . . . .	19-58
Disabling Default Assertion Success Dumping in -debug_access Option 19-59	
List of supported IEEE Std. 1800-2012 Compliant SVA Features	19-59
Support for \$countbits System Function . . . . .	19-62
Support for Real Data Type Variables . . . . .	19-63
Support for \$assertcontrol Assertion Control System Task . . . . .	19-63
Limitations . . . . .	19-64
Enabling IEEE Std. 1800-2012 Compliant Features . . . . .	19-64
Limitations . . . . .	19-65
SystemVerilog Assertions Limitations . . . . .	19-65
Debug Support for New Constructs . . . . .	19-65
Note on Cross Features . . . . .	19-66
<b>20. Using Property Specification Language</b>	
Including PSL in the Design . . . . .	20-1
Examples . . . . .	20-2
Use Model . . . . .	20-3

Examples . . . . .	20-4
Examples . . . . .	20-5
PSL Assertions Inside VHDL Block Statements in Vunit . . . . .	20-6
Introduction . . . . .	20-6
Use Model . . . . .	20-7
Limitations . . . . .	20-8
PSL Macro Support in VHDL . . . . .	20-9
Using the %for Construct . . . . .	20-9
Using the %if Construct . . . . .	20-12
Using Expressions with %if and %for Constructs . . . . .	20-13
PSL Macro Support Limitations . . . . .	20-14
Using SVA Options, SVA System Tasks, and OV Classes . . . . .	20-15
Limitations . . . . .	20-16
<b>21. Using SystemC</b>	
<b>22. C Language Interface</b>	
Using PLI . . . . .	22-2
Writing a PLI Application . . . . .	22-3
Functions in a PLI Application . . . . .	22-4
Header Files for PLI Applications . . . . .	22-5
PLI Table File . . . . .	22-6
Syntax . . . . .	22-6
Using the PLI Table File . . . . .	22-23
Enabling ACC Capabilities . . . . .	22-24
Enabling ACC Capabilities Globally . . . . .	22-24

Feedback

Using the Configuration File . . . . .	22-25
Selected ACC Capabilities . . . . .	22-28
Using VPI Routines . . . . .	22-33
Support for VPI Callbacks . . . . .	22-34
Support for the <code>vpi_register_systf</code> Routine. . . . .	22-35
Integrating a VPI Application With VCS . . . . .	22-36
PLI Table File for VPI Routines . . . . .	22-37
Virtual Interface Debug Support. . . . .	22-38
Example . . . . .	22-38
Limitations . . . . .	22-41
Unimplemented VPI Routines . . . . .	22-41
Modified VPI Features . . . . .	22-43
Using VHPI Routines. . . . .	22-46
Diagnostics for VPI/VHPI PLI Applications . . . . .	22-47
Using DirectC . . . . .	22-47
Using Direct C/C++ Function Calls . . . . .	22-49
Functioning of C/C++ Code in a Verilog Environment . . . . .	22-51
Declaring the C/C++ Function . . . . .	22-52
Calling the C/C++ Function . . . . .	22-58
Storing Vector Values in Machine Memory. . . . .	22-60
Converting Strings . . . . .	22-62
Avoiding a Naming Problem. . . . .	22-65
Using Pass by Reference. . . . .	22-66
Using Direct Access. . . . .	22-67
Using the <code>vc_hdrs.h</code> File. . . . .	22-74
Access Routines for Multi-Dimensional Arrays. . . . .	22-75

Using Abstract Access . . . . .	22-76
Using vc_handle . . . . .	22-77
Using Access Routines . . . . .	22-78
Summary of Access Routines . . . . .	22-122
Enabling C/C++ Functions . . . . .	22-127
Mixing Direct And Abstract Access . . . . .	22-129
Specifying the DirectC.h File . . . . .	22-129
Extended BNF for External Function Declarations . . . . .	22-130
 23. Support for VHDL 2002 and 2008	
VHDL 2002 Protected Type . . . . .	23-2
Use Model . . . . .	23-2
Limitations of VHDL 2002 Protected Type . . . . .	23-2
VHDL 2008 Constructs . . . . .	23-2
Array Types and Operators . . . . .	23-4
Adding Comments . . . . .	23-5
Use Clause and Aliases . . . . .	23-6
Support for Bit String Literals . . . . .	23-7
Support for TO_STRING Conversion . . . . .	23-10
Support for External Names . . . . .	23-12
Specifying The all Keyword in the Process Sensitivity List . . . . .	23-13
Support for Logical Unary Reduction Operator . . . . .	23-14
Support for Matching Relational Operators for Bit and std_ulogic . . . . .	23-15
Including Non-Static Expressions in Port Map . . . . .	23-16
Standard Environment Package . . . . .	23-18
Package Declaration and Instantiations . . . . .	23-18

Limitation . . . . .	23-22
Referencing Interface Lists . . . . .	23-22
Limitations . . . . .	23-24
Overriding the Value Assigned to a Signal. . . . .	23-27
Forcing and Releasing Values of Signals . . . . .	23-28
Forcing and Releasing Ports of a Design. . . . .	23-29
Assigning Composite Value to a Collection of Signals . . . . .	23-31
Forcing and Releasing Assignment Written in a Subprogram 23-32	
Forcing and Releasing Multiple Concurrent Assignments	23-32
Debugging the Force and Release Assignments . . . . .	23-32
Matching Case Statements . . . . .	23-33
Conditional Elaboration . . . . .	23-36
Condition Operator in an Expression. . . . .	23-43
Reading Output Port . . . . .	23-47
2008 IEEE Packages. . . . .	23-49
Overview of Additional IEEE Packages . . . . .	23-49
Resolved Elements . . . . .	23-56
Usage Example . . . . .	23-57
Conditional and Selected Assignments . . . . .	23-61
Use Model . . . . .	23-63
Usage Example . . . . .	23-63
Context Declaration . . . . .	23-67
Usage Example . . . . .	23-68
Improved I/O . . . . .	23-70
Usage Example . . . . .	23-71
Support for Implicitly Constrained Array Elements. . . . .	23-74
Usage Example . . . . .	23-75

Support for Unconstrained Element Types .....	23-76
Usage Example .....	23-78
Support for Enhanced Generics in Entity Interfaces .....	23-83
Usage Example .....	23-83
Limitation .....	23-90
Support for Slices in Array Aggregates .....	23-90
Usage Example .....	23-91
Limitation .....	23-95
Support for Type Conversion in VHDL 2008 .....	23-95
Usage Example .....	23-96
Limitation .....	23-98
Support for Case Expression Subtype.....	23-98
Example for Case Expression .....	23-98
Support for Subtypes of Ports and Parameters.....	23-103
Example for Subtypes of Ports and Parameters .....	23-103
Support for Static Composite Expressions .....	23-104
Usage Examples .....	23-105

## 24. SAIF Support

Using SAIF Files with VCS .....	24-2
SAIF System Tasks for Verilog or Verilog-Top Designs.....	24-2
The Flows to Generate a Backward SAIF File .....	24-6
Generating an SDPD Backward SAIF File.....	24-6
Generating a Non-SPDP Backward SAIF File.....	24-7
SAIF Calls That Can Be Used on VHDL or VHDL-Top Designs .	24-8
SAIF Support for Two-Dimensional Memories in v2k Designs ..	24-9

UCLI SAIF Dumping . . . . .	24-10
Criteria for Choosing Signals for SAIF Dumping . . . . .	24-10
Improving Simulation Time by Reducing the Overhead due to SAIF File Dumping . . . . .	24-11
Use Model . . . . .	24-11
Example . . . . .	24-12
Limitations . . . . .	24-13
 25. Encrypting Source Files	
IEEE Verilog Standard 1364-2005 Encryption . . . . .	25-2
The Protection Header File . . . . .	25-3
Unsupported Protection Pragma Expressions . . . . .	25-6
Other Options for IEEE Std 1364-2005 Encryption Mode . . . . .	25-6
How Protection Envelopes Work . . . . .	25-9
VCS Public Encryption Key . . . . .	25-10
Creating Interoperable Digital Envelopes Using VCS - Example . . . . .	25-11
Discontinued -ipkey Option . . . . .	25-15
IEEE VHDL Standard 1076-2008 Encryption . . . . .	25-15
VHDL 1076-2008 Encryption Use Model . . . . .	25-16
Encrypting the Entire VHDL Source Files . . . . .	25-17
Encrypting the Parts of VHDL Source Files . . . . .	25-17
Using the Protection Header File . . . . .	25-18
Options for VHDL 1076-2008 Encryption Mode . . . . .	25-20
Protection Envelopes . . . . .	25-21
The VCS Public Encryption Key . . . . .	25-22

Usage Example . . . . .	25-23
Example for Full Encryption . . . . .	25-23
Example for Partial Encryption . . . . .	25-25
Debug Protection . . . . .	25-26
Combining Encrypted and Unencrypted Code . . . . .	25-27
Assertion and Report Statements . . . . .	25-29
Hierarchy Attributes . . . . .	25-29
Profiling . . . . .	25-29
VHPI . . . . .	25-29
VPD / VCD . . . . .	25-30
Coverage . . . . .	25-30
Error Messages . . . . .	25-30
Limitations . . . . .	25-30
 128-bit Advanced Encryption Standard . . . . .	25-32
Compiler Directives for Source Protection . . . . .	25-32
Using Compiler Directives or Pragmas . . . . .	25-33
Example . . . . .	25-34
Automatic Protection Options . . . . .	25-37
Using Automatic Protection Options . . . . .	25-40
Protecting ‘include File Directive . . . . .	25-48
+autoincludeprotect . . . . .	25-48
Enabling Debug Access to Ports and Instance Hierarchy . . . . .	25-49
+autobodyprotect . . . . .	25-49
Debugging Partially Encrypted Source Code . . . . .	25-49
 Skipping Encrypted Source Code . . . . .	25-50
gen_vcs_ip . . . . .	25-51
Syntax . . . . .	25-53

Analysis Options . . . . .	25-53
Exporting The IP . . . . .	25-55
Use Model . . . . .	25-55
IP Vendor . . . . .	25-55
IP Generation . . . . .	25-55
IP User . . . . .	25-56
Licensing . . . . .	25-56
<b>26. VCS Fine-Grained Parallelism Technology</b>	
Introduction . . . . .	26-58
Use Model . . . . .	26-61
Profiling to Detect Design Suitability for Parallelism . . . . .	26-66
Limitation . . . . .	26-67
<b>27. Integrating VC Formal With Coverage and Planner</b>	
Introduction to VC Formal . . . . .	27-2
VC Formal Coverage With Verdi Coverage and Planner . . . . .	27-3
Use Model . . . . .	27-3
Collecting VC Formal Results in the Coverage Database	27-3
Measuring VC Formal Assert Status in HVP . . . . .	27-7
<b>28. Integrating VCS With Certitude</b>	
Introduction to Certitude . . . . .	28-1
VCS and Certitude Integration . . . . .	28-2
Loading Designs Automatically in Verdi with Native Certitude . . . . .	28-4

Use Model . . . . .	28-4
Points to Note . . . . .	28-5
Dumping and Comparing Waveforms in Verdi for SystemC Designs	28-5
Use Model . . . . .	28-6
Point to Note . . . . .	28-7
Reducing Compilation Time in Native Certitude With VCS Partition	
Compile Flow . . . . .	28-8
Use Model . . . . .	28-8
Example . . . . .	28-9
Limitation . . . . .	28-11
<b>29. Integrating VCS with Vera</b>	
Setting Up Vera and VCS . . . . .	29-2
Using Vera with VCS . . . . .	29-3
Usage Model . . . . .	29-4
Two-Step Flow . . . . .	29-4
Three-Step Flow . . . . .	29-4
<b>30. VCS Mixed-Signal Simulation</b>	
Introduction to VCS and CustomSim . . . . .	30-2
Analyzing a Design . . . . .	30-3
Elaborating a Design . . . . .	30-3
Running the Simulation . . . . .	30-4
Setting up the Environment . . . . .	30-4
Licenses . . . . .	30-4
Required UNIX Paths and Variable Settings . . . . .	30-4

Feedback

Use Model . . . . .	30-5
Example . . . . .	30-6
Scheduling Analog-to-Digital Events in the NBA Region . . . . .	30-7
Use Model . . . . .	30-8
Support of Verilog Force and Release Assignments on Wreal Nets	30-8
Usage Example . . . . .	30-9
Limitations . . . . .	30-10
Support for Wreal Nets in Verilog-AMS Flow . . . . .	30-11
Usage Example . . . . .	30-11
Support for SystemC Designs in Verilog-AMS . . . . .	30-13
Use Model . . . . .	30-13
Usage Example . . . . .	30-13
Support for Wildcard Character and the -exclude Option in the Mixed Signal Control Command . . . . .	30-18
Use Model . . . . .	30-18
Usage Example . . . . .	30-19
Limitation . . . . .	30-19
<b>31. Integrating VCS with Specman</b>	
Type Support . . . . .	31-2
Usage Flow . . . . .	31-4
Setting Up The Environment . . . . .	31-4
Specman e code accessing VHDL only . . . . .	31-5
Specman e Code Accessing Verilog Only . . . . .	31-7
Specman e code accessing both VHDL and Verilog . . . . .	31-9

Guidelines for Specifying HDL Path or Tick Access with VCS-Specman Interface .....	31-12
Using specrun and specview .....	31-13
Version Checker for Specman .....	31-14
Use Model .....	31-15
<b>32. Integrating VCS with Denali</b>	
Setting Up Denali Environment for VCS .....	32-1
Integrating Denali with VCS .....	32-2
Use Model .....	32-2
Use Model for VHDL Memory Models .....	32-3
Use Model for Verilog Memory Models .....	32-4
Two-Step Flow .....	32-4
Three-Step Flow .....	32-5
Execute Denali Commands at UCLI Prompt .....	32-5
<b>33. Integrating VCS with Native Low Power (NLP)</b>	
<b>34. Unified UVM Library for VCS and Verdi</b>	
Transaction/Message Recording in Verdi with VCS .....	34-4
Compilation .....	34-4
Enabling FSDB Transaction Recording .....	34-4
Simulation .....	34-5
Dumping Transactions or Messages in Verdi Flow .....	34-5

## **35. Debugging with Verdi**

Introduction . . . . .	35-2
Generating Verdi KDB . . . . .	35-3
Reading Compiled Design with Verdi . . . . .	35-5
Example . . . . .	35-7
Key Points to Note . . . . .	35-8
Limitations . . . . .	35-9
Dumping FSDB File for Various Flows . . . . .	35-10
Setting Up Verdi . . . . .	35-10
Use Model for FSDB Dumping . . . . .	35-10
Using VHDL Procedures or Verilog System Tasks . . . . .	35-13
Using UCLI . . . . .	35-13
Examples . . . . .	35-14
Interactive and Post-Processing Debug . . . . .	35-17
Prerequisites . . . . .	35-17
Interactive Simulation Debug Flow . . . . .	35-18
Key Points to Note . . . . .	35-22
Post-Processing Debug Flow . . . . .	35-23
Reducing Disk Space for Post-Process Only Debug . . . . .	35-23

## **Appendix A. VCS Environment Variables**

Simulation Environment Variables . . . . .	A-1
Setup Variables . . . . .	A-3
Analysis Setup Variables . . . . .	A-4
Compilation/Elaboration Setup Variables . . . . .	A-6
Simulation Setup Variables . . . . .	A-11

## **Feedback**

C Compilation and Linking Setup Variables . . . . .	A-18
Timescale Implementation . . . . .	A-20
Understanding `timescale . . . . .	A-21
Verilog only and Verilog Top Mixed Design . . . . .	A-25
VHDL only and VHDL Top Mixed Designs . . . . .	A-25
Setting up Simulator Resolution From Command Line . . . . .	A-26
Other Useful Timescale Related Options . . . . .	A-28
Non-Compatible Options . . . . .	A-30
Limitations . . . . .	A-30
Optional Environment Variables . . . . .	A-30
Using Environment Variables in Verilog Source Code. . . . .	A-35

## Appendix B. Analysis Utilities

The vhdlan Utility . . . . .	B-2
Using Smart Order . . . . .	B-8
Use Model . . . . .	B-8
Limitations . . . . .	B-10
The vlogan Utility . . . . .	B-11

## Appendix C. Compilation/Elaboration Options

Option for Code Generation. . . . .	C-5
Options for Accessing Verilog Libraries . . . . .	C-5
Options for Incremental Compilation . . . . .	C-8
Options for Help. . . . .	C-10
Option for SystemVerilog . . . . .	C-10
Options for SystemVerilog Assertions . . . . .	C-10

Feedback

Options to Enable Compilation of OVA Case Pragmas . . . . .	C-22
Options for Native Testbench. . . . .	C-22
Options for Different Versions of Verilog . . . . .	C-30
Option for Initializing Verilog Variables, Registers and Memories with Random Values . . . . .	C-32
Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design . . . . .	C-35
Options for Selecting Register or Memory Initialization . . . . .	C-39
Options for Using Radiant Technology. . . . .	C-40
Options for Starting Simulation Right After Compilation . . . . .	C-40
Options for Specifying Delays and SDF Files . . . . .	C-40
Options for Compiling an SDF File . . . . .	C-49
Options for Specify Blocks and Timing Checks . . . . .	C-49
Options for Pulse Filtering . . . . .	C-51
Options for Negative Timing Checks . . . . .	C-52
Options for Profiling Your Design. . . . .	C-53
Options to Specify Source Files and Compilation/Elaboration Options in a File. . . . .	C-54
Options for Compiling Runtime Options Into the Executable.	C-56
Options for PLI Applications. . . . .	C-57
Options to Enable the VCS DirectC Interface . . . . .	C-59
Options for Flushing Certain Output Text File Buffers . . . . .	C-60
Options for Simulating SWIFT VMC Models and SmartModels	C-61
Options for Controlling Messages . . . . .	C-62
Option to Run VCS in Syntax Checking Mode. . . . .	C-68
Limitations . . . . .	C-69
Options for Cell Definition . . . . .	C-70

Options for Licensing . . . . .	C-72
Options for Controlling the Linker . . . . .	C-73
Options for Controlling the C Compiler . . . . .	C-75
Options for Source Protection . . . . .	C-78
Options for Mixed Analog/Digital Simulation . . . . .	C-78
Unified Option to Change Generic and Parameter Values . . . . .	C-79
Options for Changing Parameter Values . . . . .	C-80
Checking for x and z Values in Conditional Expressions . . . . .	C-80
Options for Detecting Race Conditions . . . . .	C-80
Options to Specify the Time Scale . . . . .	C-81
Option to Exclude Environment Variables During Timestamp Checks C-86	
Options for Overriding Parameters . . . . .	C-87
Global -check_all Option . . . . .	C-93
Use Model . . . . .	C-93
Limitation . . . . .	C-94
Option to Enable Bounds Check at Compile-Time . . . . .	C-95
Option to Enable Bounds Check at Runtime . . . . .	C-96
Error-[DT-OBAE] Out of Bounds Access for Queues . . . . .	C-96
Error-[DT-OBAE] Out of Bounds Access for Dynamic Arrays	C-97
Warning-[AOOBAW] Array Out of Bounds Access for Fixed Size Unpacked Arrays . . . . .	C-98
Warning-[AOOBAW] Array Out of Bounds Access for Fixed Size Packed Arrays . . . . .	C-99
Error-[DT-OBAE] Intermediate Access for Dynamic Arrays	C-100
Warning-[AAIIW] Array Access with Intermediate Index . . . . .	C-100
Warning-[AAIIW] Array Access with Intermediate Index for Fixed Size Packed Arrays . . . . .	C-101

Option to Enable Extra Runtime Checks in VHDL . . . . .	C-102
Error-[SIMERR_FPEXCEPTION] Floating Point Exception	C-103
Error-[SIMERR_FDIVZERO_SCOPE] Divide by Zero Error	C-104
Error-[SIMERR_NEGTIME] Wrong Time Format . . . . .	C-104
Error-[SIMERR_CONSTERR_INDEXEDRANGE] Range Violation Error . . . . .	C-105
Error-[SIMERR_CONSTERR] Array Constraint Error . . .	C-106
Error-[SIMERR_INCONSISTENTIS] Subtype Constraints Inconsistencies . . . . .	C-107
Error-[SIMERR_INCONSISTENTIC] Incorrect Binding Range C-108	
Error-[SIMERR_SIZEMISMATCH] Array Size Mismatch .	C-109
General Options . . . . .	C-111
Specifying Directories for ‘include’ Searches . . . . .	C-111
Enable the VCS/SystemC Cosimulation Interface . . . .	C-112
TetraMAX . . . . .	C-112
Suppressing Port Coersion to inout . . . . .	C-112
Allow Inout Port Connection Width Mismatches . . . .	C-113
Specifying a VCD File . . . . .	C-113
Enabling Dumping . . . . .	C-113
Enabling Identifier Search . . . . .	C-113
Memories and Multi-Dimensional Arrays (MDAs) . . . .	C-114
Specifying a Log File . . . . .	C-115
Changing Source File Identifiers to Upper Case . . . .	C-115
Defining a Text Macro . . . . .	C-116
Option for Macro Expansion . . . . .	C-116
Specifying the Name of the Executable File . . . . .	C-117
Returning The Platform Directory Name . . . . .	C-118
Maximum Donut Layers for a Mixed HDL Design . . . .	C-118
Enabling feature beyond VHDL LRM . . . . .	C-118

Enabling Loop Detect . . . . .	C-118
Changing the Time Slot of Sequential UDP Output Evaluation C-120	
Gate-Level Performance . . . . .	C-120
Option to Omit Compilation of Code Between Pragmas .	C-120
Generating a List of Source Files . . . . .	C-123
Option for Dumping Environment Variables . . . . .	C-124

## Appendix D. Simulation Options

Options for Simulating Native Testbenches . . . . .	D-2
Options for SystemVerilog Assertions . . . . .	D-11
Options to Control Termination of Simulation . . . . .	D-24
Options for Enabling and Disabling Specify Blocks . . . . .	D-24
Options for Specifying When Simulation Stops . . . . .	D-25
Options for Recording Output . . . . .	D-26
Options for Controlling Messages . . . . .	D-26
Options for VPD Files . . . . .	D-27
Options for VCD Files . . . . .	D-30
Options for Specifying Delays . . . . .	D-31
Options for Flushing Certain Output Text File Buffers . . . . .	D-33
Options for Licensing . . . . .	D-34
Option to Specify User-Defined Runtime Options in a File . .	D-34
Option for the Support of Reading Gzipped Files . . . . .	D-35
Example . . . . .	D-35
Limitations . . . . .	D-37
Option for Initializing Verilog Variables, Registers and Memories at Runtime . . . . .	D-37

Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design at Runtime . . . . .	D-38
General Options. . . . .	D-40
Viewing the Compile Time Options . . . . .	D-40
Recording Where ACC Capabilities are Used . . . . .	D-40
Suppressing the \$stop System Task . . . . .	D-40
Enabling User-defined Plusarg Options . . . . .	D-41
Enabling Overriding the Timing of a SWIFT SmartModel. . . . .	D-41
Enabling feature beyond VHDL LRM . . . . .	D-41
Enabling Loop Detect. . . . .	D-41
Specifying acc_handle_simulated_net PLI Routine . . . . .	D-43
Loading DPI Libraries Dynamically at Runtime . . . . .	D-43
Loading PLI Libraries Dynamically at Runtime. . . . .	D-44
Independent Seeding Across Multiple Instances . . . . .	D-44

## Appendix E. Verilog Compiler Directives and System Tasks

Compiler Directives . . . . .	E-1
Compiler Directives for Cell Definition . . . . .	E-2
Compiler Directives for Setting Defaults . . . . .	E-3
Compiler Directives for Macros . . . . .	E-3
Compiler Directives for Delays. . . . .	E-6
Compiler Directives for Back Annotating SDF Delay Values .	E-7
Compiler Directives for Source Protection. . . . .	E-8
General Compiler Directives . . . . .	E-8
Compiler Directive for Including a Source File . . . . .	E-8
Compiler Directive for Setting the Time Scale . . . . .	E-9
Compiler Directive for Specifying a Library . . . . .	E-9

Compiler Directive for File Names and Line Numbers . . . . .	E-10
Unimplemented Compiler Directives . . . . .	E-11
<b>System Tasks and Functions. . . . .</b>	<b>E-11</b>
<b>System Tasks for SystemVerilog Assertions Severity . . . . .</b>	<b>E-11</b>
<b>System Tasks for SystemVerilog Assertions Control . . . . .</b>	<b>E-12</b>
<b>System Tasks for SystemVerilog Assertions . . . . .</b>	<b>E-13</b>
<b>System Tasks for VCD Files . . . . .</b>	<b>E-14</b>
<b>System Tasks for LSI Certification VCD and EVCD Files . . . . .</b>	<b>E-16</b>
<b>System Tasks for VPD Files. . . . .</b>	<b>E-20</b>
<b>System Tasks for SystemVerilog Assertions . . . . .</b>	<b>E-27</b>
<b>System Tasks for Executing Operating System Commands . . . . .</b>	<b>E-28</b>
<b>System Tasks for Log Files . . . . .</b>	<b>E-29</b>
<b>System Tasks for Data Type Conversions . . . . .</b>	<b>E-30</b>
<b>System Tasks for Displaying Information. . . . .</b>	<b>E-31</b>
<b>System Tasks for File I/O. . . . .</b>	<b>E-31</b>
<b>System Tasks for Loading Memories. . . . .</b>	<b>E-35</b>
<b>System Tasks for Time Scale. . . . .</b>	<b>E-36</b>
<b>System Tasks for Simulation Control. . . . .</b>	<b>E-36</b>
<b>System Tasks for Timing Checks. . . . .</b>	<b>E-37</b>
<b>Timing Checks for Clock and Control Signals . . . . .</b>	<b>E-38</b>
<b>System Tasks for PLA Modeling . . . . .</b>	<b>E-40</b>
<b>System Tasks for Stochastic Analysis . . . . .</b>	<b>E-41</b>
<b>System Tasks for Simulation Time. . . . .</b>	<b>E-42</b>
<b>System Tasks for Probabilistic Distribution . . . . .</b>	<b>E-42</b>
<b>System Tasks for Resetting VCS . . . . .</b>	<b>E-43</b>
<b>General System Tasks and Functions . . . . .</b>	<b>E-44</b>

Checks for a Plusarg . . . . .	E-44
SDF Files . . . . .	E-44
Counting the Drivers on a Net . . . . .	E-44
Depositing Values. . . . .	E-45
Fast Processing Stimulus Patterns . . . . .	E-45
Saving and Restarting The Simulation State . . . . .	E-45
Checking for X and Z Values in Conditional Expressions	E-46
Calculating Bus Widths . . . . .	E-47
Displaying the Method Stack . . . . .	E-48
IEEE Standard System Tasks Not Yet Implemented . . . . .	E-54

## Appendix F. PLI Access Routines

Access Routines for Reading and Writing to Memories . . . . .	F-2
acc_setmem_int. . . . .	F-4
acc_getmem_int . . . . .	F-5
acc_clearmem_int . . . . .	F-6
Examples . . . . .	F-6
acc_setmem_hexstr. . . . .	F-11
Examples . . . . .	F-12
acc_getmem_hexstr . . . . .	F-15
acc_setmem_bitstr. . . . .	F-16
acc_getmem_bitstr . . . . .	F-17
acc_handle_mem_by_fullname. . . . .	F-18
acc_readmem . . . . .	F-18
Examples . . . . .	F-19
acc_getmem_range. . . . .	F-21
acc_getmem_size . . . . .	F-22

acc_getmem_word_int . . . . .	F-23
acc_getmem_word_range . . . . .	F-24
<b>Access Routines for Multidimensional Arrays . . . . .</b>	<b>F-24</b>
tf_mdanodeinfo and tf_imdanodeinfo . . . . .	F-26
acc_get_mda_range . . . . .	F-27
acc_get_mda_word_range() . . . . .	F-29
acc_getmda_bitstr() . . . . .	F-30
acc_setmda_bitstr() . . . . .	F-31
<b>Access Routines for Probabilistic Distribution . . . . .</b>	<b>F-32</b>
vcs_random . . . . .	F-33
vcs_random_const_seed . . . . .	F-34
vcs_random_seed . . . . .	F-34
vcs_dist_uniform . . . . .	F-35
vcs_dist_normal . . . . .	F-35
vcs_dist_exponential . . . . .	F-36
vcs_dist_poisson . . . . .	F-37
<b>Access Routines for Returning a Pointer to a Parameter Value .</b>	<b>F-37</b>
acc_fetch_paramval_str . . . . .	F-38
<b>Access Routines for Extended VCD Files . . . . .</b>	<b>F-38</b>
acc_lsi_dumpports_all . . . . .	F-40
acc_lsi_dumpports_call . . . . .	F-41
acc_lsi_dumpports_close . . . . .	F-43
acc_lsi_dumpports_flush . . . . .	F-44
acc_lsi_dumpports_limit . . . . .	F-45
acc_lsi_dumpports_misc . . . . .	F-46

acc_lsi_dumpports_off . . . . .	F-47
acc_lsi_dumpports_on . . . . .	F-48
acc_lsi_dumpports_setformat . . . . .	F-50
acc_lsi_dumpports_vhdl_enable . . . . .	F-51
<b>Access Routines for Line Callbacks . . . . .</b>	<b>F-52</b>
acc_mod_lcb_add . . . . .	F-53
acc_mod_lcb_del . . . . .	F-55
acc_mod_lcb_enabled . . . . .	F-57
acc_mod_lcb_fetch . . . . .	F-57
acc_mod_lcb_fetch2 . . . . .	F-59
acc_mod_sfi_fetch . . . . .	F-61
<b>Access Routines for Source Protection . . . . .</b>	<b>F-62</b>
vcsSpClose . . . . .	F-66
vcsSpEncodeOff . . . . .	F-67
vcsSpEncodeOn . . . . .	F-68
vcsSpEncoding . . . . .	F-70
vcsSpGetFilePtr . . . . .	F-71
vcsSpInitialize . . . . .	F-72
vcsSpOvaDecodeLine . . . . .	F-73
vcsSpOvaDisable . . . . .	F-74
vcsSpOvaEnable . . . . .	F-75
vcsSpSetDisplayMsgFlag . . . . .	F-77
vcsSpSetFilePtr . . . . .	F-77
vcsSpSetLibLicenseCode . . . . .	F-78
vcsSpSetPliProtectionFlag . . . . .	F-79
vcsSpWriteChar . . . . .	F-80

vcsSpWriteString . . . . .	F-81
Access Routine for Signal in a Generate Block . . . . .	F-83
acc_object_of_type . . . . .	F-83
VCS API Routines . . . . .	F-83
Vcsinit() . . . . .	F-84
VcsSimUntil() . . . . .	F-84

# 1

## Getting Started

---

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

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

VCS accelerates complete system verification by delivering the fastest and highest capacity Verilog, VHDL, and mixed HDL simulation for RTL functional verification. The seamless support for mixed-language simulation of VCS provides a high performance

solution to your IP integration problems and gate-level simulation. VCS supports VHDL External Names feature introduced in the *1076-2008-IEEE Standard VHDL Language Reference Manual*.

This chapter includes the following sections:

- “[Simulator Support with Technologies](#)”
- “[Simulation Preemption Support](#)”
- “[Setting Up the Simulator](#)”
- “[Using the Simulator](#)”
- “[Default Time Unit and Time Precision](#)”
- “[Searching Identifiers in the Design Using UNIX Commands](#)”

---

## Simulator Support with Technologies

VCS supports the following IEEE standards:

- The Verilog language as defined in the *Standard Verilog Hardware Description Language* (IEEE Std 1364).
- The VHDL Language as defined in the *Standard VHDL Hardware Description Language* (IEEE VHDL 1076-1993).
- The SystemVerilog language (with some exceptions) as defined in the *IEEE Standard for SystemVerilog -- Unified Hardware Design, Specification, and Verification Language* (IEEE Std 1800™ - 2012)

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

- SystemC - VCS / SystemC Co-simulation Interface enables VCS and the SystemC modeling environment to work together when simulating a system described in the Verilog, VHDL, and SystemC languages. For more information, refer to “[Using SystemC](#)” .
- Verdi — For more information, refer to “[Using Verdi](#)” .
- Unified Command-line Interface (UCLI) — For more information, refer to “[Using UCLI](#)” .
- Built-In Coverage Metrics — a comprehensive built-in coverage analysis functionality that includes condition, toggle, line, finite-state-machine (FSM), path, and branch coverage. You can use coverage metrics to determine the quality of coverage of your verification test and focus on creating additional test cases. You only need to compile once to run both simulation and coverage analysis. For more information, refer to “[Coverage](#)” .
- DirectC Interface — this interface allows you to directly embed user-created C/C++ functions within your Verilog design description. This results in a significant improvement in ease-of-use and performance over existing PLI-based methods. VCS atomically recognizes C/C++ function calls and integrates them for simulation, thus eliminating the need to manually create PLI files.

VCS supports Synopsys DesignWare IPs, VCS Verification Library, VMC models, Vera, CustomSim, CustomSimHSIM and CustomSim FineSim. For information on integrating VCS with CustomSim, refer to the *Discovery AMS: Mixed-Signal Simulation User Guide*. For more information about CutomSim FineSim, see the *FineSim User Guide: Pro and SPICE Reference*.

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

---

## Simulation Preemption Support

VCS supports simulation preemption. If you suspend a VCS simulation, VCS waits for the safe memory point to suspend the job and checks in the license. When VCS simulation is resumed at a later time, it checks out the license and continues the simulation from the point where it was suspended. You can use `ctrl+z` or `kill -TSTP <pid>` to preempt simulation in VCS.

---

## Setting Up the Simulator

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

- “[Verifying Your System Configuration](#)”
- “[Obtaining a License](#)”
- “[Setting Up Your Environment](#)”
- “[Setting Up Your C Compiler](#)”
- “[Creating a `synopsys\_sim.setup` File](#)”
- “[Displaying Setup Information](#)”
- “[Displaying Design Information Analyzed Into a Library](#)”

---

## Verifying Your System Configuration

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

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

```
% syschk.sh
```

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

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

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

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

Note:

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

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

---

## Obtaining a License

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

To start a new license, do the following:

1. Verify that your license file is functioning correctly:

```
% lmcksum -c license_file_pathname
```

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

Note:

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

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

2. Start the license server:

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

3. Set the LM\_LICENSE\_FILE or SNPSLMD\_LICENSE\_FILE environment variable to point to the license file. For example:

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

or

```
% setenv SNPSLMD_LICENSE_FILE /u/edatools/vcs/
```

*license.dat*

**Note:**

- You can use SNPSLMD\_LICENSE\_FILE environment variable to set licenses explicitly for Synopsys tools.
- If you set the SNPSLMD\_LICENSE\_FILE environment variable, then VCS ignores the LM\_LICENSE\_FILE environment variable.

**Note:**

A single VCS license (under Synopsys' Common Licensing Program) enables you to run Verilog-only, VHDL-only, or mixed-HDL simulations.

---

## Setting Up Your Environment

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

- \$VCS\_HOME environment variable

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

```
% setenv VCS_HOME installation_path
```

- \$PATH environment variable

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

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

OR

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

- **LM\_LICENSE\_FILE or SNPSLMD\_LICENSE\_FILE environment variable:**

Set the license variable **LM\_LICENSE\_FILE** or **SNPSLMD\_LICENSE\_FILE** to your license file as shown below:

```
% setenv LM_LICENSE_FILE Location_to_the_license_file
```

OR

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

Note:

- You can use **SNPSLMD\_LICENSE\_FILE** environment variable to set licenses explicitly for Synopsys tools.
- If you set the **SNPSLMD\_LICENSE\_FILE** environment variable, then VCS ignores the **LM\_LICENSE\_FILE** environment variable.

For additional information on environment variables, see “[VCS Environment Variables](#)” .

## **Setting Up Your C Compiler**

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

RHEL 6.7, 6.8, 7.1 and 7.3, SUSE 11-SP4, 12.0, and 12-SP2, CentOS 6.6, 6.7, 6.8, 7.1.1503, 7.3.1611 platforms all include a C compiler, and VCS assumes the compiler is located in its default location (`/usr/bin`).

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

---

## **Creating a `synopsys_sim.setup` File**

VCS uses the `synopsys_sim.setup` file to configure its environment for VHDL and mixed-HDL designs. This file maps the VHDL design library names to specific host directories, sets search paths, and assigns values to simulation control variables.

When you invoke VCS, it looks for the `synopsys_sim.setup` files in the following three directories with the same order:

- Master setup directory

The `synopsys_sim.setup` file in the `$VCS_HOME/bin` directory contains default settings for your entire installation. VCS reads this file first.

- Your home directory

VCS reads the setup file in your home directory second, if present. The settings in this file take precedence over the conflicting settings in your `synopsys_sim.setup` file in the master setup directory, and carry over the rest.

- Your run directory

VCS reads the setup file in your design directory last. The settings in this file take precedence over the conflicting settings in your `synopsys_sim.setup` file in the master setup directory, and the `synopsys_sim.setup` file in your home directory, and will carry over the rest. You can use this file to customize the environment for a particular design.

**Note:**

This is the directory you invoke and run VCS from; it is not the directory where you store or generate your design files.

The key components of the setup file are the name mappings in the design libraries and the variable assignments. See the following sections for additional information.

The following rules pertain to setup files:

- Blank lines are ignored.
- Physical directory names are case-sensitive.
- All commented lines begin with two dashes (--) .
- The backslash character (\) is used for line continuation.

The following is a sample `synopsys_sim.setup` file:

```
--VCS setup file for ASIC
--Mapping default work directory

WORK > DEFAULT
DEFAULT : ./work

--Library Mapping

STATS_PKG : ./stat_work
MEM_PKG : ./mem_work
```

```
--Simulation variables  
  
TIMEBASE = ps  
  
TIME_RESOLUTION = 1NS
```

## The Concept of a Library In VCS

When you analyze a design, VCS stores the intermediate files in a design library, also called as a logical library. This logical library is pointed to a physical library, which is a physical directory in your UNIX file system. You specify this mapping in the `synopsys_sim.setup` file as shown in the following lines:

```
WORK > DEFAULT  
DEFAULT : ./worklib
```

In the above example, `WORK` is the default logical library and is mapped to the physical library `worklib`. With the above setting, by default VCS stores all the intermediate files in the library `work`, and it errors out if the library `work` does not exist in the specified path.

## Library Name Mapping

For flexibility in library naming, VCS allows you to create multiple logical libraries each one pointing to a different physical library. The syntax to map a logical library to a physical library is shown below:

```
logical_name : physical_name
```

**Note:**

Logical library names are case insensitive.

The following examples show two logical libraries ALU8 and ALU16 mapped to alu\_8bit and alu\_16bit physical libraries. During analysis, you can use the -work option to analyze the files into the respective libraries.

```
ALU8 : ./alu_8bit  
ALU16 : ./alu_16bit
```

The VCS built-in standard libraries have the following default name mappings:

```
IEEE : $VCS_HOME/$ARCH/packages/IEEE/lib  
SYNOPSYS : $VCS_HOME/$ARCH/packages/synopsys/lib
```

In these default mappings, \$ARCH is any one of the following - sparcOS5, sparc64, linux, amd64, rs6000, hp32, suse32, or suse64.

Use these built-in libraries in your design, whenever possible, to get maximum performance from VCS.

## Including Other Setup Files

To include any other setup files, specify the following in the `synopsys_sim.setup` file:

```
OTHERS = [filename]
```

Note that you cannot override the environment settings using this file. In addition, files included in this manner can be nested up to 8 levels.

If VCS is unable to open the specified file, it exits with the following error message:

```
Error: analysis preParsing vhdl-314
```

```
snps_setup fatal error: (Severity SNPS SETUP USER  
FATAL) Cannot open included setup file "user_setup.file"
```

## Using the SYNOPSYS\_SIM\_SETUP Environment Variable

You can also specify a setup file to define VCS setup variables. To do this, set the `SYNOPSYS_SIM_SETUP` variable to your setup file as shown below:

```
% setenv SYNOPSYS_SIM_SETUP my_setup
```

Note that you can use any name for this setup file; you do not need to use `synopsys_sim.setup`.

The settings in this file take precedence over conflicting settings in any regular setup file in the current directory, home directory, or installation directory, and is also searched during simulation. If the file you specify in the `SYNOPSYS_SIM_SETUP` variable cannot be opened, VCS issues the following message:

```
Warning: analysis preParsing vhdl-315  
    snps_setup message: (Severity SNPS SETUP USER WARNING)  
        Cannot open setup file "synopsys_sim.setup"
```

---

## Displaying Setup Information

To list and display all current setup information in your `synopsys_sim.setup` file, enter the following command at the prompt:

```
% show_setup
```

The full syntax of the `show_setup` command is as follows:

```
% show_setup [-v] [-lib]
```

The `show_setup` command options are:

`-v`

Displays the version number and exits.

`-lib`

Displays the library mapping.

The `show_setup` command lists setup information in alphabetical order.

The following example uses `show_setup` to check if optimizations are on for event simulation:

```
% show_setup | grep OPTIMIZE
```

The result of this command is:

```
OPTIMIZE = FALSE
```

Note:

The `show_setup` command shows the cumulative effect of reading each of the three possible `synopsys_sim.setup` files.

---

## Displaying Design Information Analyzed Into a Library

The `lplib` executable displays the following information:

- Entity name, module name, architecture name, configuration name, location of the source file, VCS version, and the timestamp information as and when the file is analyzed.

- All design unit names analyzed in the specified library.
- Architecture name of each entity and package body name of each package.

By default, `llib` lists all design units analyzed into the default logical library.

The syntax of `llib` is as follows:

```
% llib [-l] [-r] [-lib path] design_unit_name
```

The `llib` command options are:

`-l`

Displays entity name, architecture name, configuration name, location of the source file, VCS version and the timestamp for when the design file was analyzed.

`-r`

Displays architecture name of each entity, and package body name of each package.

`-lib path`

Displays the list of design units, package name, and the configuration name in the specified logical library.

`design_unit_name`

`design_unit_name` can be a module, entity, architecture, package body, or a configuration.

## Example

```
% llib -l ZERO
```

```
Library: worklibs
ENTITY          ZERO
Source file     : /u/snps/vhdl/zero.vhd
VCS Version   : Y-2006.06-SP1-5
Timestamp       : Mon Aug 13 22:31:34 2007
Library (four state only): worklibs
```

As illustrated in the example, the design unit ZERO is analyzed into the `worklibs` logical library. The `l1ib` executable also provides the location of the source file, VCS version used to analyze the design unit, and the timestamp information.

---

## Using the Simulator

You can use the following flows to simulate a design using VCS:

- [Two-step Flow](#)
- [Three-step Flow](#)

---

### Two-step Flow

The two-step flow is supported only for Verilog HDL and SystemVerilog designs. Simulating a design using two-step flow involves the following two basic steps:

- [Compiling the Design](#)
- [Simulating the Design](#)

## Compiling the Design

Compiling is the first step to simulate your design. In this phase, VCS builds the instance hierarchy and generates a binary executable `simv`. This binary executable is later used for simulation. For more information, see “[Two-step Flow](#)”.

## Simulating the Design

During compilation, VCS generates a binary executable, `simv`. You can use `simv` to run the simulation. For more information, see “[Two-step Flow](#)”.

---

## Three-step Flow

The three-step flow is supported for Verilog, VHDL, and mixed HDL designs. VCS uses the following three basic steps to compile, elaborate and simulate any Verilog, VHDL, and mixed HDL designs:

- [Analyzing the Design](#)
- [Elaborating the Design](#)
- [Simulating the Design](#)

### Analyzing the Design

VCS provides you with the `vhdlan` and `vlogan` executables to analyze your VHDL and Verilog design code. `vhdlan/vlogan` analyzes your design and stores the intermediate files in the design or a work library.

By default, the `vhdlan` option is VHDL-93 compliant, and `vlogan` is Verilog-2000 compliant. You can switch to VHDL-87 using the `-vhdl87` option with `vhdlan`. Similarly, you can switch to VHDL 2002 or VHDL 2008 by using the `-vhdl02` or `-vhdl08` option respectively. For more information, see “[Three-step Flow](#)” . You can switch to the SystemVerilog mode by using the option `-sverilog` with `vlogan`.

## **Elaborating the Design**

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

## **Simulating the Design**

Simulate your design by executing the binary simulation executable, `simv`. For more information, see “[Three-step Flow](#)” .

---

## **Basic Usage Model**

The following sections describe the basic use model for two-step flow and three-step flow:

- [Two-step Flow](#)
- [Three-step Flow](#)

## **Two-step Flow**

### **Compilation**

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

## Simulation

```
% simv [run_options]
```

## Three-step Flow

### Analysis

Always analyze Verilog before VHDL.

```
% vlogan [vlogan_options] file1.v file2.v  
% vhdlan [vhdlan_options] file3.vhd file4.vhd
```

Note:

Specify the VHDL bottommost entity first, then move up in order.

### Elaboration

```
% vcs [elaboration_options] design_unit
```

The `design_unit` can be one of the following:

module

Verilog top module name.

entity

VHDL top entity name.

entity\_archname

Name of the top entity and architecture to be simulated. By default, `archname` is the most recently analyzed architecture.

cfgname

Name of the top-level event configuration to be simulated.

## Simulation

```
% simv [run_options]
```

---

## Default Time Unit and Time Precision

The default time unit for Verilog and SystemVerilog simulation is 1 s.

The default time precision for Verilog and SystemVerilog simulation is 1 s.

For VHDL simulation there is no concept of a default time unit and delay values, for example, must have a unit name or unit of measurement, for example:

```
wait for 10.123123 ns;
```

The default time precision for an entire VHDL design is specified with the TIME\_RESOLUTION 1 ns entry in the synopsys\_sim.setup file in the VCS installation (see “[Creating a synopsys\\_sim.setup File](#)” ).

The default time precision for the VHDL part of a mixed HDL design is the smallest or finest of these two:

- What is specified with the TIME\_RESOLUTION entry in the synopsys\_sim.setup file.
- The smallest time precision from the Verilog or SystemVerilog part of the design.

You can override the default time precision with the -time\_res elaboration option.

Note:

The -time\_res option has no effect on the Verilog code.

---

## Searching Identifiers in the Design Using UNIX Commands

You can use the following `vcsfind` UNIX command to search for identifiers in your design. The `vcsfind` script is located in `$VCS_HOME/bin`. You must specify the location of the `fsearch.db` file.

```
vcsfind [<options> --] [<identifier>] [ (+/-)<search group>] +
```

Where,

options

Search options (see [Table 1-1](#)). These options must be separated by a “--” from the search query. Any change to the Verdi GUI settings has no effect on the `vcsfind` command.

*Table 1-1 Supported Search Options*

Search Option	Description
<code>--version</code>	Displays program's version number and exits
<code>-h, --help</code>	Displays help message and exits
<code>-b, --bw(Black and White)</code>	Highlights with bold and underline only, no colors.
<code>-d N, --dir_levels=N</code>	Prints <code>n</code> directory levels for every matching line. Default is 0.
<code>-f DB-FILE, --file=DB-FILE</code>	Specifies the database file. Default is <code>vcsfind.db</code>
<code>-H, --gui-help</code>	Prints help for GUI use.
<code>-l N, --limit=N</code>	Limits search to the first <code>n</code> matches. 0 means no limit. Default is 1000.

Search Option	Description
<code>-m, --match_only</code>	Matches the query pattern only. Does not display scope information.
<code>-o OUTPUT-FILE, --output=OUTPUT-FILE</code>	Outputs into a file. Default is <code>stdout/stderr</code> . This option bundles <code>stdout</code> and <code>stderr</code> , so <code>-o -</code> will redirect errors to <code>stdout</code> .
<code>-p, --plain</code>	Does not highlight matches in bold.
<code>-r, --regexp</code>	Regular expression search pattern. The pattern is interpreted as <code>^&lt;pattern&gt;\$</code> , so <code>.*</code> may be desired at the beginning and end of the pattern.
<code>-t, --translate</code>	Translation mode. Prints only the translation of the query pattern into the internal SQL query string.
<code>-u, --uclimode</code>	Enables UCLI mode. This option is used for interaction with UCLI.
<code>-v, --verbose</code>	Enables verbose mode.

`identifier`

Identifier string to be searched.

`search group`

The name of the group to be included to search or excluded from search. The following search groups are supported:

Packages, Modules, Ports, Parameters, Vars, Functions, Assertions, Types, Members, Instances

You can also use Verdi to search for the identifiers in your design.

## Examples

```
% vcsfind -f simv.daidir/debug_dump/fsearch/fsearch.db -- Top
```

**Following is the sample output:**

```
Matching modules:  
top.v:11 module Top  
    scope: Top
```

```
Matching instances:  
top.v:11 inst Top of module Top  
    scope: Top
```

```
Total: 4 results found in 0.053 seconds
```



# 2

## VCS Flow

---

This chapter describes the simulation flows supported by VCS. You can use the following flows to simulate your design using VCS:

- “Three-step Flow”
- “Two-step Flow”

---

## Three-step Flow

Simulating a design using three-step flow involves three basic steps:

- “Analysis”
- “Elaboration”
- “Simulation”

VCS uses these three steps to compile any design irrespective of the HDL, HVL, and other supported technologies used. For information on supported technologies, see [“Simulator Support with Technologies”](#).

For information on the data types supported across the language boundary for both ports and generics or parameters, see the *VCS Simulation Coding and Modeling Style Guide*.

For information on licensing options, see [“Options for Licensing”](#).

---

## Analysis

Analysis is the first step to simulate your design. In this phase, you analyze your VHDL, Verilog, SystemVerilog, and OpenVera files using `vhdlan` or `vlogan` accordingly. The following section includes a few example command lines to analyze your design files:

### Analyzing your VHDL files:

```
% vhdlan [vhdlan_options] file1.vhd file2.vhd
```

### Analyzing your Verilog files:

```
% vlogan [vlogan_options] file1.v file2.v
```

## Analyzing your SystemVerilog files:

```
% vlogan -sverilog [vlogan_options] file1.sv file2.sv  
file3.v
```

For the complete usage model, see “[Using SystemVerilog](#)” .

## Analyzing your OpenVera files:

```
% vlogan -ntb [vlogan_options] file1.vr file2.vr file3.v
```

For the complete use model, see “[Using OpenVera Native Testbench](#)” .

## Analyzing your SystemVerilog and OpenVera files:

```
% vlogan -sverilog -ntb [vlogan_options] file1.sv file2.vr  
file3.v
```

Note that you can analyze SystemVerilog files or OpenVera files along with other Verilog files in the same `vlogan` command line as shown in the examples. Unless it is required, you do not need to separately analyze these files.

In the analysis phase, VCS checks the design for the syntax errors. In this phase, VCS generates the intermediate files required for elaboration and saves these files in the design or work library pointed to by your default logical library. For information on library mapping, see “[The Concept of a Library In VCS](#)” . You can instruct VCS to save these intermediate files in a different library using the `-work` option with the `vhdlan` or `vlogan` executables.

Before you analyze your design using `vhdlan` or `vlogan`, ensure that the library mappings are defined in the `synopsys_sim.setup` file, and that the specified physical library for the logical library exists. If the physical directory does not exist, VCS exits with an error message.

VCS provides `vhdlan` and `vlogan` executable to analyze VHDL and Verilog design files, respectively. The following sections describe the usage of these two executables and some of the commonly used options.

## Using `vhdlan`

The `vhdlan` executable analyzes your VHDL design files and stores the generated intermediate files in the design or work library. The syntax for the `vhdlan` executable is as follows:

```
% vhdlan [vhdlan_options] VHDL_filename_list
```

### Commonly Used Analysis Options

This section lists some of the commonly used `vhdlan` options. For a complete list of options, see section “[The `vhdlan` Utility](#)” .

#### Command Options

`-help`

Displays usage information for `vhdlan`.

`-nc`

Suppresses the Synopsys copyright message.

`-q`

Suppresses all `vhdlan` messages.

`-version`

Displays the version number of `vhdlan` and exits without running analysis.

`-full64`

Analyzes the design for 64-bit simulation.

**-work *library***

Maps a design library name to the logical library name `WORK` that receives the output of `vhdlan`. Mapping with this command line option overrides any assignment of `WORK` to another library name in the setup file.

*library* can also be a physical path that corresponds to a logical library name defined in the setup file.

**-vhdl87**

Enables to analyze non-portable VHDL code that contains object names that are now VHDL-93 reserved words by default. VCS is VHDL-93 compliant.

**-vhdl02**

Enables to analyze the VHDL 2002 protected type. For more information, see “[Support for VHDL 2002 and 2008](#)” .

**-vhdl08**

Enables to analyze the VHDL 2008 constructs provided in the chapter “[Support for VHDL 2002 and 2008](#)” .

**-output *outfile***

Redirects standard output from VCS analysis (that usually goes to the screen) to the file that you specify as *outfile*.

**-x1rm**

Enables VHDL features beyond those described in LRM.

**-f *filename***

Specifies a file that contains a list of source files. You should specify the bottom most VHDL entity first, and then move up in order.

**-functional\_vital**

Specifies generating code for functional VITAL simulation mode.

**-l *filename***

Specifies a log file where VCS records the analyzer messages.

**-no\_functional\_vital**

Specifies generating code for full-timing VITAL simulation mode.

**VHDL\_filename\_list**

Specifies the VHDL source file names to be analyzed. If you do not provide an extension, .vhd is assumed.

**Note:**

The maximum identifier name length is 250 for package, package body and configuration names. The combined length of an entity name plus architecture name must not exceed 250 characters as well. All other VHDL identifier names and string literals do not have a limitation.

**-init\_std\_logic**

You can initialize all uninitialized VHDL signals, ports and variables of the data type STD\_LOGIC/STD\_ULOGIC (scalar/vector) with a given 9-value. A VHDL signal or variable of this type can take on the following values – 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'.

You can supply the value at `vhdlan` command line option as shown in the following command line:

```
% vhdlan hello.vhd -init_std_logic 0
```

You can also initialize the value in the `synopsys_sim_setup` file.

In the `synopsys_sim_setup` file, you can set the value to any one of the nine values to the variable `INIT_STD_LOGIC`. For example, `INIT_STD_LOGIC=0`. To create a `synopsys_sim_setup` file, see “[Creating a synopsys\\_sim.setup File](#)” .

## Using `vlogan`

Similar to `vhdlan`, the `vlogan` executable analyzes your Verilog design files and stores the generated intermediate files in the design or work library. The syntax for the `vlogan` executable is as follows:

```
% vlogan [vlogan_options] Verilog_filename_list
```

### Commonly Used Analysis Options

This section lists some of the commonly used `vlogan` options. For a complete list of options, see “[The vlogan Utility](#)” .

#### Command Options

`-help`

Displays usage information for `vlogan`.

`-nc`

Suppresses the Synopsys copyright message.

**-q**

Suppresses all vlogan messages.

**-f filename**

Specifies a file that contains a list of source files.

Note:

The maximum line length in the specified file *filename* must be less than 1024 characters. VCS truncates the line exceeding this limit and issues a warning message.

**-full64**

Analyzes the design for 64-bit simulation.

**-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.

**-l filename**

Specifies a log file where VCS records the analyzer messages.

`-liblist logical_lib1+logical_lib2+...logical_libn`

It specifies the library search order for resolving imported package definitions. The `vlogan -liblist` option restricts the libraries in which `vlogan` should search for resolving package references found while analyzing.

If the `-liblist` option is not included, `vlogan` searches all the logical libraries listed in the `synopsys_sim.setup` file.

When using multiple `vlogan` commands with the same `-work` logical library, run the commands sequentially, and if one command uses `-liblist`, then ensure that all the remaining `vlogan` commands are using the same `-liblist` argument list, as shown in the following command lines:

```
%vlogan a.v -work shared_lib -liblist  
shared_lib+ovm_lib+common_lib
```

```
%vlogan b.v -work shared_lib -liblist  
shared_lib+ovm_lib+common_lib
```

```
%vlogan c.v -work shared_lib -liblist  
shared_lib+ovm_lib+common_lib
```

`-ntb`

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

`-ntb_define macro`

Specifies any OpenVera macro name on the command line. You can specify multiple macro names using the plus (+) character.

`-ntb_fileext .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.

`-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.

`-sverilog`

Enables the analysis of SystemVerilog source code.

`-sv_pragma`

Instructs VCS to compile the SystemVerilog Assertions code that follows the `sv_pragma` keyword in a single line or multi-line comment.

`-timescale=time_unit/time_precision`

This option enables you to specify the timescale for the source files that do not contain `'timescale` compiler directive and precede the source files that contain.

Do not include spaces when specifying the arguments to this option.

`-v library_file`

Specifies a Verilog library file to search for module definitions.

`-y library_directory`

Enables you to specify a Verilog library directory to search for module definitions. VCS looks in the source files in this directory for definitions of the module and UDP instances that VCS found in your source code. However, for which it did not find the corresponding module or UDP definitions in your source code.

VCS looks in this directory for a file with the same name as the module or UDP identifier in the instance (not the instance name). If it finds this file, VCS looks in the file for the module or UDP definition to resolve the instance.

Note:

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

For example:

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

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

VCS selects `cell.v` from `rev1`.

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

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

In this case, VCS uses `rev3/cell.v` when you issue the following command:

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

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

#### *-work library*

Maps a design library name to the logical library name `WORK` that receives the output of `vlogan`. Mapping with the command-line option overrides any assignment of `WORK` to another library name in the setup file.

#### *+define+macro*

Defines a text macro. Test this definition in your Verilog source code using the `'ifdef` compiler directive.

#### *+libext+extension+*

Specifies that VCS searches only for files with the specified file name extensions in a library directory. You can specify more than one extension, separating the extensions with the plus (+) character. For example, `+libext+.v+.V+` specifies searching 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.

#### *+lint=[no] ID|none|all*

Enables messages that provides information about when your Verilog code contains something that is bad style, but is often used in designs.

`+liborder`

Specifies searching for module definitions for unresolved module instances through the remainder of the library where `vlogan` finds the instance, then searching the subsequent library on the `vlogan` command line before searching in the first library in the command line.

`+librescan`

Always specifies searching libraries for module definitions of unresolved module instances beginning with the first library in the `vlogan` command line.

`+incdir+directory`

Specifies the directories that contain the files specified with the `'include` compiler directive. You can specify more than one directory, separating each path name with the “+” character.

`+nowarnTFMPC`

Suppress the Too few module port connections warning messages during Verilog Compilation.

`+systemverilogext+ext`

Specifies a file name extension for SystemVerilog source files. If you use a different file name extension for the SystemVerilog part of your source code and you use this option, the `-sverilog` option has to be omitted.

`+verilog2001ext+ext`

Specifies a file name extension for Verilog 2001 source files.

`+verilog1995ext+ext`

Specifies a file name extension for Verilog 1995 files. Using this option allows you to write Verilog 1995 code that would be invalid in Verilog 2001 or SystemVerilog code, such as using Verilog 2001 or SystemVerilog keywords, like `localparam` and `logic`, as names.

`+warn`

Enables or disables warning messages.

*Verilog\_source\_filename*

Specifies the name of the Verilog source file.

**Note:**

The following options are `vlogan` parse options:

```
-ignore unique_checks|priority_checks|all  
-ntb_define macro  
-ntb_fileext .ext  
-sv_pragma  
-v library_file  
-y library_directory  
-sverilog  
+define+macro  
+incdir+[directory]  
+libext+extension+  
+systemverilogext+.ext  
+verilog1995ext+.ext  
+verilog2001ext+.ext  
+lint=[no] ID|none|all  
+nowarnTFMPC  
+warn
```

VCS issues a message with appropriate severity if any of these options are not allowed and used at elaboration.

## Analyzing the Design to Different Libraries

You can analyze your design to different libraries using the `-work` option with either the `vhdlan` or `vlogan` executable. However, to use this feature, map the required logical libraries to physical libraries. For information about mapping libraries, see, “[Library Name Mapping](#)”.

With the `-work` option, you can specify either the logical library name or the physical library name, specified in your `synopsys_sim.setup` file as follows:

```
% vhdlan -work libname1 VHDL_filename_list  
% vlogan -work libname1 Verilog_filename_list
```

These command lines analyze your VHDL files and Verilog files, and saves the intermediate files in the `libname1` library. VCS resolves all VHDL files having:

```
library libname1;  
use libname1.all;
```

---

## Elaboration

Elaborating is the second step to simulate your design. In this phase, using the intermediate files generated during analysis, VCS builds the instance hierarchy and generates a binary executable `simv`. This binary executable is later used for simulation.

In this phase, you can choose to elaborate the design either in the optimized mode or in the debug mode. The runtime performance of VCS is based on the mode you choose and the level of flexibility

required during simulation. Synopsys recommends you to use full debug or partial debug mode until the design correctness is achieved, and then switch to optimized mode.

In optimized mode, also called as batch mode, VCS delivers the best compile time and runtime performance for a design. You typically choose optimized mode to run regressions or when you do not require extensive debug capabilities. For more information, see [“Optimizing Simulation Performance for Desired Debug Visibility With the -debug\\_access Option”](#).

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

## Using VCS

The syntax to use `vcs` is as follows:

```
% vcs [elab_options] [libname.]design_unit
```

Where,

`libname`

The library name where you analyzed your top module, entity, or the configuration. If not specified, VCS looks for the specified `design_unit` in the list of libraries specified in the `synopsys_sim.setup` file as per the order specified. For more information, see [“Creating a synopsys\\_sim.setup File”](#).

Here, the `design_unit` can be one of the following:

`module`

Verilog top module name.

`entity`

VHDL top entity name.

`entity_archname`

Name of the top entity and architecture to be simulated. By default, `archname` is the most recently analyzed architecture.

`cfgname`

Name of the top-level configuration.

## Commonly Used Options

This section lists some of the commonly used `vcs` options. For a complete list of options, see “[Compilation/Elaboration Options](#)” .

### Options for Help

`-h` or `-help`

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

`-ID`

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

## **Options for 64-bit Elaboration**

`-full64`

Enables elaboration and simulation in 64-bit mode.

Alternatively, you can use the `VCS_TARGET_ARCH` environment variable to enable elaboration and simulation in 64-bit mode architecture. For more information, see [VCS\\_TARGET\\_ARCH](#).

## **Option to Specify Elaboration Options in a File**

`-file filename`

Specifies a file containing elaboration options.

## **Options for Starting Simulation After Elaboration**

`-R`

Runs the executable file immediately after VCS links it together.

## **Options for Changing Generics and Parameter Values**

`-gfile cmdfile`

Overrides the default values for design generics or parameters using values from the file `cmdfile`. The `cmdfile` file is an include file that contains assign commands targeting design generics.

## **Options for Controlling Messages**

`-notice`

Enables verbose diagnostic messages.

`-q`

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

-V

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

## Specifying a Log File

-l *filename*

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

---

## Simulation

During elaboration, using the intermediate files generated, VCS creates a binary executable, simv. You can use simv to run the simulation. Based on how you elaborate the design, you can run the simulation using the following two modes:

- Interactive mode
- Batch mode

For information about elaborating the design, see “[Elaboration](#)” .

## Interactive Mode

You can elaborate your design in the interactive mode, also called debug mode, in the initial phase of your design cycle. In this phase, you require abilities to debug the design issues using a GUI or

through the command line. To debug using a GUI, you can use Verdi, and to debug through the command-line interface, you can use Unified Command-line Interface (UCLI).

**Note:**

To simulate a design in the interactive mode, elaborate the design using `-debug_access+r`, `-debug_access+all`, or refer to section “[Optimizing Simulation Performance for Desired Debug Visibility With the -debug\\_access Option](#)” for more debug compile time options. For information on elaborating the design, see “[Elaboration](#)”

## **Batch Mode**

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

**Note:**

The runtime performance reduces if you use `-debug_access`. Use this option only when you require runtime debug abilities.

The following command line simulates the design in batch mode:

```
% simv
```

## **Commonly Used Runtime Options**

Use the following command line to simulate the design:

```
% simv_executable [runtime_options]
```

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

`-gui`

This option starts Verdi when `VERDI_HOME` is set.

`-ucli`

This option starts `simv` in UCLI mode.

For information on Verdi, see “[Using Verdi](#)” .

For a complete list of runtime options, see “[Simulation Options](#)” .

---

## Two-step Flow

The two-step flow is supported only for Verilog HDL and SystemVerilog designs. Simulating a design using two-step flow involves two basic steps:

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

---

### Compilation

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

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

In the optimized mode, also called as batch mode, VCS delivers the best compile time and runtime performance for a design. You typically choose optimized mode to run regressions, or when you do not require extensive debug capabilities. For more information, see [“Optimizing Simulation Performance for Desired Debug Visibility With the -debug\\_access Option”](#).

You can compile the design in debug mode, also called interactive mode, when you are in the initial phase of your development cycle, or when you need more debug capabilities or tools to debug the

design issues. In this mode, the performance is not the best that VCS can deliver. However, using some of the compile time options, you can compile your design in full debug or partial debug mode to get maximum performance in debug mode. For more information, see “[Compiling/Elaborating the Design in the Debug Mode](#)”.

For information on licensing options, refer to “[Options for Licensing](#)”.

## Using vcs

The syntax to use VCS is as follows:

```
% vcs [compile options] Verilog_files
```

### Commonly Used Options

This section lists some of the commonly used `vcs` options.

#### Options for Help

`-h` or `-help`

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

`-ID`

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

#### Options for Accessing Verilog Libraries

`-v filename`

Enables you to specify a Verilog library file. VCS looks in this file for definitions of the module and UDP instances that VCS found in your source code, however, for which it did not find the corresponding module or UDP definitions in your source code.

`-y directory`

Enables you to specify a Verilog library directory. VCS searches 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 searches 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 searches in the file for the module or UDP definition to resolve the instance.

Note:

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

For example:

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

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

VCS selects `cell.v` from `rev1`.

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

```
//top.v
```

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

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

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

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

#### +includer+directory+

Specifies the directory or directories that VCS searches for include files used in the `include compiler directive. You can specify multiple directories using the plus (+) character.

#### +libext+extension+

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

#### +liborder

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

**Note:**

+liborder and +librescan switches on elaboration command line will have impact only when the user specifies -y/-v on elaboration command line.

## **Options for 64-bit Compilation**

**-full64**

Enables compilation and simulation in 64-bit mode.

## **Option to Specify Files and Compile Time Options in a File**

**-file *filename***

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

## **Options for Verdi**

**-verdi**

This option starts Verdi.

## **Options for Starting Simulation After Compilation**

**-R**

Runs the executable file immediately after VCS links it together.

## **Options for Changing Parameter Values**

**-pvalue+parameter\_hierarchical\_name=value**

Changes the specified parameter to the specified value.

**-parameters *filename***

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

*assign value path\_to\_parameter*

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

## Options for Controlling Messages

`-notice`

Enables verbose diagnostic messages.

`-q`

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

`-V`

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

## Specifying a Log File

`-l filename`

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

## Defining a Text Macro

`+define+macro=value+`

Defines a text macro in your source code to a value or character string. You can test this definition in your Verilog source code using the `'ifdef` compiler directive.

**Note:**

The `=value` argument is optional.

For example:

```
% vcs design.v +define+USETHIS
```

The macro is used inside the source file using the `'ifdef` compiler directive. If this macro is not defined using the `+define` option, then the `else` portion in the code takes priority.

```
'ifdef USETHIS
    package p1;
    endpackage
`else
    package p2;
    Endpackage
`endif
```

---

## Simulation

During compilation, VCS generates a binary executable, `simv`. You can use `simv` to run the simulation. Based on how you compile the design, you can run your simulation using the following modes:

- Interactive mode
- Batch mode

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

## Interactive Mode

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

### Note:

To simulate the design in the interactive mode, compile the design using the `-debug_access (+<option>)` compile time option.

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

## Batch Mode

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

### Note:

The runtime performance reduces if you use the `-debug_access (+<option>)` option. Use this option only when you require runtime debug abilities.

The following command line simulates the design in batch mode:

```
% simv
```

## Commonly Used Runtime Options

Use the following command line to simulate the design:

`% executable [runtime_options]`

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

`-gui`

This option starts Verdi when `VERDI_HOME` is set.

`-ucli`

This option starts `simv` in UCLI mode.

For information on Verdi, see “[Using Verdi](#)” .

For a complete list of runtime options, see “[Simulation Options](#)” .

# 3

## Modeling Your Design

---

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

This chapter includes the following topics:

- “[Avoiding Race Conditions](#)”
- “[Race Detection in Verilog Code](#)”
- “[Race Detection Tool to Identify Race between Clock and Data](#)”
- “[Optimizing Testbenches for Debugging](#)”
- “[Creating Models That Simulate Faster](#)”

- “Creating Models That Simulate Faster”
- “Case Statement Behavior”
- “Precedence in Text Macro Definitions”
- “Memory Size Limits in the Simulator”
- “Using Sparse Memory Models”
- “Obtaining Scope Information”
- “Avoiding Circular Dependency”
- “Designing With \$lsi\_dumpports for Simulation and Test”

---

## Avoiding Race Conditions

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

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

---

## Using and Setting a Value at the Same Time

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

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

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

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

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

---

## Setting a Value Twice at the Same Time

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

```
module race;
    reg r1;
    initial #10 r1 = 0;
    initial #10 r1 = 1;
    initial
```

```

#20 if (r1) $display("may not print");
endmodule

```

The solution is to stagger the assignments to register `r1` by finite time, so that the ordering of the assignments is guaranteed. Note that using the non-blocking 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 non-blocking 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 non-zero delay in addition to the non-blocking 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.

If you are modeling flip-flops using sequential UDPs (User-Defined Primitives), note that VCS evaluates the output terminals of sequential UDP (User-Defined Primitive) in the NBA region. This can cause a race condition. The default behavior is required by the SystemVerilog LRM, IEEE Std 1800-2009.

---

## 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 non-blocking 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 is 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
```

---

## Race Detection in Verilog Code

VCS provides the following race detection tools:

- **Dynamic Race Detection Tool** - Finds the race conditions during simulation.
- **Static Race Detection Tool** - Finds the race conditions by analyzing source code during compilation.

The above two tools are described in the following sections:

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

---

## The Dynamic Race Detection Tool

This section consists of following topics:

- [Introduction to the Dynamic Race Detection Tool](#)
- [Enabling Race Detection](#)
- [The Race Detection Report](#)

- Post-Processing the Report
- Debugging Simulation Mismatches

## Introduction to the Dynamic Race Detection Tool

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

- Read - Write Race Condition
- Write - Write Race Condition

### Read - Write Race Condition

The Read - Write race condition occurs when both Read and Write on a signal take place at the same simulation time.

Example:

```
initial
#5 var1 = 0; // write operation on signal var1

initial
#5 var2 = var1; // read operation on signal var2
```

### Read

Procedural assignment in any one of the always or initial block, or a continuous assignment samples the value of signal `var1` to drive signal `var2`.

### Write

Procedural assignment in another always or initial block, or another continuous assignment assigns a new value to signal `var1`.

In the above example, at the simulation time 5, there is both read and write operation on signal `var1`. When simulation time 5 is over, you do not know if signal `var2` will have the value 0 or the previous value of signal `var1`.

### **Write - Write Race Condition**

The Write - Write race condition occurs when multiple writes on a signal take place at the same simulation time.

Example:

```
initial  
#5 var1 = 0; // write operation on signal var1  
  
initial  
#5 var1 = 1; // write operation on signal var1
```

### **Write-Write**

Value of the signal `var1` is non-deterministic when there are multiple concurrent procedural assignments on the same variable at the same simulation time.

In the above example, at simulation time 5, different initial blocks assign 0 and 1 to signal `var1`. When simulation time 5 is over, you do not know if `var1` signal 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.

**Note:**

`$dumpvars` can also expose races.

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:

- Simulation results do not match when comparing simulators
- Design modifications cause inexplicable results
- 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 should 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.

Note:

The `-race` and `-racecd` compile-time options support dynamic race detection for both pure Verilog and SystemVerilog data types.

## The Race Detection Report

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

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

The `race.unique.out` file contains only the lines 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 the 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](#)”. If that location at your site is not a perl5 interpreter, the script fails with syntax errors.

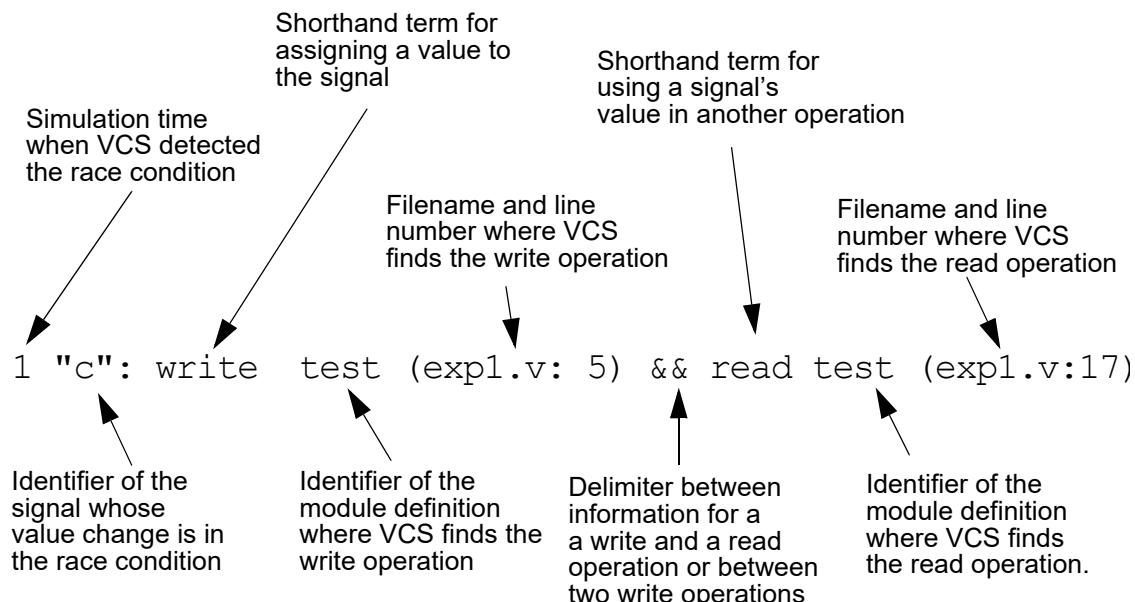
The report describes read-write and write-write race conditions. The following is an example of the contents of a small race.out file:

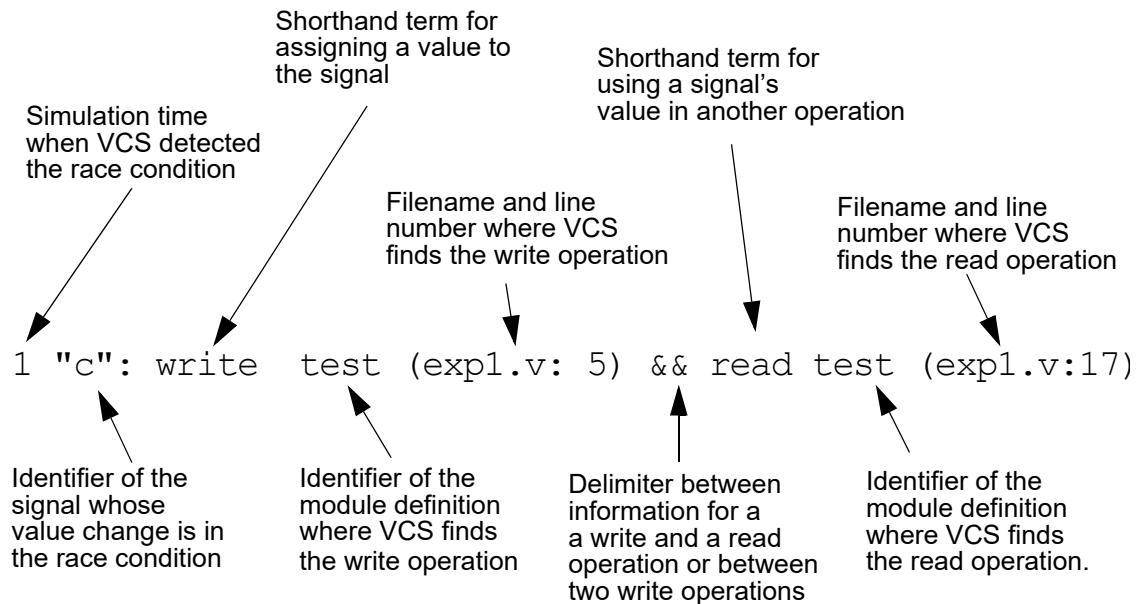
```
Synopsys Simulation VCS RACE REPORT

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 reg `c` on line 5, and the value of reg `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 as 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 should 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) 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 have 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.

Unless you specify a different name with the `-o` option, the report generated by the `PostRace.pl` script is in the `race.out.post` file.

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 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 must modify the first line of this script. This script needs a perl5 interpreter. You can find this script at the following location:

```
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 must find critical races. Critical races are 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 the 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 the number of lines in the `race.unique.out` file is smaller than the number of unique race conditions, then 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 before and 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 and a large number of unique race conditions, then the 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 before and 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:

- If the simulation values differ in the two simulations, then the race condition in the `race.out.post` file is a critical race condition.
- 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 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, then it is a critical race condition.

### **The Static Race Detection Tool**

It is possible for a group of statements to combine and form a loop, so that the loop is executed once by VCS and more than once by other Verilog simulators. This is a race condition.

These situations arise when level-sensitive sensitivity lists (event controls which immediately follow the `always` keyword in an `always` block, and which do not contain the `posedge` or `negedge` keywords) and procedural assignment statements in the `always` blocks combine with other statements such as continuous assignment or module instantiation statements to form a potential loop. It is observed that these situations do not occur if the `always` blocks contain delays or other timing information, non-blocking assignment statements, or PLI calls through user-defined system tasks.

You can use the `+race=all` compile-time option to start the static race detection tool.

Note:

The `+race=all` compile-time option supports only pure Verilog constructs.

After compilation, the static race detection tool writes the file named `race.out.static` which reports the race conditions.

The following example shows 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'.
```

---

## Race Detection Tool to Identify Race between Clock and Data

Starting with this release, VCS provides the new race detection tool that finds the race condition between clock and data and generates the diagnostic output.

This race detect tool detects race for the following conditions:

- Whenever the clock and data arrives at the same time, VCS detects the race condition and provides the RTL information.
- Race is reported when the data is updated before the clock for the same timestamp.

This race detection tool helps you to know the following:

- Glitches in the design that are encountered during simulation.
- Flops that are affected by the glitch during simulation.
- The frequency and timing of glitches in the design.

---

## Use Model

When you compile your design, you can enable the race detection tool during simulation for your entire design.

The `-hsopt=racedetect` option enables the race detection tool for your entire design.

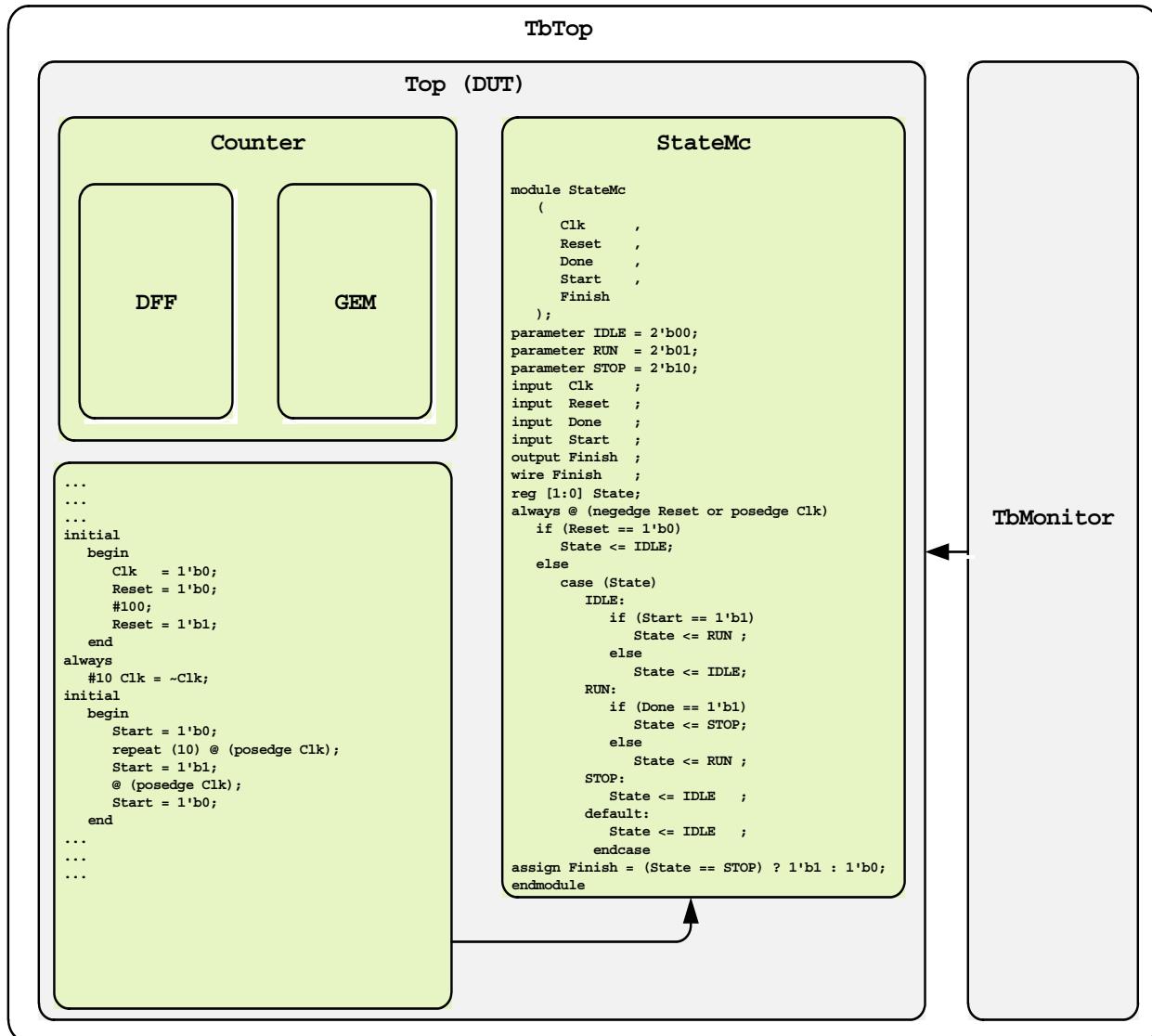
If no clock-data race is detected during simulation, VCS reports the following information:

```
Clock Data Race: No Race detected.
```

VCS stores the RTL information in the `hsRaceInfo.db` file.

## Examples

Consider the following block diagram representing the design test cases:



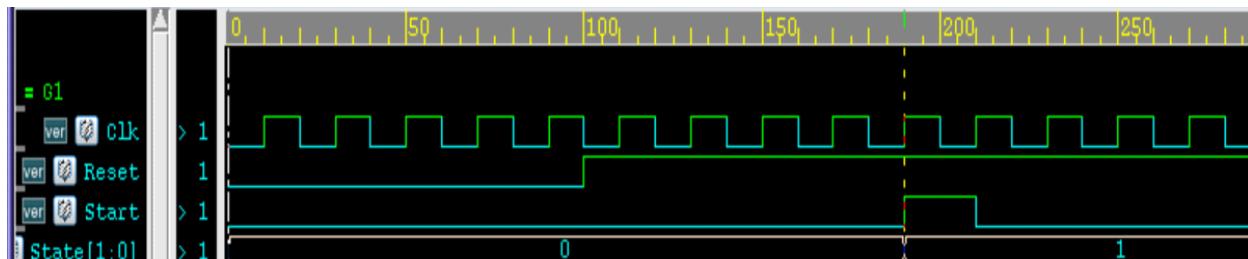
Compile the test cases as follows:

```
% vcs -debug -timescale=1ns/1ns DFF.v GEM.v Counter.v  
StateMc.v TbMonitor.v TbTop.v Top.v -hsopt=racedetect  
  
% simv  
  
% cat hsRaceInfo.db
```

It generates the following output:

```
Clock Data Race detected @ 190 Module: StateMc Instance:  
TbTop.U_TOP.U_STATEMC  
Line: 24 File: StateMc.v RTL_MOD_NAME: StateMc  
Clock Data Race detected @ 210 Module: StateMc Instance:  
TbTop.U_TOP.U_STATEMC  
Line: 24 File: StateMc.v RTL_MOD_NAME: StateMc
```

When you compile the test case with FSDB dumping enabled in Verdi, the following waveform is generated:



Note:

You need to set the environment variable `VERDI_HOME` for FSDB dumping.

---

## Limitations

The feature has the following limitations:

- RTL information is not provided for clock-only glitches.
- No race detect output file is generated if you are using any of the following VCS technologies:
  - X-propagation
  - Congruency
  - Partition Compile flow
  - Precompiled IP flow
  - Mixed-Signal design
  - FGP flow
  - `-dbsflagsbytarray` option

---

## 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 FSDB files that you can input into Verdi.

The `$dumpvars` system task specifies writing a VCD file, and the `$fsdbDumpvars` system task specifies writing a FSDB file. You can use the `vcd2fsdb` utility to also convert VCD file into FSDB, which translates the VCD file to a FSDB file and then displays the results in Verdi.

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 will 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 compiles these tasks for debugging features. For example:

```
initial  
begin
```

```
`ifdef postprocess  
$vcdpluson(0,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 FSDB file. 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 does 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
$vcplusplus(0,design_1);
$vcplusplusdeltacycleon;
$vcplusplusglitchon;
end
```

In this technique you do not include the `+define` compile-time argument on the `vcs` command line. Instead you compile the system tasks into the testbench and then enable the execution of the system

tasks with the runtime argument to the `$test$plusargs` system function. Therefore, in this example, the `simv` command line is as follows:

```
% simv +postprocess
```

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

There is a pitfall to this technique. This system function matches 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 are used at runtime — and VCS generates the `simv` executable with the corresponding overhead to prepare for these plusargs. The more fixed information VCS has at compile-time, the more VCS can optimize `simv` for efficient simulation. Alternatively, the more user control at runtime, the more overhead VCS has to add to `simv` to accept runtime options, and the less efficient the simulation.

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

---

## Combining the Techniques

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

```
`ifdef comppostprocess
initial
  if ($test$plusargs ("runpostprocess"))
    begin
      $vcddpluson(0,design_1);
      $vcsplusdeltacycleon;
      $vcddplusglitchon;
    end
`endif
```

In this instance, 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.

---

## 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:

<b>Data Type</b>	<b>Description in IEEE Std 1364-2001</b>
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 the IEEE Std 1364-2001.

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

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

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

## Avoid Strength Specifications in Continuous Assignment Statements

Omit strength specifications in continuous assignment statements. For example:

```
assign net1 = flag1;
```

Simulates faster than:

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

Continuous assignment statements are described in the IEEE Std 1364-2001.

---

## 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 as level-sensitive event controls	Page 141
<code>disable</code> statements	Page 162-164
<code>fork-join</code> block statements, also called as parallel blocks	Page 146-147

Using either blocking or non-blocking 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 non-blocking assignment statements to avoid race conditions.

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

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

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

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

### **All Sequential Controls Must Be in the Sensitivity List**

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

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

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

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

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

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

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

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

```
end
```

There is no sensitivity list event control.

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

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

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

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

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

- The sensitivity list event control is level sensitive.

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

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

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

The following code examples illustrate this rule:

#### Example 1

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

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

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

### Example 2

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

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

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

### Example 3

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

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

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

---

## Modeling Faster `always` Blocks

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

## **Place All Signals Being Read in the Sensitivity List**

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

## **Use Blocking Procedural Assignment Statements**

In VCS, blocking procedural assignment statements are more efficient.

Synopsys recommends zero delay non-blocking procedural assignment statements to avoid race conditions.

IEEE Std 1364-2001 describes blocking and non-blocking 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 Verilog 2001 Constructs**

In G-2012.09 and newer releases, Verilog 2001 or V2K source code conforms to the Verilog IEEE Std 1364-2001 instead of the Verilog IEEE Std 1364-1995.

If your Verilog code contains a V2K keyword as an identifier, you can tell VCS not to recognize V2K keywords with the `-v95` compile time option, for example:

```
module cell (..., ...);
```

The module identifier `cell` is a keyword in Verilog 2001, so to use it as an identifier, include the `-v95` compile-time option.

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

IEEE Std 1364-2001 Construct	Default
comma separated event control expressions: <code>always @ (r1, r2, r3)</code>	yes
name-based parameter passing: <code>modname #( .param_name ( value ) ) inst_name ( sig1, ... );</code>	yes
ANSI-style port and argument lists: <code>module dev ( output reg [7:0] out1, input wire [7:0] w1 );</code>	yes
initialize a reg in its declaration: <code>reg [15:0] r2 = 0;</code>	yes
conditional compiler directives: <code>'ifndef and 'elseif</code>	yes
disabling the default net data type: <code>'default_nettype</code>	yes
signed arithmetic extensions: <code>reg signed [7:0] r1;</code>	yes
file I/O system tasks: <code>\$fopen \$fsanf \$scanf and more</code>	yes
passing values from the runtime command line: <code>\$value\$plusarg system function</code>	yes
indexed part-selects: <code>reg1[8+:5]=5'b11111;</code>	yes
multi-dimensional arrays: <code>reg [7:0] r1 [3:0] [3:0];</code>	yes
maintaining file name and line number: <code>'line</code>	yes

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

---

## 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 the `casel` and `casez` statements. The standard does not specify that you can also make this substitution in the `case` statements, and you might infer that this substitution is not allowed in the `case` statements.

VCS, like other Verilog simulators, does not make this inference, and allows you to also substitute ? for z in the case statements. If you do, remember that z does not stand for “don’t care” in a case

statement, like it does in a `casez` or `casex` statement. In a `case` statement, `z` stands for the usual high impedance, and therefore so does `?`.

---

## Precedence in Text Macro Definitions

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

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

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

The following are the precedence rules for text macro definitions:

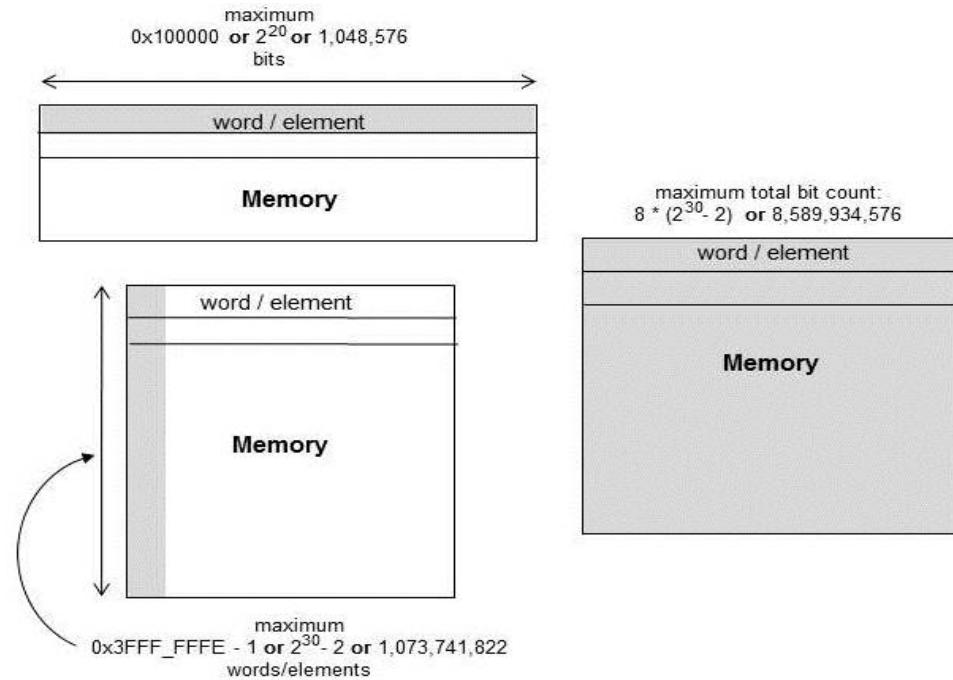
1. The ``undef` compiler directive has a higher precedence than the `+define` compile-time option.
  2. The `+define` compile-time option has a higher precedence than the ``define` and ``undefineall` compiler directives.
- 

## Memory Size Limits in the Simulator

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 8589934576.



Note:

The `reg` data type has 4 states and the `bit` data type has 2 states. The memory consumption of `reg` and `bit` variables are not the same.

For example:

```
module top;
    bit [127:0] mem_bit [0:1<<26];
    reg [127:0] mem_reg [0:1<<26];
endmodule
```

In the above example, the memory consumed by the `bit` variable `mem_bit` is:

$$2^{26} * 128 = 8,589,934,592 \text{ bits (1G Bytes)}$$

The memory consumed by the `reg` variable `mem_reg` is:

$$2 * 2^{26} * 128 = 17,179,869,184 \text{ bits (2G Bytes)}$$

The implementation limit of memory size in VCS is (2GB - 1). Therefore, a `Memory Too Large` error is issued for the `mem_reg` variable.

---

## Using Sparse Memory Models

If your design contains a large memory, the `simv` executable needs large amounts of machine memory to simulate it. However, if `/*sparse*/` is specified, the large memory does not occupy the IP space, so the above 2G-1 size limit (See “[Memory Size Limits in the Simulator](#)”) does not exist. The maximum memory size depends on address space size. If `/*sparse*/` is not specified, both full 64-bit and 32-bit VCS have the same limitation (2G-1 size limit), because even with full 64-bit, VCS still uses 32-bit IP index in back-end and runtime. So, if the memory size exceeds 2G, simulation will have errors.

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

```
reg /*sparse*/ [31:0] pattern [0:10_000_000];
integer i, j;
initial
```

```

begin
    for (j=1; j<10_000; j=j+1)
        for (i=0; i<10_000_000; i=i+1_000)
            pattern[i] = i+j;
end
endmodule

```

In simulations, this memory model uses 4 MB of machine memory with the `/*sparse*/` pragma, 81 MB without it.

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

#### 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 name of an 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 name of an 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 name of an 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 name of an 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

```

**Note:**

- Scope hierarchical names using the %m format specifier always starts with a black slash(\) and has a white space added to the Verilog instance name, when Verilog is the last instance in the hierarchy. The following example illustrates a case where scope hierarchy name uses %m format specifier.

*Example 3-1 Content of the test.v and test.vhd file*

<b>test.v</b>	<b>test.vhd</b>
<pre>module leaf(input clk);     initial \$display ("%m");     bot bt_inst();     sub_leaf inst(); endmodule: leaf  module sub_leaf();     initial \$display("%m");     bot2 inst(); endmodule  module bot2;     initial \$display("%m"); endmodule</pre>	<pre>entity top is end top;  architecture arch of top is begin     leaf_inst : entity work.leaf; end;  entity bot is end bot;  architecture arch of bot is begin     sub_leaf_inst : entity work.sub_leaf; end;</pre>

Run the files using the following commands:

- vlogan -sverilog test.v
- vhdlan test.vhd
- vcs top
- ./simv

The output generated is as follows:

\TOP.LEAF\_INST .inst

```
\TOP.LEAF_INST .inst.inst
\TOP.LEAF_INST
\TOP.LEAF_INST.bot_inst.SUB_LEAF_INST
\TOP.LEAF_INST.bot_inst.SUB_LEAF_INST .inst
```

In the output from [Example 3-1](#), you see that the scope hierarchical names starts with a black slash(\) and a white space is added to the Verilog instance name.

`\TOP.LEAF_INST inst`

---

## Returning Information About the Scope

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

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

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

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

The following example contains these system functions:

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

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

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

```
extern void showInst(input bit[31:0]); → declaration of C function named showInst

module discriminator;
task t;
reg[31:0] r; → $activeinst system function without arguments
begin
showInst($activeinst); → passed to the C function
if($activeinst("top.c1", "top.c3"))
begin
r = $activeinst;
$display("for instance %i the pointer is %s", r ? "non-zero" : "zero");
end
end
endtask

module child;
initial discriminator.t;
endmodule

module top;
```

```

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

```

In task t, the following occurs:

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

The C code is as follows:

```

#include <stdio.h>

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

```

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

During simulation, VCS and the C function display the following:

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

---

## Avoiding Circular Dependency

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

```
assign out = $random(in);

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

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

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

```
Warning-[RWSI] $random() with a 'seed' input
$random in the following statement was called with a 'seed' input
This may cause an infinite loop and an eventual crash at runtime.
"exp1.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_dumports` for Simulation and Test

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

Note:

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

---

## Dealing With Unassigned Nets

Consider the following example:

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

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

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

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

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

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

---

## Code Values at Time 0

Another issue can occur at time `0`, before values have been assigned to ports as you intended. As a result, `$lsi_dumpports` makes an evaluation for drivers when all of the users intended assignments have not been made. To correct this situation, you need to advance simulation time just enough to have your assignments take place. This can be accomplished by adding a `#1` before `$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 generates a warning message from VCS. Because top appears only once, that instance is assumed.

The second problem comes from the cross-module reference (XMR) that the force command uses. Since the module test does not 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 within test. This is done in the following way:

```

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

module top(a);

```

```

input a;
middle u1 (a);
endmodule

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

```

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

---

## Signal Value/Strength Codes

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

Testbench Level (only z drivers from the DUT)

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

DUT Level (only z drivers from the testbench)

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

Testbench Level (only z drivers from the DUT)

h	high (2 or more DUT drivers active)
Drivers Active on Both Levels	
0	low (both input and output are active with 0 values)
1	high (both input and output are active with 1 values)
?	unknown
F	tristate (input and output unconnected)
A	unknown (input 0 and output unconnected)
a	unknown (input 0 and output X)
B	unknown (input 1 and output 0)
b	unknown (input 1 and output X)
C	unknown (input X and output 0)
c	unknown (input X and output 1)
f	unknown (input and output tristate)

# 4

## Compiling/Elaborating the Design

---

This chapter describes the following sections:

- “Compiling/Elaborating the Design in the Debug Mode”
- “Optimizing Simulation Performance for Desired Debug Visibility With the `-debug_access` Option”
- “Dynamic Loading of DPI Libraries at Runtime”
- “Dynamic Loading of PLI Libraries at Runtime”
- “Key Compilation or Elaboration Features”

---

## **Compiling/Elaborating the Design in the Debug Mode**

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

- During your initial phase of the design, when you need to debug the design using debug tools, such as Verdi or UCLI.
- When you are using PLIs.
- When you use UCLI commands to force a signal to write into registers/nets.

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

```
-debug_access (+<option>),  
-debug_region=(<option>) (+<option>)
```

The following examples show how to compile the design in full, partial, and minimum debug modes:

### **Compiling/Elaborating the Design in the Partial Debug Read Mode**

```
% vcs -debug_access+r [compile_options] TOP.v
```

### **Compiling/Elaborating the Design in the Full Debug Mode**

```
% vcs -debug_access+all [compile_options] TOP.v
```

### **Compiling/Elaborating the Design With the Desired Debug Capability**

```
% vcs -debug_access<+options> [compile_options] TOP.v
```

For more information on `-debug_access` and `-debug_region` options, see the following section.

---

## Optimizing Simulation Performance for Desired Debug Visibility With the `-debug_access` Option

You can use the `-debug_access` option at compile time to selectively enable the required debug capabilities in a simulation. You can optimize simulation performance by enabling only the required debug capabilities.

This following topics describe this feature in detail:

- “[Use Model](#)”
  - “[Specifying Design Regions for `-debug\_access` Capabilities](#)”
  - “[Enabling Additional Debug Capabilities](#)”
  - “[Reduction in the Objects Being Dumped](#)”
  - “[Using `-debug\_access` With Tab Files](#)”
  - “[Unused Tab File Calls](#)”
  - “[Including Tab Files](#)”
  - “[Interaction With Other Debug Options](#)”
- 

### Use Model

Following is the use model of the `-debug_access` option:

```
-debug_access (+<option_name>) *
```

**Table 4-1** describes the supported options of `-debug_access`.

The `-debug_access` option without any additional `option_name` enables VPD and FSDB dumping capability. This option allows you to enter the UCLI prompt and use only the `run`, `quit`, and `dump` commands.

*Table 4-1 Supported Options of `-debug_access`*

Option Name	Description
r	This option enables the read capability for the entire design. This enables PLI access to <code>get value</code> , and enables the UCLI <code>get</code> command. This is the minimum debug option to invoke the Verdi interactive mode.
w	This option applies write (deposit) capability to the registers and variables for the entire design.
wn	This option applies write (deposit) capability to the nets for the entire design.
fn	This option applies force capability to the nets for the entire design. This option is equivalent to <code>-debug_access+r+fn</code> .
fwn	This option applies write (deposit) and force capability to all nets in the design. This option is equivalent to <code>-debug_access+r+wn+fn</code> .
f	This option enables the following: <ul style="list-style-type: none"><li>• Read capability on registers, variables, and nets</li><li>• Write (deposit) capability on registers and variables</li><li>• Force capability on registers, variables, and nets</li></ul> This option is equivalent to <code>-debug_access+r+w+fn+f</code>
drivers	This option enables driver debugging capability. This option is equivalent to <code>-debug_access+r+drivers</code> .
line	This option enables line debugging. It allows you to use the commands for <code>step/next</code> and <code>line breakpoints</code> . This option is equivalent to <code>-debug_access+r+line</code> .
cbk	This option enables PLI-based callbacks on nets, registers, and variables. This option is equivalent to <code>-debug_access+r+cbk</code> .

Option Name	Description
cbkd	<p>This option enables both dumping and PLI-based callbacks on dynamic nets, registers, and variables defined in classes. Class object debugging is also enabled.</p> <p>This option is equivalent to <code>-debug_access+r+line+cbkd</code>.</p>
thread	<p>This option enables the debugging of the SystemVerilog threads.</p> <p>This option is equivalent to <code>-debug_access+r+thread</code>.</p>
class	<p>This option enables debugging of the SystemVerilog classes and class objects, but the capability is also applied to the remaining portion of the design as specified by the <code>-debug_region</code> option.</p> <p>This option is equivalent to:  <code>-debug_access+r+w+thread+line+cbk+cbkd</code></p>
pp	<p>This option enables write capability on registers and variables, callbacks for the entire design, driver capability, and assertion debug capability.</p> <p>This option is equivalent to:  <code>-debug_access+w+cbk+drivers</code></p>
reverse	<p>This option enables the reverse debugging feature.</p>
all	<p>This option enables all the above options. This option is equivalent to:  <code>-debug_access+r+w+wn+f+fn+fn+drivers+line+cbk+cbkd+thread+class+pp+reverse</code></p>
-memcbk	<p>The <code>-debug_access-memcbk</code> option disables callbacks for memories and multidimensional arrays (MDAs). By default, <code>-debug_access</code> enables callbacks for memories and MDAs.</p> <p>For more information, see the "<a href="#">"Incrementally Removing Debug Capabilities"</a> section.</p>
dmptf	<p>This option enables dumping of ports and internal nodes/memories of tasks/functions.</p>

Option Name	Description
designer	<p>This option enables the following debug capabilities:</p> <ul style="list-style-type: none"> <li>• Read and write for both variables and nets</li> <li>• PLI and value change callbacks and line callback</li> <li>• FSDB/VPD dumping capability</li> <li>• Debugging or dumping of cell ports</li> </ul> <p>This option is equivalent to:  <code>-debug_access+line+r+w+wn+cbk -debug_region+cellports</code></p> <p>You can use the <code>-debug_access+designer</code> option for debugging design and simulation mismatches.</p> <p>This option does not support the following:</p> <ul style="list-style-type: none"> <li>• Class and thread debugging (no SystemVerilog Testbench debugging)</li> <li>• Driver debugging</li> </ul>
simctrl	<p>This option enables the following debug capabilities:</p> <ul style="list-style-type: none"> <li>• PLI and value change callbacks</li> <li>• Assertion control (not assertion dumping)</li> <li>• FSDB/VPD dumping capability</li> </ul> <p>You can use the <code>-debug_access+simctrl</code> option for simulation control.</p> <p>This option is equivalent to:  <code>-debug_access+r+cbk+assert_c</code></p> <p>For more information on <code>+assert_c</code>, see the “<a href="#">Assertion Debug Support</a>” section.</p> <p>This option does not support the following:</p> <ul style="list-style-type: none"> <li>• Class and thread debugging (no SystemVerilog Testbench debugging)</li> <li>• Driver debugging</li> <li>• Line stepping</li> </ul>

Option Name	Description
verbose	Reports the summary of all the -debug_access and -debug_region options passed to the vcs command line.

Example: -debug\_access+r+line+class+drivers

### Key Points to Note

- The -debug\_access options are case-insensitive.
- Read capability is disabled by default with the -debug\_access option.
- The following abbreviation of -debug\_access is supported:  
-debug\_acc
- The dynamic value change callbacks are enabled as part of the all and class options.

### Incrementally Removing Debug Capabilities

VCS allows you to remove the debug capabilities specified with the -debug\_access option. The following is the syntax to remove the debug capabilities:

-debug\_access (+<option>) \* (-<option>) \*

VCS removes the debug capabilities of the options specified with the “-” sign. For example, you can use the -debug\_acc+all-f option to remove force capability.

## **Key Points to Note**

- VCS issues a warning message when you specify the same option with `-debug_acc` for both addition and removal of debug capability. For example, VCS issues a warning message when both `-debug_acc+all-f` and `-debug_acc+f` options are specified.
- If the `-debug_acc+all` option is specified initially and later if you specify `-debug_acc-<option>`, the final option is as follows:  
`-debug_acc+all-<option>`

**For example, if both `-debug_acc+all` and `-debug_acc-f` options are specified, the final option `-debug_acc+all-f` removes the force capability.**

- VCS issues a warning message if you use `-all`, `-class`, `-designer`, or `-simctrl` options with `-debug_access`.
- If you try to remove the capability that has not been added, VCS issues a warning message saying that the specified `-<option>` is ignored.
- Multiple additions/subtractions of the same capability are treated as a single addition/subtraction. No warning message is issued in this case.
- It is recommended to use this feature with the `all`, `class`, `designer`, and `simctrl` options of `-debug_access`.

## **Assertion Debug Support**

The following assertion debug options are supported with `-debug_access`:

**Table 4-2 Assertion Debug Options of -debug\_access**

Option	Description
-assert_d	This option allows you to remove the assertion dumping capabilities from the <code>-debug_access+designer</code> and <code>-debug_access+simctrl</code> options.
+assert_c	This option enables assertion control capabilities.
+assert_f	This option enables the following: <ul style="list-style-type: none"> <li>• Dumping failures and successes for both concurrent and immediate assertions</li> <li>• Assertions control</li> </ul>

## Verdi One Search Support

The Verdi One Search feature allows you to search for design information in a database.

By default, One Search feature searches for symbols inside the `simv` executable. Shared libraries that are dynamically linked to `simv` are not searched. The `-debug_access+idents_so` option enables searching inside the shared libraries.

The Verdi One Search feature uses an internal database to quickly perform identifier search in the design. The `-debug_access+idents_db` option allows you to build the database at compile time for Verdi One Search.

## Reporting Global Debug Capability Diagnostics

You can use the `-debug_report` option at compile time to enable the reporting of debug capability added to the design.

The diagnostics are reported in an ASCII text file named debug.report. This file is generated in the current working directory and cannot be renamed during compilation. You can rename the file after compilation. There is no VCS option to rename the file.

For the Precompiled IP (PIP) flow, a corresponding report file named debug.report.<PID> (for example, debug.report.29631) is created for each IP in the flow.

The debug capability diagnostics report allows you to identify the source of debug capability and to reduce debug capability in an informed manner.

The following information is recorded in the debug diagnostics report file:

- Debug capability derived from compile options. You can view this diagnostic report after compilation.
- Debug capability declared within a tab file. This diagnostic report contains File IDs associated with the capability.
- Global Verilog functionality that enables debug capability.

## Example

Consider the following files for the global debug capability diagnostic report:

### *Example 4-1 top.sv*

```
module dutA(input clk, output reg a,b);  
  always@ (negedge clk)  
    begin  
      a=clk;
```

```

        b=a;
    end
endmodule

`celldefine
module dutB(input clk, output reg a);
    wire fgh;
    always@ (posedge clk)
        a=clk;
endmodule
`endcelldefine

module mytop;
    reg clk=0;
    wire a [0:2];
    dutA d1(clk,a[0],a[2]);
    dutB d2(clk,a[1]);
    initial begin
        clk=0;
        forever #1 clk = ~clk;
    end
    initial #100 $finish;
endmodule

```

### *Example 4-2 pli.tab*

```
acc+=frc:*
```

To compile the top.sv file, execute the following command:

```
% vcs -P pli.tab -debug_access+cbk -debug_report top.sv
```

To view the global debug capability diagnostic report, execute the following command:

```
% cat debug.report
```

Following is the debug diagnostic report file:

```

1 Here is the union of all individual -debug_access options: -debug_access+r+cbk+cbks
2
3 Global           Source          FileID
4 -----
5 acc+=frc:*
6     tab file      1
7 debug_access+cbk vcs compile command
8 +vcsd            -debug_access
9 acc+=s:*
10 acc+=s:*
11 acc+=r,cbk:*
12 +vcsd            tab file
13 acc+=frcr,frcn:*
14 acc+=s:*
15 acc+=r,cbk:*
16 █
17
18 FileID      FileName
19 -----
20 2          /global/snps_apps/vcsmx_2018.09-Beta/linux/lib/vcsdp_lite.tab
21 3          /global/snps_apps/verdi_2018.09-Beta/share/PLI/VCS/LINUX/verdi.tab
22 1          pli.tab
23
24 Modules with -debug_access caps disabled due to cell module debug caps turned off by default:
25 dutB
~
```

The tab file diagnostic report contains the file ID associated with the capability.

2  
3

This diagnostic is reported as dutB is the celldesign module and the design is not compiled with the -debug\_region+cell option.

## Limitations

This feature does not support the following compile-time options:

- The `+optconfigfile` option as it is replaced by the instance-based tab file.
- The `+acc*` option.

---

## Specifying Design Regions for -debug\_access Capabilities

You can use the `-debug_region` option to have better control over the performance of `-debug_access`. The `-debug_region` option enables you to apply debugging capabilities to the desired portion of a design.

You must use the `-debug_region` option along with the `-debug_access` option at compile time.

Following is the use model of `-debug_region`:

```
-debug_region(option_name) (option_name)*
```

[Table 4-3](#) describes the options supported by `-debug_region`.

*Table 4-3 Supported Options of `-debug_region`*

Option Name	Description	Default Functionality if <code>-debug_region</code> is not specified
+cell	Applies debug capabilities to both real cell modules and the ports of real cell modules.  Cell modules are Verilog modules that are bound with <code>`celldefine</code> and <code>`endcelldefine</code> compiler directives, as described in <i>Verilog 1800-2012 LRM, section 22.10</i> .	Debug capability is not applied to both real cell modules and the ports of real cell modules.
+cellports	Applies debug capabilities only to the ports of real cell modules.	Debug capability is not added to the ports of real cell modules.
-lib	The <code>-debug_region-lib</code> option removes debug capabilities from libraries (files passed to VCS with the preceding <code>-v/-y</code> compiler options).	Debug capabilities are applied to the libraries.

Option Name	Description	Default Functionality if -debug_region is not specified
+encrypt	Applies debug capabilities to the fully-encrypted instances (modules, programs, packages, and interfaces).	Debug capability is not applied to the fully-encrypted instances.
=sv	<p>The -debug_region=sv option applies debug capabilities only to the program, package, interface or module containing SystemVerilog constructs, and to SystemC.</p> <p>Debug capabilities are applied to SystemVerilog code inside cells and fully encrypted blocks only when the +cell option is also used.</p> <p>Debug capabilities are applied to the Verilog code inside fully encrypted modules only when the +encrypt option is also used.</p>	Debug capability is applied to Verilog, VHDL, and SystemC.
=verilog	<p>The -debug_region=verilog option applies debug capabilities to all Verilog code and to SystemC.</p> <p>Debug capabilities are applied to the Verilog code inside cells only when the +cell option is also used.</p> <p>Debug capabilities are applied to the Verilog code inside fully encrypted modules only when the +encrypt option is also used.</p>	Debug capability is applied to Verilog, VHDL, and SystemC.
=vhdl	The -debug_region=vhdl option applies debug capabilities to the VHDL code and to SystemC.	Debug capability is applied to Verilog, VHDL, and SystemC.

#### Note:

- VCS applies debug capability to the standard packages by default.
- The -debug\_region options are case-insensitive.

## **Example**

- `-debug_access+class -debug_region+cell`  
Applies class debug capabilities to the design and cell modules.

## **Key Points to Note:**

- The `-debug_region` option works only for the capabilities specified by the `-debug_access` option. It has no effect on the capabilities specified in tab files or configuration files.
- An error message is issued if you use `-debug_region` without an option.
- An error message is issued if you use `-debug_region` without `-debug_access`.
- If you use `+memcbk` and `-debug_access`, then the global debug option `+memcbk` is turned off for the following cases:
  - Inside the cells if the `-debug_region+cell` option is not specified.
  - Inside the fully-encrypted modules if the `-debug_region+encrypt` option is not specified.
  - Inside the DUT when the `-debug_region=sv` option is specified.
  - Inside the testbench when the `-debug_region=verilog` option is specified.

---

## **Enabling Additional Debug Capabilities**

This section consists of the following subsections:

- “[Driver/Load Debug Capability](#)”

- “Statement Debug Capability”
- “Force/Deposit Debug Capability”
- “Class Debug Capability”

## **Driver/Load Debug Capability**

By default, the driver/load debug support is disabled. You must use the `-debug_access+drivers` option at compile time to enable the driver/load debug support. This option enables the following capabilities:

- Active drivers
- UCLI show driver/load
- `$countdrivers`
- `$dumpports`, `$lsi_dumpports`

### **Note:**

- If you use `-debug_access` without `+drivers`, VCS automatically adds `+drivers` internally as long as `$countdrivers`, `$dumpports`, or `$lsi_dumpports` are specified in the design.
- An error message is issued when you use driver/load debug functionality without specifying `-debug_access+drivers`.

## **Statement Debug Capability**

By default, statement debugging is disabled. In UCLI, the `step`, `next`, and `file/line` breakpoints are disabled. To enable the statement debug capability, use the `-debug_access+line` option at compile time.

## **Force/Deposit Debug Capability**

By default, changing the value of a signal, variable, or net is disabled. In UCLI, the `force` command is disabled, and in VPI, the `vpi_put_value()` function is disabled. To enable value change debug capability, use the following options at compile time:

- `-debug_access+w`: For writing (depositing) values to the registers or variables.
- `-debug_access+wn`: For writing (depositing) values to the nets.
- `-debug_access+f`: For writing (depositing) values to the registers and variables, and for forcing values onto the registers, variables, and nets.
- `-debug_access+fn`: For forcing values onto the nets.

## **Class Debug Capability**

Class debugging capability enables line stepping, object IDs, thread debugging, and write capability.

This allows for object-browser debugging and the usage/display of object IDs in Verdi. It also allows for constraint debugging and thread debugging in Verdi/UCLI.

By default, class debugging is disabled. To enable class debugging, use the `-debug_access+class` option at compile time.

---

## **Reduction in the Objects Being Dumped**

The `-debug_access` option does not dump the ports of tasks and functions by default. You can use the `-debug_access+dmptf` option to dump the ports of tasks and functions.

---

## Using -debug\_access With Tab Files

If you use `-debug_access` with tab files, then the capabilities of `-debug_access` and the tab files are combined. For example, if you have a tab file with force capability applied (using `-P`) on the `testmod` module, and `-debug_access+r` is specified, then the debug access capability is applied to all instances, but the force capability is applied only to instances of the `testmod` module.

The `-debug_access` capabilities are ignored if the design is also compiled with `+applylearn`. In this case, interactive debug mode is enabled.

---

## Unused Tab File Calls

The `-debug_access` option does not apply the debug capabilities of unused tab file calls to the design. If a tab file call is marked “persistent”, then the associated debug capabilities are applied to the design.

---

## Including Tab Files

The `-debug_access` option automatically includes all the compile options required for VPD and FSDB dumping. There is no need to specify additional options to enable dumping, adding tab files, or adding PLI objects to link with `simv`.

### Dumping FSDB

If the `VERDI_HOME` environment variable is set, you can use `-debug_access` to dump FSDB. There is no need to specify Verdi PLI and tab files on the VCS command line to dump FSDB.

---

## Interaction With Other Debug Options

If you specify multiple `-debug_access` options on the same command line, then the functionality is combined. For example, specifying `-debug_acc+w -debug_acc+drivers` is equivalent to `-debug_acc+w+drivers`.

---

## Dynamic Loading of DPI Libraries at Runtime

This feature is the implementation of the SV LRM appendix on including non-Verilog or non-SystemVerilog code, through the DPI, in a design or testbench. For details, see, Annex J, “Inclusion of foreign language code” in *SystemVerilog LRM IEEE Std. 1800-2012*.

For partition compile, if you declare an import DPI function, and you do not provide the C source code on the VCS command line, VCS displays the following error message:

```
ibvcspc_test_IAwm9b.so: undefined reference to  
'my_export_dpi'  
collect2: ld returned 1 exit status
```

With dynamic loading, this error condition with partition compile and C/C++ source code does not occur.

---

## Use Model

Dynamic loading of a DPI shared library at runtime requires a number of steps before the `simv` command line. These steps are as follows:

1. For three-step flow, analyze and elaborate the Verilog or SystemVerilog code. For example:

```
%vlogan -sverilog other_options test.v  
% vcs top_level
```

For two-step flow, compile the Verilog or SystemVerilog code. For example:

```
%> vcs -sverilog other_options test.v
```

2. Compile the C code and create a shared object, for example:

```
%> gcc -fPIC -Wall ${CFLAGS} -I${VCS_HOME}/include \  
other_libraries -c test.c
```

```
%> gcc -fPIC -shared ${CFLAGS} -o test.so test.o
```

3. Load the shared object at runtime using one of the following runtime options:

-sv\_lib -sv\_root -sv\_liblist

For example, simv command lines for loading the shared object are as follows:

where the bootstrap file contains an entry specifying the location of the library

```
%> simv -sv_liblist bootstrap_file
```

```
%> simv -sv_root path_relative_or_absolute_to_shared_object \  
-sv_lib test
```

```
%> simv -sv_lib test
```

the extension for the  
shared object is omitted

where the path is relative or  
absolute to the shared object

The following is an example of a bootstrap file:

```
#!SV_LIBRARIES  
myclibs/lib1  
myclibs/lib3
```

```
proj1/clibs/lib4  
proj3/clibs/lib2
```

Where, lib1, lib2, lib3, and lib4 are shared object file names that need to be specified without extension.

---

## Dynamic Loading of PLI Libraries at Runtime

You can dynamically load a PLI library at runtime instead of linking the PLI library at compile time. For this, perform the following steps:

1. Compile the design including the PLI table file for PLI libraries with the -P compile-time option:

```
% vcs -P pli.tab design_source_files
```

2. Load the libraries dynamically at runtime, specify the libraries with the -load runtime option, and enter -load for each library:

```
% simv -load ./pli1.so -load ./pli2.so
```

In this example, there are two -load options for the libraries named pli1.so and pli2.so.

### Important:

If the PLI library is linked at compile time, the library has precedence over a PLI library loaded at runtime.

---

## Key Compilation or Elaboration Features

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

- “[Initializing Verilog Variables, Registers, and Memories](#)”
- “[Overriding Generics and Parameters](#)”
- “[Checking for x and z Values In Conditional Expressions](#)”
- “[Cross Module References \(XMRs\)](#)”
- “[Verilog Configurations and Libmaps](#)”
- “[Lint Warning Message for Missing ‘endcelldefine’](#)”
- “[Error/Warning/Lint Message Control](#)”
- “[Extracting the Files Used in Elaboration/Compilation](#)”

---

### Initializing Verilog Variables, Registers, and Memories

You can use one of the following options to initialize Verilog variables, registers, and memories in a design:

- `+vcst+initreg+random`  
This option enables initialization for an entire design.
- `+vcst+initreg+config+config_file`  
This option enables initialization for selective parts of a design.

## Initializing Verilog Variables, Registers, and Memories in an entire Design

You can use the `+vcs+initreg+random` option to initialize all bits of Verilog variables and registers defined in sequential UDPs and memories including multi-dimensional arrays (MDAs) in your design to random value 0 or 1, at time zero. The default random seed is used.

The supported data types are:

- reg
- bit
- integer
- int
- logic

To enable initialization for an entire design, the `+vcs+initreg+random` option must be specified at compile time and one of the following options must be specified at runtime:

- `+vcs+initreg+0`
- `+vcs+initreg+1`
- `+vcs+initreg+random`
- `+vcs+initreg+seed_value`

### Example 4-3

```
% vlogan [vlogan_options] file4.v file5.v file6.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

**Note:**

Specify the VHDL bottom most entity first, then move up in order.

```
% vcs +vcst+initreg+random [other_vcs_options] file1.v  
file2.v file3.v  
  
% simv +vcst+initreg+random [simv_options]
```

All Verilog variables, registers, and memories are assigned random initial values.

#### *Example 4-4*

```
% vlogan [vlogan_options] file4.v file5.v file6.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

**Note:**

Specify the VHDL bottom most entity first, then move up in order.

```
% vcs +vcst+initreg+random [other_vcs_options] file1.v  
file2.v file3.v  
  
% simv +vcst+initreg+0 [simv_options]
```

All Verilog variables, registers, and memories are assigned initial value of 0.

For more information on the `+vcst+initreg+random` compile-time option, see “[Option for Initializing Verilog Variables, Registers and Memories with Random Values](#)”.

For more information on the runtime initialization options, see “[Option for Initializing Verilog Variables, Registers and Memories at Runtime](#)”.

The initialization options may cause potential race conditions due to the initialized values specified. For more information on race condition prevention, see “[Option for Initializing Verilog Variables, Registers and Memories with Random Values](#)”.

## **Initializing Verilog Variables, Registers, and Memories in Selective Parts of a Design**

You can use the `+vcs+initreg+config+config_file` option to specify a configuration file for initializing Verilog variables, registers defined in sequential UDPs, and memories including multi-dimensional arrays (MDAs) in your design, at time zero. In the configuration file, you can define the parts of a design to apply the initialization and the initialization values of the variables.

used.

The supported data types are:

- reg
- bit
- integer
- int
- logic

To enable the initialization in selective parts of a design, you can specify the `+vcs+initreg+config+config_file` option at compile time. The `config_file` option is the configuration file used for the initialization.

If the `+vcs+initreg+config+config_file` option is specified again at runtime, then the configuration file specified at runtime overrides the configuration file specified at compile time.

### Example 4-5

```
% vlogan [vlogan_options] file4.v file5.v file6.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

Note:

Specify the VHDL bottom most entity first, then move up in order.

```
% vcs +vcst+initreg+config+test_config [other_vcs_options]  
file1.v     file2.v file3.v  
  
% simv [simv_options]
```

The configuration file, `test_config`, is used for the initialization.

### Example 4-6

```
% vlogan [vlogan_options] file4.v file5.v file6.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd  
  
% vcs +vcst+initreg+config+test_config [other_vcs_options]  
file1.v     file2.v file3.v  
  
% simv +vcst+initreg+config+my_config [simv_options]
```

The configuration file, `my_config`, is used for the initialization.

For more information on the  
`+vcst+initreg+config+config_file` option, see the following sections:

- “Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design”

- “Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design at Runtime”.

### **Configuration File Syntax:**

The syntax of the configuration file entries is as follows:

```
defaultvalue x|z|0|1|random|random seed_value

instance instance_hierarchical_name [x|z|0|1|random|
random seed_value]

tree instance_hierarchical_name depth [x|z|0|1|random|
random seed_value]

module module_name [x|z|0|1|random|random seed_value]

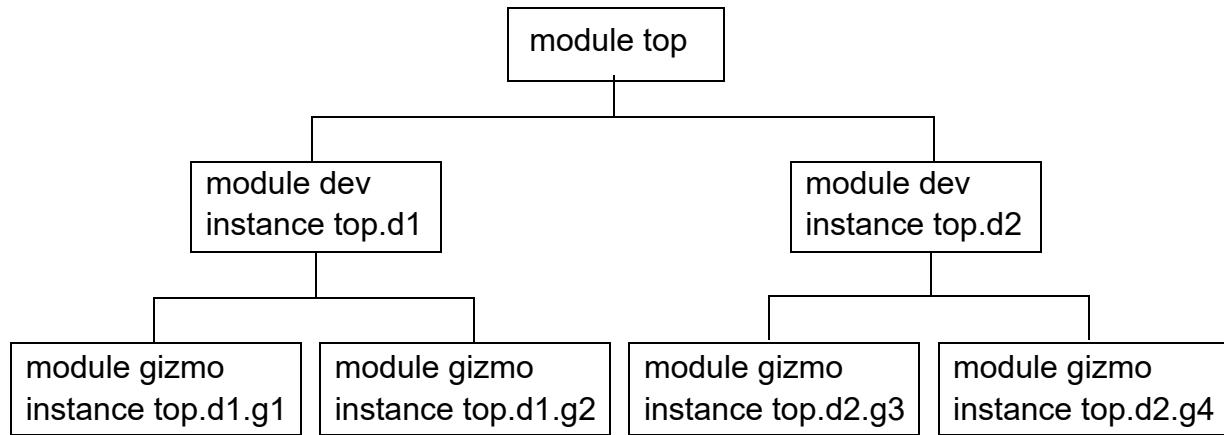
modtree module_name depth [x|z|0|1|random|
random seed_value]
```

For more information on the configuration file, see “Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design”.

### **Configuration File Example:**

Figure 4-1 shows the hierarchical diagram of a design.

*Figure 4-1 Design Hierarchy for Initializing From a Configuration File*



The following are example entries in a configuration file for the design in [Figure 4-1](#):

```
instance top.d1 0
```

Initializes variables, registers, and memories in the `top.d1` instance to value 0.

```
tree top 0 0
tree top.d1 0 x
```

The first entry initializes all variables, registers, and memories in the design to value 0. The second entry changes the initial values from 0 to x for the variables, registers and memories in the `top.d1` instance and all instances beneath `top.d1`, namely `top.d1.g1` and `top.d1.g2`.

```
module gizmo 1
```

Initializes variables, registers, and memories in all instances of the `gizmo` module to value 1, namely `top.d1.g1`, `top.d1.g2`, `top.d2.g3`, and `top.d2.g4`.

```
modtree dev 0 random
```

Initializes variables, registers, and memories in both instances of the `dev` module and all four instances beneath these instances with random values. The `top` module is not initialized.

```
modtree dev 0 random  
instance top.d1.g2 x
```

The first entry is described in the previous example. The second entry changes the initial values from random values to `x` for variables, registers, and memories in the `top.d1.g2` instance.

## Selections for Initialization of Registers or Memories

When the `+vcst+initreg+random` or `+vcst+initreg+config+config_file` option is specified at compile time, you can include one of the following initialization options:

- `+vcst+initreg+random+nomem`
- `+vcst+initreg+random+noreg`

The `+vcst+initreg+random+nomem` option disables initialization of memories or multi-dimensional arrays (MDAs). This option allows initialization of variables that do not have a dimension.

Conversely, the `+vcst+initreg+random+noreg` option disables initialization of variables that do not have a dimension. This option allows initialization of memories or MDAs.

## **Reporting the Initialized Values of Variables, Registers, and Memories**

The `VCS_PRINT_INITREG_INITIALIZATION` environment variable enables printing of all initialized variables, registers, memories, and their initialized values (represented by 0, 1, 3) to the `vcs_initreg_random_value.txt` file, where 0 indicates Boolean 0, 1 indicates Boolean 1, and 3 indicates the value `x`.

For example:

```
% setenv VCS_PRINT_INITREG_INITIALIZATION 1
```

---

## **Overriding Generics and Parameters**

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

- `-pvalue`
- `-parameters`

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

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

For example:

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

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

```
assign value path_to_the_parameter
```

Here:

assign

    Keyword that starts a line in the file.

value

    New value of the parameter.

path\_to\_the\_parameter

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

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

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

Note:

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

VCS allows you to override both generic or parameter values in the design using the compile-time option, -gfile cmd.txt.

Here, cmd.txt is an include file containing assign commands to override generic or parameter values. The syntax of the assign command is as follows:

```
assign value generics/parameters
```

**Note:**

You can also override generics at runtime. See, “[Using Verdi](#)“.

Using this option, you can override any generic or parameter of the following datatypes:

- integer
- real
- string

You can also specify more than one generic or parameter in the same line as shown below:

```
assign 1 g1 g2
```

For example:

The usage model to override the default value of a generic "WIDTH" in your top-level VHDL file to "16", is as follows:

```
% vhdlan top.vhd mem.vhd
% vcs top -gfile gen.txt
% simv
```

The include gen.txt file contains the following commands:

```
% cat gen.txt
    assign 16 WIDTH
```

Similarly, you can use the same assign commands to override the parameters in the Verilog modules as shown in the following example:

```
module top();
parameter filename="mem.txt"
```

```
initial
    $display("The filename is %s", filename);
endmodule
```

You can override the default value of the `filename` parameter in the above example, to `mem.txt`, as shown below:

```
% vhdlan top.v
% vcs top -gfile param.txt
% simv
```

The include `param.txt` file contains the following commands:

```
% cat param.txt
assign "mem2.txt" filename
```

## Usage Model

### Analysis

```
% vlogan [vlogan_options] file4.v file5.v
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

Note:

Specify the VHDL bottommost entity first, then move up in order.

### Elaboration

```
% vcs [vcs_options] top_cfg/entity/module -gfile cmd.txt
```

### Simulation

```
% simv [simv_options]
```

---

## 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 1 or 0 value. For example:

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

```
r3 = 1;
```

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

## Enabling the Checking

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

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](#)”.

## 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 4-7 False Negative Example*

```
module test;
```

```

reg r1;

initial
begin
r1=1'b0;
#1 r1=1'bx;

#0 r1=1'b1;
end

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

```

## **Example 2**

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

### *Example 4-8 False Negative Example*

```

module test;
reg r1;

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

```

```
endmodule
```

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

For example:

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

If you compile and simulate Example1 or Example2 with the `-xzcheck` elaboration/compilation option, but without the `nofalseneg` argument, VCS displays the following warning about signal `r1` transitioning to `x` or `z` value:

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

r1 false at 1

r1 true at 1
```

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

---

## Cross Module References (XMRs)

Verilog enables you to access any internal signal from any other hierarchical block without having to route it through a user interface.

VHDL does not have the language support to allow you to access internal signals from any other hierarchical block. Therefore, it is not possible to either assign or test the value of a signal deep in the design hierarchy without defining it in a global package, and then referencing it in a hierarchical block where it is used.

The `hdl_xmr` procedure (in VHDL code) and the `$hdl_xmr` system task enables you to access internal signals in a mixed HDL design and Verilog only. Therefore, you can handle the signals in the VHDL database. In a mixed HDL or Verilog only environment, you can access VHDL or Verilog signals across language boundaries using this feature.

The `hdl_xmr` procedure and the `$hdl_xmr` system task work only when the source and destination objects match in both type and size.

## **The `hdl_xmr` Procedure and the `$hdl_xmr` System Task**

The `hdl_xmr` procedure and the `$hdl_xmr` system task creates a permanent bond between the two objects, called source and destination. Each time an event occurs on the source object, the destination object is assigned a new value of the source object. Note that if the destination object has other sources, like an assignment statement, the last event value (from `hdl_xmr/$hdl_xmr` or the assignment statement) is assigned to the destination object, thus, overwriting the previous value.

When the `hdl_xmr` procedure or the `$hdl_xmr` system task is executed, source and destination objects are bound together until the end of the simulation. Therefore, it is important that `hdl_xmr/$hdl_xmr` calls are specified in the code only once.

Note:

- All these following delimiters are supported. "/", ".", ":" except for a pure VHDL design where you cannot use "." as a delimiter.
- For mixed HDL designs, you must use the `-debug_access` option for the `$hdl_xmr` system task to work.

## Data Types Supported

`hdl_xmr` and `$hdl_xmr` supports the following data types:

- Scalars, vectors, bit-selects, and part-selects (slices) are supported for both source and destination objects. Global VHDL signals are also supported.
- The following types of VHDL signals are supported with their corresponding Verilog types:
  - `integer`
  - `bit` and `bit vector`
  - `string`
  - `std_logic/std_ulogic/std_logic_vector/std_ulogic_vector`
  - Enumerated datatypes

In case of an integer type, a Verilog type of size 32, for example, `reg[31:0]`, is allowed as a matching type. Similarly, for a packed `struct std_logic_vector/std_ulogic_vector` is allowed as a matching type.

- The following SystemVerilog datatypes are supported across VCS boundary: `shortint`, `int`, `longint`, `byte`, `bit`, `logic`, and `reg`.

The following table lists the supported SystemVerilog datatypes with their matching VHDL datatypes.

*Table 4-4 SystemVerilog Datatypes With Their Matching VHDL Datatypes*

SystemVerilog Data Types	Integer	Integer Subtype	Bit vector	std_logic vector	std_ulogic vector
<b>Shortint</b>	No	No	Yes	Yes	Yes
<b>Int</b>	Yes	Yes	Yes	Yes	Yes
<b>Longint</b>	No	No	Yes	Yes	Yes
<b>Bit array</b>	Yes	Yes	Yes	Yes	Yes
<b>Logic array</b>	Yes	Yes	Yes	Yes	Yes
<b>Integer</b>	Yes	Yes	Yes	Yes	Yes

## Using the `hdl_xmr` Procedure

The syntax of the `hdl_xmr` procedure is as follows

```
hdl_xmr("source_object", "destination_object",
[verbosity]);
```

`source_object`

`source_object` can be a VHDL signal or Verilog register or net. An absolute path or a relative path to the object can be specified.

Note:

Use an absolute path instead of a relative path, if the source node resides in VHDL part of the code or if the hierarchical path has a VHDL layer.

`destination_object`

`destination_object` could be a VHDL signal or a Verilog register. An absolute path or a relative path to the object can be specified.

**Note:**

Use an absolute path instead of a relative path, if the hierarchical path contains a VHDL layer. Verilog net type as a destination object is not supported.

**verbosity**

Third optional argument to the `hdl_xmr` call is a verbosity index. If the argument is not specified then the default value is '0', otherwise possible integer values are '0' or '1'. Value '0' indicates no verbosity, and value '1' enables verbosity. If you specify '1', then every time a value of the source object is copied onto the destination object, a message is displayed.

**Note:**

To use the `hdl_xmr` procedure, you should include the XMR package in your VHDL source code as shown below:

```
Library Synopsys;
Use Synopsys.hdl_xmr_pkg.all;
```

You can call the `hdl_xmr` procedure concurrently or within a process having no sensitivity list and a wait, at the end of the process block, as shown in the following example:

```
hdl_xmr(":vh:vl:cout0","vh:coutin_xmr");
hdl_xmr("/vh/vl/cout0","/vh/in[3]", 1);
```

When there is an escaped/extended-identifier instance present in the source and if it is required to be specified, the hierarchical path must be enclosed within "\\" when a call is made.

For example, when a signal is inside an escaped instance and it is referred to as "/VH/VL/\U\_VL/U\_ESC /signall", the same signal inside the `hdl_xmr` procedure must be referred to as "/VH/

VL/\U\_VL/U\_ESC\signal1", essentially the hierarchical instance path is enclosed with "\". You can use the `hdl_xmr` procedure as shown in the following example:

```
hdl_xmr("/VH/VL/\U_VL/U_ESC\signal1", "/VH/signal2", 1);
```

Similarly, if there is an escaped instance, \U\_NETLIST/NETLIST\_TOP with the NETLSIT\_TOP/U\_AND\_2.Z signal, the same signal inside the `hdl_xmr` procedure must be referred to as:

```
hdl_xmr("/TBTOP/U_TOP/U_NETLIST/\U_NETLIST/NETLIST_TOP//NETLIST_TOP/U_AND_2.Z", "/TBTOP/CLK", 1);
```

## Using `hdl_xmr` With Generate Blocks

You can use the following syntax for names within generate blocks:

- For generate block in Verilog, the name for instances within generate block that has escaped name is "escaped\_generate\_name.instance\_name"
- For generate block in VHDL, the name for instances within generate block that has extended name is "extended\_generate\_name.instance\_name"

## Using the `$hdl_xmr` Task

The syntax of the `$hdl_xmr` procedure is as follows

```
$hdl_xmr("source_object" , "destination_object",
[verbosity]);
```

`source_object`

`source_object` could be a VHDL signal or Verilog register or net. An absolute path or a relative path to the object can be specified.

Note:

Use an absolute path instead of a relative path, if the source node resides in VHDL part of the code or if the hierarchical path has a VHDL layer.

`destination_object`

*destination\_object* could be a VHDL signal or a Verilog register. An absolute path or a relative path to the object can be specified.

Note:

Use an absolute path instead of a relative path, if the hierarchical path contains a VHDL layer. Verilog net type as a destination object is not supported.

`verbosity`

Third optional argument to the `hdl_xmr` call is a verbosity index. If the argument is not specified then the default value is '0', otherwise possible integer values are '0' or '1'. Value '0' indicates no verbosity. When verbosity is required, that is, '1' is the third argument, then every time when the value of the source object is copied on to the destination object, a message is displayed.

You can use the `$hdl_xmr` system task as shown in the following example:

```
initial begin
$hdl_xmr("vl.vh.clk", "vl.vclk");
$hdl_xmr("/vl/vh/reset_n", "/vl/vrst_n[0]", 0);
$hdl_xmr("vl:vh:state[3:0]", "vl:state[4:7]", 1);
end
```

When there is an escaped/extended-identifier instance present in the source and if it is required to be specified, the hierarchical path should be enclosed within “\\” when a call is made.

For example, when a signal is inside an escaped instance and it is referred to as "VL.\U\_VL/U\_ESC .VH.Signal1", the same signal inside the `hdl_xmr` task must be referred to as "VL.\U\_VL/U\_ESC\ .VH.Signal1", essentially the hierarchical instance path is enclosed with “\\”. You can use the `$hdl_xmr` system task as shown in the following example:

```
$hdl_xmr("VL.\U_VL/U_ESC\ .VH.Signal1", "VL.signal2", 1);
```

Similarly, if there is an escaped instance `\U_NETLIST/NETLIST_TOP` with the `NETLSIT_TOP.U_AND_2.Z` signal, the same signal inside the `$hdl_xmr` task is referred to as:

```
$hdl_xmr("top.U_NETLIST.\U_NETLIST/NETLIST_TOP\ .NETLIST_TOP.U_AND_2.Z", "top.CLK", 1);
```

## Using `$hdl_xmr` With Generate Blocks

You can use the following syntax for names within generate blocks:

- For generate block in Verilog, the name for instances within generate block that has escaped name is "`\escaped_generate_name.instance_name\\`"
- For generate block in VHDL, the name for instances within generate block that has extended name is "`\extended_generate_name\.instance_name`"

# Use Model

## Analysis

```
% vlogan [vlogan_options] file4.v file5.v file6.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

### Note:

Specify the VHDL bottom most entity first, then move up in order.

## Elaboration

```
% vcs [vcs_options] -debug_access+r+cbk top_cfg/entity/module
```

## Simulation

```
% simv [simv_options]
```

## Examples

- `hdl_xmr` with escaped/extended identifiers

Consider the following design code — `test.v` and `test.vhdl` that are cross referenced using escaped/extended identifiers.

### test.v

```
cat test.v
`timescale 1ns/1ns
module top;
reg clk;
middle `m_inst/m_inst ();
initial
begin
$hdl_xmr ("top.clk",
           "top.\`m_inst/m_inst\\\.leaf_inst.\`I1/I1\\\.vhtop", 1);
$hdl_xmr ("top.clk",
           "/top/\`m_inst/m_inst\\/leaf_inst/\`I1/I1\\/.vhtop", 1);
clk = 0;
#10 clk = 1 ;
#10 $finish;
end
```

```
endmodule

module middle ;
leaf leaf_inst();
endmodule
```

### test.vhd

```
library ieee;
use ieee.std_logic_1164.all ;

entity vhleaf is
end vhleaf ;

architecture a of vhleaf is
signal vhtop : std_logic ;
begin
end a ;

library ieee;
use ieee.std_logic_1164.all ;
library synopsys;
use synopsys.hdl_xmr_pkg.all ;

entity leaf is
end leaf ;

architecture a of leaf is
component vhleaf
end component ;
begin
hdl_xmr ("top.clk",
          "top.\m_inst/m_inst\.leaf_inst.\I1/I1\.vhtop", 1);
hdl_xmr ("top.clk",
          "/top/\m_inst/m_inst\/leaf_inst/\I1/I1\//vhtop", 1);
\I1/I1\  : vhleaf
end a ;
```

If the call ids are made from Verilog, both escaped and extended identifier must be enclosed within “\”. Whereas, if the call is made from VHDL, both escaped and extended identifier must be enclosed within “\\”. The `hdl_xmr` procedures for this example are defined as shown in the following table:

	<b>Call From</b>	<b>Format</b>	<b>hdl_xmr Procedures and Tasks</b>
Escape identifier in Verilog - `m_inst/m_inst;	Verilog	dot	<code>\$hdl_xmr ("top.clk", "top.\`m_inst/m_inst\\leaf_inst.I1/I1\\.vtop", 1);</code>
Extended identifier in VHDL - `I1/I1\ : vhleaf;		slash	<code>\$hdl_xmr ("top.clk", "/top/\`m_inst/m_inst\\leaf_inst/I1/I1\\vhleaf", 1);</code>
Escape identifier in Verilog - `m_inst/m_inst;	VHDL	dot	<code>hdl_xmr ("top.clk", "top.\`m_inst/m_inst\\leaf_inst.I1/I1.vtop", 1);</code>
Extended identifier in VHDL - `I1/I1\ : vhleaf;		slash	<code>hdl_xmr ("top.clk", "/top/\`m_inst/m_inst\\leaf_inst/I1/I1.vtop", 1);</code>

- `hdl_xmr` with generate blocks

Consider the following design code—`test.v` and `test.vhd` that are cross referenced using generate blocks.

### test.v

```
`timescale 1ns/1ns
module top;
reg clk;
middle m_inst ();
initial
begin
$hdl_xmr ("top.clk",
```

```

    "top.m_inst.\gen[0].leaf_inst\\.\GEN[0]\\I1.vhtop", 1);
$hdl_xmr ("top.clk",
"/top/m_inst/ \gen[0].leaf_inst\\/\GEN[0]\\I1/vhtop", 1);

clk = 0;
#10 clk = 1 ;
#10 $finish;
end
endmodule

module middle ;
generate
begin : \gen[0]
  leaf leaf_inst();
end
endgenerate
endmodule

```

### **test.vhd**

```

library ieee;
use ieee.std_logic_1164.all ;

entity vhleaf is

end vhleaf;

architecture a of vhleaf is

signal vhtop : std_logic ;
begin

end a ;

library ieee;
use ieee.std_logic_1164.all ;
library synopsys;
use synopsys.hdl_xmr_pkg.all ;

entity leaf is

end leaf ;

architecture a of leaf is

component vhleaf

```

```
end component ;  
  
begin  
    hdl_xmr ("top.clk",  
              "top.m_inst.\gen[0].leaf_inst\\.\\GEN[0]\\.\$1.vhtop", 1);  
    hdl_xmr ("top.clk",  
              "/top/m_inst/\gen[0].leaf_inst//\GEN[0]\\/\$1/vhtop", 1);  
  
    \GEN[0]\ : if ( true ) generate  
        \$1 : vhleaf ;  
    end generate \GEN[0]\ ;  
end a ;
```

The `hdl_xmr` tasks for this example are defined as shown in the following table:

	<b>Call From</b>	<b>Format</b>	<b>hdl_xmr tasks</b>
<pre>Generate instance in Verilog -  generate  begin : \gen[0]  leaf leaf_inst();  end  endgenerate</pre> <pre>Generate instance in VHDL- \GEN[0]\ : if ( true ) generate     I1 : vhleaf ; end generate \GEN[0]\ ;</pre>	Verilog	dot	\$hdl_xmr ("top.clk", "top.m_inst.\gen[0].leaf_in st\\.\GEN[0]\\I1.vtop", 1);
		slash	\$hdl_xmr ("top.clk", "/top/ m_inst/\gen[0].leaf_inst\\/ \GEN[0]\\I1/vtop", 1);

The `hdl_xmr` procedures for this example are shown in the following table:

	<b>Call From</b>	<b>Format</b>	<b>hdl_xmr Procedures</b>
<pre>Generate instance in Verilog -  generate  begin : \gen[0]  leaf leaf_inst();  end  endgenerate</pre> <pre>Generate instance in VHDL- \GEN[0]\ : if ( true ) generate     I1 : vhleaf ; end generate \GEN[0]\ ;</pre>	VHDL	dot	hdl_xmr ("top.clk", "top.m_inst.\gen[0].leaf_in st\\.\GEN[0]\\I1.vtop", 1);
		slash	hdl_xmr ("top.clk", "/top/ m_inst/\gen[0].leaf_inst\\/ \GEN[0]\\I1/vtop", 1);

## **\$hdl\_xmr Support for VHDL Variables**

VCS supports the usage of VHDL objects of type, variable, in the \$hdl\_xmr system task. This support enables you to use VHDL variables, as source or destination, in the \$hdl\_xmr (not hdl\_xmr in VHDL side) call.

### **Use Model**

In Verilog source, you should call \$hdl\_xmr as:

```
$hdl_xmr (<"source variable">, <"destination signal">, <verbosity_value>)
```

```
$hdl_xmr (<"source signal">, <"destination variable">, <verbosity_value>)
```

You can specify the source variable and the destination variable in a relative or an absolute path. The last integer value, `verbosity_value`, is optional. It is only used for verbosity. The variable object is the VHDL object.

To enable the support for \$hdl\_xmr with VHDL variables, you must use one of the following compile-time options:

- vcs <top> -debug\_access\* -vdbg\_watch
- vcs <top> -debug\_access+all

Note:

- In VHDL variables, you must pass the `-vdbg_watch` option along with the `-debug_access*` option. If you are using the `-debug_access+all` option, then there is no need to pass the `-vdbg_watch` option.

- For mixed HDL designs, you must use the `-debug_access*` option for the `$hdl_xmr` system task to work.

## Data Type Support and Usage Examples

The following table shows data type support and usage examples:

*Table 4-5 Data Type Support and Usage Examples*

Verilog Data Types	VHDL Data Types for Variable
<b>reg</b>	<p><b>bit/std_logic/std_ulogic</b></p> <p><b>VHDL record elements. Datatypes for record elements can be bit/ std_logic/ std_ulogic.</b></p>
<pre>module tb;  reg r1,r2; reg [0:3] r3,r4;  leaf inst1();  initial begin \$hdl_xmr("inst1.r1","r1",1); \$hdl_xmr("r2",inst1.r2",1); \$hdl_xmr("inst1.r1","r3[1:1]",1); \$hdl_xmr("r4[1:1]",inst1.r2",1); \$hdl_xmr("inst1.rec.r1","r1",1); \$hdl_xmr("r2",inst1.rec.r2",1); \$hdl_xmr("inst1.rec.r1","r3[1:1]",1) ; \$hdl_xmr("r4[1:1]",inst1.rec.r2",1); end  endmodule</pre>	<pre>entity leaf is end leaf;  architecture beh of leaf is  type pkt is record r1 : bit; r2 : std_logic; end record;  shared variable rec : pkt ; shared variable r1 : std_logic ; shared variable r2 : std_ulogic ;  begin end;</pre>

Verilog Data Types	VHDL Data Types for Variable
<b>reg vector</b>	<p><b>bit_vector/std_logic_vector/signed/unsigned/integer/natural</b></p> <p><b>VHDL record elements. Datatypes for record elements can be bit_vector/std_logic_vector/signed/ unsigned/integer/natural</b></p>
<pre>module tb;   reg [31:0] r1,r2,r3,r4;   leaf inst1();   initial   begin     \$hdl_xmr("inst1.r1","r1",1);     \$hdl_xmr("r2",inst1.r2",1);     \$hdl_xmr("inst1.r1[15:0]", "r3[31:16]       ",1);     \$hdl_xmr("r4[15:0]",inst1.r2[15:0]",       1);     \$hdl_xmr("inst1.rec.r1","r1",1);     \$hdl_xmr("r2",inst1.rec.r2",1);     \$hdl_xmr("r4[3:0]",inst1.rec.r2[3:0]       ",1);   end endmodule</pre>	<pre>entity leaf is end leaf;  architecture beh of leaf is  type pkt is record   r1 : natural;   r2 : std_logic_vector(31 downto     0); end record;  shared variable rec : pkt; shared variable r1,r2 :   std_logic_vector(31 downto 0): begin end;</pre>
<b>reg mda</b>	<p><b>vhdl mda. Base data type for array elements can be bit/std_logic/std_ulogic/bit_vector/std_logic_vector/integer/natural</b></p>

<b>Verilog Data Types</b>	<b>VHDL Data Types for Variable</b>
<pre> module tb;  reg [31:0] r1,r2,r3 [0:7] reg [31:0] r4; leaf inst1(); initial begin \$hdl_xmr("inst1.r1","r1",1); \$hdl_xmr("r2",inst1.r2",1); \$hdl_xmr("inst1.r3","r3",1); \$hdl_xmr("r4",inst1.r2[1]",1); \$hdl_xmr("inst1.r1[2]","r4",1); \$hdl_xmr("r2[2]",inst1.r2[2]",1); end endmodule </pre>	<pre> entity leaf is end leaf;  architecture beh of leaf is  type ram is array(0 to 7) of std_logic_vector(31 downto 0); type ram1 is array(0 to 7) of bit_vector(31 downto 0); type ram2 is array(0 to 7) of natural;  shared variable r1 : ram; shared variable r2 : ram1; shared variable r3 : ram2;  begin end; </pre>
<b>real</b>	<b>vhdl real</b>
<b>real</b>	<b>real field of vhdl record</b>
<b>real mda</b>	<b>real mda</b>
<b>Note: Verilog real vectors are not supported.</b>	

<b>Verilog Data Types</b>	<b>VHDL Data Types for Variable</b>
<pre> module tb;  real r1 [0:7]; real r2;  leaf inst1();  initial begin \$hdl_xmr("inst1.r1","r1",1); \$hdl_xmr("r2",inst1.r2",1); \$hdl_xmr("r2",inst1.r1[1]",1); \$hdl_xmr("inst1.r1[1]","r2",1);  end  endmodule </pre>	<pre> entity leaf is end leaf;  architecture beh of leaf is  type ram is array(0 to 7) of real;  shared variable r1 : ram; shared variable r2 : real;  begin end; </pre>
<b>packed struct</b> <b>array of packed struct</b>  <b>Data types for elements of packed struct :</b> <b>reg/logic</b>  <b>reg/logic vector</b>  <b>real</b>	<b>vhdl record</b> <b>array of vhdl records</b>  <b>Data types for elements of vhdl record:</b> <b>bit/std_logic/std_ulogic</b>  <b>bit_vector/std_[u]logic_vector/signed/</b> <b>unsigned/natural/integer</b>  <b>real</b>

<b>Verilog Data Types</b>	<b>VHDL Data Types for Variable</b>
<pre> module tb;  typedef struct packed {reg [31:0] t ; reg [15:0] b;} st;  st r1,r2; st r3 [0:1];  leaf inst1();  initial begin \$hdl_xmr("r2","inst1.r2",1); \$hdl_xmr("inst1.r1","r1",1); \$hdl_xmr("inst1.r3","r3",1); \$hdl_xmr("inst1.r3[1]","r3[1]",1); \$hdl_xmr("inst1.r3[0]","r1",1); \$hdl_xmr("r2","inst1.r3[1]"); end  endmodule </pre>	<pre> entity leaf is end leaf;  architecture beh of leaf is  type rec is record a1 : integer ; a2 : bit_vector(15 downto 0); end record;  shared variable r1,r2 : rec;  type arr is array(0 to 1) of rec; shared variable r3 : arr;  begin end beh; </pre>

---

## Verilog Configurations and Libmaps

Library mapping files are an alternative to the de facto standard way of specifying Verilog library directories and files with the `-v`, `-y`, and `+libext+ext` compile-time/analysis 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 *SystemVerilog LRM IEEE Std. 1800-2012*. It specifies that SystemVerilog interfaces can be assigned to logical libraries.

## Library Mapping Files

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

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

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

Note:

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

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

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

You specify the library mapping file with the `-libmap` during compilation/analysis.

## 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 option specified in the library mapping file overrides the +incdir option specified in the VCS command line.

## Configurations

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

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

Configurations provides the following functionalities:

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

You can define a configuration in a library mapping file or in any type of Verilog source file outside the module definition as shown in the [Example](#).

Configurations can be mapped to a logical library 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

A keyword that begins a configuration.

config\_identifier

A name you enter for the configuration.

design

A 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`

A keyword that ends a configuration.

## The default Clause

The `default` clause specifies 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 details about a specific instance. These details depend 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 the `instance` clause:

```
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 the logical library `lib2`.

## The cell Clause

The `cell` clause is similar to the `instance` clause except that it specifies details about all instances of a cell definition instead of specifying details about a particular instance. These details depend 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 have gate-level code describing the adder in a logical library named `gatelib`. If, for example, you wanted the gate-level code used for the 0 (zero) bit of the adder and the RTL level code used for the other seven bits, the configuration might appear as:

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

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

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

## The -top Compile-Time Option

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

```
vcs -top top_cfg ...
```

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` option tells VCS not to create extraneous top-level modules, that is, one that you do not specify.

## Limitations of Configurations

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

- You cannot specify source code for user-defined primitives in a configuration.
- The VPI functionality, described in Section 33.7 “Displaying library binding information” in the *SystemVerilog LRM IEEE Std. 1800-2012*, is not implemented.

## Use Model

### Analysis

```
% vlogan -libmap libmap.v [vlogan_options] file1.v \
```

```
file2.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

### Note:

Specify the VHDL bottom-most entity first, then move up in order.

### Elaboration

```
% vcs [vcs_options] top_cfg/entity/config
```

### Simulation

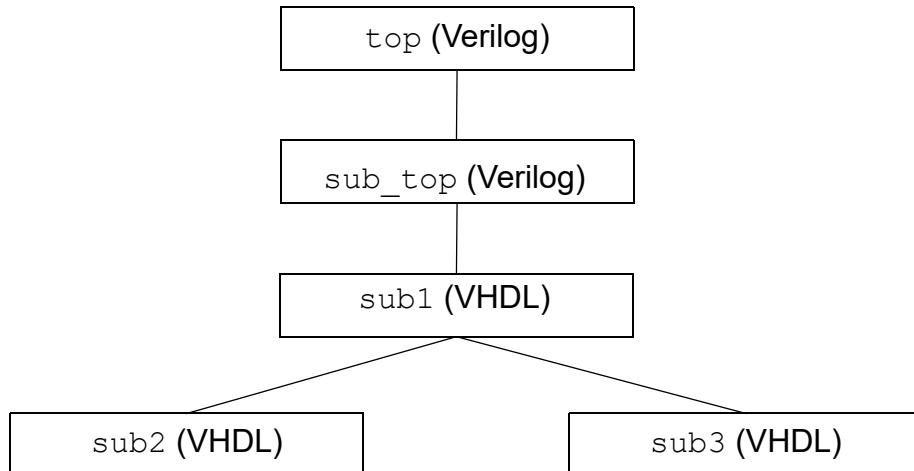
```
% simv [sim_options]
```

### Example

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.

For example, you have a design with VHDL-top design with the top entity as `top` instantiating a Verilog-top module, `sub_top`. This Verilog module, `sub_top`, further instantiates a VHDL entity, `sub1` and the VHDL entity, `sub1`, instantiates VHDL entities, `sub2` and `sub3` as shown below:

*Figure 4-2*



Suppose, you have the Verilog version of the entities, `sub1` and `sub2`, and need to compile and simulate the design with the Verilog version of `sub1` and the VHDL version of `sub2`. You can achieve this by defining configuration blocks in the Verilog source file outside the module definition or in a separate file as shown below:

To bind the Verilog version of `sub1`, define a configuration block in `top.v` (outside the module definition) as shown below:

```
//---top.v---
Module sub_top (...);
  u_sub1 sub1 (...);
endmodule

config top_cfg;
  design work.top;
  instance top.u_sub1 use work.sub1_cfg:config
endconfig
```

or in a separate file as shown below:

```
config top_cfg;
  design work.top;
  instance top.u_sub1 use work.sub1_cfg:config
```

```
endconfig
```

To bind the VHDL version of "sub2", define a configuration block in sub1.v (outside the module definition) as shown below:

```
//---sub1.v---
Module sub1(...);
  u_sub2 sub2(...);
  u_sub3 sub3(...);
endmodule

config sub1_cfg;
  design work.sub1;
  instance sub1.u_sub2 use work.CFG_SUB2_BEH:config
endconfig
```

or in a separate file as shown below:

```
config sub1_cfg;
  design work.sub1;
  instance sub1.u_sub2 use work.CFG_SUB2_BEH:config
endconfig
```

The VHDL files, sub2.vhd and sub3.vhd, are as shown below:

```
---Sub2.vhd---
Entity SUB2 is
  Port ( ... );
End SUB2;

Architecture BEH of SUB2 is
Begin
  Process
    ...
  End process;
End BEH;

Configuration work.CFG_SUB2_BEH of SUB2 is
  For BEH
```

```

        End for;
End CFG_SUB2_BEH;
---Sub3.vhd---
Entity SUB3 is
    Port ( ... );
End SUB3;
Architecture BEH of SUB3 is
Begin
    Process
        ...
    End process;
End BEH;

Configuration work.CFG_SUB3_BEH  of SUB3 is
    For BEH
        End for;
End CFG_SUB3_BEH;

```

The usage model for the above example is shown below:

## **Analysis**

```
% vlogan top.v sub1.v -libmap libmap.v

% vhdlan sub2.vhd sub3.vhd
```

## **Note:**

Specify the VHDL bottom-most entity first, then move up in order.

## **Elaboration**

```
% vcs top
```

## **Simulation**

```
% simv
```

## **Supported Features**

Verilog configuration supports the following features:

- Verilog configurations in mixed HDL designs can configure Verilog instances and boundary VHDL instances (that is, VHDL entity instantiations in a Verilog module). However, the Verilog configuration cannot configure any sub-tree below the VHDL instance in a Verilog module. To configure the sub-tree below the boundary VHDL instances, a separate Verilog configuration must be instantiated in the VHDL design unit.
- Supports direct or component instantiation. It also supports Verilog configuration specification within VHDL.
- The instance resolution happens based on the resolution rules applicable for the instantiating unit. For example, if the unit is in Verilog, then Verilog rules apply, or if the unit is in VHDL, then VHDL rules apply.
- A VHDL design can have multiple Verilog instances with same module name, but with different implementations. They should be analyzed into different logical libraries.
- A VHDL design can instantiate Verilog configuration like VHDL configuration. However, the configuration and the Verilog module that it is configuring must be analyzed in the same logical library as per parent VHDL rules.
- All configuration rules in Verilog configuration for binding instances are supported.
- While resolving the Verilog configuration, the library resolution happens as per the rules mentioned in the IEEE Verilog LRM Std 1364-2005 Section 13.3.1.5. The library order in the `synopsys_sim.setup` file for searching the Verilog or VHDL cell will be ignored.

## **Limitations of Configurations**

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

- Verilog configuration cannot have VHDL DUT in the design statement.
- Verilog configurations cannot configure pure VHDL designs.
- The hierarchical path in the instance-based rule of Verilog configuration cannot go through the VHDL instance. The hierarchical path should be pure Verilog with target Verilog or VHDL instance.
- Direct instantiation of the Verilog configuration inside a VHDL generate statement is not supported.
- SystemC with Verilog configurations is not supported for VHDL top-design topology.
- A separate compile flow with Verilog configurations used in mixed HDL designs is not supported.
- Array of instances is not supported.

## **Using the -liblist Option**

You can specify the `-liblist` option at elaboration time as follows:

```
-liblist logic_lib1+logic_lib2+
```

It specifies the library search order for unresolved module or entity instantiated in Verilog.

When `-liblist` is specified, VCS starts searching libraries in the order specified with `-liblist`. It neither honors the library order specified in the `synopsys_sim.setup` file, nor look at the other libraries present in the `synopsys_sim.setup` file. If VCS does not find the module definition in the libraries specified with the `-liblist` option, it generates an error message.

In presence of Verilog configuration, VCS first tries to resolve the instances with the configuration rules provided in the configuration file. If no instances are found, then VCS looks for `-liblist`.

In the following example, `-liblist L2` is used to find the instance, `top.level1.level2`:

## Example

```
cat level1.v
*****
module level1;
    level2 l2();
    initial $display("%l %m level1 (design)");
endmodule

cat file.v
*****
module level1;
    level2 l2();
    initial $display("%l %m level1 (library)");
endmodule

module level2;
    initial $display("%l %m level2 (library)");
    level3 l3();

```

```

endmodule

cat file1.v
*****
module level3;

initial $display("%l %m level3 (library)");

endmodule

cat dummy.v
*****
module dummy;
    level1 l();
endmodule

cat dummy1.v
*****
module dummy;
    level3 l();
endmodule

cat top.v
*****
module top;
    level1 l1();
endmodule

cat topcfg.v
*****
config topcfg;
    design L1.top;
    instance top.l1 liblist L3;
    default liblist L2 L1;
endconfig

cat synopsys_sim.setup
WORK > DEFAULT
DEFAULT : ./work
L1 : ./lib1
L2 : ./lib2
L3 : ./lib3

```

```

cat run
*****
vlogan -sverilog level1.v -work L3
vlogan -sverilog dummy1.v -v file1.v -work L3
vlogan -sverilog dummy.v -v file.v -work L2
vlogan -sverilog file1.v -work L2
vlogan -sverilog top.v -work L1
vlogan -sverilog topcfg.v -work L1
vcs L1.topcfg -diag libconfig -libmap_verbose -liblist L2

```

The following options enable you to search in different order:

- `-liblist_work`
- `-liblist_nocelldiff`

The `-liblist_work` elaboration option enables VCS to first search in the parent work library when attempting to resolve a design cell. This option is relevant only when the Verilog configuration file is not used.

The `-liblist_nocelldiff` elaboration option disables the differentiation between design cell and library cell. You can use this option at elaboration time to override the default precedence order.

## Design Cells and Library Cells

VCS designates Verilog modules as either design cells or library cells. When analyzing files without `-v/-y`, Verilog modules are treated as design cells.

When analyzing files using `-v/-y`, Verilog modules resolved from `-v/-y` are treated as library cells.

For example:

- Consider the following command-line:

```
% vlogan a.v
```

If `a.v` contains module `a`, then `a` is treated as the design cell.

- Consider the following command-line where `a.v` is passed with `-v`:

```
% vlogan top.v -v a.v
```

If `top.v` needs module `a` defined in `a.v`, then `a` is treated as the library cell.

By default, design cells take precedence over the library cells while searching for a module definition. That is, if a library cell and a design cell have the same module name, then VCS first searches for the design cell in the applicable libraries. If VCS cannot find the design cell, it will then search for the library cell.

For example, consider the following command:

```
% vcs top.v b_prime.v -v b.v
```

If both `b_prime.v` and `b.v` define the `b` module needed by `top.v`, then VCS gives first preference to the `b` module defined in `b_prime.v` while resolving the instance of the `b` module defined inside the `top` module.

You can use the `-liblist_nocelldiff` option at elaboration time to override the default precedence order. This option does not differentiate between design cells and library cells.

Note:

- If the design and library cell definitions of a module are dumped in the same library, then VCS stores the last dumped design definition. For example, consider the following commands:

```
% vlogan level1.v -work L1  
  
% vlogan top.v -v level1.v -work L1
```

In this case, both design cell and library cell definitions of the level1 module are analyzed in library L1 one after another.

Same is applicable for multiple design definitions also and VCS stores the last dumped design definition.

- Design cells take precedence over library cells if the `-liblist_nocelldiff` is not passed at elaboration time.

## Library Search Order Rules

This section describes the library search order rules for module definitions when Verilog configuration file is used with `-v/-y` and `-liblist` options. This section consists of the following subsections:

- “[Library Search Order Rules for Mixed HDL Designs](#)”
- “[Library Search Order Rules for Verilog or SystemVerilog Designs](#)”

### Library Search Order Rules for Mixed HDL Designs

VCS searches for the same language instances as that of the parent. If the same language child instance is not available, then VCS searches for other language instance.

For example, if both Verilog and VHDL child instances are available in the same library, a VHDL parent gives priority for a VHDL child instance over a Verilog child instance. Also, the Verilog parent gives priority for the Verilog child instance over the VHDL child instance. Further, VHDL parent search order is defined by the VHDL language and Verilog parent search order for various conditions are explained in detail in the subsequent sections.

For any Verilog parent design cell, Verilog configuration is honored until the VHDL instance is hit. Subsequent VHDL hierarchy can be configured using VHDL configurations until Verilog instance is hit. The subsequent Verilog hierarchy beneath VHDL can be configured using appropriate Verilog configurations. In absence of appropriate Verilog configuration, regular search order is followed.

The subsequent sections discuss library search order rules for mixed HDL designs with and without the Verilog configuration file. The library search order rules are listed in order of priority, where the first rule is given preference before the second rule.

## **Library Search Order Rules for Mixed HDL Designs Without Configuration File**

Library Search order rules for mixed HDL designs without a configuration file are as follows:

### **1. VHDL-VHDL Scenario:**

Binding rule of VHDL is applied. The `-liblist` option does not have any impact on this scenario.

### **2. VHDL-Verilog Scenario:**

Binding rule of VHDL is applied. The `-liblist` option does not have any impact on this scenario.

3. Verilog-VHDL Scenario:

The VHDL instantiation gets resolved to a VHDL entity as per the library order specified in `-liblist`.

The `-liblist` option is applicable for Verilog parent only.

### **Library Search Order Rules for Mixed HDL Designs With Configuration File**

Library Search order rules for mixed HDL designs with a configuration file are as follows:

1. VHDL-VHDL Scenario:

Binding rule of VHDL is applied. The `-liblist` option does not have any impact on this scenario.

2. VHDL-Verilog Scenario:

Binding rule of VHDL is applied. The `-liblist` option does not have any impact on this scenario.

3. Verilog-VHDL Scenario:

For Verilog-top designs, Verilog cells are searched, if not found, then VHDL cells are searched as per the rules. For any Verilog instantiated in the VHDL, the configuration rules are not applicable.

## **Library Search Order Rules for Verilog or SystemVerilog Designs**

The library search order rules are listed in order of priority, where the first rule is given preference before the second rule. Each rule contains multiple sub-rules, which are listed in the order of priority. VCS first determines the rule of highest priority, and then searches the sub-rules in that rule, in the order of priority.

This section consists of the following subsections:

- “[Library Search Order Rules for Verilog or SystemVerilog Designs With a Configuration File](#)”
- “[Library Search Order Rules for Verilog or SystemVerilog Designs Without a Configuration File](#)”

### **Library Search Order Rules for Verilog or SystemVerilog Designs With a Configuration File**

#### **RULE1: `uselib**

1. Any library cell pre-resolved from a `uselib file during vlogan:
  - a. If a cell is resolved from `uselib, then no configuration rules are applied to an instance in it if it was pre-resolved in the `uselib library. For usage example, see “[Example-1](#)”
  - b. If the `uselib instance is configured to resolve a design cell, then configuration rules are applied for any hierarchy under it. For usage example, see “[Example-2](#)”
  - c. If a cell is resolved from `uselib, and an instance in it was not pre-resolved from `uselib during analysis, then if it finds a design cell during elaboration, it is taken. In this case, the hierarchy underneath `uselib can be configured. For usage example, see “[Example-3](#)”

- d. If a design cell for an instance is found during `vlogan` itself, VCS does not look for ``uselib` or `-v/-y`. For usage example, see "[Example-4](#)"

## RULE2: Instance use Clause

- 1. Configuration instance `use` clause
  - a. VCS looks for an instance in the `use` clause specified in the configuration. For usage example, see "[Example-5](#)"
  - b. If not found, VCS generates an error message.

## RULE3: Instance liblist Clause

- 1. Configuration instance `liblist` clause
  - a. Resolves the design cell using the configuration instance `liblist` clause, if the `liblist` argument is not empty. For usage example, see "[Example-6.1](#)"
  - b. Looks for the design cell in the parent cell library, if `liblist` argument is empty. For usage example, see "[Example-6.2](#)"
  - c. Looks for the design cell in libraries specified with `-liblist` at the command line. For usage example, see "[Example-6.3](#)"
  - d. If the cell was analyzed using `-v` or `-y` in the same `vlogan` invocation as the parent.
  - e. Looks for the library cell in the configuration instance `liblist` clause, if the `liblist` argument is not empty.
  - f. Looks for the library cell in the parent cell library, if the `liblist` argument is empty.

- g. Looks for library cells in the libraries specified with -liblist at the command line.

#### **RULE4: Cell use Clause**

1. Configuration Cell use Clause
  - a. Looks for the design cell in the use clause.
  - b. If not found, VCS generates an error message.

#### **RULE5: Cell liblist Clause**

1. Configuration cell liblist clause
  - a. Resolves the design cell using the configuration cell liblist clause, if the liblist argument is not empty.
  - b. Looks for the design cell in the parent cell library, if the liblist argument is empty.
  - c. Looks for the design cell in the libraries specified with -liblist at the command line.
  - d. If the cell was analyzed using -v or -y in the same vlogan invocation as the parent.
  - e. Looks for the library cell in the configuration cell liblist clause, if the liblist argument is empty.
  - f. Looks for the library cell in the libraries, if the liblist argument is not empty.
  - g. Looks for the libraries specified with -liblist at the command line.

## **RULE6: Configuration Inherited liblist Clause**

1. Configuration inherited liblist clause (derived from instance liblist or from cell liblist)
  - a. Looks for the design cell in the configuration liblist (instance or cell clause)
  - b. Looks for the design cell in the parent cell library, if the liblist argument is empty.
  - c. Looks for the libraries specified with -liblist at the command line.
  - d. If the cell was analyzed using -v or -y in the same vlogan invocation as the parent.
  - e. Looks for the library cell in the configuration cell liblist clause.
  - f. Looks for the library cell in the parent cell library, if the liblist argument is empty.
  - g. Looks for the libraries specified with -liblist at the command line.

## **RULE7: Configuration Default liblist Clause**

1. Configuration default liblist clause
  - a. Looks for the design cell in the configuration default liblist. For usage example, see "[Example-7.1](#)".
  - b. Looks for the design cell in the parent cell library, if there are no arguments in the configuration default liblist.

- c. Looks for the design cell definition in the libraries specified with `-liblist` at the command line. For usage example, see “[Example-7.2](#)”.
- d. If the cell was analyzed using `-v` or `-y` in the same vlogan invocation as the parent.
- e. Looks for the library cell definition in the configuration default liblist.
- f. Looks for the library cell definition in the parent cell library, if there are no arguments in the configuration default liblist.
- g. Looks for the library cell definition in the libraries specified with `-liblist` at the command line. For usage example, see “[Example-7.3](#)”.

## **RULE8: No Configuration Rule Applies**

- 1. No configuration rule applies
  - a. Looks for the design cell in the parent cell library. For usage example, see “[Example-8.1](#)”.
  - b. Looks for the libraries specified with `-liblist` at the command line. For usage example, see “[Example-8.2](#)”.
  - c. Looks for the library cell in the parent cell library.
  - d. Looks for the libraries specified with `-liblist` at the command line.

## **Library Search Order Rules for Verilog or SystemVerilog Designs Without a Configuration File**

### **-liblist**

Specify `-liblist` at the command line.

Search Order:

1. Look in the libraries specified at the command line.
2. If not found, VCS generates an error message.

For usage example, see "[Example-9](#)".

### **-liblist\_work**

This option tells VCS to first look in the parent `work` library when trying to resolve a design cell. That is, while resolving module instances, this option ensures to always start search for child modules from the parent work library before searching in the other analyzed libraries.

This elaboration option is relevant only for non-configuration flow (that is, when you do not use the Verilog configuration file).

Search Order:

1. Look for the design cell in the parent cell library.
2. Look for the libraries specified in `-liblist` at the command line, or look for the libraries specified in the `synopsys_sim.setup` file if `-liblist` is not passed.
3. Not applicable in the configuration flow. It does not apply in the configuration flow. It is ignored if top is configured.

4. In case of , `-liblist_work` applies only for Verilog modules, and does not have any impact on the VHDL module.

For usage example, see "[Example-10](#)".

### **-liblist\_work and -liblist**

`-liblist_work` takes precedence over `-liblist`.

### **-liblist\_nocelldiff**

If this option is used, VCS does not differentiate between design cells and library cells. By default, VCS prefers design cells over library cells while resolving instances. It follows the configuration rules as applied to the `-liblist` options. For usage example, see "[Example-11](#)".

## **Example Testcase Files**

Consider the following design files:

**Testcase:** `file1.v`

```
module a(in1, in2, out1);
    input in1, in2;
    output out1;
    initial
        $display ( "Instance %m Module a from file1.v %l");
        b b1(in1, in2, out1);
endmodule

module t1;

    reg r1, r2;
    wire w1;

    a a1(r1,r2,w1);

```

```
initial
$display ( "Instance %m Module t1 from file1.v %l");
endmodule
```

**Testcase:** *file2.v*

```
module a(in3, in4, out2);
input in3, in4;
output out2;

initial
$display ( "Instance %m Module a from file2.v %l");
b b1(in3,in4,out2);
endmodule

module t2;
reg r2, r4;
wire w2;

a a1(r2,r4,w2);

initial
$display ( "Instance %m Module t2 from file2.v %l");
endmodule
```

**Testcase:** *modb\_1.v*

```
module b(in1, in2, out1);
input in1, in2;
output out1;
initial
$display ( "Instance %m Module b from modb_1.v %l");
endmodule
```

**Testcase:** *modb\_2.v*

```
module b(in3, in4, out2);
input in3, in4;
output out2;
initial
```

```
$display ( "Instance %m Module b from modb_2.v %l");
endmodule
```

**Testcase:** *top.v*

```
module top;

    t1 t1_1();
    t2 t2_1();

    initial
        $display ( "Instance %m Module top from top.v %l");
endmodule
```

## Usage Examples for Library Search Order Rules for Verilog or SystemVerilog Designs

### Usage Examples for RULE1

This section describes the examples for sub-rules listed under “[RULE1:`uselib](#)” of the [“Library Search Order Rules for Verilog or SystemVerilog Designs”](#) section.

#### Example-1

Consider the following design files:

**Testcase:** *file.v*

```
module level1;
    level2 l2();
    initial $display("%l %m level1 (uselib)");
endmodule

module level2;
    initial $display("%l %m level2 (uselib)");
endmodule
```

**Testcase:** level2.v

```
module level2;
    initial $display("%l %m level2 (design)");
endmodule

module M;
    initial $display("%l %m M");
endmodule
```

**Testcase:** top.v

```
module top;
    `uselib file="file.v"
    level1 l1();
endmodule
```

**Configuration file:** cfg.v

```
config topcfg;
    design L1.top;
    instance top.l1.12 use L2.level2;
endconfig
```

**VCS commands:**

```
mkdir L1 L2
vlogan -sverilog level2.v -work L2
vlogan -sverilog top.v -work L1
vlogan -sverilog cfg.v -work L1
vcs L1.topcfg -diag libconfig -libmap_verbose
simv
```

**synopsys\_sim.setup file:**

```
WORK > DEFAULT
DEFAULT : ./work
L1 : ./lib1
L2 : ./lib2
```

```
L3 : ./lib3
```

## Description

Library search order rules to resolve the `top.11.12` instance are as follows:

1. The configuration file `cfg.v` specifies a configuration rule for the `top.11.12` instance (as the `top.11.12` instance uses `L2.level2`), which indicates to resolve the `12` definition from the `L2.level2` library.
2. As per the search rule, as the cell `level1` (instance `11`) is resolved from the `'uselib file in top.v`, instance `12` under `11` is also resolved from the `'uselib library` only.
3. Instance `12` cannot be resolved from the configuration rule as given in `cfg.v`.
4. Configuration verbose output shows that `12` is resolved from the `'uselib library`.

```
instance: top.11.12
  rule: `uselib
  module: L1.level2
```

## Example-2

Consider the following design files:

*Testcase: file.v*

```
module level1;
    level2 l2();
    initial $display("%l %m level1 (uselib)");
endmodule

module level2;
```

```
    initial $display("%l %m level2 (uselib)");
endmodule
```

**Testcase:** *level1.v*

```
module level1;
    level2 l2();
    initial $display("%l %m level1 (design)");
endmodule

module M;
    initial $display("%l %m M");
endmodule
```

**Testcase:** *top.v*

```
module top;
    `uselib file="file.v"
    level1 l1();
endmodule
```

**Configuration file:** *cfg.v*

```
config topcfg;
    instance top.l1 use L2.level1;
    instance top.l1.l2 use L2.M;
endconfig
```

**VCS commands:**

```
mkdir L1
vlogan -sverilog level1.v -work L2
vlogan -sverilog top.v -work L1
vlogan -sverilog topcfg.v -work L1
vcs L1.topcfg -diag libconfig -libmap_verbose
```

***synopsys\_sim.setup:***

```

WORK > DEFAULT
DEFAULT : ./work
L1 : ./lib1
L2 : ./lib2
L3 : ./lib3

```

## Description

Library search order rules to resolve the `top.11` instance are as follows:

1. As there is a configuration rule mentioned in the `cfg.v` file (The `top.11` instance uses `L2.level1`) to resolve the definition from `L2.level1`, instance `11` is resolved as per the configuration rule.
2. The `top.11.12` instance is also resolved as per the configuration rule and uses `L2.M`.

```

instance: top.11
rule: instance top.11 uses L2.level1;
module: L2.level1

instance: top.11.12
rule: instance top.11.12 uses L2.M;
module: L2.M

```

## Example-3

Consider the following design files:

*Testcase: file.v*

```

module level1;
    level2 l2();
    initial $display("%l %m level1 (uselib)");
endmodule

module level2;

```

```
    initial $display("%l %m level2 (uselib)");
endmodule
```

**Testcase:** *level2.v*

```
module level2;
    initial $display("%l %m level2 (design)");
endmodule

module M;
    initial $display("%l %m M");
endmodule
```

**Testcase:** *top.v*

```
module top;
    `uselib file="file.v"
    level1 l1();
endmodule
```

**Configuration file:** *cfg.v*

```
config topcfg;
    design L1.top;
    instance top.l1.l2 use L2.M;
endconfig
```

**VCS commands:**

```
mkdir L1 L2
vlogan -sverilog level2.v -work L2
vlogan -sverilog top.v -work L1
vlogan -sverilog cfg.v -work L1
vcs L1.topcfg -diag libconfig -libmap_verbose
```

***synopsys\_sim.setup file:***

```
WORK > DEFAULT
```

```
DEFAULT : ./work
L1  : ./lib1
L2  : ./lib2
L3  : ./lib3
```

## Description

Library search order rules to resolve the `top.11.12` instance are as follows:

1. The `level1` (instance `11`) cell is resolved from the ``uselib` file, but the `12` instance is not resolved during `vlogan` analysis.
2. Configuration rules are applied to resolve the `12` instance, and the instance is resolved from the `L2.M` library.

```
instance: top.11.12
rule: instance top.11.12 use L2.M;
module: L2.M
```

## Example-4

Consider the following design files:

**Testcase:** `file.v`

```
module level1;
    level2 l2();
    initial $display("%l %m level1 (uselib)");
endmodule
```

**Testcase:** `level1.v`

```
module level1;
    initial $display("%l %m level1 (design)");
endmodule

module level2;
    initial $display("%l %m level2 (design)");
```

```

endmodule
module M;
    initial $display("%l %m M");
endmodule

```

**Testcase:** *top.v*

```

module top;
    `uselib file="file.v"
    level1 l1();
endmodule

```

**Configuration file:** *cfg.v*

```

config topcfg;
    design L1.top;
endconfig

```

**VCS commands:**

```

mkdir L1 L2
vlogan -sverilog level2.v -work L2
vlogan -sverilog top.v level2.v -work L1
vlogan -sverilog cfg.v -work L1
vcs L1.topcfg -diag libconfig -libmap_verbose

```

***synopsys\_sim.setup:***

```

WORK > DEFAULT
DEFAULT : ./work
L1 : ./lib1
L2 : ./lib2
L3 : ./lib3

```

## Description

To resolve `top.l1`, VCS does not look at ``uselib` or `-v/-y` as the design cell `level2` for an instance `l2` is found during `vlogan` itself.

```
instance: top.l1
rule: parent cell's library
module: L1.level1
```

## Usage Examples for RULE2

This section describes the examples for sub-rules listed under “[RULE2: Instance use Clause](#)” of the “[Library Search Order Rules for Verilog or SystemVerilog Designs](#)” section.

Consider the design files and hierarchy mentioned in the “[Example Testcase Files](#)” section.

### Example-5

Consider the following configuration file and VCS commands:

*Configuration file:* `cfg.v`

```
config topcfg;
    design lib3.top;
    instance top.t2_1.a1 use LIB2.a
endconfig
```

*VCS commands:*

```
vlogan modb_1.v -work lib_b1 +v2k -q
vlogan modb_2.v -work lib_b2 +v2k -q
vlogan file1.v -work lib1 +v2k -q
vlogan file2.v -work lib2 +v2k -q
vlogan top.v -work lib3 +v2k -q
vlogan +v2k -work work cfg.v -q
vcs work.cfg -liblist lib1+lib2+lib3+lib_b1+lib_b2 -diag
libconfig
```

### Description

Library search order rules to resolve the `top.t2_1.a` instance are as follows:

1. VCS searches for a module definition of `a` in `LIB2.a` as provided by the configuration rules.
2. VCS search completes once it finds the definition in `LIB2.a`.

```
instance: top.t2_1.a1
rule: instance top.t2_1.a1 use LIB2.a;
module: LIB2.a
```

## Usage Examples for RULE3

This section describes the examples for sub-rules listed under “[RULE3: Instance liblist Clause](#)” of the “[Library Search Order Rules for Verilog or SystemVerilog Designs](#)” section.

### Example-6

Consider the design files and hierarchy mentioned in the “[Example Testcase Files](#)” section.

*Configuration file:* `cfg.v`

```
config cfg;
  design lib3.top;
  instance top.t2_1.a1 liblist
    instance top.t1_1.a1 liblist lib1
endconfig
```

*VCS commands:*

```
vlogan +v2k modb_2.v -work lib_b2
vlogan +v2k modb_1.v -work lib_b1
vlogan +v2k file2.v -work lib2
vlogan +v2k file2.v -work lib3
vlogan +v2k file1.v -work lib1
```

```
vlogan +v2k -work lib3 top.v
vlogan +v2k cfg.v
vcs work.cfg -liblist lib2+lib1+lib3+lib_b1+lib_b2 -diag
libconfig -libmap_verbose
simv
```

## Example-6.1

VCS resolves the design cell using the configuration instance liblist clause if liblist is not empty.

Consider the design files, the configuration file, and VCS commands mentioned in “[Example-6](#)”.

### Description

Library search order rules to resolve the `top.t1_1.a1` instance are as follows:

1. Search for the module definition of `a` in liblist `lib1` as provided by the configuration rule in `cfg.v` (instance `top.t1_1.a1` liblist `lib1`)
2. Search completes if the definition is found in `lib1`.
3. Configuration verbose output:

```
instance: top.t1_1.a1
rule: instance top.t1_1.a1 liblist lib1;
module: LIB1.a
```

## Example-6.2

VCS looks for the design cell in parent cell library, if the liblist argument is empty.

Consider the design files, the configuration file, and VCS commands mentioned in “[Example-6](#)”.

## Description

Library search order rules to resolve the `top.t2_1.a1` instance are as follows:

1. The `liblist` argument in the configuration rule is empty (`instance top.t2_1.a1 liblist`).
2. If the argument after `liblist` is empty, then VCS looks for the module definition in the parent cell library.
3. VCS looks for the module definition of `a` in the parent cell library which is `t2_1`.
4. Parent cell library of `t2_1` is in `LIB3`.
5. VCS tries to resolve the module definition of `a` also in `LIB3`.
6. Search completes if the definition of `a` is found.
7. Configuration verbose output

```
instance: top.t2_1.a1
rule: instance top.t2_1.a1 liblist;
module: LIB3.a
```

## Example-6.3

VCS looks for libraries specified in `-liblist` at the command line.

Consider the design files, the configuration file, and VCS commands mentioned in “[Example-6](#)”.

## Description

1. For the `top.t1_1.a1.b1` instance, there is no specific rule applied in the configuration file.

2. VCS looks for the module definition `a` first in the parent cell library, if the default configuration is not present in the configuration file. In this example, the parent cell library of the `a` module is `lib1`. VCS does not find the `b` module definition in `lib1`.
3. VCS then looks for libraries specified at `-liblist` at the command line in the order provided.
4. VCS searches in `LIB2`. If not found in `LIB2`, VCS searches in `LIB1`. Then, VCS searches in `LIB3`. If it finds the definition, search gets complete.
5. Configuration verbose output:

```
instance: top.t1_1.a1.b1
rule: default library search order
module: LIB_B1.b
```

## Usage Examples for RULE7

This section describes the examples for sub-rules listed under “[RULE7: Configuration Default liblist Clause](#)” of the “[Library Search Order Rules for Verilog or SystemVerilog Designs](#)” section.

### Example-7

Consider the design files and hierarchy mentioned in the “[Example Testcase Files](#)” section.

*Configuration file:* `cfg.v`

```
config cfg;
  design lib3.top;
  instance top.t2_1.a1 liblist
    default liblist lib_b2 lib2;
endconfig
```

*VCS commands:*

```

vlogan +v2k modb_2.v -work lib_b2
vlogan +v2k modb_1.v -work lib_b1
vlogan +v2k file2.v -work lib2
vlogan +v2k file2.v -work lib3
vlogan +v2k file1.v -work lib1
vlogan +v2k -work lib3 top.v
vlogan +v2k cfg.v
vcs work.cfg -liblist lib2+lib1+lib3+lib_b1+lib_b2 -diag
libconfig -libmap_verbose

```

## **Example-7.1**

VCS looks for the design cell in the configuration default liblist.

Consider the design files, the configuration file, and VCS commands mentioned in “[Example-7](#)”.

### **Description**

Library search order rules to resolve the `top.t1_1.a1.b1` instance are as follows:

1. VCS looks for default libraries specified in the default liblist in the configuration file (`default liblist lib_b2 lib1 lib2`)
2. VCS finds the definition in `LIB_B2`, and the search gets completed.

```

instance: top.t1_1.a1.b1
rule: default liblist lib_b2 lib2;
module: LIB_B2.b

```

## **Example-7.2**

VCS looks for the design cell definition in the libraries specified with `-liblist` at the command line.

Consider the design files, the configuration file, and VCS commands mentioned in “[Example-7](#)”.

## Description

Library search order rules to resolve the `top.t1_1` instance are as follows:

1. VCS searches in the default configuration `liblist` provided in the configuration file, `cfg.v` (default `liblist lib_b2 lib2`).
2. VCS could not find in any of the default libraries `lib_b2` and `lib2` definition of `t1_1`.
3. VCS looks for the libraries specified with `-liblist` at the command line (`lib2+lib1+lib3+lib_b1+lib_b2`).
4. VCS starts looking for the `t1_1` definition and finds in `LIB1`.
5. VCS search completes.

```
instance: top.t1_1
rule: default library search order
module: LIB1.t1
```

## Example-7.3

VCS looks for the library cell definition in the libraries specified in the `-liblist` at the command line.

Consider the design files mentioned in “[Example-7](#)”.

*Configuration file:* `cfg.v`

```
config cfg;
    design lib3.top;
    instance top.t2_1.a1 liblist;
```

```
instance top.t1_1.a1 liblist lib3
default liblist lib_b2 lib1 lib2;
endconfig
```

### VCS commands:

```
vlogan +v2k file2.v -work lib3 -v modb_2.v
vlogan +v2k file2.v -work lib3
vlogan +v2k file1.v -v modb_1.v -work lib_b1
vlogan +v2k file1.v -work lib1
vlogan +v2k -work lib3 top.v
vlogan +v2k cfg.v
vcs work.cfg -liblist lib1+lib3+lib_b1+lib_b2 -diag
libconfig -libmap_verbose
```

### Description

Library search order rules to resolve the `top.t2_1.a1.b1` instance are as follows:

1. VCS searches in the default library list `lib_b2`, `lib`, and `lib2`. However, it could not find the definition.
2. VCS then looks at the libraries specified in `-liblist` at the command line in the order provided `lib1+lib3+lib_b1+lib_b2`.
3. VCS finds the definition in `LIB3` and search gets completed.

```
instance: top.t2_1.a1.b1
rule: default library search order
module: LIB3.b
```

### Usage Examples for RULE8

This section describes the examples for sub-rules listed under “[RULE8: No Configuration Rule Applies](#)” of the “[Library Search Order Rules for Verilog or SystemVerilog Designs](#)” section.

## **Example-8**

Consider the design files and hierarchy mentioned in the “[Example Testcase Files](#)” section.

*Configuration file:* `cfg.v`

```
config cfg;
    design lib3.top;
endconfig
```

*VCS commands:*

```
vlogan +v2k modb_2.v -work lib_b2
vlogan +v2k modb_1.v -work lib_b1
vlogan +v2k file2.v -work lib2
vlogan +v2k file2.v -work lib3
vlogan +v2k file1.v -work lib1
vlogan +v2k -work lib3 top.v
vlogan +v2k cfg.v
vcs work.cfg -liblist lib2+lib1+lib3+lib_b1+lib_b2 -diag
libconfig -libmap_verbose
```

### **Example-8.1**

VCS looks for the design cell in the parent cell library.

Consider the design files, the configuration file, and VCS commands mentioned in “[Example-8](#)”.

### **Description**

Library search order rules to resolve the `top.t2_1.a1` instance are as follows:

1. No configuration rule applies for the `top.t2_1.a1` instance, and there is no default liblist configuration present. Therefore, VCS looks at the parent cell library.
2. Parent cell library for the parent module `t2` is `LIB3`.
3. VCS finds the definition of `a` in `LIB3` and the search gets completed.

```
instance: top.t2_1.a1
rule: parent cell's library
module: LIB3.a
```

### **Example-8.2**

VCS looks for the libraries specified with `-liblist` at the command line.

Consider the design files, the configuration file, and VCS commands mentioned in “[Example-8](#)”.

### **Description**

Library search order rules to resolve the `top.t2_1.a1.b1` instance are as follows:

1. No configuration rules applies for instance `top.t2_1.a1.b1`.
2. VCS tries to look into the parent cell library but could not find the definition.
3. VCS looks for the libraries mentioned with `-liblist` at the command line.
4. VCS searches in the libraries specified in the order specified at the command line.

5. VCS finds the definition in LIB\_B1, and then the search gets completed.

```
instance: top.t2_1.a1.b1
rule: default library search order
module: LIB_B1.b
```

## Usage Examples for Library Search Order Rules for Verilog or SystemVerilog Designs Without a Configuration File

This section describes the examples for library search order rules for Verilog or SystemVerilog designs without a configuration file.

### Example-9

Consider the design files and hierarchy mentioned in the “[Example Testcase Files](#)” section.

*VCS commands:*

```
vlogan +v2k modb_2.v -work lib_b2
vlogan +v2k modb_1.v -work lib_b1
vlogan +v2k file2.v -work lib2
vlogan +v2k file1.v -work lib1
vlogan +v2k -work lib3 top.v
vcs top -liblist lib1+lib2+lib3+lib_b1+lib_b2 -diag
libconfig -libmap_verbose -liblist_nocelldiff
```

### Description

Library search order rules to resolve the `top.t2_1.a1.b1` instance are as follows:

1. Looks for the definition of module b.

2. Starts scanning the library files provided with `-liblist` from left to right.
3. Looks in library `lib1`. If it does not find in `lib1`, it looks in library `lib2`. If it does not find in `lib2`, it looks in library `lib3`.
4. Looks in library `lib_b1`, finds the definition of `b`. The search gets completed.

```
instance: top.t2_1.a1.b1
rule: default library search order
module: LIB_B1.b
```

## Example-10

Consider the design files and hierarchy mentioned in the “[Example Testcase Files](#)” section.

*VCS commands:*

```
vlogan +v2k modb_2.v -work lib_b2
vlogan +v2k modb_2.v -work lib3
vlogan +v2k modb_1.v -work lib_b1
vlogan +v2k modb_1.v -work lib3
vlogan +v2k file2.v -work lib2
vlogan +v2k file2.v -work lib3
vlogan +v2k file1.v -work lib1
vlogan +v2k file1.v -work lib3
vlogan +v2k -work lib3 top.v
vcs top -liblist_work -diag libconfig -libmap_verbose
```

## Description

Library search order rules to resolve the `top.t2_1.a1.b1` instance are as follows:

1. Looks for the design cell `b`.

2. As `-liblist_work` is present at elaboration time, VCS first tries to search the parent work library.
3. Parent module of `b` is module `a`, and is available in the `LIB3` library.
4. VCS tries to search the definition of `b` in `LIB3`.
5. VCS finds the definition of `b` in `LIB3`, and the search gets completed.

```
instance: top.t2_1.a1.b1
rule: parent cell's library
module: LIB3.b
```

### **Example-11**

Consider the design files and hierarchy mentioned in the “[Example Testcase Files](#)” section.

*VCS commands:*

```
vlogan +v2k modb_2.v -work lib_b2
vlogan +v2k modb_1.v -work lib_b1
vlogan +v2k file2.v -v modb_2.v -work lib2
vlogan +v2k file2.v -work lib3
vlogan +v2k file1.v -work lib1
vlogan +v2k -work lib3 top.v
vcs top -liblist lib1+lib2+lib3+lib_b1+lib_b2 -diag
libconfig -libmap_verbose -liblist_nocelldiff
```

### **Description**

Library search order rules to resolve the `top.t2_1.a1.b1` instance are as follows:

1. Looks for the definition of module `b` in library `LIB2` (library cell) and in `LIB3` (design cell).

2. With `-liblist_nocelldiff`, definition of `b` is resolved from library `LIB2` as it is available in `LIB2`.
3. `LIB2` comes first in the order of `-liblist` (`lib1+lib2+lib3+lib_b1+lib_b2`) provided at elaboration time.

```
instance: top.t2_1.a1.b1
rule: default library search order
module: LIB2.b
```

4. With no presence of `-liblist_nocelldiff` in the same example, definition of `b` is resolved from the `LIB_B1`(design cell) library as it is available in `LIB_B1`.

```
instance: top.t2_1.a1.b1
rule: default library search order
module: LIB_B1.b
```

## Lint Warning Message for Missing ‘endcelldefine’

You can tell VCS to display a lint warning message if your Verilog or SystemVerilog code contains a `'celldefine` compiler directive without a corresponding `'endcelldefine` compiler directive and vice versa.

You enable this warning message with the `+lint=CDUB` VCS compile-time option `vlogan` command-line option. The `CDUB` argument stands for “Compiler Directives Unbalanced.”

The examples in this section show the warning message and the source code that results in its display.

### *Example 4-9 Source Code With Missing ‘endcelldefine’*

```
'celldefine
module mod;
```

```
endmodule
```

In this example, there is no corresponding `endcelldefine compiler directive.

In VCS two-step flow, if you execute the following vcs command:

```
vcs exp1.v +lint=CDUB
```

VCS displays the following Lint warning message:

```
Lint-[CDUB] Compiler directive unbalanced  
exp1.v, 1  
Unbalanced compiler directive is detected : `celldefine  
has no matching `endcelldefine.  
Please make sure that all directives are balanced.
```

In VCS, vlogan also displays this lint warning message when you execute the following command:

```
vlogan exp1.v +lint=CDUB
```

The source code in [Example4-10](#) does not display this warning message when you include the +lint=CDUB.

#### *Example 4-10 Source Code With 'celldefine and 'endcelldefine*

```
`celldefine  
module mod;  
endmodule  
`endcelldefine
```

It does not display the warning message because there is the `endcelldefine compiler directive after the `celldefine compiler directive in the source code.

Instead of the `endcelldefine compiler directive, you can substitute the `resetall compiler directive, as shown in [Example4-11](#).

*Example 4-11 Source Code With ‘celldefine and ‘resetall*

```
`celldefine  
module mod;  
endmodule  
'resetall
```

The source code in both [Example4-10](#) and [Example4-11](#) does not result in the warning message when you include the +lint=CDUB option.

Also with the +lint=CDUB option, if your source code contains the `endcelldefine compiler directive without the preceding and corresponding `celldefine compiler directive, you see a similar warning message.

*Example 4-12 ‘endcelldefine Without a Preceding and Corresponding ‘celldefine*

```
module mod;  
endmodule  
'endcelldefine
```

With the +lint=CDUB option, this source code results in the following lint warning message:

```
Lint-[CDUB] Compiler directive unbalanced  
exp6.v, 3  
Unbalanced compiler directive is detected : `endcelldefine  
has no matching `celldefine.  
Please make sure that all directives are balanced.
```

With the `+lint=CDUB` option, it is not just that the number of `'endcelldefine` compiler directives must be equal to the number of `'celldefine` compiler directives. The `'endcelldefine` compiler directive must follow the `'celldefine` compiler directive before there is another `'celldefine` compiler directive.

*Example 4-13 Equal Number of 'celldefine and 'endcelldefine, But Not in the Required Sequence*

```
'celldefine  \\ line 1
module mod;
endmodule

`celldefine
module schmodule;
endmodule

`endcelldefine

`endcelldefine \\ line 11
```

In Example4-13, the number of `'celldefine` compiler directives matches the number of `'endcelldefine` compiler directives, but they are not in a corresponding sequence, which results in the following lint warning messages:

```
Lint-[CDUB] Compiler directive unbalanced
exp5.v, 1
    Unbalanced compiler directive is detected : `celldefine
    has no matching `endcelldefine.
    Please make sure that all directives are balanced.
```

```
Lint-[CDUB] Compiler directive unbalanced
exp5.v, 11
    Unbalanced compiler directive is detected : `endcelldefine
    has no matching `celldefine.
    Please make sure that all directives are balanced.
```

## Limitation

The `'celldefine/'endcelldefine` compiler directives must be matched serially. Recursive `'celldefine/'endcelldefine` directives are not supported with the `+lint=CDUB` option and keyword argument, for example:

### *Example 4-14 Recursive 'celldefine/'endcelldefine Compiler Directives*

```
'celldefine  
'celldefine  
module dev (...,...);  
'celldefine  
'celldefine  
module dev (...,...);  
...  
endmodule  
'endcelldefine  
'endcelldefine
```

[Example4-14](#) shows redundant and unnecessary `'celldefine` and `'endcelldefine` compiler directives, but does not prevent compilation. The `+lint=CDUB` option and keyword argument triggers the unbalanced message of Lint compiler directives when VCS reads another `'celldefine` directive before reading an `'endcelldefine` directive,

---

## Error/Warning/Lint Message Control

You can control error, warning, and lint messages in the following two ways:

- For `-error`, `-suppress`, `+lint`, and `+warn` compile options, see [“Controlling Error/Warning/Lint Messages Using Compile-Time Options” on page 112](#).

- With a configuration file that you specify with the following compile-time option:

`-msg_config=message_configuration_file_name`

Using a configuration file, you can control lint, warning, and error messages that VCS displays according to the following:

- by source file name
- by module name
- by design subhierarchy

See “[Controlling Error/Warning/Lint Messages Using a Configuration File](#)”.

## Controlling Error/Warning/Lint Messages Using Compile-Time Options

The `-error`, `-suppress`, `+lint`, and `+warn` options control error and warning messages. With them, you can:

- Disable the display of any lint, warning, or error messages.
- Disable the display of specific messages.
- Limit the display of specific messages to a maximum number that you specify.

To control the display of specific messages, you need their message IDs. A message ID is the character string in a message between the square brackets [ ]. In [Figure 4-3](#), the message ID is MFACF.

Note:

The `-error` option is also a runtime option.

*Figure 4-3 Message IDs*

message ID  
Warning-[MFACF] Missing flag argument  
Argument for flag 'verboseLevel' is missing in config statement, it will be ignored.  
Config file : error\_id0\_id1.cfg, starting at line 4.

The new compile-time options for controlling messages and their syntax are as follows:

```
-error=[no]message_ID[:max_number],...|none|all  
-error=all,noWarn_ID|noLint_ID  
+warn=[no]message_ID[:max_number],...|none|all  
+lint=[no]message_ID[:max_number],...|none|all  
-suppress[=message_ID,...]
```

Note:

The `-error` option is also a runtime option. However, only the following feature is supported at runtime:

```
-error=[no]message_ID[:max_number],...
```

These compile-time options and their arguments are described in the following sections:

- “Controlling Error Messages”
- “Controlling Lint Messages”

- “[Suppressing Lint, Warning, and Error Messages](#)”
- “[Error Conditions and Messages That Cannot Be Disabled](#)”
- “[Using Message Control Options Together](#)”

## **Controlling Error Messages**

You can control error messages with the `-error` option in the following ways:

- Limit the number of occurrences of an error message to a number you specify. For this, specify the message ID as an argument to the `-error` option along with the specified maximum number of occurrences.
- Disable the display of all error messages which are downgradable with the `none` argument.
- Enables the display of all errors/warnings/lint messages with the `all` argument to the `-error` option.

## **Upgrading Lint and Warning Messages to Error Messages**

If you enter the message ID for a warning or lint message as an argument to the `-error` option, VCS upgrades the condition causing the warning or lint message to an error condition and an error message.

## **Controlling Warning Messages**

Like error messages, you can control warning messages with the `+warn` option in the following ways:

- Limit the number of occurrences of a warning message to a number you specify. For this, specify the message ID as an argument to the `+warn` option along with the specified maximum number of occurrences.
- Disable the display of a particular warning message by entering the keyword `no` as an argument and appending to this keyword the message ID, for example:

`+warn=noTFIPC`

This option disables the display of the error message with the TFIPC message ID.

**Important:**

- Do not enter a maximum number of occurrences, even if 0, if also appending the `no` keyword to the message ID.
- Disable the display of all warning messages with the `none` argument to the `+warn` option.
- Enable the display of all warning messages with the `all` argument to the `+warn` option.
- Controls the display of all notes. For example,

`+warn=noFCICIO`

This option suppresses the display of the following note:

Note-[FCICIO] Instance coverage is ON

## Upgrading Lint Messages to Warning Messages

**Important:**

- All lint/warning messages are suppressible. But only some of the error messages can be downgraded or suppressed.

- You cannot downgrade all error conditions and messages to a warning condition and message. Entering a message ID for an error message that cannot be downgraded as an argument to the `+warn` option results in VCS ignoring the message ID and displaying a warning message similar to the following:

```
Warning-[CSMC] Cannot set message count
Failed to set display count for message id 'TFAFTC'
because cannot set count
for non-warning ID in '+warn' switch.
Specified count is ignored.
```

For an example of this warning see “[Example 4: An Error Message That Cannot Be Controlled](#)”.

This warning message is in response to the `+warn=TFAFTC:2` option, when `TFAFTC` is the ID for the following error message:

```
Error-[TFAFTC] Too few arguments to function/task call
tfatc_err.v, 9
"wrFlld4(.bus(1));"
The above function/task call is not done with sufficient
arguments.
```

## Controlling Lint Messages

Like error and warning messages, you can control lint messages with the `+lint` option in the following ways:

- You can limit the number of occurrences of a lint message to a number you specify. For this, specify the message ID as an argument to the `+lint` option along with the specified maximum number of occurrences.

You can enter a maximum of 0 to disable any display of the message specified by the message ID, see “[Example 2: Reducing the Number of lint Messages](#)”.

**Important:**

- Do not enter a maximum number of occurrences, even if 0, if also appending the `no` keyword to the message ID.
- Disable the display of all lint messages with the `none` argument to the `+lint` option.
- Enable the display of all lint messages with the `all` argument to the `+lint` option.

**Important:**

You cannot downgrade an error or warning condition and message to a lint condition and message.

## Suppressing Lint, Warning, and Error Messages

The `-suppress` option suppresses lint, warning, and error messages. The `-suppress` option with no argument should suppress all warnings/lint and downgradable error messages

If you enter a message ID argument, and the message is downgradable, VCS does not display that message. You can enter the ID for any lint, warning, or downgradable error message.

The `-suppress` option gives you a message control option that takes a higher precedence when you enter more than one of these options: `-error`, `+warn`, or `+lint`. For more details, see “[Using Message Control Options Together](#)”.

**Note:**

The `-error` option is also a runtime option.

## Error Conditions and Messages That Cannot Be Disabled

Some error conditions always terminate compilation without creating an executable and cannot be controlled or suppressed by the `-error` or `-suppress` options.

- Syntax errors
- Fatal error messages, those from error conditions that immediately halt compilation

## Using Message Control Options Together

If you are entering more than one of these message control options, you will need to know their precedence when used together. The order of precedence from highest to lowest is as follows:

1. The `-suppress` option with no arguments, suppresses all possible messages and cannot be overridden by another message control option.
2. The `none` argument has a higher precedence than specifying `all` or a message ID.
3. The order on the `vcs` command line

The following options and arguments have the same intrinsic precedence:

<code>-suppress=messageID</code>	
<code>-error=messageID:max</code>	<code>-error=all</code>
<code>+warn=messageID:max</code>	<code>+warn=all</code>
<code>+lint=messageID:max</code>	<code>+lint=all</code>

Because they have equal intrinsic precedence, the order on the `vcs` command line determines relative precedence. The first of these options on the command line has the least precedence and the last of these has the most.

## Message Control Examples

The following examples show how to use these options:

### Example 1: Reducing the Number of Warning Messages

If you have small SystemVerilog source file named as `diff_clk_wosvaext.sv` with the following content:

```
1 module top #(Pa = 1);
2 bit a, c, clk;
3 wand b1;
4 wand c1;
5
6 clocking cb2 @ (posedge clk);
7 endclocking
8
9 sequence S2();
10 @ (cb2)
$past ($past (a,, $stable ($isunknown(1'bx), @ (negedge
clk)), @ (posedge clk)), , $sampled (a), @ (negedge clk));
11 endsequence
12
13 property P1();
14 @ (cb2, posedge clk iff ($stable (b1, @ (posedge clk)))
$stable ($past (b1,, , @ (posedge clk)), @ (negedge clk)));
15 endproperty
16
17 A1: assume property (@ (S2)      S2 );
18 A2: assume property (@ (S2) P1());
19 A3: assume property (@ (cb2) disable iff ($stable (c1)) P1);
20 A4: assume property (@ (cb2) disable
iff ($sampled ($past (c1,, , @ (clk)))) first_match (S2));
21
22 sequence S3();
```

```

23 @ (cb2) S2() ##1 @(negedge clk) $stable(b1 || $sampled(c1),
@(posedge clk));
24 endsequence
25
26 A5: cover property ( @(S2) S3);
27 initial begin
28 a = 1;
29 repeat (20)
30 #5 clk = !clk;
31 end
32 endmodule

```

If you compile the above system Verilog file with the following command,

```
vcs -sverilog diff_clk_wosvaext.sv
```

**VCS displays the following warning messages:**

```

Warning-[SVA-LCDNAWPSC] Lead and property/sequence clocks
differ
diff_clk_wosvaext.sv, 17
top
    Leading clock of expression does not agree with property/
    sequence clock.
    Leading clock will be applied.
    property/sequence clock: S2
    leading clock: posedge clk

```

```

Warning-[SVA-LCDNAWPSC] Lead and property/sequence clocks
differ
diff_clk_wosvaext.sv, 18
top
    Leading clock of expression does not agree with property/
    sequence clock.
    Leading clock will be applied.
    property/sequence clock: S2
    leading clock: top.cb2,posedge clk iff $stable(b1,
@(posedge clk))

```

```

Warning-[SVA-LCDNAWPSC] Lead and property/sequence clocks
differ

```

```

diff_clk_wosvaext.sv, 19
top
    Leading clock of expression does not agree with property/
sequence clock.
    Leading clock will be applied.
    property/sequence clock: posedge clk
    leading clock: top.cb2,posedge clk iff $stable(b1,
@(posedge clk))

Warning- [SVA-LCDNAWPSC] Lead and property/sequence clocks
differ
diff_clk_wosvaext.sv, 26
top
    Leading clock of expression does not agree with property/
sequence clock.
    Leading clock will be applied.
    property/sequence clock: S2
    leading clock: posedge clk

```

VCS displays the same warning four times, if you want to control the number of warning messages, you can use the compile-time option  
`+warn=warn_ID:n...`

For example,

```
vcs -sverilog +warn=SVA-LCDNAWPSC:1 diff_clk_wosvaext.sv
```

VCS limits the warning messages to one.

```

Warning- [SVA-LCDNAWPSC] Lead and property/sequence clocks
differ
diff_clk_wosvaext.sv, 17
top
    Leading clock of expression does not agree with property/
sequence clock.
    Leading clock will be applied.
    property/sequence clock: S2
    leading clock: posedge clk

```

## Example 2: Reducing the Number of lint Messages

If you have small SystemVerilog source file named as `top.sv` with the following content:

```
1 `celldefine
2 module sub;
3 endmodule
4
5 `celldefine
6 module sub1;
7 endmodule
8
9 `celldefine
10 module top;
11 sub inst();
12 sub1 inst1();
13 endmodule
```

By default, all lint messages are disabled if you want to enable the lint message, you need to use the `+lint=lint_ID` compile-time option. For example:

```
vcs -sverilog +lint=CDUB top.sv
```

VCS displays the following lint messages during compilation:

```
Lint-[CDUB] Compiler directive unbalanced
top.sv, 1
    Unbalanced compiler directive is detected : `celldefine
has no matching
    `endcelldefine.
    Please make sure that all directives are balanced.
```

```
Lint-[CDUB] Compiler directive unbalanced
top.sv, 5
    Unbalanced compiler directive is detected : `celldefine
has no matching
    `endcelldefine.
    Please make sure that all directives are balanced.
```

```

Lint-[CDUB] Compiler directive unbalanced
top.sv, 9
    Unbalanced compiler directive is detected : `celldefine
has no matching
`endcelldefine.
    Please make sure that all directives are balanced.

```

If you want to control the number of lint messages printed in the compile time, you can use `+lint=lint_ID:n...`. For example:

```
vcs -sverilog +lint=CDUB:1 top.sv
```

Now, VCS controls the number of lint messages printed to one:

```

Lint-[CDUB] Compiler directive unbalanced
top.sv, 1
    Unbalanced compiler directive is detected : `celldefine
has no matching
`endcelldefine.
    Please make sure that all directives are balanced

```

### **Example 3: Upgrading Multiple Warnings to One Error**

Consider a Verilog file named `tfpic.v` with the following contents:

```

module top();
wire a,b,c;
child child_position_instance(a,b);
child child_name_instance(.b(b));
endmodule

module child( input a, input b, input c);
endmodule

```

The module `child` has three input ports, but the module instantiation statements have only two or one port connection.

If you compile the `vcs tfpic.v` source file without message control, VCS displays the following during compilation:

```
Warning-[TFIPC] Too few instance port connections
  The following instance has fewer port connections than the
  module definition
    "tfipc.v", 3: child child_position_instance(a, b);

Warning-[TFIPC] Too few instance port connections
  The following instance has fewer port connections than the
  module definition
    "tfipc.v", 4: child child_name_instance( .b (b));

Warning-[TFIPC] Too few instance port connections
  The following instance has fewer port connections than the
  module definition
    "tfipc.v", 4: child child_name_instance( .b (b));
```

If you recompile specifying that message ID, TFIPC is upgraded to an error, and set to display this error message only once:

```
vcs tfpic.v -error=TFIPC:1
```

VCS displays the following error message only once:

```
Error-[TFIPC] Too few instance port connections
  The following instance has fewer port connections than the
  module definition
    "tfipc.v", 3: child child_position_instance(a, b);

1 error
```

#### **Example 4: An Error Message That Cannot Be Controlled**

Consider a Verilog file named `tfatf_err.v` with the following content:

```
module top;
  task wrFld4(input string fldName, input int bus = 0, input
  string fldName2);
```

```

        $display("In wrFld4");
endtask
task wrFld4_2(input int bus = 0,input string fldName);
    $display("In wrFld4");
endtask
initial begin
    wrFld4(.bus(1));                      // this is line 9
    wrFld4(,1);                           //          10
    wrFld4_2(.bus(1));                   //          11
end
endmodule

```

The `wrFld4` task has three input ports and the `wrFld4_2` task has two input ports. However, the task enabling statements for them have only one connection.

VCS displays the following during compilation:

```

Error-[TFAFTC] Too few arguments to function/task call
tfatc_err.v, 9
"wrFld4(.bus(1));"
    The above function/task call is not done with sufficient
    arguments.

Error-[TFAFTC] Too few arguments to function/task call
tfatc_err.v, 10
"wrFld4(, 1);"
    The above function/task call is not done with sufficient
    arguments.

Error-[TFAFTC] Too few arguments to function/task call
tfatc_err.v, 10
top, "wrFld4(, 1);"
    The above function/task call is not done with sufficient
    arguments.

Error-[TFAFTC] Too few arguments to function/task call
tfatc_err.v, 11
top, "wrFld4_2(1);"
    The above function/task call is not done with sufficient
    arguments.

```

The error message with the ID TFAFTC is displayed four times. If you recompile while specifying that this error message gets displayed only once:

```
vcs tfatc_err.v -sverilog -error=TFAFTC:1
```

VCS displays the following:

```
Warning-[CSMC] Cannot set message count
Failed to set display count for message id 'TFAFTC' because
it cannot be
suppressed.
Specified count is ignored.

Parsing design file 'tfatc_err.v'

Error-[TFAFTC] Too few arguments to function/task call
tfatc_err.v, 9
"wrFld4(.bus(1));"
The above function/task call is not done with sufficient
arguments.

Error-[TFAFTC] Too few arguments to function/task call
tfatc_err.v, 10
"wrFld4(, 1);"
The above function/task call is not done with sufficient
arguments.

Error-[TFAFTC] Too few arguments to function/task call
tfatc_err.v, 10
top, "wrFld4(, 1);"
The above function/task call is not done with sufficient
arguments.

Error-[TFAFTC] Too few arguments to function/task call
tfatc_err.v, 11
top, "wrFld4_2(1);"
The above function/task call is not done with sufficient
arguments.
```

```
1 warning  
4 errors
```

None of the error messages are disabled and there is a warning saying that VCS cannot limit the display of the message.

### **Example 5: Syntax Using the -suppress Option**

Consider a SystemVerilog file `example.sv` with the following content:

```
1 module top;  
2 wire [5:0]data;  
3 longint result,result1,result2,result3,result4;  
4 assign data = 6'h2345;  
5 initial  
6 begin  
7 result = $clog2(4294967296);      //2 ** 32  
8 result4 = $clog2(2147483648);      //2 ** 31  
9 result3 = $clog2(1073741824);      //2 ** 30  
10 result1=2**16;  
11 result2=result1*result1;  
12 $display("clog: %0d result2 %0d \n",result,result2);  
13 $display("clog3: %0d \n",result3);  
14 $display("clog43: %0d \n",result4);  
15 end  
16 endmodule
```

If you compile this file as follows:

```
vcs -sverilog exmaple.sv
```

VCS displays the following warning messages:

```
Warning-[TMBIN] Too many bits in Based Number  
example.sv, 4  
The specified width is '6' bits, actually got '16' bits.  
The offending number is : '2345'.
```

```
Warning-[DCTL] Decimal constant too large
example.sv, 7
    Decimal constant is too large to be handled in compilation.
        Absolute value 4294967296 should be smaller than
2147483648.
```

```
Warning-[DCTL] Decimal constant too large
example.sv, 8
    Decimal constant is too large to be handled in compilation.
        Absolute value 2147483648 should be smaller than
2147483648.
```

If you are using the `-suppress` option with the command line all warning messages are suppressed.

For example, if you use the following command:

```
vcs -sverilog -suppress example.sv
```

The `-suppress` option suppresses all warning/lint/downgradable error messages.

## Controlling Error/Warning/Lint Messages Using a Configuration File

Using a configuration file, you can control lint, warning, and error messages that VCS displays according to the following:

- Source file name
- Module name
- Design subhierarchy

You control these messages with entries in a configuration file that you specify with the following compile-time option:

```
-msg_config=message_configuration_file_name
```

In this message configuration file, the basic rules are as follows:

- Each configuration entry is enclosed in braces or curly brackets { } ; for example:

```
{ +warn=noTFIPC;  
  +file=$VCS_HOME/vmm.sv;  
}
```

This entry specifies disabling the warning message with the TFIPC message ID about the content of the vmm.sv source file in the VCS installation.

- Each entry can have only one message operation *command*, beginning with one the following keywords:

```
+lint +warn -error -suppress
```

- There can be multiple control conditions specified in the same entry, beginning with the following keywords:

```
+file +module +tree -file -module and -tree
```

- Message operation commands and control conditions that begin with + are for including something; those that begin with - are for excluding something.
- The message operation command and control conditions are separated with a semicolon ; or white space or return.

Note:

- Any + control condition, such as `+file`, cannot be used together with its corresponding – control condition, such as `-file`, in the same configuration entry.
- VCS reports an error condition if you specify conflicting control conditions for the same message ID.

This sections consists of the following subsections:

- “Controlling Lint Messages”
- “Controlling Warning Messages”
- “Controlling Error Messages”
- “Upgrading Lint and Warning Messages to Error Messages”
- “Downgrading Error Messages to Warning Messages”
- “Suppressing All Types of Messages”
- “Enabling and Disabling by Source File”
- “Enabling and Disabling by Module Definition”
- “Enabling and Disabling by Subhierarchy”

## Controlling Lint Messages

Lint messages are disabled by default so the lines in a configuration file enable their display.

To enable lint messages with a message operation command in a configuration entry that begins with `+lint=arguments`, use the following arguments:

`+lint=all`

To specify that the lines that follow enable the display of all lint messages.

```
+lint=ID1, ID2...
```

A comma separated list of lint message IDs to specify that you want to enable these specific lint messages, for example:

```
+lint=CDUB, NCEID
```

This list of IDs enables the display of the lint messages with CDUB and NCEID message IDs.

```
+lint=none
```

To specify that the lines that follow disable the display of all lint messages for a particular control condition in a configuration entry.

```
+lint=all, noID1, noID2...
```

A comma separated list of message IDs, each preceded by `no` with no space between `no` and the IDs, to disable these specified lint messages in a configuration entry.

Note the following about the `+lint` message operation command:

- It suppresses lint messages for the specified modules (see “[Enabling and Disabling by Module Definition](#)”) when you enter the `+lint=none` message operation command.
- It suppresses the specific lint messages for the specified modules when you enter the `+lint=no ID` message operation command.

## Controlling Warning Messages

To disable warning messages with the `+warn=arguments` message operation command, use the following arguments:

`+warn=none`

To specify that the lines that follow disable the display of all warning messages.

`+warn=no ID1, no ID2...`

A comma separated list of message IDs, each preceded by `no` with no space between `no` and the IDs, to specify that you want to disable these specific warning messages, for example:

`+warn=noMFACF, noCSMC`

This list of IDs disables the display of the warning messages with MFACF and CSMC message IDs.

Note the following about the `+warn` message operation command:

- It suppresses warning messages for the specified modules when you enter the `+warn=none` message operation command.
- It suppresses the specific warning messages for the specified modules when you enter the `+warn=no ID` message operation command.

## Controlling Error Messages

Error messages, like warning messages, are enabled by default. You can use the configuration file to do the following:

- Upgrade lint and warning messages to error messages.

- Downgrade applicable error messages to warning messages (not all error messages are downgradable).

## **Upgrading Lint and Warning Messages to Error Messages**

To upgrade lint and warning messages to error messages, use the `-error=arguments` message operation command in the configuration entry. The arguments you can enter are as follows:

`-error=all`

Upgrades all lint and warning messages to error messages.

`-error=ID1, ID2...`

A comma separated list of lint and warning message IDs to upgrade them to error messages, for example:

`-error=CDUB, MFACF`

This list of IDs upgrades the lint message with the ID of `CDUB` and the warning message with the ID of `MFACF` to error messages.

## **Downgrading Error Messages to Warning Messages**

To downgrade error messages to warning messages, use a message operation command in the configuration entry that begins with:

`-error=noID1, noID2...`

The comma separated list is a list of error message IDs, preceded by the keyword `no`, for example:

`-error=noURMI, noETMFBC`

Not all error messages are downgradable. If you enter an error message ID for a non-downgradable error message, you receive a different error message indicating that it is not downgradable.

**Important:**

You cannot downgrade all error messages to warning messages with the following line:

```
-error=none
```

## Suppressing All Types of Messages

You can disable the display of all types of messages - such as informational, lint, warning, and error messages. For this, enter a line in the configuration file beginning with the `-suppress` or `-suppress=arguments` message operation command except the error messages that cannot be downgraded.

**Note:**

The `-suppress` message operation command cannot suppress non-downgradable error messages.

The arguments you can enter are as follows:

`-suppress` without an argument

Suppress all downgradable messages. This message operation command is the equivalent of the `-error=none` message operation command.

`-suppress=ID1, ID2...`

A comma separated list of message IDs to suppress specific lint, warning, or error messages, for example:

```
-suppress=CDUB, CSMC
```

This list of IDs suppresses the display of the lint message with the CDUB ID and the warning message with the CSMC IDs.

## Enabling and Disabling by Source File

You can enable or disable lint, warning, and error messages for specific source files. For this, add the following control conditions to message operation command:

`+file=source_file_list`

`source_file_list` is a comma separated list of source files without spaces between them, for example:

`+file=top.sv,introctr.sv,arbit.sv`

This control condition specifies that the messages enabled in the preceding message operation command are enabled only for the source files named `top.sv`, `introctr.sv`, and `arbit.sv`.

`-file=source_file_list`

This control condition is similar to but opposite from

`+file=source_file_list`. This control condition specifies the source files not affected by the message operation command.

## Enabling and Disabling by Module Definition

You can enable or disable messages for specific module definitions. For this, add the following control conditions to a message operation command in a configuration entry:

`+module=module_name_list`

The module name list is a comma separated list of module names, for example:

```
+module=top,introctr,arbit
```

This control condition specifies that the messages enabled in the message operation command are enabled for the contents of the modules named `top`, `introctr`, and `arbit`.

```
-module=module_name_list
```

This control condition is similar to but opposite from `+module=module_name_list`. This control condition specifies the module definitions not affected by the message operation command.

## Enabling and Disabling by Subhierarchy

Consider a scenario in which your design includes sub-hierarchies, such as in a Verilog library file that has a top-level module and module definitions hierarchically under it, or some other discrete set of module definitions in a hierarchy with a top-level module, such as in design re-use in a larger design. To enable or disable messages for these subhierarchies, specify the top-level module definition with the following control conditions:

```
+tree=module_name_list
```

The module name list is a comma separated list of top-level module names, for example:

```
+tree=introctr,arbit
```

This control condition specifies that the messages enabled in the message operation command are enabled for module definitions `introctr` and `arbit` and the module definitions hierarchically under them.

```
-tree=module_name_list
```

This control condition is similar to but opposite from `+tree=module_name_list`. This control condition specifies the subhierarchies not affected by the message operation command.

---

## Extracting the Files Used in Elaboration/Compilation

To extract the Extensible Markup Language (XML) files, which are required to create the `top` module, use the `-metadump` compile-time option. Its syntax is as follows:

For two-step flow:

```
% vcs -metadump <design_top>
```

For three-step flow:

```
% vlogan <analyze_options> <source_file>
% vcs -metadump <design_top>
```

Using this syntax, you can generate the list of files required to create the top-level module and create `simv`.

The `-metadump` option generates XML files from which you can get information about all files with the file name and the information about the line number to resolve `simv`.

The reporting files are in the XML file format. The `verilogMetadata.xml` file is extracted for the Verilog portion of the design, and the `vhdlMetadata.xml` file is extracted for the VHDL portion of the design. These files can be accessed from the current working directory.

**Note:**

This section describes the feature in the context of the Unified Use Model (UUM) flow only. The two-step flow is not supported in this implementation.

## **XML File Format**

This section describes the format of the XML document. Synopsys does not provide a parser for the XML file and it is suggested to choose to process the file in the way you want.

There are four main sections in the XML file as described in this section.

### **Section 1**

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<opml version="1.0">
<head>
<title>VCS Dump File for Post Process</title>
<vcsVersion> H-2013.06-SP1 (ENG)</vcsVersion>
<dateCreated>Tue May 14 13:35:00 2013</dateCreated>
</head>
```

The top-level `head` section describes the basic statistical information about the file, such as the VCS version that is used and the date on which it is created. This information can be used to keep track of when the list of files was extracted.

### **Section 2**

The following example provides the collection of files that are listed in the `<filelist>` section. These are the complete set of files that are used in the entire design.

```
<fileList>
```

```

<file fid="0" path="/remote/path1/directory1/Macro/m.h" />
<file fid="1" path="/remote/path2/directory2/y.h" />
<file fid="2" path="/remote/path1/directory1/Macro_n.h" />
<file fid="3" path="/remote/path1/directory1/Macro/x.h" />
<file fid="4" path="=/remote/path1/directory1/Macro/
test.v" />
</fileList>

```

Each of the file fid point to a specific path on the file system from where it is picked up. This section also provides a list of all the files that make up simv.

In this example, file fid ="1" refers to the /remote/path2/directory2/y.h file on the file system. The file fid="0" refers to the m.h file in some other location. The list of files included in this section are both include files and elaboration files.

### Section 3

This section provides the information about the files that are included by other files as understood by the VCS parser. It provides a list of include files that are used for elaboration of the design. Note that the list of include files are non-unique. If a file is included by many files, it is displayed as separate lines in this section of the XML file.

```

<!-- Include file list for the whole design -->
<includeFileList>
<incfile fid="0" lineno="10" includeID="3" />
<incfile fid="4" lineno="1" includeID="0" />
<incfile fid="4" lineno="2" includeID="2" />
<incfile fid="4" lineno="3" includeID="3" />
</includeFileList>

```

For example,

```
<incfile fid="0" lineno="10" includeID="3" />
```

is interpreted as follows:

file fid="0", that is, /remote/path1/directory1/Macro/m.v includes file fid = "3", (/remote/path1/directory1/Macro/x.h) on line number 10.

Each file picked up by the parser is reported as explained.

In this example, it is noted that file fid="3" (that is /remote/path1/directory1/Macro/x.h) is included by multiple files, and therefore, shows up multiple times in the list.

## Section 4

This section is a unified unique list of files that are included in the design using the `include directive from the previous section, which is a list of unique includeIDs.

This section displays only a subset of the list presented in Section 3. As mentioned earlier, includedfile fid="3" has been included multiple times by many files and can be seen in the XML entries in the previous section. However, it is reported once in this section.

```
<uniqIncludeFileList>
<includedfile fid="0" />
<includedfile fid="2" />
<includedfile fid="3" />
</uniqIncludeFileList>
```

### Example

If the top module is specified as a design top in elaboration, VCS gathers top.v and header.v elaboration files. The top.v file is gathered as a non-include file and header.v is gathered as an include file. If these two files are provided as input to tools, such as vlogan, multiple-definition errors might occur.

**top.v**

```
`include "header.v"
module top;
bottom bot();
endmodule
```

**header.v**

```
module bottom;
endmodule
not_used.v

module not_used;
endmodule
```

**VCS Command Line:**

```
% vcs top.v
% vcs header.v
% vcs not_used.v
% vcs top -sverilog -metadump
% simv
```

**VCS generates the `verilogMetadata.xml` file, which can be accessed from the current working directory and contains the following in <body>:**

```
<fileList>

<file fid="0" path="/remote/xxxx/yyy/Documents/header.v" />
<file fid="1" path="/remote/xxxx/yyy/Documents/top.v" />

</fileList>
```

**The `verilogMetadata.xml` file lists only `header.v` and `top.v`. However, it does not list the `not_used.v` file as it is not used in simulation.**

**Note:**

For any tool that is capable of parsing the Verilog file and substitute the `include directive, non-include files are sufficient to work.

**Example**

**vllogic.v**

```
module vllogic(a,b,c,d,e);
  input a,b,c;
  output d,e;
  wire d,e;
  assign d = a ^ b ^ c;
  assign e = ((a&b) | (b&c) | (a&c));
endmodule
```

**vltop.v**

```
`include "vllogic.v"
module vltop;
  reg a,b,c;
  wire d,e;
  vhent vh1(.a(a), .b(b), .c(c), .d(d), .e(e));
  initial begin
    $monitor("a=%b b = %b c=%b d=%b e=%b", a,b,c,d,e,$time);
    #2 a = 'b1; b = 'b1; c= 'b0;
    #2 a = 'b0; b = 'b1; c= 'b1;
    #2 a = 'b1; b = 'b0; c= 'b0;
    #5 $finish;
  end
endmodule
```

**vhnet.vhd**

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;
entity vhent is
  port(a,b,c: in std_logic;
```

```

d,e: out std_logic);
end entity;
architecture structural of vhent is
component vlogic
port(a,b,c: in std_logic;
d,e: out std_logic);
end component;
begin
vh1: vlogic port map(a,b,c,d,e);
end structural;

```

### **not\_used.vhd**

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;

entity orGate is
    port(      A, B : in std_logic;
               F : out std_logic);
end orGate;

architecture func of orGate is
begin
    F <= A or B;
end func;

```

### **VCS Command Line:**

```

% vhdlan not_used.vhd
% vlogan vlogic.v
% vlogan vtop.v
% vhdlan vhnet.vhd
% vcs vtop -metadump
% simv

```

**VCS generates the verilogMetadata.xml file for the Verilog code and the vhdlMetadata.xml file for the VHDL code, which can be accessed from the current working directory.**

**The vhdlMetadata.xml file lists the following:**

```
<fileList>
<file fid="0" path="/remote/xx/y/yyy/metadump1/vhnet.vhd" />
</fileList>
```

However, the XML file does not list the `not_used.vhd` file as it is not used in simulation.

**Note:**

To extract the text files instead of XML files, use the `-metadump_txt` compile-time option.

## **Limitations**

The following are the limitations with the `-metadump` option:

- The order of elaboration files cannot be guaranteed. For example, in a three-step flow, if you swap file A and file B for `vlog`, one or more of the following can happen:
  - For the undefined macro references, syntax errors occur if file A uses a macro defined in file B, and file A does not include file B either directly or recursively.
  - A different default net type is picked up if file A uses a default net type defined in file B, and file A does not include file B either directly or recursively.
  - Error might occur in other related compiler directives.

Even with correct ordering of file A and file B, other problems might arise.

For example, in an elaboration, top1 is named design top, and top1.v gets copied over to a new directory for insertion and the timescale of top1.v is 1ns/1ns. In the original analyzed database (AN.DB), the timescale could either be 1ps/1ps or 1ns/1ns based on the order of analysis of top1.v and top2.v.

#### top1.v

```
`timescale 1ns/1ns
module top1;
    initial begin
        $display("At %t Hello world\n", $time);
    end
endmodule
```

#### top2.v

```
`timescale 1ps/1ps
module top2;
    initial begin
        $display("Hello world\n");
    end
endmodule
```

The correct ordering does not guarantee simulation of file copies and happens in the same manner as that of original files.

- Elaboration files do not include configuration files, synopsys\_sim.setup, other non-Verilog files, and non-VHDL files.
- The -metadump option does not work with partition compile flow. The metadump file can be generated with single compile flow and the same file works in both single compile and partition compile flow.



# 5

## Simulating the Design

---

This chapter describes the following:

- “Using Verdi”
- “Using UCLI”
- “Reporting Forces/Injections in a Simulation”
- “Key Runtime Features”

As described in the section “[Simulation](#)”, you can simulate your design in either interactive mode or batch mode. To simulate your design in interactive mode, you can use Verdi and UCLI. To simulate your design in batch mode, you must use UCLI. For more information, refer to the section entitled, “[Batch Mode](#)”.

---

## Using Verdi

Verdi provides you with a graphical user interface to debug your design. Using Verdi, you can debug the design in interactive mode or in post-processing mode. You must use the same version of VCS and Verdi to ensure problem-free debugging of your simulation.

In the interactive mode, apart from running the simulation, Verdi allows you to do the following:

- View waveforms
- Compare waveforms
- Trace drivers and loads
- View schematics and path schematics
- Execute UCLI/Tcl commands
- Set breakpoints (line, time, event, and so on)
- Line stepping

However, in post-processing mode, an FSDB file is created during simulation, and you use Verdi to:

- View waveforms
- Compare waveforms
- Trace drivers and loads
- View schematics and path schematics

**Set `VERDI_HOME` and perform the following:**

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

```
% simv -gui
```

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

```
% verdi -ssf novas.fsdb -nologo
```

Note:

The interactive mode of Verdi is not supported when you are running VCS slave mode simulation.

For information on generating an FSDB dump file, see [“Debugging with Verdi”](#).

For more information on using Verdi, see *Verdi and Siliti Command Reference Guide* under Verdi documentation in SolvNet.

---

## Using UCLI

Unified Command Line Interface (UCLI) provides a common set of commands for interactive simulation. UCLI is the default command line interface for batch mode debugging in VCS.

UCLI commands are based on Tcl, therefore you can use any Tcl command with UCLI. You can also write Tcl procedures and execute them at the UCLI prompt. Using UCLI commands, you can do the following:

- Control simulation

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

UCLI commands are built based on Tcl. Therefore, you can execute any Tcl command or procedures at the UCLI prompt. This provides you with more flexibility to debug the design in interactive mode. The following command starts the simulation from the UCLI prompt:

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

When you execute the above command, VCS takes you to the UCLI command prompt. To invoke UCLI, ensure that you specify the `-debug_access+r` option during compilation/elaboration. You can then use the `-ucli` option at runtime to enter the UCLI prompt at time 0 as follows:

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

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

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

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

**Note:**

UCLI is not supported when you are running VCS slave mode simulation.

**Note:**

You can use the `-ucli` option at runtime even if you have not used some form of the `-debug_access` option during compilation, but in this case only the `run` and `quit` UCLI commands are supported.

Note the following behavioral changes when UCLI is the default command-line interface:

- Command line options, such as `simv -i` or `-do`, only accept UCLI commands
- Interrupting the simulation using `Ctrl+C` takes you to the UCLI prompt by default for debugging your designs
- UCLI include file options (`-i` or `-do`) expect the next argument to be a UCLI script.

```
%> simv -ucli -i ucli_script.inc
```

- The `-R` feature in VCS continues to take you to the old CLI UI, unless you explicitly add `-ucli` to VCS command line.

## **-ucli2Proc Option**

By default, UCLI runs in the `simv` process. There are a few scenarios which may require running UCLI in its own process, and this is enabled using the `-ucli2Proc` option:

- In SystemC designs, you must specify the `-ucli2Proc` command, if you want to call ‘cbug’ in batch mode (ucli). VCS issues a warning message if you do not specify this command
- When you issue a `restore` command inside a `-i/-do/source`, you must pass the `-ucli2Proc`. This situation is only applicable when there are commands following the `restore` commands that need to be executed in the `do` script
- Any usage of `start/restart/finish/checkpoint/config endofsim/reversedebug` from UCLI needs the `-ucli2Proc` command

For more information about UCLI, click the link [Unified Command-line Interface \(UCLI\)](#) if you are using the VCS Online Documentation.

If you are using the PDF interface, see *ucli\_ug.pdf* to view the UCLI User Guide.

## Options for Debugging Using Verdi and UCLI

`-debug_access`

Gives best performance with the ability to generate FSDB/VPD/VCD files for post-process debug. It is the recommended option for post-process debug.

`-debug_access (+<option>)`

Allows you to have more granular control over the debug capabilities in a simulation. The `-debug_access` option enables the dumping of the VPD and FSDB files for post-process debug.

You can specify additional options with the `-debug_access` option to selectively enable the required debug capabilities. You can optimize the simulation performance by enabling only the required debug capabilities.

For more information on the `-debug_access` option, see [“Optimizing Simulation Performance for Desired Debug Visibility With the `-debug\_access` Option” section](#).

`-debug_region (=<option>) (+<option>)`

Allows you to have better control over the performance of `-debug_access`. This option enables you to apply debugging capabilities to the desired portion of a design.

You must use the `-debug_region` option along with the `-debug_access` option at compile time. For more information on `-debug_region`, see [“Optimizing Simulation Performance for Desired Debug Visibility With the `-debug\_access` Option” section](#).

`+fsdbfile+filename`

FSDB dumping option. This option allows you to specify the FSDB file name. If this option is not specified, the default FSDB file name is `novas.fsdb`.

**Example:** `+fsdbfile+test1.fsdb`

`-vpd_file <file_name>`

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.

`+fsdb+dump_limit=size`

Specifies a size limit for the FSDB file. The default and minimum size limit is 10MB.

**Example:** +fsdb+dump\_limit=15

**-vpd\_fileswitchsize <size\_in\_MB>**

Specifies a size for the VPD file. When the VPD file reaches this size, VCS closes this file and opens a new one with the same size.

---

## Reporting Forces/Injections in a Simulation

VCS provides the details of all the forces applied on your design during the simulation in a user-defined ASCII text file. This feature helps you to debug forces by allowing you to view all the forces that are effective in a simulation.

The following sections describe this feature in detail:

- [Use Model](#)
- [Reporting Force/Deposit/Release Information](#)
- [Generating Force List Report for Desired Instance Hierarchies](#)
- [Displaying Different ID for Each Verilog Force](#)
- [Displaying Source Information for VHDL Forces](#)
- [Displaying Source Information for External Forces](#)
- [Viewing Force Information in Interactive Debug Mode](#)
- [Reporting \\$deposit Value Changes](#)
- [Limitations](#)

---

## Use Model

Perform the following steps to use this feature:

1. Use the `-force_list` option at compile time, as shown below, to allow the force reporting feature to record language forces/releases.

```
% vcs <debug_option> filename.v -force_list  
<other_vcs_options>
```

Where,

`<debug_option>`

Reporting external (PLI) forces and deposits requires a minimum of read and callback debug capability. This corresponds to `-debug_access+r+cbk`.

Note:

This step does not enable force reporting feature by itself. You must use `-force_list <filename>` at runtime, as shown in the following step. For more information, see [Table 5-1](#).

2. Use the `-force_list` option at runtime, as shown below, to enable force reporting feature and generate an ASCII text file containing information about the forces/deposits/releases applied during the simulation in time order.

```
% simv -force_list <filename>
```

Where, `filename` is the user-defined ASCII file name. It can be relative path or absolute path. Compression is disabled by default. Use the `-force_list_compress` option at runtime to compress the resulting log file with the gzip compression. The log file is saved with the same name, but changes its filename extension by appending `.gz` at the end of it.

For example, for the following command:

```
% simv -force_list report.log -force_list_compress
```

the output file is: `report.log.gz`

Use gunzip to uncompress a force list file. For example, uncompress the above output file as follows:

```
gunzip report.log.gz
```

This results in the original file `report.log` which is uncompressed.

## Key Points to Note

- If you use the `-force_list` option at runtime, but not at compile time, only external forces are logged
- If you use the `-force_list` option at both compile time and runtime, then both language forces and external forces are logged
- For mixed signal designs, the `-force_list` option captures the force events on the VHDL/VHPI/hdl\_xmr targets. Pure VHDL designs are not supported
- [Table 5-1](#) describes the usage of the `-force_list` option in detail.

*Table 5-1 Usage of the -force\_list Option*

<b>-force_list at compile time</b>	<b>-force_list at runtime</b>	<b>Language forces logged</b>	<b>External forces logged</b>
No/Yes	No	No	No
No	Yes	No	Yes
Yes	Yes	Yes	Yes

---

## Reporting Force/Deposit/Release Information

The ASCII text file consists of the following parts:

- A header section that includes the information given in [Table 5-2](#), associated with an ASCII character ID that is 1 to 4 characters long. For more information, see “[Header Section](#)”.
- A time order sorted list of force/release/deposits, as they occur during the simulation, indexed by the ID shown in the header section. For more information, see “[Event List Section](#)”.

**Table 5-2 Force Capture and Log Information**

Force type	Time	Instance name of the target node	Module name where force occurred	File / Line data logged	Value
Language force/ release/ deposit	Simulation time when the node was forced or released.	Hierarchical node name being forced. Example: top.test.child2.a	Name of the module. Example: top	Full path of the file where the force statement occurs, and the line number of the statement in the source file. For example: /home/work/test.v:1234	Value represented as binary except for int, real, and string types.  Binary value is prefixed with 'b.  Release will not have values.
VPI/ACC/UCLI force/ release/ deposit	Simulation time when the node was forced or released.	Hierarchical node name being forced.	Not Applicable	Not Applicable	Value represented as binary except for int, real, and string types.  Binary value is prefixed with 'b.  Release will not have values.

## Handling Forces on Bit/Part Select and MDA Word

If the target of the force is bit-select, part-select, or mda word, the appropriate indices is included in the target node name, for example, as follows:

Bit select	top.a.b[2]
Part select	top.a.b[0:3]
MDA bit select	top.c.d[2][3][4]

```
MDA part select          top.c.d.[2][3][1:4]
```

Forces recorded can be any object supported by language and PLI forces.

Expressions are evaluated only if they contain constants and/or parameters. For example, `top.a[(1+paramb)*2]` is evaluated to determine the resulting constant index.

Expressions are not evaluated if they contain variables. For example, consider the following code. In this case, only the base vector is captured.

```
logic [0:9] top.a;
for (i= 0; i < 10; i++)
begin
    force top.a[i] = 1; // captures the entire vector
for top.a and not a select of top.a
end
end
```

## Handling Forces on Concatenated Codes

Forces consisting of more than one signal on a signal line are split up on signal basis.

For example, `force {a,b} = 2'b11` results in two header entries, one for `a` and one for `b`. Both `a` and `b` display the same file/line number, but carry different IDs since they are different nodes.

## Output Format

The ASCII text file output consists of the following two sections:

- Header Section

- Event List Section

## Header Section

The header section contains mapping between forced object list and unique ASCII ID. This section is divided into two parts: Language Forces and External Forces.

Language forces are unique by statement, whereas external forces are unique by node. Multiple language forces on the same node from different lines result in multiple header entries for that node.

Multiple external forces result in one external force header entry for that node. Nodes with both language and external forces have entries in both Language Forces and External Forces parts.

For a unique node, only single ID is used for all entries in header for that particular node.

If VCS finds unsupported force/release event, it labels such event with a reason.

Following is the display format for Language Forces and External Forces:

Language Forces:

ID	Target	Module	File:Line
----	--------	--------	-----------

External Forces (VPI/ACC/UCLI):

ID	Target
----	--------

## Header Example

Header Section

Language Forces

ID	Target	Module	File	Line
1	top.child1.a	top	/home/user/top.v:10	** NO_VALUE_CHANGE full mda **
2	top.child1.child2.foo	child	/home/user/child.v:125	
3	top.child1.a_real	child	/home/user/child.v:127	

External Forces

ID Target

4	top.child1.a_int
5	top.child1.b[0:3]
6	top.child1.b[2]

## Event List Section

This section displays the following information:

- Time during which the value is forced
- ID from the header
- Type of the force
- Value of force or deposit

Release value is not displayed. [Table 5-3](#) lists the phrases of the acronyms used in the Event List section.

*Table 5-3 List of Acronyms Displayed in the Event List Section*

Acronym	Phrase
LF	Language Force
LR	Language Release
LD	Language Deposit/write
EF	External Force
ER	External Release
ED	External Deposit

### Event List Example

```

ID      Type    Value
----   Time:   1   ---
2       LF      'b0
5       EF      'b1111
6       EF      'b1
4       ED      25
----   Time:   2   ---
3       EF      2.14
3       ER
5       LR
5       LD      'b0110
---    Time:   3   ---

```

The `Value` column displays the integer and real values as decimal values, strings as ASCII characters, and rest of them as binary values prefixed with '`b`'. Long value strings are line-wrapped.

### Usage Examples

#### Example-1

Consider the following testcase `test.v` and the `test.ucli` file which contains UCLI forces.

*Example 5-1 Design Testcase test.v*

```
module top;

reg clk,rst,d;
wire q;

DUT dut (clk,rst,d,q);

always #1 clk = ~clk;

initial begin
    clk = 0 ; rst =0 ; d=0 ;

    #5 force rst = 1;
    #5 release rst ;
    #10 force dut.q =1 ;
    #10 release dut.q ;
    #100 $finish ;
end
endmodule

module DUT (clk,rst,d,q);
input clk,rst,d;
output q;
wire q;
reg q_reg;

assign q = q_reg;

always @ (posedge clk)
if (rst) begin
    q_reg <= 0;
end else begin
    q_reg <= d;
end

endmodule
```

*Example 5-2 test.ucli*

```
run 30
```

```
force top.dut.q 1  
release top.dut.q  
run
```

**Compile the test.v code, as follows:**

```
% vcs -debug_access+r+cbk -sverilog -force_list  
test.v
```

**Run the simulation, as follows:**

```
% simv -ucli -i test.ucli -force_list report.log
```

**Use the following command to view the report.log file:**

```
% cat report.log
```

**Below is the content of the report.log file:**

```
VCS Force List  
Header Section  
  
Language Forces  
  
ID Target    Module    File Line  
  
1 top.rst top force.v 13  
1 top.rst top force.v 14  
2 top.dut.q top force.v 15  
2 top.dut.q top force.v 16  
  
External Forces  
  
ID    Target  
  
2 top.dut.q  
  
Event List Section
```

```

----- Time: 0 -----
----- Time: 5 -----
1  LF 'b1
----- Time: 10 -----
1  LR
----- Time: 20 -----
2  LF 'b1
----- Time: 30 -----
2  EF 'b1
2  LF 'b1
2  ER
2  LR
2  LR

```

## **Example-2**

This example describes the usage of `-force_list` to capture force events on the `hdl_xmr` targets.

Consider the following files:

### *Example 5-3 Testcase test\_1.v*

```

`timescale 1 ns/1 ns
module ex1 (main_bus bus);

reg reg_hdlxmr;
reg [7:0] reg_hdlxmr_vec;

endmodule

```

### *Example 5-4 Testcase test\_2.v*

```

module vtop (wa,wb,wc);

inout wa,wb;
input wc;
main_bus bus (.wa(wa), .wb(wb), .control1(wc));

ex1 ex11 (.bus(bus));

reg source_inter_reg;

```

```

byte vbyte1=8'd1,vbyte2=8'd1;
shortint vshort1=16'd1,vshort2=16'd1;
int vint1=32'd1,vint2=32'd1;
longint vlong1=64'd1,vlong2=64'd1;
bit[0:7] vbit1,vbit2;
bit[7:0] vbit3,vbit4;
logic[0:7] vlogic1=8'd1,vlogic2=8'd1;
logic[7:0] vlogic3=8'd1,vlogic4=8'd1;

logic l1=1;

initial
begin
# 10 ;
bus.interfacebyte2 = 8'd10 ;
bus.interfaceshort2 = 16'd10 ;
bus.interfacelong2 = 64'd10 ;
bus.interfacebit4 = 8'd10 ;
bus.interfacebit3 = 8'd10 ;
bus.interfaceint1 = 5 ;
bus.interfaceint2 = 5 ;
bus.interfacelogic3 = 8'd10 ;
end

endmodule

```

### *Example 5-5 Testcase interface\_datatypes.v*

```

typedef enum {red,blue,green,yellow} color1;
interface main_bus (inout wa, wb, input control1);

reg interface_reg;
byte interfacebyte1 ,interfacebyte2 ;
shortint interfaceshort1 ,interfaceshort2 ;
integer interfaceint1 ;
int interfaceint2 ;
longint interfacelong1 , interfacelong2 ;
bit[0:7] interfacebit1 ,interfacebit2 ;
bit[7:0] interfacebit3 , interfacebit4 ;
logic[0:7] interfacelogic1 ,interfacelogic2 ;
logic[7:0] interfacelogic3 ,interfacelogic4 ;

```

```

logic interfacell;
logic[0:1] interfacef1,interfacef2;
color1 intfcolor1 , intfcolor2 ;
modport driver (inout wa,
                inout wb,
                input control1);

modport slave (inout wa,
                inout wb,
                input control1);

endinterface

```

### *Example 5-6 Testcase test.vhd*

```

LIBRARY IEEE;
USE STD.TEXTIO.all;
USE IEEE.STD_LOGIC_TEXTIO.all;
USE IEEE.std_logic_1164.ALL;
Library Synopsys;
USE Synopsys.hdl_xmr_pkg.all;

entity ex2_vhtop  is
port (wa,wb:INOUT std_logic;control1:in std_logic);
end entity;

architecture arch of ex2_vhtop is
component vtop is
port (wa,wb:INOUT std_logic;wc:in std_logic);
end component;
signal w_vector:std_logic_vector(0 to 10);
signal wire_wa,wire_wb,wire_control:std_logic:='0';
signal vhd_sig,vhd_sig1:std_ulogic_vector(0 to 7);
type color_vhd is (red,blue,green,yellow);
signal vhd_enum,vhd_enum1:color_vhd;
signal vhbytel,vhbyte:std_logic_vector(0 to 7) ;
signal vhshort1,vhshort2:std_logic_vector(0 to 15);
signal vhlond1,vhlond2:std_logic_vector(0 to 63);
signal vhbit1,vhbit2:std_logic_vector(7 downto 0);
signal vhd_sig3,vhd_sig4:std_logic_vector(7 downto 0);
signal vhint1,vhint2:integer;

```

```

signal vhbit3,vhbit4:bit_vector(7 downto 0);
signal vh_ustd3,vh_ustd4:std_ulogic_vector(7 downto 0);

begin

--Source interface signals
hdl_xmr(":ex2_vhtop:g1:bus:interfacebyte2","vhbyte1",1);
hdl_xmr(".ex2_vhtop.g1.bus.interfaceshort2","vhshort1",1);
hdl_xmr(".ex2_vhtop.g1.bus.interfacelong2","vhlong1",1);
hdl_xmr(".ex2_vhtop.g1.bus.interfacebit4","vhdsig4",1);
hdl_xmr(".ex2_vhtop.g1.bus.interfacebit3","vhbit3",1);
hdl_xmr(".ex2_vhtop.g1.bus.interfaceint1","vhint1",1);
hdl_xmr(".ex2_vhtop.g1.bus.interfaceint2","vhint2",1);
hdl_xmr(".ex2_vhtop.g1.bus.interfacelogic3","vh_ustd3",1);

g1:vtop port map(wire_wa,wire_wb,wire_control);
wa <= wire_wa;
wb <= wire_wb;

end arch;

```

**Execute the following elaboration commands:**

```

% vlogan -sverilog interface_datatypes.v
% vlogan -sverilog test_1.v
% vlogan -sverilog test_2.v
% vhdlan test.vhd
% vcs ex2_vhtop -debug_access+r+cbk

```

**Execute the following simulation command:**

```
% simv -force_list force.log
```

The following output (force.log) shows the force events on the hdl\_xmr targets:

```
VCS Force List
Header Section
```

## External Forces

ID Target

```
1 /EX2_VHTOP/VHBYTE1  
2 /EX2_VHTOP/VHSHORT1  
3 /EX2_VHTOP/VHLONG1  
4 /EX2_VHTOP/VHD_SIG4  
5 /EX2_VHTOP/VHBIT3  
6 /EX2_VHTOP/VHINT1  
7 /EX2_VHTOP/VHINT2  
8 /EX2_VHTOP/VH_USTD3
```

## Event List Section

---

## **Generating Force List Report for Desired Instance Hierarchies**

The `-force_list` option allows you to generate force list report for the desired instance hierarchies in the design. This feature helps you to get the force list report for the selected hierarchies.

Use the following option at runtime to generate force list report for the desired instance hierarchies in the design:

```
-force_list_hier <list_file>
```

Where, `<list_file>` is the configuration file. This file allows you to specify the hierarchy and levels of hierarchy to be dumped in the resultant `-force_list` output file.

**Note:**

To use this feature, you must first compile your designs with the `-force_list` option.

Following is the syntax of the `<list_file>` statements:

```
+/-<instance> <depth>
```

Where,

```
+<instance>
```

Fully rooted path name to a Verilog instance. Dumps the forces on the signals in the instance tree to the report.

```
<depth> for +<instance>
```

Specify the level/levels of hierarchy to be dumped. For example, <depth> of 1 means only the signals in <instance> are dumped. <depth> of 0 means all levels of sub-instances are dumped.

-<instance>

The forces on signals in the instance tree are filtered out. They are not dumped to the report.

<depth> for -<instance>

Specify the level/levels of hierarchy to be filtered out. For example, <depth> of 1 means only the signals in <instance> are filtered out. <depth> of 0 means all levels of sub-instances are filtered out.

## Examples

The following are the examples:

- Example-1: Generating force list report for the desired instance hierarchies
- Example-2: Reporting \$deposit value changes

### Example-1: Generating force list report for the desired instance hierarchies

*Example 5-7 test.v*

```
module tb();
    reg r;
    wire oA, oB;
    dut dut(r, oA, oB);
    initial begin
        r = 0;
        #10 force tb.dut.instA.oA = 1;
```

```

#10 force tb.dut.instA.instB.oB = 1;
end
endmodule

module dut(input r, output oA, oB);
    A instA(r, oA, oB);
endmodule

module A(input r, output oA, oB);
    assign oA = r;
    B instB(r, oB);
endmodule

module B(input r, output oB);
    assign oB = r;
endmodule

```

**Example 5-8 hierarchyConfig.cfg**

```
+tb.dut.instA 1
```

**Compile test.v as follows:**

```
% vcs -debug_access -sverilog -force_list test.v
```

**Simulation command:**

```
% simv -force_list=uniqueID forceReport.txt -
force_list_hier hierarchyConfig.cfg
```

**Following is the forceReport.txt output:**

```
VCS Force List
Header Section
```

```
Language Forces
```

ID	Target	Module	File	Line
----	--------	--------	------	------

```
1 tb.dut.instA.oA tb test.v 7
```

```
Event List Section
```

```
---- Time: 10 ----  
1 LF 'b1
```

## Example-2: Reporting \$deposit value changes

*Example 5-9 test.v*

```
module mymod(o,i,phi);  
    output o;  
    input i,phi;  
  
    and(o,i,phi);  
  
    initial begin  
        $deposit(o,1'b1);  
    end  
endmodule  
  
module test;  
  
    reg i,phi;  
    wire o;  
  
    mymod m1(o,i,phi);  
  
    initial begin  
        i=0; phi=0;  
        #5 i=1;  
        #5; phi=1;  
        #5; phi=0;  
        #10; $finish;  
    end  
endmodule
```

Compile test.v as follows:

```
% vcs -debug_access -sverilog -force_list test.v
```

Simulation command:

```
% simv -force_list=uniqueID forceReport.txt
```

Following is the frc.txt output:

```
VCS Force List
    Header Section

    Language Forces

    ID   Target     Module      File Line

    1  test.m1.o mymod prim.v 8

    Event List Section

    ----- Time: 0 -----
    1  LD 'b1
```

---

## Displaying Different ID for Each Verilog Force

The force list feature provides separate IDs for each source of a Verilog force, release, or deposit in the Language Forces section of the force list file.

## Enhanced Force List Report

The enhanced force list report looks as follows:

- VCS displays a unique ID for each Verilog force in the Language Forces section.
- The unique ID is used in the Event List Section.

Following is the sample report:

```

VCS Force List
    Header Section

    Language Forces

ID  Target      Module      File Line

1 test.a test t.v 3
2 test.a test t.v 4

    Event List Section

---- Time: 0 ----
---- Time: 1 ----
1  LF 'b0
---- Time: 2 ----
2  LF 'b1

```

## Use Model

To enable this feature, perform the following:

- At compile-time, use the following options:
  - The `-force_list` option. This option enables the force list feature
  - Debug capability `-debug_access+r+cbk` or higher
  - The `-sverilog` option, otherwise old Verilog style force may be missing
- At runtime, use the following option to enable the new enhancements:
   
`-force_list=uniqueID`

---

## Displaying Source Information for VHDL Forces

The force list report is enhanced to mark the VHDL force/release with the file name and line number of the corresponding force/release assignment statement. For example, see the following report:

```
VCS Force List
    Header Section

    VHDL Language Forces

    ID   Target        File           Line
    1   /top/et/S_1    forceRelease.vhd  23
    2   /top/et/S_1    forceRelease.vhd  25
    3   /top/et/S_2    forceRelease.vhd  28

    Event List Section

    ---- Time: 0 ----
    1  LF 'b1
    ---- Time: 1 ----
    2  LR
    ---- Time: 2 ----
    3  LF 2
    ---- Time: 3 ----
    4  LR
    5  LF 32
```

## Use Model

To enable this feature, perform the following:

- At compile-time, use the following options:

- The `-force_list` option. This option enables the force list feature
  - Debug capability `-debug_access+r+cbk` or higher
  - The `-sverilog` option, otherwise old Verilog style force may be missing
- At runtime, use the following option to enable the new enhancements:  
`-force_list=uniqueID`

---

## Displaying Source Information for External Forces

VCS provides detailed information about the origin of an external force, release, or deposit in the force list report.

The force list report is enhanced to mark an external force, release, or deposit with the file name and line number of the corresponding force, release, or deposit assignment statement.

[Table 5-4](#) describes this enhancement in detail.

**Table 5-4 Displaying Source Information for External Forces**

	<b>Origin of the force, release, or deposit</b>	<b>Description</b>
1	hdl_xmr_force, hdl_xmr_release, hdl_xmr	Force, release, or deposit that originates from hdl_xmr* procedures is marked with the Verilog/VHDL source file name and line number calling the hdl_xmr procedure, if one exists.
2 UCLI	UCLI	Force, release, or deposit that originates from sourcing a file with the UCLI Tcl do command is marked with the TCL source file name and line number. There is no file name and line number information when the Tcl source command is used.
		Force, release, or deposit originating from an interactive command is marked as INTERACTIVE, and no file name and line number is given.
		Force, release, or deposit originating from mhpi_ucliTclExec is marked as UCLI_FROM_MHPI, and no file name and line number is given.
		No file name and line number is given for the force command executed in the UCLI stop -command script.
3 VPI, VHPI, MHPI, VCSD, and Third-party tools		Third-party tools and other PLI applications are typically not run from a design file name/line number. Third-party tools are run from a VPI/VHPI synchronization callback. Therefore, there is no design file name/line number information.
		Force, release, or deposit originating from the user-defined system function activation is marked with the design file name and line number for the activation, if one exists.
		If a force, release, or deposit originates from a simulation callback, no file name and line number is given.

## Enhanced Force List Report

Following is the sample report:

External Forces

ID      Target

1 top.tsanadigmld.vccsa      t.v, 11

```
2 top.tsanadigmld.vccst    t.v, 12
3 top.tsanadigmld.vcc_ts   t.v, 13
```

Following is the output when the UCLI force is used:

```
top.tsanadigmld.vccsa    my.tcl, 11
```

Following is the output when the VPI force is used in a third-party PLI called with the \$user\_pli user-defined task in the testbench.v Verilog file:

```
top.tsanadigmld.vccsa  testbench.v, 11
```

## Use Model

To enable this feature, perform the following:

- At compile-time, use the following options:
  - The `-force_list` option. This option enables the force list feature
  - Debug capability `-debug_access+r+cbk` or higher
  - The `-sverilog` option, otherwise old Verilog style force may be missing
- At runtime, use the following option to enable the new enhancements:  
`-force_list=uniqueID`

---

## Viewing Force Information in Interactive Debug Mode

VCS allows you to view all active forces at the current simulation time in interactive debug mode using the `find_forces` UCLI command.

### Use model

Compile the design as follows:

```
% vcs -debug_access+r+cbk -sverilog -force_list  
test.v
```

Run the simulation as follows:

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

Use the `find_forces` UCLI command at the `ucli` prompt. This command reports the current active forces in the design or scope.

```
ucli% find_forces <option_name>
```

Following is the use model of the `find_forces` UCLI command:

```
find_forces <nid>  
  
find_forces -scope <scope_name> [-level  
<level_number>] [-file <file_name>]
```

Where,

- `nid` is a nested identifier (hierarchical path name).
- `-scope` allows you to specify the full scope path name to find force.

- `-level` allows you to specify the level of hierarchy to do force search. Default value is 0 (search all sub-scopes).
- `-file <file_name>`: Specify a file to report the result. By default, report is generated as UCLI output.

Examples:

- `ucli% find_forces -scope tb`  
**Searches tb scope only**
- `ucli% find_forces -scope tb/DUT/* -level 3`  
**Searches DUT and additional 2 levels under DUT**
- `ucli% find_forces tb/DUT/clk_i_node`  
**Specifies whether the node clk\_i\_node is forced**
- `ucli% find_forces -scope tb/DUT/* -level 0 file.txt`  
**Searches all levels under DUT and writes to file.txt.**

Note:

- You must specify a signal name when `-scope` is not specified. Otherwise, VCS issues an error message.
- A force is reported only if it is forced. If the force start time (for example, using `hdl_xmr_force`) is beyond current time, but the actual command was already executed, it will not be reported in the force list.

## Example

Consider the following files:

### *Example 5-10 test.sv*

```
module test;
    reg [0:3] a;
    reg clk;
    wire aa,b;
    assign aa=a[1];

    dut dt (clk,aa,b);
    initial
    begin
        clk=0;
        a = 4'b1010;
        #10 a= 4'b1110;
        #10 force a[0] =1'bX;
        #10 release a[0];
        #100 $finish;
    end
    initial begin
        forever #1 clk = ~clk;
    end
endmodule
module top;
reg [0:3] a;
    reg clk;
    wire aa,b;
    assign aa=a[1];

    dut dt (clk,aa,b);
endmodule
module dut(input clk,a,output b);
    assign b = a;
endmodule
```

### **Example 5-11 frc.tcl**

```
run 1
catch {find_forces -scope test} err
puts $err
force test.a 3'b1010
catch {find_forces -scope test -level 0} err
puts $err
run 1
force test.dt.a 3'b0011
run 1
catch {find_forces -scope test.dt -file frc.txt} err
puts $err
exec cat frc.txt
run
quit
```

**Execute the following commands:**

```
%vcs -debug_access+r+cbk+f -sverilog -force_list
test.sv

%./simv -force_list -ucli -i frc.tcl
```

**Output:**

```
ucli% run 1
1 s
ucli% catch {find_forces -scope test} err
0
ucli% puts $err

ucli% force test.a 3'b1010
ucli% catch {find_forces -scope test -level 0} err
0
ucli% puts $err
{test.a[0]} freeze 'b1 Design
{test.a} freeze 'b1010 External

ucli% run 1
```

```
2 s
ucli% force test.dt.a 3'b0011
ucli% run 1
3 s
ucli% catch {find_forces -scope test.dt -file frc.txt} err
0
ucli% puts $err
{test.dt.a} freeze 'b1 External
Result successfully reported to: frc.txt
ucli% exec cat frc.txt
{test.dt.a} freeze 'b1 External
ucli% run
```

---

## Reporting \$deposit Value Changes

The `-force_list` option dumps only the `$deposit` value change events in the force list report. It does not dump regular value changes caused by the design activity. A unique ID is assigned for each `$deposit` statement in the force list report. This feature helps you to identify value changes caused by `$deposit`.

---

## Limitations

- This feature is not supported for pure VHDL designs.
- The `-R` option is not supported.
- Language forces from encrypted code are not reported.
- Nodes that are forced by more than one line in the Verilog source are not analyzed for the exact line that is driving the value. If a force/release event occurs on the node, VCS records the event, but not the file and line that exactly caused the event. VCS only lists the possible force drivers, not the exact driver.

- Force on entire mda is not supported for an event list capture. For example, consider the following code:

```
logic [1:0] foo [1:0][1:0] = 8'b11111111;
logic [1:0] baz [1:0][1:0];
initial begin
#1 force baz = foo;      //force entire mda baz
end
```

VCS does not record this value change on the force at #1, it just captures the header information for this force. VCS provides a comment in the header saying that value changes for this force event will not be recorded.

- Values in the event list for the language forces represent the current value at the time of the force, and not necessarily the forced value. This becomes an issue when multiple force statements influence one or more but not all of the same bits in the part selects and full vectors. As an example of this limitation, consider the following code:

```
logic foo [0:3] = 4'b0000;
Initial begin
#0 force foo[0] = 1;      //creates force list header
item 1 for the force on bit select of foo[0].
#1 force foo[0:2] = 'b111; //creates force list header
item2 for the force on part select foo[0:2], note bit 0.
#1 release foo[0:2] = 'b000; //creates a new header entry
using the same id as the above line.
#1 foo[0:2] = 'b000;      //reset bits
#1 force foo[0] = 1; //creates a new header entry using
same id as the first force on this node.
end
```

Following is the output report for the above code. Comments are added to the following report for illustration purposes only.

Language Forces	ID	Target	Module	File	Line
-----------------	----	--------	--------	------	------

```

1 top.foo[0] top test.v 11
2 top.foo[0:2] top test.v 12
2 top.foo[0:2] top test.v 17
1 top.foo[0] top test.v 18
          Event List Section
----- Time: 0 -----
2 LF 'b100      //at time 0, force occurs on id 1, so
values for bits 1 and 2 on id2 are not forced.
1 LF 'b1        //this is the real force at this time,
but forcelist cannot know whether the force is from id 1
or id 2.
----- Time: 1 -----
2 LF 'b111      // this is the real force, no issue here
because all values are forced.
1 LF 'b1
----- Time: 2 -----
2 LR           // release fires, this is ok.
----- Time: 4 -----
2 LF 'b100      //similar to time 0, bits 1 and 2 are not
forced but reported.
1 LF 'b1        //this is the real force.

```

---

## Key Runtime Features

Key runtime features includes:

- “Overriding Generics at Runtime”
- “Passing Values from the Runtime Command Line”
- “Saving and Restarting the Simulation”
- “Specifying Long Time Before Stopping the Simulation”
- “Using -f Runtime Option”
- “Resolving RTL Simulation Races in Mixed HDL Designs”
- “Resolving RTL Simulation Races in Verilog Designs”
- “Preventing Time 0 Race Conditions”
- “Supporting Simulation Executable to Return Non-Zero Value on Error Results”
- “Supporting Memory Load and Dump Task Verbosity”

---

### Overriding Generics at Runtime

Using the `-g`, `-gen` or `-generics` runtime option, you can change the following types of VHDL generics at runtime:

- Any generic that stays in VHDL and is not propagated directly or indirectly into Verilog
- Any generic that does not shape the tree or define the widths of ports through mixed HDL boundary

- Generics like delays, file names, and timing checks control

The use model is as follows:

```
% simv -g generics_file
```

The `-g`, `-gen` or `-generics` option takes a command file as an argument. You must specify the hierarchical path of the generic and a new value to override. A sample `generics_file` is as follows:

```
% cat generics_file
assign 1 /TOP/LEN
assign "OK.dat" /TOP/G1/vhdl1/FILE_NAME
assign (4 ns) /TOP/G1/VHDL1/delay
assign 16 /TOP/width
assign 4 /TOP/add_width
```

## Use Model

### Analysis

```
% vlogan [vlogan_options] file1.v file2.v
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

Note:

Specify the VHDL bottommost entity first, then move up in order.

### Elaboration

```
% vcs [vcs_options] top_cfg/entity/config
```

### Simulation

```
% simv [sim_options] -g cmd.file
```

## Example

Consider the following example:

```
--spmem.vhd---

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.All;

ENTITY spmem IS
generic ( add_width : integer := 3;
           delay : time := 2 ns;
           file_name : string := "empty.dat";
           WIDTH : integer := 8);

PORT (
      clk       : IN std_logic;
      reset     : IN std_logic;
      add      : IN std_logic_vector(add_width -1 downto 0);
      Data_In   : IN std_logic_vector(WIDTH -1 DOWNTO 0);
      Data_Out  : OUT std_logic_vector(WIDTH -1 DOWNTO 0);
      WR        : IN std_logic);
END spmem;

ARCHITECTURE spmem_v1 OF spmem IS

TYPE data_array IS ARRAY (integer range <>) OF
                           std_logic_vector(7 DOWNTO 0);

SIGNAL data : data_array(0 to (2** add_width) ) ;

BEGIN -- spmem_v1

PROCESS (clk, reset)
BEGIN -- PROCESS

  IF (reset = '0') THEN
    data_out <= (OTHERS => 'Z');

  ELSIF clk'event AND clk = '1' THEN
    IF (WR = '0') THEN


```

```

        data(conv_integer(add)) <= data_in after delay;
    END IF;
    data_out <= data(conv_integer(add));
END IF;

END PROCESS;

END spmem_v1;

--TOP.vhd---

library IEEE;
use IEEE.std_logic_1164.all;

entity top is
generic ( add_width : integer := 3;
           delay : time := 2 ns;
           file_name : string := "empty.dat";
           WIDTH : integer := 8;
           LEN : integer := 1  );
PORT (
      clk      : IN  std_logic;
      reset    : IN  std_logic;
      add      : IN  std_logic_vector(add_width -1 downto 0);
      Data_In  : IN  std_logic_vector(WIDTH -1  DOWNTO 0);
      Data_Out : OUT std_logic_vector(WIDTH -1  DOWNTO 0);
      WR       : IN  std_logic);
END top;

architecture top_arch of top is
component spmem
generic ( add_width : integer := 3;
           delay : time := 2 ns;
           file_name : string := "empty.dat";
           WIDTH : integer := 8);
PORT (
      clk      : IN  std_logic;
      reset    : IN  std_logic;
      add      : IN  std_logic_vector(add_width -1 downto 0);
      data_in  : IN  std_logic_vector(WIDTH -1  DOWNTO 0);
      data_out : OUT std_logic_vector(WIDTH -1  DOWNTO 0));
end component;

```

```

        WR      : IN  std_logic);
END component;

begin -- top_arch

G1: if LEN=1 generate
    INST1 : spmem generic map (add_width,delay,file_name,width)
                  port map (clk,reset,add,data_in,data_out,wr);
end generate G1;

G2: if LEN=2 generate
    INST2 : spmem generic map (add_width,delay,file_name,width)
                  port map (clk,reset,add,data_in,data_out,wr);
end generate G2;

end top_arch;

```

In the above example, you can override the generics at runtime. The use model is as follows:

## **Analysis**

```
% vhdlan spec_mem.vhd TOP.vhd
```

### **Note:**

Specify the VHDL bottommost entity first, then move up in order.

## **Elaboration**

```
% vcs TOP
```

## **Simulation**

```
% simv -g generics_file
```

Following is the `generics_file`:

```

assign 1 /TOP/LEN
assign "OK.dat" /TOP/G1/INST1/FILE_NAME
assign (4 ns) /TOP/G1/INST1/delay

```

```
assign 16 /TOP/width  
assign 4 /TOP/add_width
```

As per `generics_file`, VCS overrides the generics `LEN`, `width`, and `add_width` in the `TOP.vhd` file, and `FILE_NAME` and `delay` generics defined in the `spmem.vhd` file.

---

## 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 `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
```

---

## Using -f Runtime Option

You can use the `-f <filename.f>` runtime option to specify user-defined arguments in a file. These arguments are those that you specify on the `simv` command line. This option works for all mixed HDL designs, pure VHDL, and pure Verilog designs.

## Limitations

- Nested file inclusion is not supported
- Environment expansion is not supported
- Complex string options are not supported
- You cannot specify multiple options on the same line. This is illustrated in the below example:

```
%simv -f <filename.f>  
  
filename.f  
  
-ova_report  
-lca  
-cm_name foo  
...  
...
```

---

## Saving and Restarting the Simulation

You can use the `$save` and `$restart` system tasks to save the checkpoints of the simulation at arbitrary times. The resulting checkpoint files can be executed at a later time, causing simulation to resume at the point immediately following the save.

Note:

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

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

## Save and Restart Example

[Example 5-12](#) illustrates the basic functionality of save and restart.

The \$save call is scheduled and executed as the last event in the current time. Therefore, the events delayed with #1 are the first to be processed upon restart.

### *Example 5-12 Save and Restart Example*

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

Now compile the example source file:

```
% vcs test.v
```

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:

```
% simv -r test.chk
```

VCS displays the following:

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

## Save and Restart File I/O

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

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

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

## Save and Restart With Runtime Options

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

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

The following example illustrates this in greater detail. The first run optimizes simulation speed by omitting the `+dump` option. If a bug is found, the latest checkpoint file is run with the `+dump` option 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");
        #1 processargs;
    end
endmodule
// The design itself here
module top();
    ....
endmodule

```

---

## Specifying Long Time Before Stopping the Simulation

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

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

`+vcs+stop+<first argument>+<second argument>`

`+vcs+finish+<first argument>+<second argument>`

This procedure is as follows:

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

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

In this example:

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

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

In this example, you would narrow down to 2.

3. Multiply  $2^{32}$  with the second argument (that is, 2), and then subtract the obtained result from the large time value (that is, subtract  $2 \times 2^{32}$  from the large *time* value), as shown below:

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

This difference is the first argument.

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

+vcs+stop+1410065408+2

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

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

VCS displays the following:

Hi=2, Lo=1410065408

---

## Preventing Time 0 Race Conditions

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

For example, consider the following code:

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

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

    assign q1=rq1;
    assign q2=rq2;

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

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

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

---

## Resolving RTL Simulation Races in Mixed HDL Designs

A race between data and clock signal occurs when both signals change at the same simulation time and both are input to the same sequential element (flip-flop or latch). However, it is expected that the clock arrives before data and samples the previous settled value of data. When clock arrives after or at the same cycle as data, the new value of data is sampled which causes incorrect results.

VCS helps resolve these RTL simulation races in mixed VHDL-Verilog designs. The following section illustrates how to resolve race conditions.

## Recommended Approach to Resolve Race Conditions

It is recommended to use the following methodologies to resolve the race conditions:

1. Use the VHDL `-sn=+rdr` race detector option and try to manually fix the RTL code. If it is not possible to modify the RTL code, use the race detector output as an input to the configuration file. For the use model, see section “[Using the VHDL Race Detector](#)”.
2. Use the `-sn=+rfx` automatic option to accelerate the scalars (clocks) over the boundary and insert NBA delays only for the data drivers that directly provide stimulus to the Verilog boundary. For information about the using the configuration file, see section “[Using the Race Configuration File](#)”.

## Using the VHDL Race Detector

If a simulation race is already detected, delay the data to fix the race. To resolve the simulation race using the VHDL race detector, execute the following steps:

1. To insert delays, modify the VHDL code and add “`after 0.0 sec`” on drivers for data signals reported as racy. The delay can be a non-zero delay or a non-blocking assignment zero delay specified as follows in the source code:

```
signal <= data after 0.0 sec;
```

This is legal VHDL and works with any tool. However, VCS recognizes it as a VHDL Non-Blocking Assignment (NBA). To enable VHDL NBA assignment, use the `vcs -sn=+nbavhd` option.

2. Enable the race detector using the `-sn=+rdr` option, as shown in the following command:

```
% vcs -sn=+rdr; simv
```

This option generates the `snps_vcs_vhdl_race.log` file, which can be used as an input to the configuration file to insert NBA delays on those signals.

## Using the Race Configuration File

To enable the configuration input, use the `-sn=+nbacfg` option with `snps_vcs_vhdl_nba.cfg` as an input file. The configuration input file contains a list of signals for which NBA delays are inserted.

The configuration file with the NBA delays can be generated by the race detector using the following commands:

```
% vcs -sn=+rdr; simv  
% cp snps_vcs_vhdl_race.log snps_vcs_vhdl_nba.cfg  
% vcs -sn=+nbacfg; simv
```

OR

The configuration file with the NBA delays can be generated by the automatic race solution along with the log file (`-sn=+rfx+nbalog`). This can be used as a starting list and enables you to tune the configuration list:

```
% vcs -sn=+rfx+nbalog; simv  
% cp snps_vcs_vhdl_nba.log snps_vcs_vhdl_nba.cfg  
% vcs -sn=+nbacfg; simv
```

---

## Resolving RTL Simulation Races in Verilog Designs

A race between data and clock signal occurs when both signals change at the same simulation time and both are input to the same sequential element (flip-flop). However, it is expected that the clock

arrives before data and samples the previous settled value of data. When clock arrives after or at the same cycle as data, the new value of data is sampled which causes incorrect results.

VCS helps resolve these RTL simulation races in Verilog design. The following section illustrates how to resolve race conditions.

## **Recommended Approach to Resolve Race Conditions**

It is recommended to use the Verilog `-deraceclockdata` option to enable the clock-data resolution for your entire design. For more information, see section “[Using Clock-Data Resolution](#)”.

### **Using Clock-Data Resolution**

Using clock-data resolution ensures that the previous value of the data is always sampled.

This significantly improves the verification productivity. There is no simulation mismatch due to races on flops while migrating to a new release or modifying options that are provided to the VCS command line.

### **Use Model**

To enable the clock-data resolution for your entire design, use the Verilog `-deraceclockdata` option.

```
% vcs -deraceclockdata <other vcs options>
```

By default, the `-deraceclockdata` option samples memory up to 8 MB. You can use the `-deraceclockdata=fullmem` option to remove the restriction on the memory size.

## Example

Consider the following test case:

```
module dff(q, d, clk, clr);
    output q;
    input d, clk, clr;
    reg q;

    always @ (posedge clk ,negedge clr) begin
        if (!clr)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule

//=====top module=====
module top(clk, d, clr, out);
    input clk, d, clr;
    output out;
    wire clk2;
    wire q1;

    dff div2(clk2, ~clk2, clk, clr);
    dff f1(q1, d, clk, clr);
    dff f2(out, q1, clk2, clr);
endmodule

//=====testbench=====
module tb;
    reg clk;
    reg clr;
    reg d;

    wire dout;
    top t1(clk,d,clr,dout);
    initial begin
        clk = 1'b0;
        d = 1'b0;
        clr <= 1'b0;
```

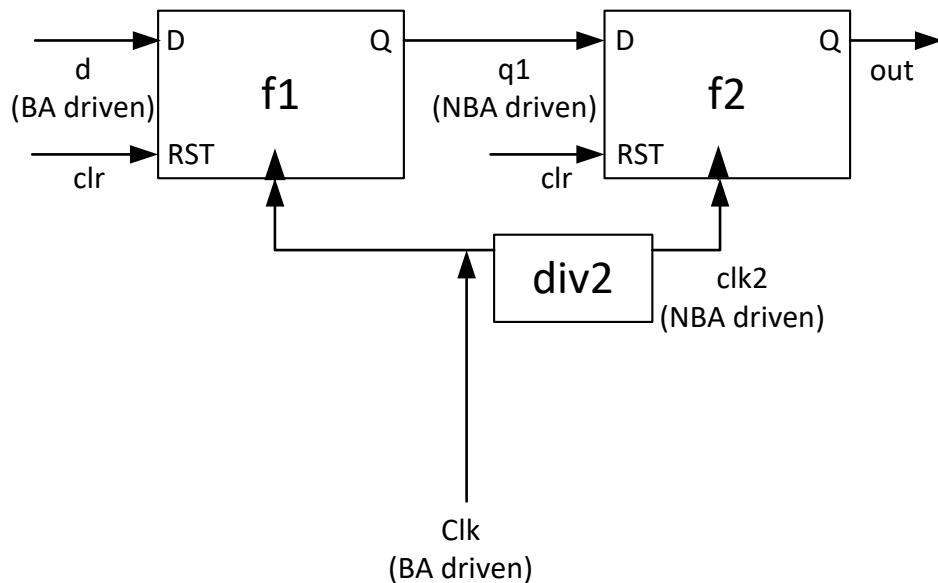
```

fork
    forever clk = #5 ~clk;
    #25 d = 1'b1;
    #45 d = 1'b0;
    #9 clr <= 1'b1;
join_none

#70 $finish();
end
endmodule

```

This example has three flops. For f1 flop, the clock and the data changes at the same time in the blocking assignment region. For f2 flop, both clock and data are changing in the NBA region.



Compile and run the test case using the following command line:

```
% vcs -sverilog -deraceclockdata test.v
% simv
```

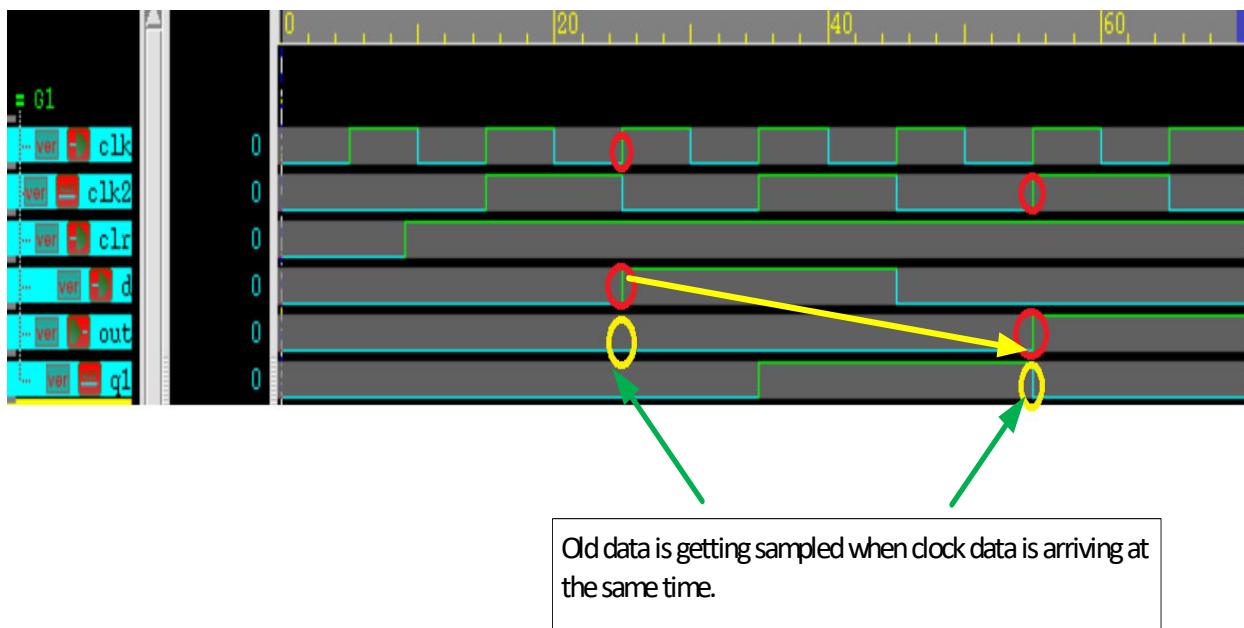
To compile the test case with FSDB dumping enabled, use the following command line:

```
% vcs -sverilog -deraceclockdata test.v -debug_access  
-debug_region+cell +vcs+fsdbon
```

Note:

You need to set the `VERDI_HOME` environment variable for FSDB dumping.

Import the design and load the generated FSDB file into the Verdi platform. The following waveform is generated:



As shown in the waveform, it is able to consistently sample the clock first using the `-deraceclockdata` option. Therefore, it picks up the previous value of the data.

## Limitations

The feature has the following limitations:

- The feature is supported for inferred flops in Verilog only.
- Process blocks containing following constructs are ignored:
  - Call to impure function
  - Containing delays
  - Having immediate assertions inside
  - Dynamic variables
  - System Task Calls other than \$display and \$monitor
- Flops modeled using both blocking assignment region and non-blocking assignment region (in same process) are not supported.
- UDPS with synchronous control are not supported.

---

## **Supporting Simulation Executable to Return Non-Zero Value on Error Results**

Simulation executable generated by VCS returns non-zero value in case of errors, fatal errors, and assertion failures.

The simulation executable return values on errors, fatal errors, and assertion values are:

- 0 (no indication)
- 1 (as in runtime crash or system crash)
- 2 (error)
- 3 (fatal)

The possible scenarios and return error value for the scenarios are listed in the following table:

*Table 5-5 List of Scenarios and Return Error Values*

Scenario	Return Error Value
\$fatal/UVM_FATAL/OVM_FATAL/VMM_FATAL	3
\$error/UVM_ERROR/ OVM_ERROR/ VMM_ERROR/ Errors promoted from warning messages to errors	2
NLP ERROR	2
Assertion failure Verilog	2
\$warning /UVM_WARNING/ OVM_WARNING/ VMM_WARNING	0
NLP WARNING	0
Unique/priority RT warnings	0
-xzcheck	0

Note:

The `-assert quiet` and `-assert quiet1` runtime options cannot override the exit status in case of assertion failures. The exit status value still must be value 2 in case of assertion failures.

Use of `-error` runtime option generates non-zero return values in case of errors, fatal errors, as well as in case of errors resulting from warning messages.

If the messages are suppressed using the `-error` runtime option, non-zero return values are not generated.

If a simulation has several errors, fatal errors, and/or assertion failures, the most severe status must be returned. For example, if simulation has both `$error` and `$fatal` messages, the returned status must be of value 3 as in case of `$fatal` scenario.

## Use Model

The following is the use model for this feature:

- Compile time

The compile time is same as the previous use model. There is no change needed.

- Runtime

```
% simv -exitstatus  
% echo $status
```

This returns a value based on the type of exit.

## Limitations

The following are the limitations with this feature:

- VHDL assertions are not supported
- VMM is not supported

---

## Supporting Memory Load and Dump Task Verbosity

If you use the `-diag sys_task_mem` compile-time option, `$writememh`, `$writememb`, `$readmemh` and `$readmemb` system tasks get into a verbose mode and displays the following information:

- Full path of the file being written by `$writememh` and `$writememb` and being read by `$readmemh` and `$readmemb`, such as `/foo/bar/filename`.

- The full hierarchical instance path of the module from where \$writememh, \$writememb, \$readmemh, and \$readmemb is invoked, such as soc.a1.b1.c1.my\_module\_instance. All instances are reported.
- The name of the module template from where it is displayed, such as my\_module.
- The full path to the file that includes the module that contains the \$writememh, \$writememb, \$readmemh, and \$readmemb, such as /baz/qux/my\_module.vs.

Syntax:

```
$writememh ( filename , memory_name [ , start_addr [ ,  
finish_addr ] ] ) ;
```

```
$writememb ( filename , memory_name [ , start_addr [ ,  
finish_addr ] ] ) ;
```

The system tasks \$writememh and \$writememb dump memory array contents to files that are readable by \$readmemh and \$readmemb respectively.

```
$readmemh ( mem_name , start_address , finish_address ,  
string { , string } ) ;
```

```
$readmemb ( mem_name , start_address , finish_address ,  
string { , string } ) ;
```

The system tasks \$readmemh and \$readmemb load data into memory mem\_name from a character string.

The \$readmemh and \$readmemb system tasks take memory data values and addresses as string literal arguments. These strings take the same format as the strings that appear in the input files and are

passed as arguments to \$readmemh and \$readmemb. The start\_address and finish\_address indicate the bounds for the data to be stored in the memory.

## Use Model

The following is the use model to support memory load and dump task verbosity:

```
%vcs -diag sys_task_mem
```

The following examples illustrate the usage of \$writemem and \$readmem system tasks:

*Example 5-13 Example to illustrate \$writemem system task*

```
//test.v
module test;
reg [31:0] mem[0:11];

initial begin
    $writememh("./datah.dat", mem);
end
endmodule
```

Run the example using the following commands:

```
% vcs test.v -diag sys_task_mem
% simv
```

It generates the following output:

```
Note-[STASK_WMEM] Encountered Memory Write Task
/home/user/task/test.v, 5
    At module test, Instance test
        Writing to file /home/user/task/datah.dat.
```

*Example 5-14 Example to illustrate \$readmem system task*

```
//test.v
```

```
module test;
reg [31:0] mem[0:11];

initial begin
    $readmemh("./data.dat", mem);
end
endmodule
```

**Run the example using the following commands:**

```
% vcs test.v -diag sys_task_mem
% simv
```

**It generates the following output:**

```
Note-[STASK_RMEM] Encountered Memory Read Task
/home/user/task/test.v, 5
At module test, Instance test
Reading from file /home/user/task/data.dat.
```



# 6

## The Unified Simulation Profiler

---

The unified simulation profiler reports the amount of CPU time and machine memory used by the Verilog, SystemVerilog, and VHDL parts of the design. For SystemC parts, the unified profiler reports just the CPU times.

This information can be in summary form and per module definition and module instance. It also can be based on constructs, such as `always` procedures.

The reports are written by the `profrpt` profile report generator. These reports can be in text or HTML format or both.

Note:

You need a correct installation of Python version 2.3 or newer for `profrpt` to generate HTML reports. If the version of Python is older than v 2.3, or the installation was done incorrectly, then `profrpt` might not function properly.

The major sections in this unified profiler documentation are as follows:

- “The Use Model”
- “HTML Profiler Reports”
- “Constraint Profiling Integrated in the Unified Profiler”
- “Performance/Memory Profiling for Coverage Covergroups”
- “Limitations”

For examples of SystemC profiler reports, see “[SystemC Views](#)” .

---

## The Use Model

The use model for the unified simulation profiler is as follows:

1. Compile your design using the `-simprofile` compile-time option.

### **Important:**

If this is not the first compilation of your design, delete the `csrc` and `simv.daidir` directories and `simv` executable file before this step. Incremental compilation is not yet supported for the unified profiler.

2. At runtime, you can enter the `-simprofile` runtime option with a keyword argument or sub-option to specify the type of data VCS collects during the simulation. These keyword arguments or sub-options are as follows:

`time`

The `time` argument specifies collecting CPU time profile information.

`mem`

The `mem` argument specifies collecting machine memory profile information.

`noprof`

Tells VCS not to collect profiling information at runtime. Synopsys recommends entering this runtime option and keyword argument or sub-option instead of simply omitting the `-simprofile` runtime option. See “[Omitting Profiling at Runtime](#)” .

`noreport`

Tells VCS to collect profile information at runtime but not write the `profileReport.html` file or the `profileReport` directory after simulation, see “[Omitting Profile Report Writing after Runtime](#)” .

After simulation, but before you run the `profrpt` profile report generator there is a summary of profile information available to you, see “[Post Simulation Profile Information](#)” .

3. Run the `profrpt` profile report generator using the `profrpt` command and its command-line options.
4. Review the reports created by the `profrpt` profile report generator; see “[Running the profrpt Profile Report Generator](#)” for more information.

---

## Omitting Profiling at Runtime

If you compiled the design to collect profile data by entering the `-simprofile` compile-time option, but decide to forgo the performance cost of collecting profile data during simulation, you enter the `-simprofile noprof` runtime option and the keyword argument.

When you do so, VCS does not create the `profileReport.html` file or the `profileReport` directory, but does create the `simprofile_dir` directory. However, this `simprofile_dir` directory remains empty.

Omitting the `-simprofile` runtime option after compiling with the `-simprofile` compile-time option is not recommended.

If you compile your design with the `-simprofile` compile-time option, but omit the `-simprofile` runtime option when you run the simulation, VCS by default, creates the `simprofile_dir` and `profileReport` directories and write the `profileReport.html` in the current directory that only contains information about the simulation time.

The `simprofile_dir` directory never contains any information that you can read. It does contain database files that come with a performance cost.

If `simprofile_dir`, the directory `profileReport`, and the file `profileReport.html` exist and you run the test, then VCS renames them to `simprofile_dir.integer`, `profileReport.integer`, `profileReport.integer.html` respectively, so that the files are not overwritten.

---

## Omitting the `-simprofile` Runtime Option

The `-simprofile` compile-time option has optional arguments. With no arguments its purpose is to enable VCS to collect profile information at runtime. Also at runtime, you can also enter the `-simprofile` option along with the `time` or `mem` arguments.

For example:

```
%> simv -simprofile time
```

or

```
%> simv -simprofile mem
```

The `time` argument allows you to collect CPU time profile information.

The `mem` argument allows you to collect machine memory profile information.

Note:

Synopsys does not recommend collecting both CPU time and machine memory profile information in the same simulation.

The `-simprofile` compile-time option also has optional arguments:

```
-simprofile=time
```

Specifies compiling the design and testbench for collecting both CPU time and machine memory profile information. Then at runtime specifies collecting CPU time profile information.

`-simprofile=mem`

Specifies compiling the design and testbench for collecting both CPU time and machine memory profile information. Then at runtime, specifies collecting machine memory profile information.

With these arguments, you can omit the `-simprofile` runtime option if you want to collect the type of profile information that you have specified at compile time.

If at runtime you want VCS to collect different types of profile information than the type you specified at compile time, you can specify different type with the runtime option.

For example:

```
%> vcs source.v -simprofile=time
```

```
%> simv -simprofile mem
```

---

## Omitting Profile Report Writing after Runtime

If you have compiled your design to collect profile data and want VCS to collect profile data during simulation, but you do not want VCS to write the `profileReport.html` file or the `profileReport` directory, enter the `-simprofile noreport` runtime option and the keyword argument.

When you do this you still have the profile database and can obtain profile information after simulation with a `profrpt` command line.

---

## Specifying a Directory for the Profile Database

By default, VCS creates the profile database and the directory named `simprofile_dir` that contains all the profile information gathered during simulation, in the directory that contains the `simv` executable.

You can specify a different directory for the profile database with the `-simprofile_dir_path pathname` runtime option.

For example:

```
% simv -simprofile time -simprofile_dir_path /tmp/SUBDIR1
```

To use this option, the directories in the pathname must already exist. VCS does not create the directories in the pathname.

---

## Post Simulation Profile Information

After simulation, but before you run the `profrpt` profile report generator, VCS provides a summary report of profile information.

At the end of simulation VCS writes the `simprofile_dir` and `profileReport` directories and the `profileReport.html` file.

The `simprofile_dir` directory contains the databases that are read by the `profrpt` profile report generator to write profile reports in a separate step after the simulation.

The `profileReport.html` file can tell you the total simulation time and the location of the profiler databases.

---

## Specifying the Name of the Profile Report

By default, VCS writes the profile report named `profileReport.html` and the corresponding `profileReport` directory that contains the profile report information. You can enter the `-simprofile_report reportname` runtime option to specify a different name for this file and directory.

For example:

```
% simv -simprofile_report memory_rppt_default_constraints
```

This example creates the following profile report and report directory named `memory_rppt_default_constraints.html` file and `memory_rppt_default_constraints` directory respectively.

---

## Running the profrpt Profile Report Generator

You run the `profrpt` profile report generator with the `profrpt` command line. The syntax of this command line is as follows:

```
profrpt simprofile_dir -view view1[+view2[+...]]  
[-h|-help] [-format text|html|ALL] [-output <name>]  
[-filter percentage] [-snapshot [delta|incr|delta+incr]]  
[-timeline [dynamic_memory_type_or_class +...]]
```

Where:

`simprofile_dir`

Specifies the profile database directory that VCS writes at runtime. The default name is `simprofile_dir`. You can enable the writing of this database with the `-simprofile` compile-time option and specify the kind of data in the database with the `-simprofile` runtime option.

`-view view1 [+view2 [+...]]`

Specifies the views you want to see in the reports, see “[Specifying Views](#)”. You must specify this option.

`-h | -help`

Displays help information about the `profrpt` command-line options.

`-format text | html | ALL`

Specifies whether the report files are text files, HTML files, or in both formats (by specifying the `ALL` keyword). The default format is HTML. Some views, like the accumulative views, are only available in HTML format.

`-output <name>`

Specifies the name of the output directory for the profile report. If the report format is in HTML (which is the default format), `profrpt` writes in the current directory that is an HTML index file with the name `<name>.html`. This HTML index file contains hypertext links to the HTML files in the output directory.

If you omit the `-output` option, the default name of the output directory is `profileReport` and the default name of the HTML index file is `profileReport.html`.

Any currently existing `profileReport` directory and `profileReport.html` file is renamed by `profrpt` to `profileReport.integer` directory and `profileReport.integer.html`. The integer value is incremented to differentiate it from the current `profrpt` output.

For more information on the `-output` option, see “[The Output Directories and Files](#)” .

`-filter percentage`

Specifies the minimum percentage of the machine memory or the CPU time that a module, instance, or construct needs to use before `profrpt` enables reporting about it in the output views and reports. The default limit is 0.5%. For a more granular report enter a small percentage, such as `-filter 0.0001`.

Note:

Filtering is not applicable to the time and memory summary views and the dynamic memory timeline report.

`-snapshot [delta|incr|delta+incr]`

Specifies writing a series of snapshot profile reports for SystemVerilog dynamic memories. It writes a snapshot report each time a dynamic memory uses a specified different amount of machine memory. For information on specifying this amount, and more on the snapshot mechanism, see “[The Snapshot Mechanism](#)” .

`-timeline [dynamic_memory_type_or_class +...]`

Specifies two things:

- Timeline reports for SystemVerilog dynamic memories.
- Snapshot reports using the default delta threshold of 5%.

If you omit the *dynamic\_memory\_type\_or\_class +...* argument or arguments, `profrpt` writes all the dynamic class timeline views.

For information on the keyword arguments or sub-options for specifying the types of SystemVerilog dynamic memories in the timeline reports, see “[Specifying Timeline Reports](#)” .

---

## Specifying Views

You must enter the `-view` option on the `profrpt` command line.

The views you can specify with the `-view` option depend on the type of report that `profrpt` is writing, which depends on the argument to the `-simprofile` runtime option.

The arguments and the views that you can specify are as follows:

CPU Time views:

`time_summary`

To specify writing the time summary view.

`time_inst`

To specify writing the time instance view that shows the CPU time used by the various module, program, and interface instances in a design. For VHDL, this view also reports the CPU time used by the various entity/architecture instances in a design.

`time_mod`

To specify writing the time module view that shows the CPU time used by the various module, program, and interface definitions in a design. For VHDL, this view also reports the CPU time used by the various entity/architecture definitions in a design.

`time_constr`

To specify writing the time construct view that shows the CPU time used by constructs, such as the `always` procedures.

`time_solver`

To specify generating the Time Constraint Solver view.

`time_callercallee`

To specify the Caller/Callee view for CPU time information.

`time_all`

Specifies writing all the supported CPU time views.

Machine memory views:

`mem_summary`

To specify writing the peak memory summary view, which is when your design used the most machine memory.

`mem_inst`

To specify writing the peak memory instance view.

`mem_mod`

To specify writing the peak memory module view.

`mem_constr`

To specify writing the peak memory construct view.

`dynamic_mem`

To specify writing the dynamic memory peak view.

`dynamic_mem+stack`

To specify writing the dynamic memory peak view, and machine memory stack traces. The stack traces can help you determine which callers consume the most memory. For more information, see “[Stack Trace Report Example](#)” .

`mem_solver`

To specify generating the Memory Constraint Solver view.

`mem_callercallee`

To specify the Caller/Callee view for the machine memory information.

`mem_all`

To specify writing all the supported machine memory views. This argument also enables machine memory stack traces.

Both memory and time profiler:

`ALL`

To specify writing all supported views. The `profrpt` output is the HTML or text files for all these views, including machine memory stack traces.

---

## The Snapshot Mechanism

The keyword arguments, or sub-options, that you include after the `-snapshot` option control the snapshot mechanism. They are as follows:

*delta*

A numerical value (not a keyword) specifying the delta threshold for another snapshot. For example, `-snapshot 8.5` specifies a delta threshold of 8.5%. Thus, `profrpt` writes another snapshot report when a dynamic memory uses 8.5% more machine memory or 8.5% less machine memory.

*incr*

A keyword specifying the generation of another snapshot only when the machine memory for a SystemVerilog dynamic memory increases by 5%.

*delta+incr*

Specifies another snapshot when the amount of machine memory used by a SystemVerilog dynamic memory increases (but not decreases) by the specified delta threshold.

If you enter no arguments or sub-options, the profiler uses the default delta threshold of 5%, and enables a new snapshot when the amount of machine memory used by a SystemVerilog dynamic memory increases or decreases by 5%.

---

## Specifying Timeline Reports

The `-timeline` option specifies writing timeline reports. The keyword arguments or sub-options that you include after the `-timeline` option specify the types of SystemVerilog dynamic memories in the timeline reports. You can also specify a SystemVerilog class by name. Its dynamic memories are included in the timeline reports.

The arguments or sub-options for the `-timeline` option are as follows:

`vcs_ST`

keyword for string dynamic memories

`vcs_ET`

keyword for event dynamic memories

`vcs_DA`

keyword for dynamic arrays

`vcs_SQ`

keyword for queues

`vcs_AA`

keyword for associative arrays

`class`

a class name, not a keyword, specifying a class

`ALL`

keyword specifying all types of dynamic memories

If you enter the `-timeline` option without an argument or sub-option, `profrpt` writes timeline reports for all dynamic memories. Thus, the keyword `ALL` as an argument or sub-option is same as entering no argument or sub-option.

---

## Recording and Viewing Memory Stack Traces

You can use the unified profiler to record stack traces whenever machine memory is allocated. The stack traces can help you determine which callers consume the most memory.

You can enable memory stack traces with the `dynamic_mem+stack`, `mem_all`, or `ALL` arguments to the `profrpt -view` option. For more information, see “[Stack Trace Report Example](#)” .

---

## Reporting PLI, DPI, and DirectC Function Call Information

Profile information is reported for each PLI, DPI, or DirectC function called by your Verilog or SystemVerilog code.

There are views for this profile information. These views are in the menu for views in the left pane. The new views are as follows:

- Time PLI/DPI/DirectC (see “[Profiling Time Used by Various Parts of the Design](#)” ).
- Memory PLI/DPI/DirectC (see “[Profiling Memory Used by Various Parts of the Design](#)” ).

You can tell the `profrpt` report generator to write these views with the `time_pli` or `mem_pli` arguments (or sub-options) to the `-view` option.

The `profrpt` report generator also writes the Time PLI/DPI/DirectC view with the `time_all` and `ALL` arguments. It also writes the Memory PLI/DPI/DirectC view with the `mem_all` or `ALL` arguments.

SystemC code is not included in these views.

This profile capability has the following limitations:

- Details of VHDL code that calls external language functions are not reported.
- The location of the external language call is not reported.
- Text format reports are not supported; only the HTML format is supported.

## Compiling and Running the Profiler Example

The following example illustrates the new runtime and memory usage reporting.

If your Verilog code contains user-defined system tasks for PLI functions like those shown in [Example 6-1](#) for a file named `pli.v`:

### *Example 6-1 Verilog System Tasks for PLI Functions*

```
module top;
initial
begin
    $foo_200M();
    $foo_400M();
    $foo_200M();
    $foo_400M();
```

```

$foo_200M();
$foo_400M();
$foo_200M();
$foo_400M();
end
endmodule

```

Then, the pli.tab file looks like [Example 6-2](#).

### *Example 6-2 PLI Tab File*

```

$foo_200M call=foo_200M_func
$foo_400M call=foo_400M_func

```

The C code in example file pli.c looks like [Example 6-3](#).

### *Example 6-3 C File*

```

#include "stdio.h"
void foo_200M_func()
{
    char *a = NULL;
    int i = 0;
    for (i = 0; i < 100; i++)
        a = (void *) malloc (1024*2048);
}
void foo_400M_func()
{
    char *a = NULL;
    int i = 0;
    for (i = 0; i < 100; i++)
        a = (void *) malloc (2048*2048);
}

```

To compile these example files, use the following command line:

Two-step Flow:

```
% vcs -simprofile -P pli.tab pli.v pli.c
```

Three-step Flow:

```
% vlogan pli.v  
% vcs top -simprofile -P pli.tab pli.c
```

## Profiling Time Used by Various Parts of the Design

To profile the CPU time used by various parts of the design, use the `-simprofile time` option:

```
% simv -simprofile time
```

VCS creates:

- `simprofile_dir` database directory for the CPU time profile information.
- `profileReport.html` file and `profileReport` directory for the CPU time post-simulation profile report information.

To generate the Time PLI/DPI/DirectC view, use the `profprt` command with the following options:

```
% profprt -view time_pli simprofile_dir
```

To view the Time PLI/DPI/DirectC reports, open the `profileReport.html` file. In the left pane, select the `simprofile_dir` database and the Time PLI/DPI/DirectC view (see [Figure 6-1](#)).

This view shows the following:

- The `$foo_200M()` and `$foo_400M` PLI user-defined system tasks that call the `foo_200M_func()` and `foo_$00M_func()` C functions.
- CPU time that they use.
- Percentage of the total CPU time of the simulation.

*Figure 6-1 Time PLI/DPI/DirectC View*

---

Time PLI/DPI/DirectC View		
Name	Time	Percentage
PLI		
\$foo_200M	2.97 ms	1.56 %
<a href="#">\$foo_200M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v4</a>	2.97 ms	1.56 %
total	2.97 ms	1.56 %

Page: 1

## Profiling Memory Used by Various Parts of the Design

To profile the machine memory used by various parts of the design, simulate using the `-simprofile mem` option:

```
% simv -simprofile mem
```

To generate the Memory PLI/DPI/DirectC view, use the `profrpt` command with the following options:

```
% profrpt -view mem_pli simprofile_dir
```

To view the Memory PLI/DPI/DirectC reports, open the `profileReport.html` file. In the left pane, select the `simprofile_dir` database and the Memory PLI/DPI/DirectC view (see [Figure 6-2](#)).

This view shows the following:

- The `$foo_200M()` and `$foo_400M` PLI user-defined system tasks that call the `foo_200M_func()` and `foo_$00M_func()` C functions.
- Machine memory that they used.
- Percentage of the total machine memory needed during the simulation.

*Figure 6-2 Memory PLI/DPI/DirectC View*

Memory PLI/DPI/DirectC View (clock:10)			
	Name	Size	Percentage
PLI		3600.00 MB	99.06 %
	\$foo_400M	2400.00 MB	66.04 %
	\$foo_400M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:5	400.00 MB	11.01 %
	\$foo_400M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:7	400.00 MB	11.01 %
	\$foo_400M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:9	400.00 MB	11.01 %
	\$foo_400M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:11	400.00 MB	11.01 %
	\$foo_400M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:14	400.00 MB	11.01 %
	\$foo_400M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:16	400.00 MB	11.01 %
	\$foo_200M	1200.00 MB	33.02 %
	\$foo_200M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:4	200.00 MB	5.50 %
	\$foo_200M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:6	200.00 MB	5.50 %
	\$foo_200M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:8	200.00 MB	5.50 %
	\$foo_200M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:10	200.00 MB	5.50 %
	\$foo_200M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:13	200.00 MB	5.50 %
	\$foo_200M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:15	200.00 MB	5.50 %
	total	3600.00 MB	99.06 %

Page: 1

Here, the clock reference (clock:0) specifies the clock cycle of the peak memory usage (in this example, time 0).

---

## The Output Directories and Files

The `-output <name>` option and the argument specifies two things:

- The name of the directory for the profiler reports

- If the profiler is writing HTML reports (the default format), the name of the HTML index file that the `profrpt` writes in the current directory. This index file is `<name>.html`.

If you omit the `-output` option, the default name of the output directory is `profileReport` and the default name of the HTML index file is `profileReport.html`.

As explained in “[Post Simulation Profile Information](#)”, VCS writes the `profileReport` directory and the HTML index file `profileReport.html` at the end of simulation. So, if you omit the `-output` option, `profrpt` renames this directory and file `profileReport.integer` and `profileReport.integer.html` and then writes a new `profileReport` directory and `profileReport.html` file. This new directory and the file contain post-processing information from the database.

If the specified directory and file already exists, a warning message is generated and the `profrpt` creates a new output directory and file and renames the older output `name.integer` and `name.integer.html` to differentiate them from the new directory and file.

## The Enhanced Accumulative Views

The accumulative views displaying the accumulated CPU time or the machine memory profile information from two or more databases have been expanded to more accumulative views. These new views are:

- the accumulative summary view
- the accumulative module view

- the accumulative instance view
- the accumulative construct view

To generate a report showing the accumulated results from more than one profile database, follow these steps:

1. Write a file that lists the profile databases.
2. Run the `profrpt` profile report generator without entering a profile database on the command line. Instead, enter the `-f` option specifying the file that lists the databases.

```
% profrpt -view time_all -f time_db_list \
-output accum_time
```

In this example, the `-f` option specifies the file `time_db_list`, which contains a list of databases:

```
simprofile_dir
simprofile_dir.1
simprofile_dir.2
```

The `profrpt` profile report writing utility writes `accum_time.html`.

The `profrpt` utility only writes this accumulative view in HTML format. The text file format is not supported.

*Figure 6-3 The Right Pane of the Accumulated CPU Time View in accum\_time.html*

Time Summary		
Component	Time	Percentage
KERNEL	779.6 ms	51.97%
VERILOG	700.0 ms	46.67%
HSIM	19.8 ms	1.32%
PROFILE	622.0 us	0.04%
<b>total</b>	<b>1.5 s</b>	<b>100%</b>

This view shows the accumulated CPU times.

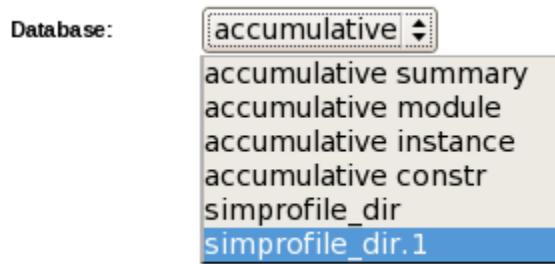
*Figure 6-4 The Left Pane of the Accumulated CPU Time Views in accum\_time.html*



The **Database** drop-down menu contains:

- selections for each of the accumulated databases
- selections for the module, instance and construct points of view.

*Figure 6-5 The Database Drop-Down Menu in the Left Pane*



The following selections display the accumulated information from different points of view:

### **module summary**

Contains a section for each module definition.

### **instance summary**

Contains a section for each module instance.

### **construct summary**

Contains a section for each type of construct, such as initial and always blocks.

Also from the drop-down menu, you can select the accumulated simulation time profile database. If you select one of these databases, the view changes to the time summary view for that database.

This example shows how to generate accumulative views of the CPU time profile data. The same can be done for machine memory profile data.

**Example 6-4 Code Example for the Accumulative View**

```
// test.v
module dut (input reg in[0:3], output reg out);
wire c1, c2;
assign c1 = in[0] & in[1];
assign c2 = in[2] & in[3];
or o1 (out, c1, c2);
endmodule

//tb.v
module tb1;
reg in[0:3], out;

dut d1 (in, out);
initial begin
in[0]=1; in[1]=0; in[2]=1; in[3]=0;
end
always #5 in[0] = ~in[0];
always #6 in[1] = ~in[1];
always #7 in[2] = ~in[2];
always #8 in[3] = ~in[3];
initial begin
$monitor($time,"in[0]=%b, in[1]=%b, in[2]=%b, in[3]=%b,
out=%b\n", in[0], in[1], in[2], in[3], out);
end
endmodule
```

Also for this example you have the following -i UCLI command files:

```
// run1
run 10000
quit

// run2
run 100000
quit
```

To run this example, enter the following command lines:

```
% vlogan -nc -sverilog test.v tb1.v
```

```
% vcs tb1 -sverilog -nc -simprofile
```

```
% simv -simprofile time -ucli -i run1
```

```
% simv -simprofile time -ucli -i run2
```

```
% profrpt -view time_all -f file
```

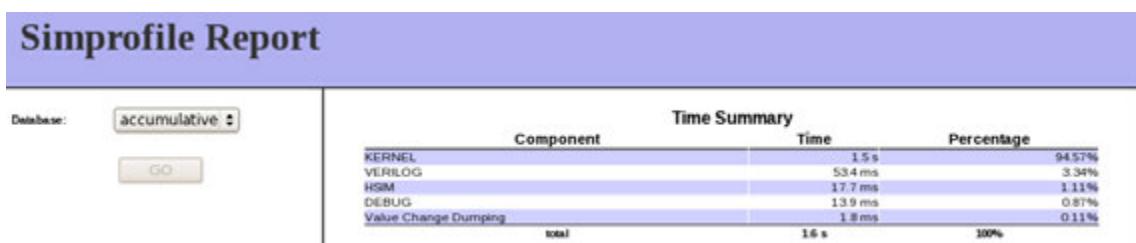
VCS writes the first profile database

VCS writes the second profile database

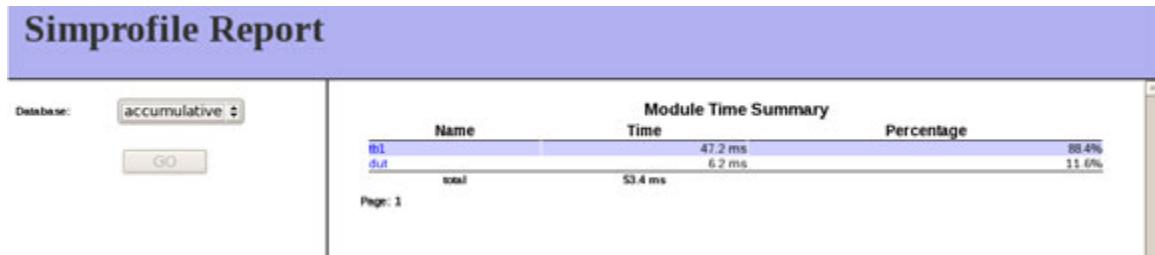
The `time_all` argument specifies, among other views, the accumulative time view.

Open the `profileReport.html` file to see the accumulative time view

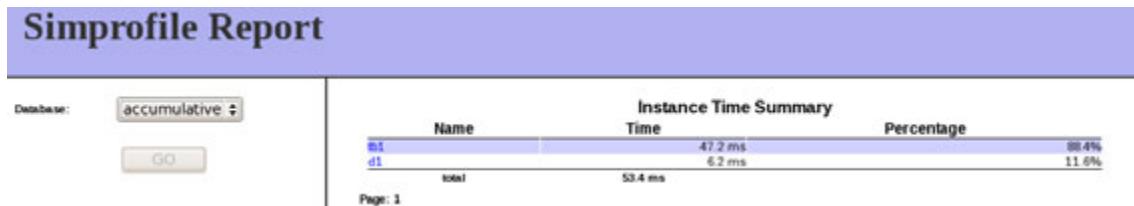
Figure 6-6 The Accumulative Summary View



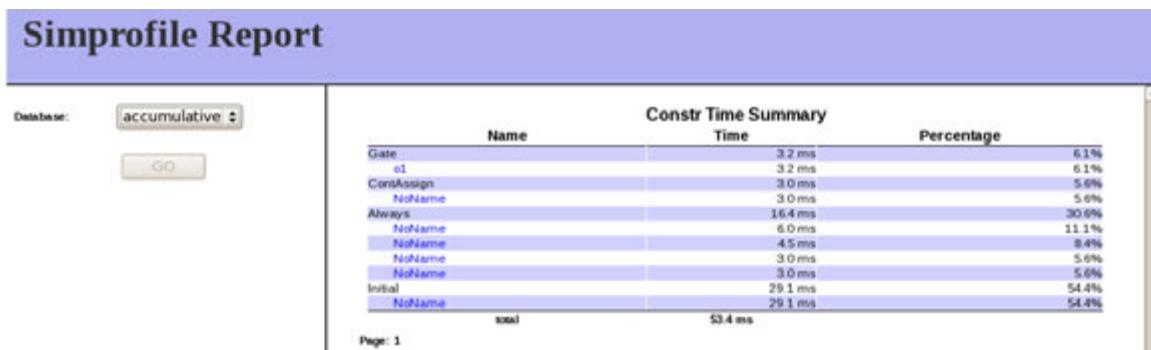
*Figure 6-7 The Accumulative Module View*



*Figure 6-8 The Accumulative Instance View*



*Figure 6-9 The Accumulative Construct View*



The accumulative views are only available in HTML format.

---

## The Comparative View

To generate a report that compares the results of two profile databases from two different simulations, include the `-diff` option and enter both databases on the `profrpt` command line.

Of the two specified databases, the first is the target database and the second is the reference database.

For example:

```
% profrpt -view ALL -diff simprofile_dir simprofile_dir.1 \
-output diff
```

The `profrpt` profile report writing utility in this example writes the `diff.html` file to compare the two profile databases.

Figure 6-10 The Comparative View in `diff.html`

Time Summary			
Component	Time	Reference	Gap
VERILOG	81.6 s	81.6 s	17.9 ms
PLI/DPI/DirectC	32.2 s	35.2 s	-3.0 s
KERNEL	574.4 ms	335.2 ms	239.2 ms
HSIM	39.9 ms	23.9 ms	15.9 ms
Value Change Dumping	16.0 ms	4.0 ms	12.0 ms
total	114.4 s	117.1 s	-2.7 s

Memory Summary			
Component	Memory	Reference	Gap
PLI/DPI/DirectC	511.5 MB	511.5 MB	0 B
KERNEL	35.6 MB	35.6 MB	0 B
Value Change Dumping	6.3 MB	6.3 MB	0 B
VERILOG	5.6 MB	5.6 MB	0 B
HSIM	2.1 MB	2.1 MB	0 B
total	561.1 MB	561.1 MB	0 B

Dynamic Summary			
Component	Dynamic	Reference	Gap
String	4.1 MB	4.1 MB	0 B
Class	608.0 KB	608.0 KB	0 B
SmartQueue	8.3 KB	8.3 KB	0 B
total	4.7 MB	4.7 MB	0 B

In this example view, the target database is the newer `simprofile_dir` directory and the reference database is the older `simprofile_dir.1` directory.

The significant differences are in the CPU times. [Figure 6-11](#) shows magnifications of the CPU time as part of the comparative view.

*Figure 6-11 Magnifications of the Values in the Comparative View*

The screenshot shows a comparative view of CPU times. On the left, there is a vertical list of components: VERILOG, PLI/DPI/DirectC, KERNEL, HSIM, and Value Change Dumping. Below this is a table comparing the execution times of these components in two databases. The table has columns for Time, Reference, and Gap.

Time	Reference	Gap
81.6 s	81.6 s	17.9 ms
32.2 s	35.2 s	-3.0 s
574.4 ms	335.2 ms	239.2 ms
39.9 ms	23.9 ms	15.9 ms
16.0 ms	4.0 ms	12.0 ms
<b>114.4 s</b>	<b>117.1 s</b>	<b>-2.7 s</b>

As you can see in [Figure 6-11](#), the reference database needed a full 3.0 seconds more of CPU time to execute its PLI application, but in the other components the target database needed more CPU time.

Green negative gap values indicate that the reference database values exceed the target database. Red values indicate that the target database exceeds the reference database.

## Limitations

- The comparative view only compares the information in the summary views.
- The comparative view is only supported in HTML format.

---

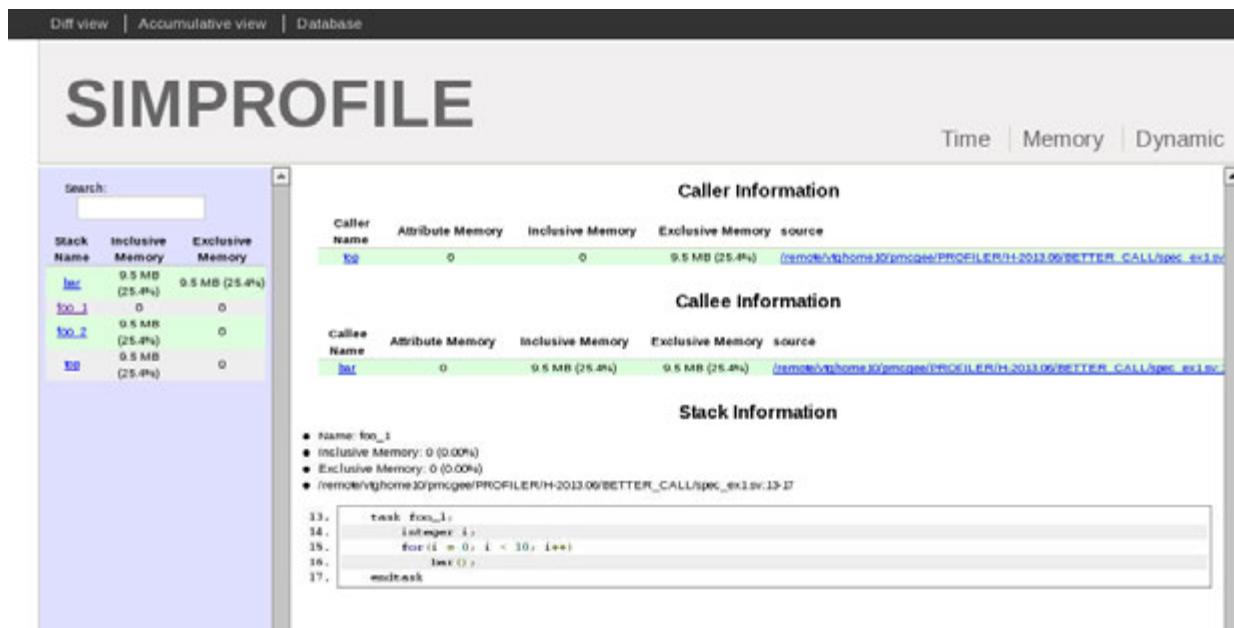
## The Caller-Callee Views

The unified simulation profiler has new views that show you the hierarchical constructs that call other hierarchical constructs (these are caller constructs) and the hierarchical constructs that are called by other constructs (these are callee constructs). These new views are as follows:

- The Caller-Callee Memory View, see [Figure 6-12](#)
- The Caller-Callee Time view

The concepts and the organization is same in both of these views.

*Figure 6-12 The Caller-Callee Memory View*



The caller-callee views are only supported in HTML format.

For example, a hierarchical construct, such as a user-defined task can contain a task enabling statement that starts another user-defined construct.

Other views can tell you when a construct consumes the most CPU time or machine memory. But the reason a construct uses so much of these resources is not apparent. Consider the source code in [Example 6-5](#).

### *Example 6-5 Tasks That Consume Resources*

```

module top;
    integer a[$];
    integer i;

    initial begin

```

```

    i = 0;
    foo_1();
    foo_2();
end

task bar;
    a.push_back(i);
    i++;
endtask

task foo_1;
    integer i;
    for(i = 0; i < 10; i++)
        bar();
endtask

task foo_2;
    integer j;
    for(j = 0; j < 1024 * 1024; j++)
        bar();
endtask

endmodule

```

When you run the above code and profile for machine memory, you can view and examine the Memory Construct view, as shown in [Figure 6-13](#).

*Figure 6-13 The Memory Construct View*

Memory Construct View (clock:0)			
Name	Size	Percentage	
Task bar	22.11 MB 21.83 MB	34.01 %	33.59 %
total	22.11 MB	34.01 %	

Page: 1

In the Memory Construct View, you can see that the task named bar consumed most of the machine memory during simulations. it is only the construct that uses enough resources to warrant inclusion in this file.

The question remains why is task bar called (or enabled) so many times that it consumes so much of this resource? The unified profiler has a new view that shows us why. It is called the Memory Caller-Callee view (and its corresponding Caller-Callee Time view).

Let us consider different panes of the Caller-Callee Memory view as shown in [Figure 6-12](#).

*Figure 6-14 The Left Pane of the Caller-Callee Memory View*

Stack Name	Inclusive Memory	Exclusive Memory
<a href="#">bar</a>	9.5 MB (51.5%)	9.5 MB (51.5%)
<a href="#">foo_1</a>	0	0
<a href="#">foo_2</a>	9.5 MB (51.5%)	0
<a href="#">top</a>	9.5 MB (51.5%)	0

Unlike other left panes of views, which are a place for selecting profile databases and views, this left pane contains:

- a search field for searching for constructs, such as modules, tasks, and other hierarchical constructs in the call stack
- A list of hierarchical constructs and their inclusive and exclusive memory usage.

As shown in [Figure 6-14](#), the call stack for this example contains user-defined tasks bar `foo_1`, `foo_2`, and top-level module `top`.

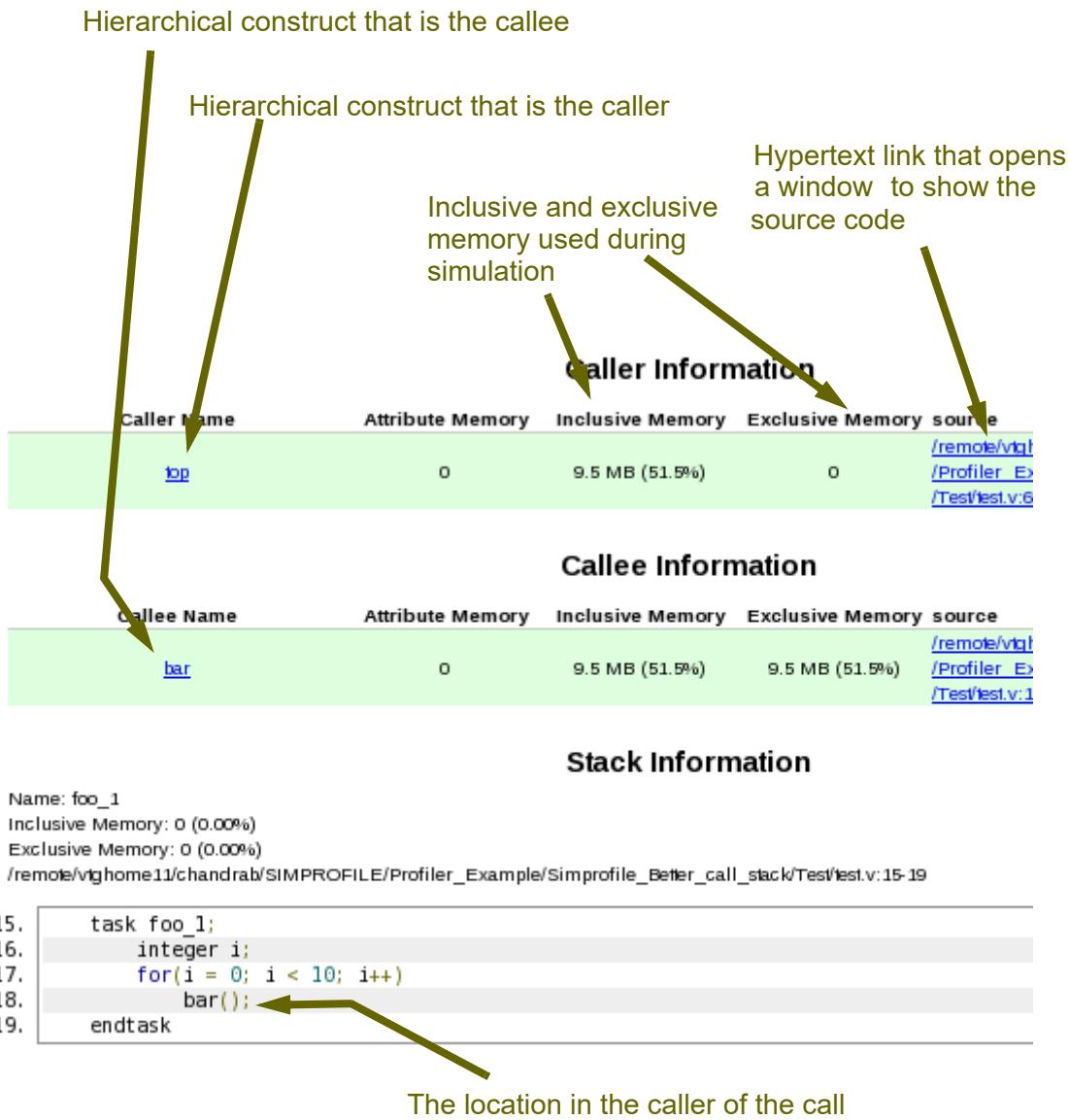
Task bar is a callee, tasks `foo_1`, `foo_2`, and module `top` are callers.

As in other views, inclusive is the resource used by the hierarchical construct and all such other constructs hierarchically under it. Exclusive is the amount or the resource used by the construct itself.

For an extensive list of callers and callees, there is a search field.

As shown in [Figure 6-14](#), task bar uses most of the exclusive machine memory that is used by the code example.

*Figure 6-15 The Right Division of the Caller-Callee Memory View*



In Figure 6-15, the caller of the task bar is top-level module top that contains task foo\_1 and foo\_2, which contain task enabling statement that calls the task bar. The **Stack Information** section of the view shows the source code of task foo\_1.

As in other views, inclusive memory is the amount of memory used by the hierarchical construct and those other hierarchical constructs that are under it in the design hierarchy.

As in other views, exclusive memory is the amount of memory used solely by the hierarchical construct.

Also as in other views, attribute memory is the amount of memory used by each caller construct in the design hierarchy.

---

## HTML Profiler Reports

Profiler reports are by default in HTML format.

The following are the examples of these reports based on the SystemVerilog code as shown in [Example 6-6](#):

### *Example 6-6 Profiler SystemVerilog Code Example*

```
program tb_top;
    logic [255:0]          Squeue_data_info[$];
    logic [255:0]          temp;

    class PACKET;
        rand reg [255:0] packet_val;
    endclass

    initial
        begin
            for(int y = 0 ; y < 1000 ; y++)
                begin
                    PACKET packet_inst;
```

```

packet_inst = new();
packet_inst.randomize();
#1;

Squeue_data_info.push_back(packet_inst.packet_val);
#1;

end

repeat(10)
$display("DEBUG==> Pushed 1000");

for(int y = 0 ; y < 500 ; y++)
begin

#1;
temp = Squeue_data_info.pop_front();
#1;

end
repeat(10)
$display("DEBUG==> Popped 500");

for(int y = 0 ; y < 10000 ; y++)
begin
    PACKET packet_inst;

packet_inst = new();
packet_inst.randomize();
#1;

Squeue_data_info.push_back(packet_inst.packet_val);
#1;

end

repeat(10)
$display("DEBUG==> Pushed 10000");

```

```

for(int y = 0 ; y < 5000 ; y++)
begin
    PACKET packet_inst_2;

#1;
temp = Squeue_data_info.pop_front();
#1;

end
repeat(10)
$display("DEBUG==> Popped 5000");

for(int y = 0 ; y < 100000 ; y++)
begin
    PACKET packet_inst;

packet_inst = new();
packet_inst.randomize();
#1;

Squeue_data_info.push_back(packet_inst.packet_val);
#1;

end

repeat(10)
$display("DEBUG==> Pushed 100000");

for(int y = 0 ; y < 50000 ; y++)
begin
    PACKET packet_inst_2;

#1;
temp = Squeue_data_info.pop_front();
#1;

end
repeat(10)
$display("DEBUG==> Popped 50000");

```

```

for(int y = 0 ; y < 1000000 ; y++)
begin
    PACKET packet_inst;

    packet_inst = new();
    packet_inst.randomize();
    #1;

    Squeue_data_info.push_back(packet_inst.packet_val);
    #1;

end

repeat(10)
$display("DEBUG==> Pushed 1000000");

for(int y = 0 ; y < 500000 ; y++)
begin
    PACKET packet_inst_2;

    #1;
    temp = Squeue_data_info.pop_front();
    #1;

end
repeat(10)
$display("DEBUG==> Popped 500000");

$finish;

end

endprogram

```

**This code was compiled and simulated for CPU time profile information with the following command lines:**

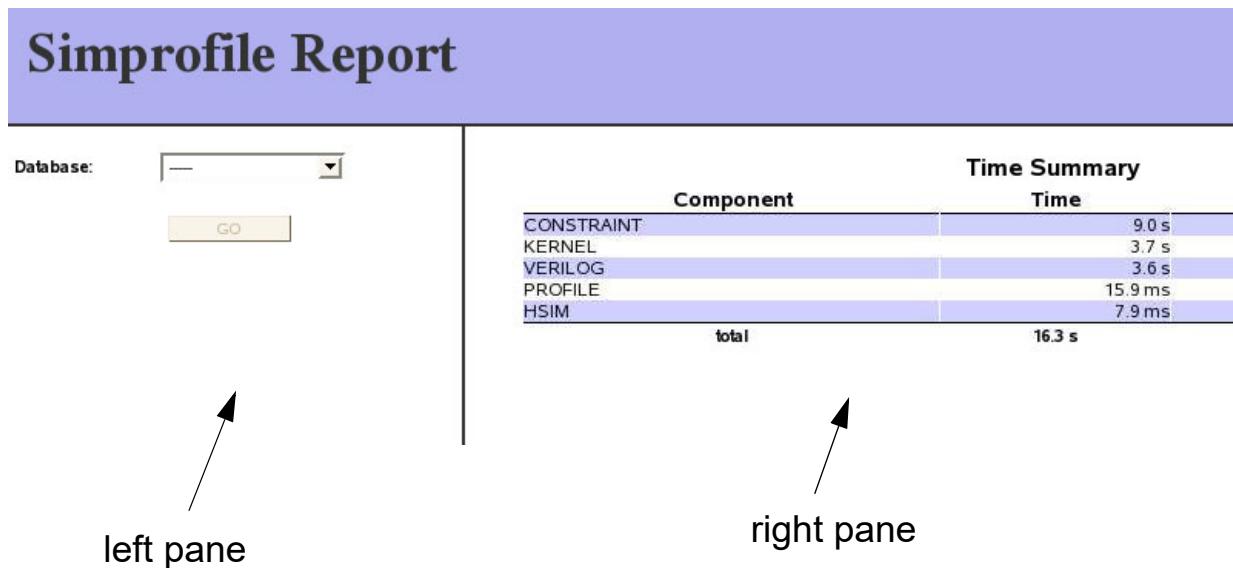
```
% vcs smart_queue.v -simprofile -sverilog
```

```
% simv -simprofile time
```

The profrpt command line is as follows:

```
profrpt simprofile_dir -view time_all -timeline ALL
```

Figure 6-16 The profileReport.html File for CPU Time Profile Information



The profileReport.html file contains two panes:

- The left pane is for specifying the profile database and the view you want to see.
- The right pane is for displaying the profile information.

*Figure 6-17 The Left Pane of the simprofileReport.html file*

**Database:**

The **Database** field is a drop-down menu. You can select the only database in this example so far, the `simprofile_dir` directory. Doing so adds the **View** field to the left pane and the default view, which in this case is the Time Summary view. Then, click the **GO** button.

*Figure 6-18 The Left Pane of the simprofileReport.html file*

**Database:**

**View:**

click here

*Figure 6-19 The Right Pane of the simprofileReport.html file for CPU Time Summary Information*

### Time Summary View

Component	Time	Percentage
CONSTRAINT	9.03 s	55.40 %
KERNEL	3.65 s	22.42 %
VERILOG	3.59 s	22.03 %
Program	3.59 s	22.03 %
<b>total</b>	<b>16.30 s</b>	<b>100%</b>

Components, in this case, are consumers of CPU time during simulation. The components in this example are as follows:

#### CONSTRAINT

The CPU time needed to solve and simulate the SystemVerilog constraint blocks.

Also the CPU time used for calls to the `randomize()` method, like in this example, are included in this component. These calls to `randomize()` are taking most of the CPU time reported for this component, and in turn this component used most of the CPU time.

#### KERNEL

The CPU time needed by the VCS kernel. This CPU time is separate from the CPU time needed to simulated your Verilog or SystemVerilog, VHDL, SystemC, or C or C++ code for your design and testbench.

## VERILOG

The CPU time needed by VCS to simulate this example's SystemVerilog code, which is a program block. For Verilog and SystemVerilog there are sub-components. In this example view, there is only one sub-component named Program.

This example consists of a SystemVerilog program block that used 22.03% of the CPU time.

Possible other sub-components are Module, Interface, UDP, and Assertion, for the CPU time used by Verilog and SystemVerilog definitions for module, interface, user-defined primitive, package and assertion.

Other possible components are as follows:

## DEBUG

The CPU time needed by VCS to simulate this example with the debugging capabilities of Verdi and the UCLI or to write a simulation history VCD or FSDB file.

## Value Change Dumping

The CPU time needed by VCS to write a simulation history VCD or VPD file. This component is always accompanied by the DEBUG component. This component has the following sub-components:

## VPD

The CPU time needed by VCS to write a VPD file.

## VCD

The CPU time needed by VCS to write a VCD file.

## VHDL

For VCS only, the CPU time needed to simulate the VHDL code design.

## PLI/DPI/DirectC

The CPU time needed by VCS to simulate the C/C++ in a PLI, DPI, or DirectC application.

## HSIM (Hybrid Simulation)

This is about the CPU time used by HSOPT (Hybrid Simulation Optimization). The HSIM bucket indicates the CPU time consumption of design constructs that are optimized by HSOPT. It has become prominent in GLS design/RTL.

The HSIM cost is more with GLS design because most constructs are optimized by HSOPT. But it cannot be zero because there are some global HSIM activities.

## COVERAGE

The CPU time needed for functional coverage (testbench and assertion coverage). Code coverage is not part of this component.

## SystemC

The CPU time needed for SystemC simulation.

If you select the Time Module view in the **View** field in the left pane, then click the **GO** button again. The right pane changes to show the following view:

*Figure 6-20 The CPU Time Module View*

Time Module View					
Module	Inclusive Time	Percentage	Exclusive Time	Percentage	
tb_top	1.90 s	24.54 %	1.90 s	24.54 %	
Initial	1.90 s	24.54 %	1.90 s	24.54 %	
<0.50 %	0.00us	0.00 %	0.00us	0.00 %	
total	1.90 s	24.54 %	1.90 s	24.54 %	

Page: 1

click here

As explained earlier, modules not only include Verilog and SystemVerilog modules, but can also include SystemVerilog programs and interfaces, and for VHDL, it can also include entity/architectures.

*Figure 6-21 The Expanded CPU Time Module View*

Time Module View		
Module	Time	Percentage
tb_top	3.59 s	22.03 %
NoName	3.45 s	21.16 %
total	3.59 s	22.03 %

Page: 1

In this example view, program block `tb_top` used 3.59 seconds of CPU time, which was 22.03% of the CPU time used by the simulation.

The program name is a hypertext link to expand the display in this view. If you click the hypertext link, you can see the scopes inside the program block.

The scopes inside the program block are begin-end blocks that are not named. Thus, profrpt calls them all NoName. These begin-end blocks use most of the CPU time used by the program block. In this example view, NoName is not a hypertext link.

Other possible scopes inside a module are fork-join blocks and user-defined tasks and functions.

If you select the Time Construct view in the **View** field in the left pane, then click the **GO** button again. The right pane changes to show the following view:

Figure 6-22 The CPU Time Construct View

Time Construct View		
Name	Time	Percentage
▶ Initial	3.45 s	21.16 %
total	3.51 s	21.55 %
Page: 1 click here		

In this example view, the only construct is an initial block. This initial block uses 21.55% of the CPU time.

For Verilog and SystemVerilog, the constructs in this view can include initial procedures, always procedures (including the SystemVerilog always procedures such as `always_comb`), SystemVerilog final procedures, user-defined tasks, and user-defined functions.

For VHDL, the constructs in this view are processes in architectures.

The initial keyword is a hypertext link to expand the display in this view. If you click the hypertext link, you can see the scopes inside the initial procedure.

*Figure 6-23 The Expanded CPU Time Construct View*

Time Construct View		
Name	Time	Percentage
▼ Initial	3.45 s	21.16 %
NoName	3.45 s	21.16 %
total	3.51 s	21.55 %

**Page:** 1

In this example view, the scopes inside the initial procedure are begin-end blocks that are not named. Thus, the profiler calls them all NoName. These begin-end blocks use most of the CPU time used by the program block. In this example view, NoName is not a hypertext link.

If you select the Time Instance view in the **View** field in the left pane, then click the **GO** button again. The right pane changes to show the following view:

*Figure 6-24 The CPU Time Instance View*

Time Instance View					
Instance	Inclusive Time	Percentage	Exclusive Time	Percentage	
►tb_top	3.59 s	22.03 %	3.59 s	22.03 %	
total	3.59 s	22.03 %	3.59 s	22.03 %	

Page: 1

This view shows CPU times and percentages for the instances in the design. These are instances of Verilog and SystemVerilog modules and also instances of SystemVerilog interfaces and VHDL entity/architectures.

This view shows for an instance the inclusive and exclusive time and percentage values.

The inclusive time and percentage is for the percentage of CPU time used by this instance and all instances that are hierarchically under it in the design hierarchy.

The exclusive time and percentage is for the CPU time used by this instance alone, not counting the instances that are hierarchically under this instance.

In the above figure, there is only one instance of program tb\_top. Thus, the inclusive and exclusive values are the same, which are 3.59 seconds and 22.03% of the CPU time.

The instance name tb\_top is not a hypertext link.

There is no PLI, DPI, or DirectC code. Therefore, there is no information in the Pli/DPI/DirectC view. There is also no information in the Dynamic Timeline view because this view is for machine memory information and you do not collect machine memory profile information in the profile database.

You can now simulate for machine memory profile information as shown:

```
% simv -simprofile mem
```

The profrpt command line is as follows:

```
% profrpt simprofile_dir -view mem_all -timeline ALL
```

The `-timeline` option specifies the snapshot reports.

The profile report generator, `profrpt`, rewrites the `profileReport.html` file for the machine memory information. Therefore, you should re-open this file.

In the left pane, in the **Database** field, select the `simprofile_dir` profile database directory again. This adds the following fields to the left pane:

- the **View** field that is at the default selection of the Memory Summary view
- the **Snapshot** field that is at the default selection of the peak machine memory snapshot. The example shows the 33rd snapshot at simulation time 2000000.

Then click the **GO** button.

*Figure 6-25 The Machine Memory Summary View for the Peak Snapshot*



**Memory Summary View (clock:2333001)**

Component	Size	Percentage
VERILOG	113.86 MB	51.52 %
Program	113.86 MB	51.52 %
Package	1.08 KB	0.00 %
KERNEL	36.76 MB	16.63 %
CONSTRAINT	7.33 MB	3.32 %
HSIM	1.05 MB	0.47 %
Library/Executable	62.00 MB	28.05 %
VCS	59.00 MB	26.70 %
Third-party	3.00 MB	1.36 %
total	<b>221.00 MB</b>	<b>100%</b>

Components, in this view, are consumers of machine memory during simulation. This view reports the amount of machine memory used by each component and their percentage of the total machine memory used in the snapshot. In this example view, this is the peak snapshot.

In this snapshot, the preponderance of the machine memory is used by the VERILOG and KERNEL components.

The components in this view are as follows:

## VERILOG

The machine memory needed by VCS to simulate this example's SystemVerilog code, which is a program block at the peak snapshot. The following are the sub-components:

## **Program**

The machine memory needed to simulate the SystemVerilog program block in the code example.

## **Package**

Usually the machine memory needed to simulate a SystemVerilog package.

In this case this is an anomaly; reporting a small amount of machine memory for a package when there is no package in the code example. You can ignore these anomalies.

The other possible sub-components are Module, Interface, UDP, and Assertion.

## **KERNEL**

The machine memory used by the VCS kernel. This is separate machine memory from the machine memory needed to simulate the code in the code example.

## **CONSTRAINT**

The machine memory needed to solve and simulate the SystemVerilog constraint blocks, but also counted in this component are calls to the `randomize()` method.

## **HSIM**

This is about the machine memory used by Hybrid Simulation Optimizations (HSOPT). Memory profiling is not impacted much because dynamic memory allocation is not expected in HSIM except during the initialization.

## **Library/Executable**

This is the sum of the VCS and Third-party sub-components.

### **VCS**

Memory consumed by VCS executable and library. It consists of `simv` and all the libraries provided by VCS. Most of these libraries are located in `$VCS_HOME/lib`.

### **Third-party**

Memory consumed by user-provided libraries and global libraries, such as `libc.so`. This includes all other libraries.

## **COVERAGE**

This component is for functional coverage. A small percentage of machine memory is reported here even though there is no functional coverage code in the design. This is the machine memory needed for functional coverage enabling optimizations, which are default optimizations.

Code coverage is not reported in this component. The machine memory used for code coverage is in the VERILOG (or VHDL) components.

Other possible components, if you change the example source code and entered different options, are as follows:

## **DEBUG**

This component is for the machine memory needed by VCS to simulate this example with the debugging capabilities of Verdi and the UCLI or to write a simulation history VCD or FSDB file.

## **PLI/DPI/DirectC**

This component is for the machine memory needed by VCS to simulate the C/C++ code in a design.

## **SystemC**

The machine memory needed for SystemC simulation.

For VHDL and mixed-HDL designs, there is an additional possible component:

## **VHDL**

The machine memory needed to simulate VHDL code of a design.

So far you have looked at the machine memory summary view for the peak snapshot. There is a summary view for other snapshots.

For example, if you select the 10th snapshot, as shown:

*Figure 6-26 Selecting the 10th Snapshot*

<b>Database:</b>	<input type="text" value="simprofile_dir"/>
<b>View:</b>	<input type="text" value="Memory Sumr"/>
<b>Snapshot:</b>	<input type="text" value="#10 (clock:193)"/>
<input type="button" value="GO"/>	

Then click the **GO** button, the right pane shows the machine memory summary view for this snapshot.

*Figure 6-27 The Machine Memory Summary View for the 10th Snapshot*

Memory Summary View (clock:193131)		
Component	Size	Percentage
KERNEL	29.99 MB	67.64 %
VERILOG	11.59 MB	26.14 %
Program	11.57 MB	26.09 %
Package	153.64 KB	0.34 %
HSIM	2.01 MB	4.53 %
CONSTRAINT	768.03 KB	1.69 %
COVERAGE	316 B	0.00 %
total	44.33 MB	100%

This view shows the machine memory used by the various components in the 10th snapshot.

Now, back in the left pane, you can return to the peak snapshot, the 33rd in the **Snapshot** field and select the Memory Module view in the **View** field, then click the **GO** button. The right pane changes to the machine memory module view for the peak snapshot.

Figure 6-28 The Machine Memory Module View for the Peak Snapshot

Module	Size	Percentage
<a href="#">tb_top</a>	116.14 MB	73.92 %
total	116.78 MB	74.33 %

Page: 1

click here

As explained earlier, modules not only include Verilog and SystemVerilog modules, but can also include SystemVerilog programs and interfaces, and for VHDL, include entity/architectures.

In this example view, program block `tb_top` used 116.14 MB of machine memory, which is 74.33% of the machine memory used to simulate the peak snapshot.

The program name is a hypertext link to expand the display in this view. If you click the hypertext link, you can see the scopes inside the program block.

*Figure 6-29 The Expanded Machine Memory Module View for the Peak Snapshot*

Memory Module View (clock:2000000)		
Module	Size	Percentage
tb_top	116.14 MB	73.92 %
NoName	116.14 MB	73.92 %
<b>total</b>	<b>116.78 MB</b>	<b>74.33 %</b>

Page: 1

In this example view, the scope inside the program block is a begin-end block that is not named. Therefore, profrpt calls it NoName. This begin-end block uses most of the machine memory used by the program block. In this example view, NoName is not a hypertext link.

Other possible scopes inside a module are fork-join blocks and user-defined tasks and functions.

There is a machine memory module view for each snapshot.

If in the left pane, you select the Memory Constant view and then click the **GO** button, the right pane changes to the machine memory construct view for the peak snapshot.

Figure 6-30 The Machine Memory Construct View for the Peak Snapshot

Memory Construct View (clock:2000000)		
Name	Size	Percentage
▶ Initial	116.14 MB	73.92 %
Total	116.14 MB	73.92 %
Page: 1		

click here

In this example view, the only construct is an initial block. This initial block uses, at the peak snapshot, 116.14 MB of machine memory, which is 73.92% of the total machine memory use at the peak snapshot.

For Verilog and SystemVerilog, the constructs in this view can include `initial` procedures, `always` procedures (including the SystemVerilog `always` procedures such as `always_comb`), SystemVerilog `final` procedures, user-defined tasks, and user-defined functions.

For VHDL, the constructs in this view are processes in the architectures.

The `initial` keyword is a hypertext link to expand the display in this view. If you click hypertext link, you can see the scope inside the `initial` procedure.

*Figure 6-31 The Expanded Machine Memory Construct View for the Peak Snapshot*

Memory Construct View (clock:2000000)		
Name	Size	Percentage
▼Initial	116.14 MB	73.92 %
NoName	116.14 MB	73.92 %
<b>total</b>	<b>116.14 MB</b>	<b>73.92 %</b>

Page: 1

In this example view, the scope inside the initial procedure is a begin-end block that is unnamed. Therefore, profrpt calls it NoName. This begin-end block use all of the machine memory used by the initial procedure. In this example view, NoName is not a hypertext link.

There is a machine memory construct view for each snapshot.

If you select the Memory Instance view in the **View** field in the left pane, then click the **GO** button again, the right pane changes to show the following view:

*Figure 6-32 The Machine Memory Instance View for the Peak Snapshot*

Memory Instance View (clock:2000000)				
Instance	Inclusive Size	Percentage	Exclusive Size	Percentage
►tb_top	116.14 MB	73.92 %	116.14 MB	73.92 %
<b>total</b>	<b>116.16 MB</b>	<b>73.93 %</b>	<b>116.16 MB</b>	<b>73.93 %</b>

Page: 1

This view shows the machine memory used and percentages for the instances in the design at the peak snapshot. These are instances of Verilog and SystemVerilog modules and also instances of SystemVerilog programs and interfaces and VHDL entity/architectures.

This view shows for an instance the inclusive and exclusive machine memory used and percentage values for the peak snapshot.

The inclusive machine memory amount and percentage is the percentage of machine memory used by this instance and all instances that are hierarchically under it in the design hierarchy.

The exclusive machine memory amount and percentage is the machine memory used by this instance alone, not counting the instances that are hierarchically under this instance.

In this example view, there is only one instance of program `tb_top`. So, the inclusive and exclusive values are the same, which are 116.16 MB and 73.93% of the machine memory.

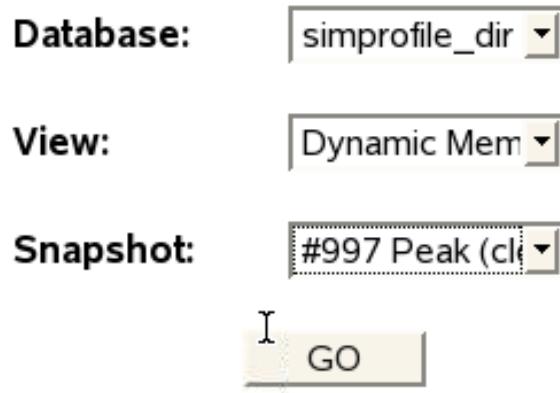
The instance name `tb_top` is not a hypertext link.

Like the machine memory summary, module, and construct views, there is a machine memory instance view for each snapshot.

In this example view, there is no PLI, DPI, or DirectC code so there is no information in the Pli/DPI/DirectC view.

If you select the Dynamic Memory view in the **View** field in the left pane, the **Snapshot** field automatically changes to snapshot #997.

*Figure 6-33 The Left Pane After Selecting the Dynamic Memory View*



Snapshot #997 is the peak snapshot for dynamic objects.

If you click the GO button again, the right pane changes to show this view.

*Figure 6-34 The Dynamic Memory View for the Peak Snapshot*

Dynamic Memory View (clock: 1998612)				
Dynamic Object	Instance Number	Memory	Percentage	
► PACKET	31552	29.68 MB	87.12 %	
► SmartQueue	N/A	4.39 MB	12.88 %	
► String	1	48 B	0.00 %	
		34.07 MB	100%	

Page: 1       click here

The peak machine memory dynamic view shows the machine memory that was used by dynamic objects at their peak machine memory consumption. This is not the peak machine memory consumption of the entire design and testbench, just the peak machine memory consumption of their dynamic objects.

The dynamic objects include dynamic and associative arrays and queues.

In this view is a SystemVerilog queue and string.

Smart Queues are a concept in the *OpenVera Language Reference Manual: Testbench*. The `profrpt` profile report generator lists SystemVerilog queues as Smart Queues. In this example view, there is only one SystemVerilog queue. It is declared as follows:

```
logic [255:0] Squeue_data_info[$];
```

`Squeue_data_info`, in this peak machine memory dynamic view, uses 4.39 MB of machine memory, which is 12.88% of the machine memory used at this peak by this queue.

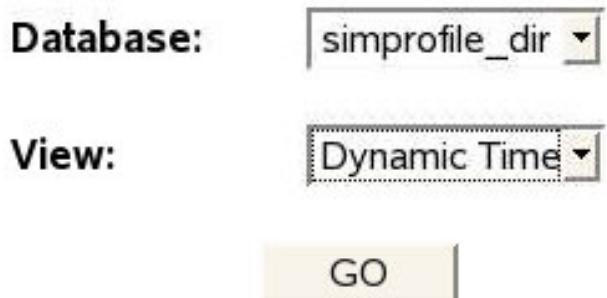
The `profrpt` profile report generator cannot report the number of instances of this queue.

The string entry is for a small amount of machine memory and can be ignored.

There is a dynamic object machine memory view for each snapshot.

If you select the Dynamic Timeline view in the View: field in the left pane, the Snapshot: field disappears.

*Figure 6-35 The Left Pane After Selecting the Dynamic Timeline View*



If you click the **GO** button again, the right pane changes to show this view.

Figure 6-36 The Machine Memory Dynamic Timeline View

Column for snapshots

Dynamic Memory Timeline										Display percentage
	Clock	Assoc-Aarry	Dynamic Array	Smart Queue	Event	Mailbox	String	Class	Total	
#0	0	0 B	0 B	0 B	0 B	0 B	48 B	0 B	48 B	
#1	1	0 B	0 B	288 B	0 B	0 B	48 B	104 B	440 B	
#2	36	0 B	0 B	200 B	0 B	0 B	48 B	1.93 KB	2.17 KB	
#3	1182	0 B	0 B	3.83 KB	0 B	0 B	48 B	60.13 KB	64.01 KB	
#4	1246	0 B	0 B	3.83 KB	0 B	0 B	48 B	63.38 KB	67.26 KB	
#5	1314	0 B	0 B	3.83 KB	0 B	0 B	48 B	66.84 KB	70.71 KB	
#6	1384	0 B	0 B	3.83 KB	0 B	0 B	48 B	70.39 KB	74.27 KB	
#7	1458	0 B	0 B	3.83 KB	0 B	0 B	48 B	74.15 KB	78.02 KB	
#8	1536	0 B	0 B	3.83 KB	0 B	0 B	48 B	78.11 KB	81.98 KB	
#9	1618	0 B	0 B	3.83 KB	0 B	0 B	48 B	82.27 KB	86.15 KB	
#10	1704	0 B	0 B	3.83 KB	0 B	0 B	48 B	86.64 KB	90.52 KB	
#11	1794	0 B	0 B	3.83 KB	0 B	0 B	48 B	91.21 KB	95.09 KB	
#12	1888	0 B	0 B	3.83 KB	0 B	0 B	48 B	95.98 KB	99.86 KB	
#13	1921	0 B	0 B	11.41 KB	0 B	0 B	48 B	97.61 KB	109.06 KB	
#14	2104	0 B	0 B	7.58 KB	0 B	0 B	48 B	106.95 KB	114.58 KB	
#15	2218	0 B	0 B	7.58 KB	0 B	0 B	48 B	112.74 KB	120.37 KB	
#16	2338	0 B	0 B	7.58 KB	0 B	0 B	48 B	118.84 KB	126.46 KB	
#17	2464	0 B	0 B	7.58 KB	0 B	0 B	48 B	125.23 KB	132.86 KB	
#18	2596	0 B	0 B	7.58 KB	0 B	0 B	48 B	131.94 KB	139.56 KB	
#19	2734	0 B	0 B	7.58 KB	0 B	0 B	48 B	138.95 KB	146.57 KB	

Page:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42

hypertext links for page numbers

This view unlike the previous machine memory views is not for a specific snapshot, but for all snapshots in the profile database.

In this example view, there are multiple pages. The page numbers at the bottom of the view are hypertext links to show the different pages. In this view, there are many pages because there are hundreds of snapshots in the database.

Notice that there is a significant increase in the machine memory for the queue in snapshot 13.

You can scroll to the right and click page 50, which includes the dynamic object machine memory peak snapshot, and the right pane changes to show this page.

*Figure 6-37 Page 50 of the Machine Memory Dynamic Timeline View*

Dynamic Memory Timeline								Display percentage
Clock	Assoc-Aarry	Dynamic Array	Smart Queue	Event	Mailbox	String	Class	Total
#980	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.83 MB 6.22 MB
#981	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.52 MB 5.91 MB
#982	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.23 MB 5.61 MB
#983	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	968.30 KB 5.33 MB
#984	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	695.20 KB 5.07 MB
#985	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	435.70 KB 4.81 MB
#986	1936896	0 B	0 B	4.39 MB	0 B	0 B	48 B	189.21 KB 4.57 MB
#987	1945230	0 B	0 B	4.39 MB	0 B	0 B	48 B	423.31 KB 4.80 MB
#988	1950072	0 B	0 B	4.39 MB	0 B	0 B	48 B	669.20 KB 5.04 MB
#989	1955156	0 B	0 B	4.39 MB	0 B	0 B	48 B	927.37 KB 5.29 MB
#990	1960494	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.17 MB 5.56 MB
#991	1966098	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.45 MB 5.84 MB
#992	1971982	0 B	0 B	4.39 MB	0 B	0 B	48 B	1.74 MB 6.13 MB
#993	1978160	0 B	0 B	4.39 MB	0 B	0 B	48 B	2.05 MB 6.43 MB
#994	1984648	0 B	0 B	4.39 MB	0 B	0 B	48 B	2.37 MB 6.76 MB
#995	1991460	0 B	0 B	4.39 MB	0 B	0 B	48 B	2.71 MB 7.09 MB
#996	1998612	0 B	0 B	4.39 MB	0 B	0 B	48 B	3.06 MB 7.45 MB
#997	1998612	0 B	0 B	4.39 MB	0 B	0 B	48 B	29.68 MB 34.07 MB

---

## Display of Parameterized Class Functions and Tasks in Profiling Reports

The reports generated by the unified simulation profiler display functions and tasks of parameterized classes that are defined in a package or in the global scope.

For example:

```
class vector #(int size = 1);
    rand bit [size-1:0] a;
    bit [size-1:0] a_arry[];

    constraint num { a > 1; }

    task obj_disp();
        $display("%0d : Object v%0d : %p", $time, size, this);
    endtask

    function void disp_count();
        int i;
        for (i=0; i<1000000; i++) begin
            this.randomize();
            a_arry[i] = a;
        end
    endfunction
endclass

program prog;
    vector #(2) v2 = new;
    vector #(3) v3 = new;
    vector #(4) v4 = new;

    initial begin
        v2.disp_count();
        v3.disp_count();
        v4.disp_count();
        v2.obj_disp();
        v3.obj_disp();
    end

```

```

    v4.obj_disp();
end
endprogram

```

In the above example, the parameterized class `vector` is defined in the global scope. In the profiling report, the instance `_global_` is displayed in the Time Instance View and the class function `disp_count()` is displayed in the Time Module View.

**Note:**

For objects of the same parameterized class, profiling data for their functions and tasks are combined and displayed as a single entry.

[Figure 6-38](#) is the Time Module View.

*Figure 6-38 Time Module View*

Time Module View				
Module	Inclusive Time	Percentage	Exclusive Time	Percentage
<code>_global_</code>	1.23 s	2.65 %	1.23 s	2.65 %
<code>Function</code>	1.23 s	2.65 %	1.23 s	2.65 %
<code>disp_count</code>	1.23 s	2.65 %	1.23 s	2.65 %
<0.50 %	0.00us	0.00 %	0.00us	0.00 %
total	1.23 s	2.65 %	1.23 s	2.65 %

Construct Information	
Construct Name	<code>disp_count</code>
Time	1.23 s
Construct Type	Function
Parent Module	<code>global</code>
Source Information	/home/gaurav/Downloads/TESTS/vcs1412/key_acct/8439_simprofile_task_fn/testv14-20

---

## Hypertext Links to the Source Files

The pathnames of source files in any of the HTML views are hypertext links. Clicking on one of these links opens a new window of the browser to display that source file. This section describes and illustrates this feature.

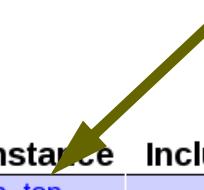
Note:

The hypertext link to the source files feature is not implemented for SystemC/C/C++ source files.

To use this feature, do the following:

1. Compile a design with the `-simprofile` option.
2. Run the simulation with the `-simprofile time/mem/time+mem` option and keyword argument to enable VCS to collect time/memory/time and memory profile information.
3. Run the `profprt` utility to create the HTML views.
4. Open the `profileReport.html` file.
5. Select a profile database in the left pane.
6. Select the Time Instance view.

Figure 6-39 Time Instance View



click here

Instance	Inclusive Time	Percentage	Exclusive Time	Percentage
►tb_top	3.52 s	23.46 %	3.52 s	23.46 %
total	3.52 s	23.46 %	3.52 s	23.46 %

Page: 1

7. Click an instance in the view.

This adds this information about the instance to the bottom of the HTML page:

Instance Name	a reiteration of the instance name
Exclusive Time	the CPU time used by the instance
Exclusive Percentage	the percentage of the total CPU time that was used by this instance
Inclusive Time	the CPU time used by the instance and all instances under it in the design hierarchy
Inclusive Percentage	the percentage of the total CPU time that was used by this instance and all instances under it in the design hierarchy
Master Module	the name of the top-level module in the design hierarchy
Child Instance Number	the number of instances under this instance in the design hierarchy
Source Information	the path to the source file and line number of the header of the module, interface, or program definition

The **Source Information** is in blue text in this expanded view because it is a hypertext link to the source code, as shown in [Figure 6-40](#).

*Figure 6-40 Time Instance View Expanded*

Instance Information		
Instance Name		tb_top
Exclusive Time		3.52 s
Exclusive Percentage	click here	23.46 %
Inclusive Time		3.52 s
Inclusive Percentage		23.46 %
Master Module		tb_top
Child Instance Number		0
Source Information	<a href="#">/file_system/big_design/VCS_user_files/smart_queue.v:2</a>	

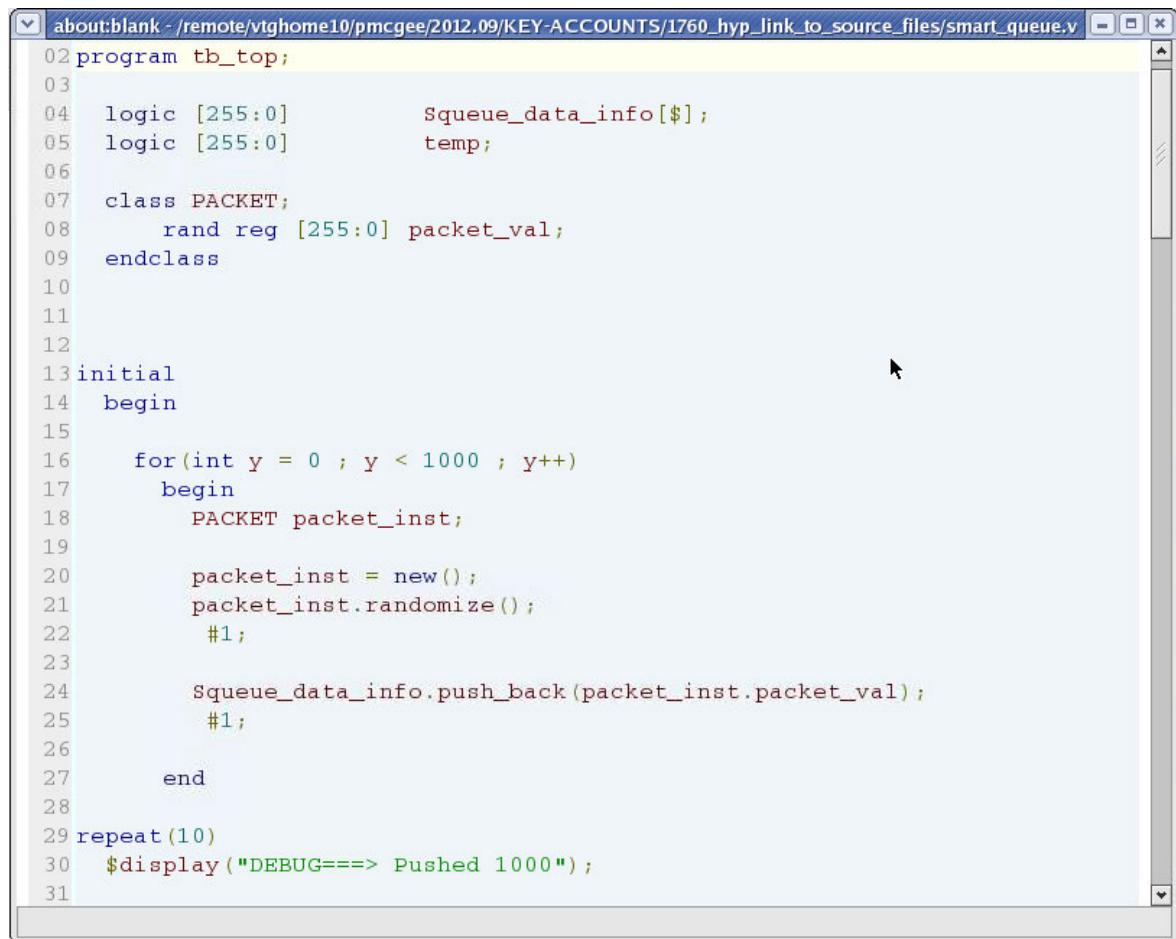
In this example view, source information for the instance is the program definition for instance tb\_top in /file\_system/big\_design/VCS\_user\_files/smart\_queue.v on line 2.

8. Click the blue path name of the source file and line number, this is a hypertext link. The browser opens a new window to display the source file, as shown in [Figure 6-41](#).

Note:

To display the source file in new window, you should open the source file with Firefox 3.\* web browser.

*Figure 6-41 New Source File window*



A screenshot of a computer screen showing a "New Source File" window. The window title is "about:blank - /remote/vtghome10/pmcgee/2012.09/KEY-ACCOUNTS/1760\_hyp\_link\_to\_source\_files/smart\_queue.v". The code editor displays the following Verilog code:

```
02 program tb_top;
03
04     logic [255:0]           Squeue_data_info[$];
05     logic [255:0]           temp;
06
07     class PACKET;
08         rand reg [255:0] packet_val;
09     endclass
10
11
12 initial
13 begin
14
15     for(int y = 0 ; y < 1000 ; y++)
16         begin
17             PACKET packet_inst;
18
19             packet_inst = new();
20             packet_inst.randomize();
21             #1;
22
23             Squeue_data_info.push_back(packet_inst.packet_val);
24             #1;
25
26         end
27
28 repeat(10)
29 $display ("DEBUG====> Pushed 1000");
31
```

The program header is `program tb_top;` in line 2, has a lighter background.

The lines in this source file window also shows the line numbers.

---

## Single Text Format Report

Text format views are merged together into a text file named `profileReport.txt` in the current directory.

You can specify text format reports with the `-format text` or `-format all` option and argument on the `profrpt` command line.

If you run the `profrpt` report generator more than once, the utility overwrites the `profileReport.txt` file in the current directory so that its profile information is from the last run.

When you specify text format reports the `profrpt` utility also creates separate text files for each view in the profile report directory, these separate text files for each view have names such as `PeakMemInstanceView.txt` or `TimeConstr.txt`.

---

## Stack Trace Report Example

The following file, named `check.v`, is used to produce a sample stack trace report.

```
class Packet;
    bit[100000:0] b;
    function new();
        b = 0;
    endfunction
endclass

Packet pp[int];
    int cindex = 0;
reg r;

program p;
    function Packet AllocPacket();
        begin
            AllocPacket = new;
        end
    endfunction

task A;
    begin
```

```

        fork
            B();
            C();
        join
    end
endtask

task B;
    int i;
    Packet lpp[int];
begin
    $display("B called");
    for (i=0; i < 100000; i++)
        pp[i] = AllocPacket();
end
endtask

task C;
    int i;
    Packet lpp[int];
begin
    $display("C called");
    for (i=0; i < 10000; i++)
        lpp[i] = AllocPacket();
end
endtask

initial
begin
    A();
end
endprogram

```

The following command sequence generates the stack trace report for the `check.v` example:

```

% vcs check.v -simprofile -sverilog

% simv -simprofile mem

% profrpt simprofile_dir -view dynamic_mem+stack

```

[Figure 6-42](#) shows the HTML stack trace report for the check.v example. The stack trace information is at the bottom of the view.

*Figure 6-42 The Machine Memory Dynamic Object View for the Peak Snapshot*

Dynamic Memory View (clock: 0)				
Dynamic Object	Instance Number	Memory	Percentage	
▼ Packet	110000	1315.92 MB	100.00 %	
▶ AllocPacket	100000	1196.29 MB	90.91 %	
▶ AllocPacket	10000	119.63 MB	9.09 %	
▶ AssociativeArray	N/A	128 B	0.00 %	
▶ String	1	48 B	0.00 %	
		1315.92 MB	100%	

Page: 1

Stack Information		
#0	AllocPacket	/file_system/big_design/VCS_user_files/stack.sv:15
#1	C	stack.sv:44
#2	A	stack.sv:23
#3	p	stack.sv:50

---

## SystemC Views

The following views are from a SystemC co-simulation after running the `profrpt` profile report generator.

The code examples for these views is in the `$VCS_HOME/doc/examples/systemc/vcs/vcs_profiler`. There is a minor change to one of the files to show the name for a begin-end block in `sv_mod.sv` as follows:

```

module sv_mod(iclk);
    input iclk;
    static int count=0;
    int i;

    always @(posedge iclk)
begin: bel
    count++;
    $display("SV:Executing on pos edge @%d",count);
    for(i=0;i<1000*100000000;i++)
    ;
end

endmodule

```

*Figure 6-43 The Time Summary View*

### Time Summary View

Component	Time	Percentage
VERILOG	282.53 s	77.85 %
Module	282.53 s	77.85 %
Package	999.89us	0.00 %
SystemC	79.85 s	22.00 %
KERNEL	505.94 ms	0.14 %
HSIM	12.00 ms	0.00 %
PLI/DPI/DirectC	999.89us	0.00 %
<b>total</b>	<b>362.90 s</b>	<b>100%</b>

As you would expect from reading the SystemVerilog and SystemC files in this example, most of the CPU time was used by the SystemVerilog and SystemC modules.

A small amount of CPU time was used by The VCS kernel.

A small amount of CPU time was reported used by a SystemVerilog package, writing a VPD file, and PLI, DPI, or a DirectC application, even though these are not present in this example. Notice that they all take 0.00% of the CPU time. You can ignore these anomalies.

If our example writes a VPD file or contains a PLI, DPI, or DirectC application, you might see significant values for the CPU times in this view.

*Figure 6-44 The Time Module View*

Time Module View		
Module	Time	Percentage
►sv_mod	282.53 s	77.85 %
►sc_mod	79.85 s	22.00 %
►sv_top	2.00 ms	0.00 %
►std	999.89us	0.00 %
►_global_	0.00us	0.00 %
total	282.53 s	77.85 %

Page: 1

This view shows the CPU times and percentages for the main consumers of CPU times, the `sv_mod` SystemVerilog module and the `sc_mod` SystemC module. A small amount of time is used by the top-level module `sv_top`.

The `std` and `_global_` modules are from the internals of VCS and when seen in this view should be ignored.

If you click these module names, the view expands to show scopes inside these module definitions.

*Figure 6-45 The Expanded Time Module View*

Time Module View		
Module	Time	Percentage
sv_mod	282.53 s	77.85 %
be1	282.53 s	77.85 %
NoName	0.00us	0.00 %
iclk	0.00us	0.00 %
sc_mod	79.85 s	22.00 %
mythread	79.85 s	22.00 %
sv_top	2.00 ms	0.00 %
NoName	2.00 ms	0.00 %
NoName	0.00us	0.00 %
std	999.89us	0.00 %
_global_	0.00us	0.00 %
total	282.53 s	77.85 %

Page: 1

In the `sv_mod` SystemVerilog module:

- The begin-end block `be1` consumes all of the CPU time of the module.
- There is an extraneous process call `NoName` that consumes no CPU time and can be ignored.
- The `iclk` input port in `sv_mod` is shown as a process, such as the begin-end block of the code. If `sv_mod` have other ports that are not clock signals, `profrpt` does not show them as processes.

In the `sc_mod` SystemC module, `mythread()` is the SystemC variant of a named block in Verilog or SystemVerilog and represents the code (like in a SystemVerilog `always` procedure, but is shown in this view rather than in the Time Construct view). The implementation of this function is in the `.cpp` file.

*Figure 6-46 The CPU Time Construct View*

Time Construct View		
Name	Time	Percentage
► Always	282.53 s	77.85 %
► Initial	2.00 ms	0.00 %
► Task	0.00us	0.00 %
► Function	0.00us	0.00 %
► Port	0.00us	0.00 %
<b>total</b>	<b>282.53 s</b>	<b>77.85 %</b>

Page: 1

The CPU time construct view shows the CPU times and percentages used by the always and initial procedures in the design and also the port in the design.

In this example view, Task and Function do not refer to a user-defined task and function, but rather refer to the internals of VCS and do not consume any CPU time. If this example contained user-defined tasks or functions, they are listed as a Task or Function here.

*Figure 6-47 The Time Instance View*

Time Instance View				
Instance	Inclusive Time	Percentage	Exclusive Time	Percentage
► sv_top	282.53 s	77.85 %	2.00 ms	0.00 %
► sv_top.sc_mod_inst	79.85 s	22.00 %	79.85 s	22.00 %
► std	999.89us	0.00 %	999.89us	0.00 %
► _global_	0.00us	0.00 %	0.00us	0.00 %
<b>total</b>	<b>282.53 s</b>	<b>77.85 %</b>	<b>282.53 s</b>	<b>77.85 %</b>

Page: 1

In this view, as it initially appears, you see the SystemVerilog top-level instance `sv_top`. You can also see the SystemC instance `sv_top.sc_mod_inst` because it is a SystemC instance in this SystemVerilog.

As in previous views, `std` and `_global_` are from the internals of VCS and can be ignored.

If you click the top-level module `sv_top`, you can see the `sv_mod_inst` instance.

*Figure 6-48 The Expanded CPU Time Instance View*

### Time Instance View

Instance	Inclusive Time	Percentage	Exclusive Time	Percentage
▼ <code>sv_top</code>	282.53 s	77.85 %	2.00 ms	0.00 %
► <code>sv_mod_inst</code>	282.53 s	77.85 %	282.53 s	77.85 %
► <code>sv_top.sc_mod_inst</code>	79.85 s	22.00 %	79.85 s	22.00 %
► <code>std</code>	999.89us	0.00 %	999.89us	0.00 %
► <code>_global_</code>	0.00us	0.00 %	0.00us	0.00 %
total	<b>282.53 s</b>	<b>77.85 %</b>	<b>282.53 s</b>	<b>77.85 %</b>

Page: 1

Figure 6-49 The PLI/DPI/DirectC View

Time PLI/DPI/DirectC View		
Name	Time	Percentage
PLI		
\$sc_mod_init	999.89us	0.00 %
\$vcdplusfilter	0.00us	0.00 %
\$msglog	0.00us	0.00 %
\$lsl_dumpports	0.00us	0.00 %
\$countdrivers	0.00us	0.00 %
\$vcsmemprof	0.00us	0.00 %
\$start_toggle_count	0.00us	0.00 %
\$report_toggle_count	0.00us	0.00 %
\$set_toggle_region	0.00us	0.00 %
\$toggle_start	0.00us	0.00 %
\$toggle_stop	0.00us	0.00 %
\$toggle_reset	0.00us	0.00 %
\$toggle_report	0.00us	0.00 %
\$read_lib_saif	0.00us	0.00 %
\$read rtl_saif	0.00us	0.00 %
\$set_gate_level_monitoring	0.00us	0.00 %
DPI	0.00us	0.00 %
DirectC	0.00us	0.00 %
total	999.89us	0.00 %

This view for the PLI shows both VCS internal functions and user-written PLI functions.

In this example view, all functions are VCS internal functions. You can look for ones that consume the significant CPU time. The `$vcdplusmsglog` system function, not in this example, can consume significant CPU time.

For SystemC, there is an additional CPU time view, the SC (SystemC) OverHead View.

*Figure 6-50 The SC OverHead View*

Time SC-OverHead View		
Name	Time	Percentage
SC-Value-OverHead	0.00us	0.00 %
SC-Kernel-OverHead	0.00us	0.00 %
SC-Spawn-OverHead	0.00us	0.00 %
<b>total</b>	<b>0.00us</b>	<b>0.00 %</b>

**Page: 1**

This data depends on the test case. It can be that kernel overhead becomes an issue and it can be compared against the Verilog kernel overhead.

The `sc-value` overhead is time taken to transfer data from one domain to another, such as to or from SystemC to or from Verilog, or SystemVerilog, or VHDL. This can be expensive when there is large amount of data, such as with a large vector signal or a large multidimensional array. Also spawning of processes can take time and accumulate sc-overhead.

Kernel overhead from SystemC, Verilog, SystemVerilog or VHDL, can become an issue when your code does not consume much CPU time and there is significant overhead to keep the co-simulation running. Usually you want these CPU time values to be low.

---

## Constraint Profiling Integrated in the Unified Profiler

Constraint profiling is integrated in the unified profiler. This integration adds the following views to the profile reports:

- the Time Constraint Solver view
- the Memory Constraint Solver view

These views tell you in detail the calls to the `randomize()` method that use the most CPU time or the most machine memory. With this information you can consider revising your constraints on the random variables to use less of these resources.

---

## Changes to the Use Model for Constraint Profiling

To tell `profrpt` to generate these views the following arguments are added to the use model:

The `profrpt -view` option's arguments now include:

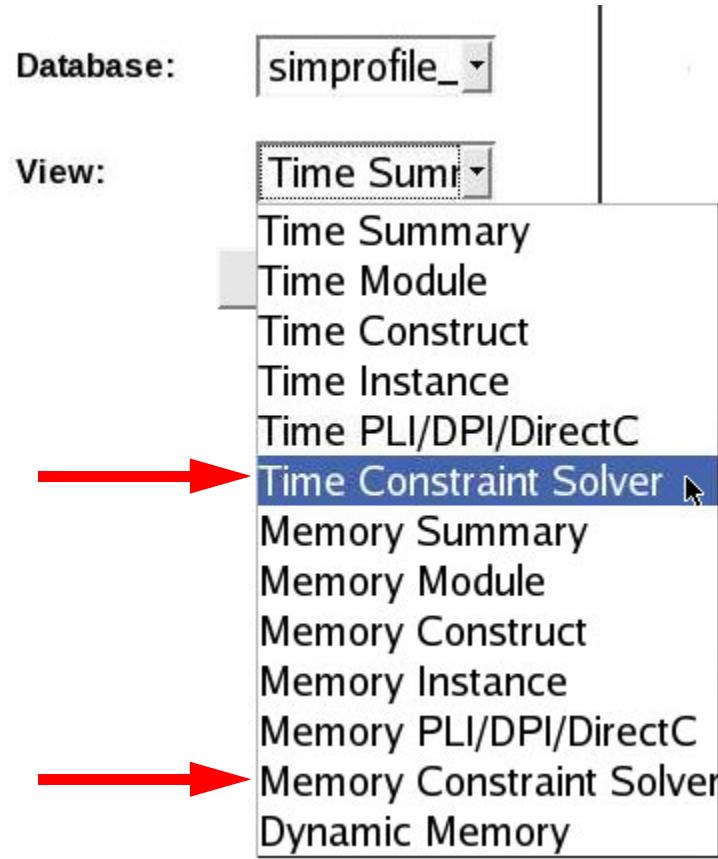
- `time_solver` to specify generating the Time Constraint Solver view
- `mem_solver` to specify generating the Memory Constraint Solver view.

The `time_all` and `mem_all` arguments also generate these views.

The left pane of the `profileReport.html` file, after selecting a profile database, contains a drop-down menu for views. This menu now contains the following for constraint profiling:

- the Time Constraint Solver view
- the Memory Constraint Solver view

*Figure 6-51 New Constraint Views*



The following sections describe these views.

---

## **The Time Constraint Solver View**

The following is an example of the Time Constraint Solver View.

*Figure 6-52 Example Time Constraint Solver View*

### Time Constraint Solver View

Total user time: 11.670seconds  
 Total system time: 0.120seconds  
 Total randomize time: 0.030seconds  
 Total randomize count: 2

#### Top randomize calls based on cpu runtime

File:line@visit	serial#	time (sec)	variables	constraints	cnst blocks
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1	0.030	27	37	9
<a href="#">/env/nvs_atapi_env.sv:120@1</a>	2	0.000	3	3	1

#### Top randomize calls based on cumulative cpu runtime

File:line	calls	time (sec)
<a href="#">/env/nvs_atapi_env.sv:118</a>	1	0.030
<a href="#">/env/nvs_atapi_env.sv:120</a>	1	0.000

#### Top partitions based on cpu time

File:line@visit	Rand.Partition	cpu time (sec)	variables	constraints	cnst blocks
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.1	0.03	2	4	2
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.2	0.00	1	1	1
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.3	0.00	7	12	3
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.4	0.00	5	10	3
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.5	0.00	2	1	1
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.6	0.00	1	1	1
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.7	0.00	1	1	1
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.8	0.00	2	1	1
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.9	0.00	2	1	1
<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1.10	0.00	2	1	1

Constraint solver profile	
Solver	Time (sec)
Core Solver (default)	0.030
Core Solver (mode=1)	0.000
Core Solver (FAST)	0.000
Problem Generation	0.000

Top partitions based on BDD size					
File:line@visit Rand.Partition	peak BDD size	final BDD size	variables	constraints	cnst blocks

Parts of this view in [Figure 6-52](#) are described in detail in [Figure 6-53](#), [Figure 6-54](#), [Figure 6-55](#), [Figure 6-56](#), [Figure 6-57](#), and [Figure 6-58](#).

*Figure 6-53 Introductory information is at the top of the view*

Time Constraint Solver View	
Total user time:	11.670seconds
Total system time:	0.120seconds
Total randomize time:	0.030seconds
Total randomize count:	2

### Total user time:

Specifies the total CPU time to simulate the design and testbench. In this example view, it is 11.670 seconds.

### Total system time:

Specifies the total CPU time used by VCS when not simulating the design or testbench. In this example view, it is 0.12 seconds.

### Total randomize time:

Specifies the CPU time that VCS needs to execute the `randomize()` method calls in the design. In this example view, it is 0.03 seconds.

### Total randomize count:

Specifies the number of entries of the `randomize()` method in the SystemVerilog source code. In this example view, it is 2.

Figure 6-54 Top randomize calls based on CPU time

#### **hypertext link** Top randomize calls based on cpu runtime

File:line@visit	serial#	time (sec)	variables	constraints	cnst blocks
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1	0.030	27	37	9
<a href="#">./env/nvs_atapi_env.sv:120@1</a>	2	0.000	3	3	1

This section of the view is for the `randomize()` entries in the source code that use the most CPU time. There is a separate line for each entry. There are two such entries in this code example. So, they are listed here.

The columns in this section are for the following values:

#### **File:line@visit**

Specifies the following three things:

File

Specifies the path name for the source file that contains the entry. In this example view, the first line is for a source file with the `/env/nvs_atapi_env.sv` path name.

#### line

Specifies the line number in the source file that contains the entry. In this example view, the entry is on line 118.

#### @visit

An execution of the entry. There can be multiple executions of the same entry throughout a simulation. In this example view, the first line is for the first execution or visit of the entry.

If VCS executes the entry in the code three times, the line in this section could begin with:

`/env/nvs_atapi_env.sv:118@3`

#### **Important:**

The `File:line@visit` part of a line is in blue because this part is a hypertext link. When you click the link, the browser opens a new window showing the source file with the line specified at the top.

#### **serial#**

The series in this column is the order in which VCS executes the calls to the `randomize()` method. In this example view, line 118 contains the first call and line 120 contains the second call.

**Note:**

This section of the view is for the calls that used the most CPU time. However, these top users are not always the first or second randomize() calls that VCS executes.

### **time (sec)**

The amount of CPU time used by the call.

### **variables**

The number of rand or randc variables randomized by a call.  
Not all such variables in a class are randomized by a call.

### **constraints**

The number of constraints in the class that are randomized by a call.

### **cnst blocks**

The number of constraint blocks that contain these constraints.

**Note:**

In the following example, there is one constraint block and four constraints, as shown:

```
constraint reasonable_on_latencies {
    dior_to_data_place_time < 10;
    data_prepare_time       < 10;
    dior_to_data_place_time > 0;
    data_prepare_time       > 0;
} //end constraint reasonable_on_latencies
```

Figure 6-55 Top randomize calls based on cumulative CPU runtime

**hypertext link**

Top randomize calls based on cumulative cpu runtime		
File:line	calls	time (sec)
<a href="#">./env/nvs_atapi_env.sv:118</a>	1	0.030
<a href="#">./env/nvs_atapi_env.sv:120</a>	1	0.000

VCS can execute or visit a call to the `randomize()` method in a specific location of the source code more than once. If it does so, VCS keeps track of the cumulative CPU time used by these multiple executed calls and `profrpt` reports this cumulative time in this section.

This section reports:

- The location of the call in a hypertext link that opens a new window displaying the source code.
- The number of calls or visits to this location.
- The cumulative CPU time used by the calls.

Figure 6-56 Top partitions based on CPU time

hypertext link

Top partitions based on cpu time						
File:line@visit	Rand.Partition	cpu time	variables	constraints	cnst	blocks
		(sec)				
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.1	0.03	2	4	2	
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.2	0.00	1	1	1	
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.3	0.00	7	12	3	
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.4	0.00	5	10	3	
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.5	0.00	2	1	1	
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.6	0.00	1	1	1	
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.7	0.00	1	1	1	
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.8	0.00	2	1	1	
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.9	0.00	2	1	1	
<a href="#">./env/nvs_atapi_env.sv:118@1</a>	1.10	0.00	2	1	1	

VCS has a constraint solver to determine the possible values that conform to your constraints. To solve these problems, the constraint solver divides its work into partitions. This section reports the number of partitions in a problem.

In this example view, this section reports the visit to the `randomize()` method in the example source file at `/env/nvs_atapi_env.sv` on line 118.

The constraint solver divided its work into 10 partitions. The `profrpt` utility generate reports for each partition:

- the CPU time needed to solve the partition
- the number of random variables in the partition
- The number of constraints
- The number of constraint blocks that contained these constraints

*Figure 6-57 Constraint solver profile*

Constraint solver profile	
Solver	Time (sec)
Core Solver (default)	0.030
Core Solver (mode=1)	0.000
Core Solver (FAST)	0.000
Problem Generation	0.000

The total randomize time is further broken down into the different internal solvers and problem generation. This information might indicate where you can revise your constraints and randomize calls to improve the total CPU time.

*Figure 6-58 Top partitions based on BDD size*

Top partitions based on BDD size				
File:line@visit Rand.Partition	peak BDD size	final BDD size	variables	constraints
	cnst	blocks		

This part of the constraint profile report is empty unless VCS uses the solver (mode=1) in the randomization. When it uses mode=1, this section shows some memory footprint information of different randomize calls executed under this solver (mode=1). You specify using the mode=1 solver with the `+ntb_solver_mode=1` runtime option and argument.

No information is in this example section because the default solver is doing the constraint solving for this example.

---

## The Memory Constraint Solver View

The following is an example of the memory constraint solver view.

*Figure 6-59 Example Memory Constraint Solver View*

Memory Constraint Solver View						
Largest memory increment: 640KB						
Top randomize calls based on memory increment						
File:line@visit	serial#	mem incr (KB)	variables	constraints	cnst blocks	
/env/nvs_atapi_env.sv:118@1	1	640	27	37	9	
/env/nvs_atapi_env.sv:120@1	2	8	3	3	1	

Top recurring randomize calls based on memory increment		
File:line	calls	mem incr (KB)
/env/nvs_atapi_env.sv:118	1	640
/env/nvs_atapi_env.sv:120	1	8

Parts of this view, [Figure 6-59](#), are described in detail in [Figure 6-60](#) and [Figure 6-61](#).

In the beginning, the view shows the size of the largest memory increment during the simulation, as shown:

*Figure 6-60 Largest memory increment*

## Memory Constraint Solver View

Largest memory increment: 640KB

In this example view, the largest increase in machine memory is an increase of 640 KB.

The view follows with the `randomize()` entries that cause the largest increase in the use of machine memory, as shown:

*Figure 6-61 Top randomize calls based on memory increment*

Top randomize calls based on memory increment						
<b>hypertext link</b>	File:line@visit	serial#	mem	variables constraints	cnst blocks	
			incr (KB)			
	<a href="#">/env/nvs_atapi_env.sv:118@1</a>	1	640	27	37	9
	<a href="#">/env/nvs_atapi_env.sv:120@1</a>	2	8	3	3	1

The columns in this section are as follows:

**File:line@visit**

Specifies the following three things:

**File**

Specifies the path name for the source file that contains the entry. In this example view, the first line is for a source file with the path name `/env/nvs_atapi_env.sv`.

### **line**

Specifies the line number in the source file that contains the entry. In this example view, the entry is on line 118.

### **@visit**

A visit is an execution of the call. There can be multiple executions of the same call throughout the simulation. In this example view, the first line is for the first execution, or visit of the call.

If VCS executes the entry in the code three times, the line in this section could begin with:

[/env/nvs\\_atapi\\_env.sv:118@3](#)

### **Important:**

The File:line@visit part of a line is in blue because this part is a hypertext link. When you click the link, the browser opens a new window showing the source file with the line specified at the top.

### **serial#**

The series in this column is the order in which VCS executes the calls to the `randomize()` method. In this example view, line 118 contains the first call and line 120 contains the second call.

### **Note:**

This section of the view is for the calls that use the most machine memory and these top users are not always the first or second `randomize()` calls that VCS executes.

### **mem incr (KB)**

Specifies the amount of additional machine memory that VCS needs when it executes the call.

## variables

The number of `rand` or `randc` variables randomized by a call.  
Not all such variables in a class are randomized by a call.

## constraints

The number of constraints in the class that are randomized by a call.

## cnst blocks

The number of constraint blocks that contain these constraints.

## hypertext link



Top recurring randomize calls based on memory increment		
File:line	calls	mem incr (KB)
<a href="#">./env/nvs_atapi_env.sv:118</a>	1	640
<a href="#">./env/nvs_atapi_env.sv:120</a>	1	8

The next section is for the `randomize()` calls that VCS executes the most. There are two `randomize()` entries in this example. Each entry is executed only once. The calls that are executed once are shown in this section because the code example does not contain calls that execute more frequently during the simulation.

This section reports:

- The path to the source file and the line number of the call.
- The number of times VCS executes a call.
- The amount of additional machine memory VCS needs to execute the call.

---

## Performance/Memory Profiling for Coverage Covergroups

This is an extension to the Unified Simulation Profiler to increase the granularity at which it reports the coverage related data. It provides the total time/memory taken by each covergroup across all its instantiations and the time/memory taken by individual instances of each covergroup.

The data reported for a covergroup or a covergroup instance includes the time/memory spent in instantiating and initializing the covergroup instances and the time/memory spent in sampling the covergroup and the associated processing of the bins.

A covergroup instance is defined as the covergroup instantiation that is uniquely determined by an external reference as defined by the SystemVerilog LRM. This is also the lowest granularity at which time/memory data is reported. If a covergroup is instantiated multiple times on the same line of code, then the time/memory data is gathered for all those instances. Similarly, if a covergroup is instantiated within the same scope in different branches using the same handle, then the time/memory data is gathered for all those instances.

---

## Use Model

The naming mechanism should be similar to URG.

For covergroups: *declaring scope name::covergroup name*

For covergroup instances, you must provide a full hierarchical path including both static and dynamic components for embedded covergroup definitions.

---

## Example

The covergroups for which the time/memory data are provided:

```
my_mod::my_static_cg  
my_class1::my_cg
```

The covergroup instances for which a separate time/memory data is provided:

```
top.i1.cg1  
top.i2.cg1  
top.i1.cg2  
top.i2.cg2  
top.i1.mc1_top.my_cg  
top.i2.mc1_top.my_cg  
top.i1.mc2_top.mc1.my_cg  
top.i2.mc2_top.mc1.my_cg  
top.i1.mc2_top.mc2.my_cg  
top.i2.mc2_top.mc2.my_cg
```

---

## HTML Profiler Reports

Profiler reports are by default in HTML format.

The following sections provide the covergroup enhancements for each of the views.

## Default Summary View

When a default HTML Simprofile Report is loaded, the Default Summary View is opened.

The coverage component is split into two new components — Functional Coverage and Code Coverage. The Covergroup captures the total time/memory spent in all the instantiated covergroups for the run. The Code Coverage component captures the time/memory spent in segments of code coverage collection to be determined later. The full code coverage data is collected and reported.

*Figure 6-62 Default Summary View*

Time Summary View		
Component	Time	Percentage
VERILOG	13.57 s	97.17 %
Functional Coverage	8.68 s	62.14 %
Module	4.16 s	29.80 %
Function Coverage Kernel	730.69 ms	5.23 %
KERNEL	383.31 ms	2.74 %
HSIM	11.98 ms	0.09 %
total	13.97 s	100%

## Time/Memory Summary View

To access the Time/Memory Summary View, click **Time/Memory Summary** option in the left pane of the Simprofile Report.

This view is similar to Default Summary View. To view more information, see “[HTML Profiler Reports](#)” .

## Time/Memory Module View

To access the Time/Memory Module view, click **Time/Memory Module** option in the left pane of the Simprofile Report.

Expanding a module/interface/program/package provides the data for the covergroups instantiated in it. The data for each covergroup captures the total time/memory spent in all instances of that covergroup across all the instances of the scope. As shown in above example, the `my_class1::my_cg` covergroup is instantiated thrice in the `my_mod` module; once as part of `mc1_top`, an object of class `my_class1`, and twice as part of `mc2_top`, an object of class `my_class2`. There are two instances of `my_mod` in the design. The data presented for `my_class1::my_cg` under `my_mod` is the cumulative data from all the six instances of the covergroup.

The covergroups are further expanded to provide data for each cover item (coverpoint or cross) in the covergroup.

**Figure 6-63 Time/Memory Module View**

Module	Time Module View			
	Inclusive Time	Percentage	Exclusive Time	Percentage
▼top	12.84 s	91.94 %	12.84 s	91.94 %
▼CoverGroup	8.68 s	62.14 %	8.68 s	62.14 %
▼cg	8.68 s	62.14 %	8.68 s	62.14 %
▼cpt1_cp	964.27 ms	6.90 %	964.27 ms	6.90 %
▼cpt4_cp	745.67 ms	5.34 %	745.67 ms	5.34 %
▼cpt3_cp	730.69 ms	5.23 %	730.69 ms	5.23 %
▼cpt2_cp	715.72 ms	5.12 %	715.72 ms	5.12 %
▼cg_cc_0	428.23 ms	3.07 %	428.23 ms	3.07 %
▼cg_cc	404.28 ms	2.89 %	404.28 ms	2.89 %
▼Initial	233.58 ms	1.67 %	233.58 ms	1.67 %
►NoName	233.58 ms	1.67 %	233.58 ms	1.67 %
<0.50 %	0.00us	0.00 %	0.00us	0.00 %
total	12.84 s	91.94 %	12.84 s	91.94 %

Page: 1

In Time/Memory Module view, click a covergroup to view the details of that covergroup in the **Construct Information** pane. These include the name of the covergroup, the scope in which it is declared (package, module, programs, interface, checker, or class), the total time/memory taken by all the covergroup instances in all the instances of the instantiating scope, and the file and line number for the declaration of the covergroup. Click the source file/line information to get the appropriate file and move the cursor to the appropriate line.

## Time/Memory Construct View

To access the Time/Memory Construct View, click **Time/Memory Construct** option in the left pane of the Simprofile Report.

A new covergroup entry is added to the existing constructs. When a covergroup is expanded, it lists all the covergroups declared in the design. The data displayed for each covergroup is the cumulative data across all the instances of that covergroup regardless of where it is instantiated.

The covergroups are further expanded to provide data for each cover item (coverpoint or cross) in the covergroup.

*Figure 6-64 Time/Memory Construct View*

Time Construct View		
Name	Time	Percentage
▼ CoverGroup		62.14 %
cg	8.68 s	62.14 %
▼ CoverPoint		22.59 %
cpt1_cp	964.27 ms	6.90 %
cpt4_cp	745.67 ms	5.34 %
cpt3_cp	730.69 ms	5.23 %
cpt2_cp	715.72 ms	5.12 %
▼ CoverCross		5.96 %
cg_cc_0	832.51 ms	3.07 %
cg_cc	428.23 ms	2.89 %
▼ Initial		1.67 %
NoName	233.58 ms	1.67 %
total	12.90 s	92.37 %

Page: 1

In Time/Construct View, click a covergroup to provide the details of the covergroup in the **Construct Information** pane. These include name of the covergroup, scope in which it is declared (package, module, programs, interface, checker, or class), total time/memory taken by all the covergroup instances of this covergroup in the entire design, and file and line number for the declaration of the covergroup. Click the source file/line information to get the appropriate file and move the cursor to the appropriate line.

## Time/Memory Covergroup View

To access the Time/Memory Covergroup View, click the **Time/Memory Covergroup** option in the left pane of the Simprofile Report. It provides information for the functional covergroups and the time/memory information both at the covergroup definition level and at the covergroup instance level. The time/memory data for the covergroup definition includes the time/memory spent in all the

instances of that covergroup in the entire design, whereas the time/memory data for the covergroup instance includes only the time/memory spent in that particular instance of the covergroup.

*Figure 6-65 Time/Memory Coverage View*

Time Coverage View			
Name	Time	Percentage	Source Information
cg	8.69 s	62.24 %	<a href="#">/remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example</a> <a href="#">/Simprofile_Fcov_Support/Test/test.v:4-17</a>
cpt1_cp(CoverPoint)	965.72 ms	6.91 %	<a href="#">/remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example</a> <a href="#">/Simprofile_Fcov_Support/Test/test.v:5-5</a>
cpt4_cp(CoverPoint)	746.79 ms	5.35 %	<a href="#">/remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example</a> <a href="#">/Simprofile_Fcov_Support/Test/test.v:11-11</a>
cpt3_cp(CoverPoint)	731.79 ms	5.24 %	<a href="#">/remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example</a> <a href="#">/Simprofile_Fcov_Support/Test/test.v:10-10</a>
cpt2_cp(CoverPoint)	716.79 ms	5.13 %	<a href="#">/remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example</a> <a href="#">/Simprofile_Fcov_Support/Test/test.v:9-9</a>
cg_cc_0(CoverCross)	428.88 ms	3.07 %	<a href="#">/remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example</a> <a href="#">/Simprofile_Fcov_Support/Test/test.v:16-16</a>
cg_cc(CoverCross)	404.88 ms	2.90 %	<a href="#">/remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example</a> <a href="#">/Simprofile_Fcov_Support/Test/test.v:15-15</a>
total	8.69 s	62.24 %	

Page: 1

## Limitations

The following technologies are not supported in the unified profiler:

- The behavior becomes unpredictable if you fork child processes or threads in your C code, which might be called through PLI/DPI/ DirectC interfaces.
- Incremental compilation is not yet supported for the unified profiler.
- OpenVera is not officially supported, VCS provides some information for reference but the name of the programs and constructs might be a bit different from the original one.

- Code coverage is not yet supported. The time and memory used by code coverage is counted to the corresponding HDL code.
- The accumulative views are available only in HTML format.
- The caller-callee views are available only in HTML format.
- No break down information is available for analog simulations. The information is available only in the summary form.
- No instance information is available for SystemVerilog Assertions (SVAs). Only time module view is displayed that helps you to determine which caller consume the most memory or time.

---

## Reporting Debug Capabilities for Each Module

VCS integrates the profiler report with debug capacities profile that shows the debug capacities enabled and used by each module. The debug capabilities that you enable are collected on the compile time. On runtime, the capabilities actually used by modules are recorded. Simprofile automatically analyzes the database generated on runtime as well as the compile-time data and shows the profile in the final report.

---

### Use Model

The following is the use model for reporting debug capabilities:

- Compile time

The compile time use model remains same as that of the previous use model for reporting debug capabilities. There is no change needed for reporting debug capabilities.

```
%vcs -simprofile
```

- **Runtime**

Debug aware profile is enabled automatically when you enable time profiler at runtime:

```
%vcs -simprofile <time/mem>
```

- **Profp**

To generate debug aware profile report, invoke **profrpt** using one of the following commands:

```
%profrpt -view plilearn <other profrpt options>
```

or

```
%profrpt -view ALL <other profrpt options>
```

Separate file **PliLearn.txt** is generated for PLI debug view. The switch at **profrpt** is **profrpt -view plilearn plilearn+mem\_mod**, which considers both time and memory to generate report.

---

## HTML Reports

The debug capacity profile presents a separate view. This is called as the ACC capacity view in the final HTML report. This view consists of two sets of data:

- **Capacities Statistics (PLI Debug Capability View)**

For each capability, the following three sets of statistics data are generated:

- Enabled modules
- Enabled and used modules
- Enabled but not used modules

The HTML reports are shown in [Figure 6-66](#):

**Figure 6-66 PLI Debug Capability View in HTML Report**

Click this blue colored hyperlink to display the time taken by the module below HTML report.

Stat.	read	read_write	callback	callback_all	force	static
enabled module no.	9	6	8	2	3	7
percentage	100.00 %	66.67 %	88.89 %	22.22 %	33.33 %	77.78 %
enabled module time	7.17 s	28.92 ms	7.15 s	0.00us	19.28 ms	7.15 s
percentage	89.32 %	0.36 %	89.08 %	0.00 %	0.24 %	89.08 %
used module no.	0	0	2	0	3	0
percentage	0.00 %	0.00 %	22.22 %	0.00 %	33.33 %	0.00 %
used module time	0.00us	0.00us	0.00us	0.00us	19.28 ms	0.00us
percentage	0.00 %	0.00 %	0.00 %	0.00 %	0.24 %	0.00 %
unused module no.	9	6	6	2	0	7
percentage	100.00 %	66.67 %	66.67 %	22.22 %	0.00 %	77.78 %
unused module time	7.17 s	28.92 ms	7.15 s	0.00us	0.00us	7.15 s
percentage	89.32 %	0.36 %	89.08 %	0.00 %	0.00 %	89.08 %
Capabilities enabled but no module uses: read, read_write, callback_all, static						
Capabilities enabled and used by all modules: force						
<a href="#">show full module list</a>						

Percentage = enabled module number / Total number of modules enabled. For example, 9/9 for read, 6/9 for read\_write, and so on.

Click this hyperlink to display the full module view with each capability enabled.

Each data set includes module number, the percentage in total module number, modules exclusive time, and the percentage in total execution time.

As shown in the figure, percentage is calculated as follows:

Percentage = enabled module number / Total number of modules enabled

To display the full module view with each capability enabled, click the module number, as shown in [Figure 6-67](#):

VCS also provides the following aggregated statistics:

- Capabilities enabled but no module uses. This means that the capacities are enabled by some modules but are not used in any of the modules.
- Capabilities enabled and used by all modules. This means that the capacities are enabled by some modules and are used in all these modules.
- Module level debug capacities (Full Module View)

The reports indicate which module needs which capacity. Capacities that are used by all modules or by no modules are extracted for clarity.

The slot labels are:

- **Enabled:** Enabled but not used
- **Used:** Enabled and used
- **Empty slot:** Not enabled

**Figure 6-67 Full Module List View in HTML Report**

Module	Exclusive Time	Full Module View						static
		read	read_write	callback	callback_all	force	Used	
TbTop4	7.29 s	Enabled		Enabled				Enabled
DutTop2	19.27 ms	Enabled	Enabled				Used	
Top	9.63 ms	Enabled	Enabled	Enabled				Enabled
TbTop	0.00us	Enabled		Enabled				Enabled
mid1	0.00us	Enabled	Enabled	Used	Enabled			
bot1	0.00us	Enabled	Enabled	Used	Enabled		Enabled	
TbTop1	0.00us	Enabled	Enabled	Enabled			Used	Enabled
TbTop2	0.00us	Enabled		Enabled				Enabled
TbTop3	0.00us	Enabled	Enabled	Enabled			Used	Enabled

## Text Reports

There are large number of modules in a design. Therefore, the module level data do not fit into the size of the text report. So, the text report includes only the capacities statistics data as shown in [Figure 6-68](#).

**Figure 6-68 Text Report**

PLI Debug Capability View						
Stat.	read	read_write	callback	callback_all	force	static
enabled module no.	9	6	8	2	3	7
percentage	100.00 %	66.67 %	88.89 %	22.22 %	33.33 %	77.78 %
enabled module time	7.32 s	28.90 ms	7.30 s	0.00 us	19.27 ms	7.30 s
percentage	89.62 %	0.35 %	89.39 %	0.00 %	0.24 %	89.39 %
used module no.	0	0	2	0	3	0
percentage	0.00 %	0.00 %	22.22 %	0.00 %	33.33 %	0.00 %
used module time	0.00 us	0.00 us	0.00 us	0.00 us	19.27 ms	0.00 us
percentage	0.00 %	0.00 %	0.00 %	0.00 %	0.24 %	0.00 %

## Limitations

The following are the limitations with this feature:

- Module level data is available only for text reports.
- No detail data, such as source code is available in module list view in HTML or text reports.

- Only support capabilities, such as `read`, `read_write`, `callback`, `callback_all`, `force`, `static` are reported. Other debug capabilities like `line callback` are not reported.
  - Only the original module is shown for parameterized modules.
- 

## Supporting Line-Based CPU Time Profiler

You can generate line-based profile report. This helps you to generate more accurate profile report and also helps you to efficiently identify the line in a module or construct that has taken most of the time.

---

### Use Model

The use model remains the same as that of the previous use model.

Line-based profile is enabled automatically along with the simprofile. The line-based profile report is generated in the source information window that is displayed when you click a module or a construct.

If you select the **Time Module View** or the **Time Construct View** in the **View** field in the left pane and then click the **GO** button, the right pane changes to show the **Time Module View** or the **Time Construct View** respectively as shown in [Figure 6-69](#) and [Figure 6-70](#).

Figure 6-69 The CPU Time Module View

Time Module View				
Module	Inclusive Time	Percentage	Exclusive Time	Percentage
tb_top	1.90 s	24.54 %	1.90 s	24.54 %
►Initial	1.90 s	24.54 %	1.90 s	24.54 %
<0.50 %	0.00us	0.00 %	0.00us	0.00 %
total	1.90 s	24.54 %	1.90 s	24.54 %

Page: 1

Figure 6-70 The CPU Time Construct View

Time Construct View		
Name	Time	Percentage
►Initial	3.45 s	21.16 %
total	3.51 s	21.55 %

Page: 1

In **Time Module View** or **Time Construct View**, click the hyperlink of the source information in the **Construct Information** pane as shown in Figure 6-71:

Figure 6-71 Construct Information Pane

**Construct Information**

Construct Name		disp_count
Time		1.23 s
Construct Type		Function
Parent Module		global
Source Information	<a href="#">/TESTS/vcs1412/key_acct/8439_simprofile_task_fnitestv14-20</a>	

click here

The line-based profile report is displayed in a new browser as shown in Figure 6-72:

*Figure 6-72 Line-Based Profile Report*

```
01 class Packet;
02
03 bit[100000:0] b;
04
0.13s 5.7% 05 function new();
0.01s 0.4% 06 b = 0;
0      0     07 endfunction
08
09 endclass
10
```

The first column displays the CPU time consumed by that line and the second column displays the percentage. The CPU time of a construct is not reported in the source view. The CPU time of a construct is equal to the total CPU time of all the lines in a construct.

---

## Limitations

The following are the limitations with this feature:

- Supports only the CPU time profile.
- Supports only the HTML form for the line-based profile.

---

## Supporting Simulation Time Slice Based Profiler

VCS allows you to generate profile report for a specific time period and helps you to limit the size of the database.

For example, you can generate a profile report for the time period when the simulation is very slow (before reset) or you can generate a profile report for the time period when the simulation is occupying huge memory.

---

## Use Model

The following is the use model for reporting debug capabilities:

- Compile time

The compile time use model remains the same as that of the previous use model for reporting time slice profile. There is no change needed for reporting time slice profile.

```
%vcs -simprofile
```

- Runtime

```
%simv -simprofile -simprofile_start <t+ht> -  
simprofile_stop <t+ht>
```

where,

```
-simprofile_start <t+ht>
```

Turns on the simulation profile dumping at simulation time  $t$ .  $ht$  is the high 32 bits. If  $ht$  is 0, then it can be omitted.

```
simprofile_stop <t+ht>
```

Turns off the simulation profile dumping at simulation time  $t$ .  $ht$  is the high 32 bits. If  $ht$  is 0, then it can be omitted.

## Example 1

```
% simv -simprofile time -simprofile_start 1+50 -  
simprofile_stop 1+60
```

Here,

```
start time is è 1 * 2 ^32 + 50 = 4294967346ns  
stop time is è 1 * 2 ^32 + 60 = 4294967356ns
```

## Example 2

```
% simv -simprofile time -simprofile_start 50 -  
simprofile_stop 60
```

Here,

```
start time is è 50 = 50ns  
stop time will be è 60 = 60ns
```

- Profrpt

To generate time slice profiler report, invoke profrpt as follows:

```
% profrpt -start <start-time> -stop<stop-time> <other  
profrpt options>
```

where,

-start <time>

Specifies the starting time (in simulation units) when the report generation should begin. By default, the start time is 0.

-stop <time>

Specifies the stopping time (in simulation units) when the report generation should end. By default, the report generation stops at the latest time available in the simulation profile database.

**Figure 6-73 HTML Report - Time/Memory Controlled During Simulation**

Current Database Information	
Build Date	Apr 02 2015 20:41:37
Compile	J-2015.09-Alpha
Version	
Runtime	J-2015.09-Alpha
Version	
Create	
Date	Fri Apr 3 02:48:34 2015
Profile Start	290000
Profile Stop	29271423

All individual profiler reports contain the profile start and stop duration and time consumed is during this specified time only.

```
%simv -simprofile time -simprofile_start 290000 -  
simprofile_stop 29271423
```

**Figure 6-74 HTML Report - Time/Memory Controlled During Profrpt**

##### VCS build date:	
compiler version:	Apr 02 2015 20:41:37
Runtime version:	J-2015.09-Alpha
Machine Name:	J-2015.09-Alpha
Profile runner:	vgintsb56
Profile data:	meenar
Creation date:	/remote/vtghome12/meenar/SIMPROFILE/time_slice_based
Profile start:	ofiler/BM_sel/WB_DMA_SVD/simprofile_dir
Profile stop:	Fri Apr 3 03:07:05 2015
	291000
	29270000

Profile start time and stop time is profrpt time.

All individual reports are for this duration only.

```
%simv -simprofile time -simprofile_start 290000 -  
simprofile_stop 29271423  
%profrpt -start 291000 -stop 29270000 <all other  
options>
```

## Diagnostics

The diagnostics feature sends you a message indicating when the simulation profile is turned on and turned off in the simulation log.

Note- [ON-SIMPROF] Simprofile is turned on

simulation profile is turned on at simulation time  
<time>

For example, simulation profile is turned on at simulation time 290000.

Note-[OFF-SIMPROF] Simprofile is turned off  
simulation profile is turned off at simulation time  
<time>

For example, simulation profile is turned off at simulation time 2971423.

---

## Limitations

The following are the limitations with this feature:

- Multiple start and stop on the same command line is not supported during profrpt stage.
  - Multiple start and stop on the same command line is not supported during simulation stage.
- 

## Isolating the Cost of Garbage Collection

VCS isolates the CPU time consumed by the garbage collection.

The cost of garbage collection is reported in the **Time Summary View** report. It is displayed as a sub-category under the KERNEL category.

---

## Use Model

The use model remains the same as that of the previous use model.

*Figure 6-75 The Time Summary View*

Time Summary View	
Component	Percentage
VERILOG	90.45%
Module	90.45%
KERNEL	8.82%
Garbage Collection	1.08%
HSIM	0.73%
TOTAL	100.00%

Time consumption for garbage collection under KERNEL.

---

## Isolating the Cost of Loading Design Database

In some large designs, loading the design database consumes lot of time and memory.

VCS isolates the CPU time and memory consumed for loading the design database.

The cost of loading the design database is reported in the **Time PLI/DPI/DirectC View** and **Peak Memory PLI/DPI/DirectC View** reports. It is displayed as a sub-category under the **PLI** category.

---

## Use Model

The use model remains the same as that of the previous use model.

**Figure 6-76 Time PLI/DPI/DirectC View**

Time PLI/DPI/DirectC View	
Name	Percentage
PLI	5.91 %
\$countdrivers	5.91 %
\$countdrivers /remote/vgr9/REGRUN/TD	4.72 %
/unit_RUNTIME/simprofile/RTVIR_profile/	
test1.v:9885	
Load Design Database	1.57 %
\$countdrivers /remote/vgr9/REGRUN/TD	1.18 %
/unit_RUNTIME/simprofile/RTVIR_profile/	
test1.v:9760	
Load Design Database	0.79 %
Total:	5.91 %

Cost of loading the design database.

**Figure 6-77 Peak Memory PLI/DPI/DirectC View**

Peak Memory PLI/DPI/DirectC View		
Name	Percentage	Memory
PLI	30.10 %	4.19 M
\$countdrivers	30.10 %	4.19 M
\$countdrivers /remote/vgr9/REGRUN/TD	28.32 %	3.95 M
/unit_RUNTIME/simprofile/RTVIR_profile/		
test1.v:9885		
Load Design Database	4.34 %	618.50 K
\$countdrivers /remote/vgr9/REGRUN/TD	1.75 %	249.40 K
/unit_RUNTIME/simprofile/RTVIR_profile/		
test1.v:9760		
Load Design Database	1.02 %	144.83 K
Total:	30.10 %	4.19 M

Cost of loading the design database.

## Support for Third-Party Shared Library Profiler Report

Simprofile report displays the detailed information of the memory cost of the individual shared library besides the total memory cost of all third-party shared libraries.

---

## Use Model

The use model remains the same as that of the previous use model.

*Figure 6-78 Memory Size of the Individual Third-Party Shared Library*

Peak Memory Summary View		
Component	Memory	Percentage
HSIM	5.90 M	4.84%
KERNEL	3.31 M	2.72%
VERILOG	576	0.00%
Module	436	0.00%
Package	140	0.00%
Anonymous	27.78 M	22.77%
Library/Executable	85.00 M	69.67%
VCS	82.00 M	67.21%
Third-party	3.00 M	2.46%
libc-2.5	1.34 M	1.10%
libstdc++	908.00 K	0.73%
libncurses	288.00 K	0.23%
libm-2.5	164.00 K	0.13%
libpthread-2.5	92.00 K	0.07%
libnss_files-2.5	48.00 K	0.04%
libgcc_s-4.1.2-20080825	48.00 K	0.04%
libdl-2.5	20.00 K	0.02%
TOTAL	122.00 M	100.00%

Profile report provide the detailed information for each individual library.

---

## Enhancements to VHDL Unified Simulation Profiler Report

VCS offers module view and construct view under a new sub-category called `process` added to the `VHDL` component. Also, there is a new sub-category called `VHDL-Kernel` added to the `Kernel` component. The previous categories, such as `simulation` and `monitor_evaluation` in the `VHDL` summary view is deprecated.

These enhancements help to analyze the time and memory consumption for VHDL with details using all profiler views, such as time/memory summary view, module view, and construct view. There is no change to the current simprofile usage model.

The following illustrations show the enhanced summary view, module view and construct views:

*Figure 6-1 Time Summary View*

Component	Time Summary		Percentage
	Time		
VERILOG	515.5 ms		36.30%
HSIM	476.6 ms		33.56%
KERNEL	427.9 ms		30.14%
total	1.4 s		100%

*Figure 6-2 Time Module View*

▼IQ_OP_REG	19.45 ms	1.37 %	19.45 ms	1.37 %
▼Always	19.45 ms	1.37 %	19.45 ms	1.37 %
►NoName {line:27606}	9.73 ms	0.68 %	9.73 ms	0.68 %
►NoName {line:27702}	9.73 ms	0.68 %	9.73 ms	0.68 %

*Figure 6-3 Time Construct View*

Time Construct View		
Name	Time	Percentage
▼VHDL Process	234.67 ms	26.67 %
SIM_MDL	222.93 ms	25.33 %
TOP_CLOCK_P1	11.73 ms	1.33 %
►Port	11.73 ms	1.33 %
total	246.40 ms	28.00 %

The construct view allows you to view the VHDL processes that consumed the most simulation time.

## Limitations

The feature has the following limitations:

- VHDL 2008 constructs are not supported.

---

## Enhancing the HSIM View With Simulation Profiler Report

In the simprofile summary view, the HSIM component consumes a significant part of gate-level simulation (GLS) and RTL designs.

In the previous releases, you cannot identify the bucket that consumes more time within the HSIM component. Therefore, this lack of visibility becomes an issue when you measure the simulation performance. [Figure 6-4](#) shows the Time summary view with the percentage allocation for HSIM, for releases before VCS N-2017.12.

*Figure 6-4 Time Summary View*

Component	Time Summary	
	Time	Percentage
VERILOG	515.5 ms	36.30%
HSIM	476.6 ms	33.56%
KERNEL	427.9 ms	30.14%
total	1.4 s	100%

Starting with this release, VCS offers better visibility for designs with a high HSIM usage using the following methods:

- Maps the HSIM cost back to regular Verilog constructs.

- Breaks down the global HSIM bucket into small HSIM buckets inside each module or instance.

*Figure 6-5 Updated Time Summary View*

Time Summary		
Component	Time	Percentage
VERILOG	921.6 ms	64.00%
KERNEL	451.2 ms	31.33%
HSIM	67.2 ms	4.67%
total	1.4 s	100%

Using these methods, the global HSIM cost is reduced from 33.56% to 4.67%. See [Figure 6-5](#) for the *Time summary view* with the HSIM percentage allocation encircled for releases VCS O-2018.09 and later.

## Benefits

- In the *Time Summary view*, the cost of global HSIM bucket is decreased.
- In *Time Module view*, time cost on each module is increased. As the HSIM cost is mapped back to regular constructs, more regular constructs are visible under each module. When the HSIM cost cannot be mapped back to a regular construct, a new HSIM construct is added to the module.

[Figure 6-6](#) and [Figure 6-7](#) for the *Time Module* views in VCS N-2017.12 release and prior, and O-2018.09 onwards respectively.

Figure 6-6 Time Module View in VCS N-2017.12 and prior

▼IQ_OP_REG	19.45 ms	1.37 %	19.45 ms	1.37 %
▼Always	19.45 ms	1.37 %	19.45 ms	1.37 %
►NoName {line:27606}	9.73 ms	0.68 %	9.73 ms	0.68 %
►NoName {line:27702}	9.73 ms	0.68 %	9.73 ms	0.68 %

Figure 6-7 Updated Time Module View

▼IQ_OP_REG	100.00 ms	7.25 %	100.00 ms	7.25 %
▼Hsim	60.00 ms	4.35 %	60.00 ms	4.35 %
►Hsim {Module:IQ_OP_REG}	60.00 ms	4.35 %	60.00 ms	4.35 %
▼Always	40.00 ms	2.90 %	40.00 ms	2.90 %
►NoName {line:27606}	20.00 ms	1.45 %	20.00 ms	1.45 %
►NoName {line:27702}	10.00 ms	0.72 %	10.00 ms	0.72 %
►NoName {line:27811}	10.00 ms	0.72 %	10.00 ms	0.72 %



# 7

## Diagnostics

---

This chapter covers various diagnostic tools and provides instructions on how to use these tools.

The following tasks are covered in this chapter:

- “Using Diagnostics”
- “Compile-time Diagnostics”
- “Runtime Diagnostics”
- “Post-Processing Diagnostics”
- “Sparse Memory Diagnostics”

---

# Using Diagnostics

This section describes the following topic:

- “[Using -diag Option](#)”

---

## Using -diag Option

Use the `-diag` option to enable the libconfig/timescale diagnostic messages at compile-time and VPI/VHPI diagnostic messages at runtime. The `-diag` option supports compile-time diagnostics on the `vcs` command-line and runtime diagnostics on the `simv` command-line.

### Syntax

Following is the syntax of the `-diag` option:

```
-diag <diag_arg>[,diag_arg][,diag_arg]..
```

Where, `diag_arg` is a diagnostic argument. [Table 7-1](#) lists the supported diagnostic arguments.

**Table 7-1 Supported Diagnostic Arguments**

Argument	Use Model	Description
libconfig	vcs -diag libconfig	Enables the library binding diagnostics. For more information, see “ <a href="#">Libconfig Diagnostics</a> ”.
timescale	vcs -diag timescale	Enables timescale diagnostics. For more information, see “ <a href="#">Timescale Diagnostics</a> ”.
vpi	simv -diag vpi	Enables VPI diagnostics. For more information, see “ <a href="#">Diagnostics for VPI/VHPI PLI Applications</a> ”.
vhpi	simv -diag vhpi	Enables VHPI diagnostics. For more information, see “ <a href="#">Diagnostics for VPI/VHPI PLI Applications</a> ”.
all	vcs -diag all	Enables the libconfig and timescale diagnostics.
	simv -diag all	Enables the vpi and vhpi diagnostics.
help	vcs -diag help simv -diag help	<p>Displays the following help message:</p> <p>Usage for -diag flag: <b>-diag &lt;option&gt;,&lt;option&gt;,...</b></p> <p>Options:</p> <ul style="list-style-type: none"> <li><b>all</b> Enable all diagnostics</li> <li><b>help</b> Display this message</li> <li><b>libconfig</b> Library binding diagnostics (compile time)</li> <li><b>timescale</b> Timescale diagnostics (compile time)</li> <li><b>vpi</b> VPI diagnostics (simulation time)</li> <li><b>vhpi</b> VHPI diagnostics (simulation time)</li> </ul>

---

## Compile-time Diagnostics

This section describes the following topics:

- “Libconfig Diagnostics”
- “Timescale Diagnostics”
- “Generating Information on Unused Libraries at vlogan”
- “Generating Information on Unused Libraries at VCS”
- “Obtaining Statistics on Package Utilization”

---

### Libconfig Diagnostics

You can use the `libconfig` option, as shown below, to enable libconfig diagnostics:

```
% vcs -diag libconfig
```

This option provides the library binding diagnostics at compile-time. It generates physical mappings of user-defined libraries and the default work library specified by VCS.

For each VHDL/Verilog instance, this option generates the instance name, location, binding rule, and entity-architecture pair/module to which it is bound.

### Example

Consider the following test case:

```
leaf.vhd
```

```

=====
entity leaf is
end entity leaf;

architecture behv of leaf is
begin
end architecture;

mid.vhd
=====
entity mid is
end entity mid;

architecture behv of mid is
  component leaf
    end component leaf;
begin
  a0: leaf;
end architecture;

top.v
=====
module top();
  mid inst1 ();
endmodule

```

**Perform the following commands:**

```

% vhdlan leaf.vhd -work lib1
% vhdlan mid.vhd -work lib1
% vlogan top.v -work lib2
% vcs top -diag libconfig -l log

```

**Following is the output:**

```

Setup library mapping:
  DEFAULT : /remote/vtghome13/diag./work/
  LIB1 : /remote/vtghome13/diag./lib/
  LIB2 : /remote/vtghome13/diag./lib/

```

```
Work logical library name set to 'DEFAULT'.
Default library search order:
    DEFAULT
    LIB1

    instance: LIB1.top
                "/remote/vtghome13/diag/top.v", 1
        rule: Top Module
    module: LIB1.top
                "/remote/vtghome13/diag/top.v", 1
```

Top Level Modules:

```
    top
    instance: top.inst1
                "/remote/vtghome13/diag/top.v", 3
        rule: Direct Instantiation
entity: LIB1.MID
                "/remote/vtghome13/diag/mid.vhd", 3
architecture: BEHV
                "/remote/vtghome13/diag/mid.vhd", 6

    instance: top.inst1.A0
                "/remote/vtghome13/diag/mid.vhd", 10
        rule: Default Binding
    entity: LIB1.LEAF
                "/remote/vtghome13/diag/leaf.vhd", 4
architecture: BEHV
                "/remote/vtghome13/diag/leaf.vhd", 7
```

#### Note:

- If the **-l** option is specified, the output is dumped into the corresponding text file.
- If the **-sml** option is specified, smartlog output is also dumped into the corresponding smartlog file.

---

## Timescale Diagnostics

You can use the `timescale` option, as shown below, to enable timescale diagnostics:

```
% vcs -diag timescale
```

This option generates timescale diagnostic message for each module during VCS elaboration phase. This allows you to understand how VCS has scaled delays in its design, and helps you to quickly identify, localize and fix the timescale issues.

Note:

- The output is printed on the `STDOUT` by default.
- If the `-l` option is specified, the output is dumped into the corresponding text file.
- If the `-sm` option is specified, smartlog output is also dumped into the corresponding smartlog file.

## Example

### Example 1: Module has `timescale

Consider the following test case `test.v`, which contains module `test` with ``timescale as 1ns/1ns`:

```
`timescale 1ns/1ns
module test;
    initial
        $printtimescale;
endmodule
```

Enabling timescale diagnostics at elaboration time using `-diag timescale`:

```
% vcs test.v -diag timescale
```

Following is the output:

```
Parsing design file 'test.v'
Top Level Modules:
    test
TimeScale is 1ns/1ns
module 'test' gets time unit '1ns' from source code '/remote/
vgscratch7/timescale_diag/tests/cft/sva_bind/l1_sv/
Source/test.v', 1
module 'test' gets time precision '1ns' from source code '/
remote/vgscratch7/timescale_diag/tests/cft/sva_bind/
l1_sv/Source/test.v', 1
Starting vcs inline pass...
1 module and 0 UDP read.
recompiling module test
if [ -x ..//simv ]; then chmod -x ..//simv; fi
g++ -o ..//simv -melf_i386 -m32      -Wl,-whole-archive     -
-Wl,-no-whole-archive _vcsobj_1_1.o  5NrI_d.o
...
..//simv up to date
```

From the above output, you can figure out which module gets what timescale at elaboration, and also the reason why and from where the module got that timescale.

```
module 'test' gets time unit '1ns' from source code '/remote/
vgscratch7/timescale_diag/tests/cft/sva_bind/l1_sv/
Source/test.v', 1
module 'test' gets time precision '1ns' from source code '/
remote/vgscratch7/timescale_diag/tests/cft/sva_bind/
l1_sv/Source/test.v', 1
```

In the above example, as mentioned `timescale 1ns/1ns on line# 1, the module gets the time unit of 1ns and time precision of 1ns.

## **Example 2: Passing -timescale from vcs command-line**

Consider the following testcase test.v:

```
module test;
    initial
        $printtimescale;
endmodule
```

Perform the following command:

```
% vcs test.v -diag timescale -timescale=1ns/1ns
```

Following is the output:

```
Parsing design file test.v
Top Level Modules:
    test
TimeScale is 1ns/1ns
module 'test' gets time unit '1ns' from vcs command option
module 'test' gets time precision '1ns' from vcs command option
Starting vcs inline pass...
1 module and 0 UDP read.
recompiling module test
if [ -x ..//simv ]; then chmod -x ..//simv; fi
g++ -o ..//simv -melf_i386 -m32      -Wl,-whole-archive      -
-Wl,-no-whole-archive _vcsobj_1_1.o  5NrI_d.o
...
..//simv up to date
```

In the following command, timescale is passed at elaboration using the -timescale option.

```
% vcs test.v -diag timescale -timescale=1ns/1ns
```

The diagnostics message printed on the output is as follows:

```
module 'test' gets time unit '1ns' from vcs command option
```

```
module 'test' gets time precision '1ns' from vcs command
option
```

---

## Generating Information on Unused Libraries at vlogan

VCS allows you to generate information on unused libraries at vlogan stage for three-step flow. VCS displays a lint message if there are more libraries specified than required with the `-liblist` option at vlogan stage.

### Use Model

You can use the `-diag libdepends` option at vlogan stage, as shown, to generate information on unused libraries at vlogan stage:

```
% vlogan -diag libdepends
```

VCS displays a lint message `ULILL-L` for unused libraries and prints the information on required libraries in `vlogan_statistics.out` file in the library path.

### Usage Example

Consider the following examples:

#### *Example 7-1 Generating Unused Libraries at vlogan stage*

```
tb.v
=====
`define SUB d.u.s
package p1;
int option;
wire w1;
endpackage

module tb;
```

```

import p1::*;
`ifdef INIT initial
$display("TB1 is running"); `endif
dut d();
bot bot1();
endmodule

module bot();
import p1::*;
class B;
endclass
B b = new;
endmodule

```

**Perform the following command:**

```
% vlogan -sverilog tb.v -diag libdepends -liblist temp2 -liblist temp3
```

**VCS generates the following lint message at compile time and shares the list of unused libraries passed at vlogan step using the `-liblist` option:**

```

Lint-[ULILL-L] Unused library in liblist
Current analysis step does not require the following
libraries:
temp2
temp3

```

**VCS also prints the following information of the used libraries in a file. This file is generated as <library\_folder>/vlogan\_statistics.out file.**

```

Module: tb
Required Packages:
    p1
Required Libraries:
    DEFAULT
Module: bot

```

```
Required Packages:  
    p1  
Required Libraries:  
    DEFAULT
```

---

## Generating Information on Unused Libraries at VCS

VCS generates information on unused libraries passed at VCS stage. VCS displays a lint message that provides the information on the unused libraries mentioned in the `synopsys_sim.setup` file. The lint message also shows information on the amount of disk space consumed by each of the unused libraries.

### Use Model

You can use the `-diag libusage` option at VCS stage, as shown, to generate the information on unused libraries at VCS stage:

```
% vcs -diag libusage
```

VCS displays a lint message `LINU-L` for unused libraries and prints the information on required libraries in `elaboration_statistics.out` file in the library path.

#### *Example 7-2 Generating Unused Libraries at VCS stage*

```
tb.v  
=====  
`define SUB d.u.s  
package p1;  
int option;  
wire w1;  
endpackage  
  
module tb;  
import p1::*;  
`ifdef INIT initial
```

```

$display("TB1 is runing"); `endif
dut d();
bot bot1();
endmodule

module bot();
import p1::*;

class B;
endclass
B b = new;
endmodule

synopsys_sim.setup
=====
WORK > DEFAULT
DEFAULT:./work
lib3:./lib3

```

**Perform the following commands:**

```
% vlogan -sverilog tb.v
% vcs tb -diag libusage
```

VCS generates the following lint message at compile time and shares the list of unused libraries passed at vcs step using the **-libusage** option

```
Lint-[LINU-L] Library included but not used
None of the cells from the following libraries has been used:
lib3 (disk size: 4593 bytes)
```

VCS also prints the following information of the used libraries in a file. This file is generated as **<library\_folder>/elaboration\_statistics.out** file.

```
Library Cell Usage:
DEFAULT: (7/8)
```

As shown in the output, 7 out of 8 cells are used. To get details on the used cells, use the `-diag libdepends=verbose` option.

---

## Obtaining Statistics on Package Utilization

VCS obtains statistics on consumption of packages in the design. The statistics shows information on number of classes, tasks, and functions from the package that are actually used in the design.

### Use Model

You can use the `-diag pkgusage` compile-time option, as shown, to obtain statistics on package utilization.

```
% vlogan -diag pkgusage
```

VCS prints the information on package utilization in `elaboration_statistics.out` file in the library path.

#### *Example 7-3 Obtaining Statistics on Package Utilization*

```
tb.v
=====
`define SUB d.u.s
package p1;
int option;
wire w1;
endpackage

package p12;
import p1::*;
function f();
    $display(option);
endfunction
endpackage
module tb;
    import p12::*;

```

```
`ifdef INIT initial $display("TB1 is runing"); `endif
      dut d();
initial begin
      f(); // XMR
end
endmodule
```

**Perform the following commands:**

```
% vlogan -sverilog tb.v
% vcs tb -diag pkgusage
```

**VCS prints the following information of the used libraries in a file.  
This file is generated as <library\_folder>/  
elaboration\_statistics.out file.**

```
Package usage diagnostics:std: Classes: 0/6, Functions: 0/
0, Tasks: 0/0
p1: Empty

Package usage diagnostics:std: Classes: 0/6, Functions: 0/
0, Tasks: 0/0
p1: Empty
p12: Classes: 0/0, Functions: 1/1, Tasks: 0/0
```

---

## Runtime Diagnostics

This section describes the following topics:

- “[Diagnostics for VPI/VHPI PLI Applications](#)”
  - “[Keeping the UCLI/Verdi Prompt Active After a Runtime Error](#)”
  - “[Diagnosing Quickthread Issues](#)”
- 

### Diagnostics for VPI/VHPI PLI Applications

As per LRM, VPI/VHPI remain silent when an error occurs. The application checks for error status to report an error. If error detection mechanisms are not in place, the C code of the application must be modified and recompiled. In addition, you may need to recompile the HDL code, if required.

However, you can use the following new runtime diagnostics options to make the PLI application to report errors without code modification:

- `-diag vpi`
- `-diag vhpi`

Furthermore, reporting provides you the information related to the HDL code context, wherever applicable, to help fix problems with a faster turnaround time.

Note:

- If the `-l` option is specified, the output is dumped into the corresponding text file.

- If the `-sml` option is specified, smartlog output is also dumped into the corresponding smartlog file.

For example, consider the following test case `tokens.v` and files `value.tab` and `value.c`.

*Example 7-4 tokens.v*

```

module top;
    reg r;

    initial begin
        #5;
        $putValue("sys_top.rst", 1'b1);

        #1 $finish;
    end
endmodule

module sys_top;
    wire rst;

    assign db.A = rst;
endmodule

module db;
    wire Y;
    wire A;

    my_buf b1(Y, A);

    initial begin
    end
endmodule

module my_buf(Y, A);
    output Y;
    input A;

```

```

    buf #5 (Y, A);
endmodule

```

*Example 7-5 value.tab*

```
$putValue  call=put_value  acc=rw:top
```

*Example 7-6 value.c*

```

#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include "sv_vpi_user.h"

void put_value() {
    vpiHandle sysTfH, argI, objH, valueH;
    s_vpi_value value;
    s_vpi_time time_s;
    int format;
    p_vpi_value value_p;
    p_vpi_time time_p;

    sysTfH = vpi_handle(vpiSysTfCall, 0x0);

    argI = vpi_iterate(vpiArgument, sysTfH);

    objH = vpi_scan(argI);
    valueH = vpi_scan(argI);

    if (vpi_get(vpiType, objH) == vpiConstant) {
        value.format = vpiStringVal;
        vpi_get_value(objH, &value);
        vpi_free_object(objH);
        if(strcmp(value.value.str, "null")) {
            objH = vpi_handle_by_name(value.value.str, 0x0);
        } else {
            objH = 0x0;
        }
    }
}

```

```

    time_p = 0x0;
    value.format = vpiIntVal;
    vpi_get_value(valueH, &value);
    value_p = &value;

    vpi_put_value(objH, value_p, time_p, vpiNoDelay);
}

```

**Compile and run the tokens.v code as follows:**

**Two-step Flow:**

```

% vcs -sverilog +vpi -P value.tab value.c tokens.v

% simv -diag vpi

```

**Three-step Flow:**

```

% vlogan tokens.v

% vcs top -sverilog +vpi -P value.tab value.c

% simv -diag vpi

```

Here, the user application tries to write a value on the sys\_top.rst signal, but there is no write permission enabled on sys\_top. So VPI generates an error message and prints the HDL information, as follows:

```
Error-[VPI-WPNEN] VPI put value error
At time 5, in PLI routine called from tokens.v, 6
  In vpi_put_value call, write permission not enabled.
  Please add capability 'wn' to signal 'sys_top.rst' of module 'sys_top'.
  Please refer to the VCS User Guide, Section 'Specifying ACC Capabilities
  PLI functions' in the chapter 'Using PLI' for further details.

At time 5, in the PLI application '$putValue' called from tokens.v, 6:
  vpiSeverity - vpiError
  PLI Routine - vpi_put_value
  Reference Object - rst
  Reference Scope - sys_top
  Reference vpiType - vpiNet
  Path - /remote/us01home17/[REDACTED]/12-09/VPI_EM/tokens.v, 14
  Delay Propagation Method - 1
```

---

## Keeping the UCLI/Verdi Prompt Active After a Runtime Error

VCS allows you to debug an unexpected error condition by not exiting and keeping the UCLI or Verdi command prompt active for debugging commands.

Verdi or UCLI command prompt remains active when there is an error condition, allowing you to examine the current simulation state (the simulation stack, variable values, and so on) so you can debug the error condition.

### UCLI Use Model

If simv is executed from UCLI, perform the following steps to enable this feature:

1. Specify the following UCLI configuration command in a Tcl file ( see [Example 7-8](#)) or in `$HOME/.synopsys_ucli_prefs.tcl` file:

```
config onfail enable [failure_type]
```

Where `failure_type` is optional. It allows you to specify the failure type. [Table 7-1](#) lists the types of failures which are normally observed during an unexpected runtime error.

**Table 7-2 Types of Failures**

Failure Type	Failure Description
sysfault	Assertion or signal (including segfault)
{error <regex>}	Error for which the tag matches regex. The tag of an error can be seen in the error message (Error-[TAG]).
fatal	Fatal error for which VCS currently dumps a stack trace.
all	All failures (default)

**Note:**

- You can divide the configuration of `onfail` into multiple configuration commands.
- You can use the `config onfail disable` configuration command to disable this feature.

**Example**

The following command enables you to catch system faults, DT.\* errors, and NOA errors:

```
config onfail enable sysfault {error DT.*} {error NOA}
```

You can also specify the above command as three different configuration commands:

```
config onfail enable sysfault
config onfail enable {error DT.*}
config onfail enable {error NOA}
```

2. Use the following UCLI command to get a UCLI prompt when a runtime error occurs:

```
% simv -ucli -i file_name.tcl
```

or

```
% simv -ucli  
ucli% do file_name.tcl
```

Where *file\_name.tcl* is the Tcl file that contains the config onfail enable command and run script (see [Example 7-8](#)).

Note:

You must run the simulation using the `run` command by specifying it in a Tcl file. You can also specify the config onfail enable command in the same Tcl file, but instead, if you use `simv -ucli` at the UNIX prompt to run the simulation, then UCLI exits when there is a failure.

## Automating User Actions on Failure

You can create the `onfail` routine to automate some actions (like printing specific message, collecting data into a file, and so on) when an unexpected crash happens during runtime. You can create this routine in your script or in the `.synopsys_ucli_prefs.tcl` file.

If you declare this routine, and the `onfail` configuration is enabled, then `simv` calls the `onfail` routine before going into the UCLI prompt. If you do not want to go into the UCLI prompt, you can call the UCLI `exit` command from the routine.

## Verdi Use Model

By default, Verdi enables the `onfail` configuration on all types of failures. Verdi systematically enables the `onfail` configuration on all error types.

When you enable the onfail configuration, simv stays active and continue to respond to Verdi queries. Also, Verdi shows the location of the error with the simulation pointer (pink arrow in the source view), and the Stack pane shows the current HDL stack. You can use value annotation to obtain signal values in order to debug the issue.

## UCLI Usage Example

Consider the following test case `test.v`. This code causes simv to exit during simulation:

*Example 7-7 UCLI Prompt on Error Test Case (test.v)*

```
module test;
    class Packet;
        int _a;

        function void set (int a);
            _a = a;
        endfunction
    endclass

    initial begin
        Packet pkt;
        reg a;
        pkt.set(a);
    end
endmodule
```

Compile the `test.v` file, as shown below:

Two-step Flow:

```
% vcs -sverilog -debug_access+all test.v
```

Three-step Flow:

```
% vlogan test.v
```

```
% vcs test -sverilog -debug_access+all
```

If you run the above test case using the `simv -ucli` command, VCS generates the following NOA error message:

*Figure 7-1 NOA Error Message*

```
Error-[NOA] Null object access
test.v, 13
  The object is being used before it was constructed/allocated.
  Please make sure that the object is newed before using it.

#0 in unnamed$$_0 at test.v:13
#1 in test

          V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:      0.790 seconds;           Data structure size:  0.0Mb
Mon Jan 23 02:59:51 2012
```

Create the following Tcl file to catch the above error and analyze it inside an onfail routine:

*Example 7-8 Tcl File (test.tcl)*

```
onfail {
    set err_msg "Stopped in "
    append err_msg [scope]
    puts $err_msg
}
config onfail enable {error NOA}
run
```

Run the `test.tcl` file using the following command to keep the UCLI prompt active after the NOA error, as shown in [Figure 7-2](#):

```
% simv -ucli -i test.tcl
```

*Figure 7-2 Viewing the UCLI Prompt After Failure*

```
ucli% config -onfail enable {error NOA}
ucli% run

Error-[NOA] Null object access
test.v, 13
The object is being used before it was constructed/allocated.
Please make sure that the object is newed before using it.

#0 in unnamed$$_0 at test.v:13
#1 in test

file test.tcl, line 7: System Fault
Stopped in test

Pause in file test.tcl, line 7
pause% █
```

The onfail routine is executed after the NOA error is generated.

## Limitations

- You cannot specify an onfail routine to be executed on error in Verdi.

---

## Diagnosing Quickthread Issues

VCS is now equipped with a better mechanism to report VCS runtime crashes caused by certain problems with quickthreads used during VCS runtime. You will get clear feedback as to what went wrong and which thread is causing the crash thereby enabling you to take specific action to circumvent the issue.

## Diagnosing Quickthread Issues in DPI

While calling an import DPI routine that either calls any SystemVerilog blocking/time consuming task or is declared as a context task, VCS creates quickthread with a runtime memory of 256 KB (by default) for the call chain that originated from the import DPI routine. Such import DPI routine is also called a heavy weight DPI routine. If the call chain from the import DPI routine uses more memory than the pre-allocated buffer (due to large local variables and/or a deep call chain), it causes a segmentation fault and the following runtime error message is generated:

Note-[VCS-QTHREAD-OVERRUN] Stack of quickthread maybe too small

The simulation received a fatal segmentation violation signal SEGV and will end because it accessed protected stack guard memory. This memory belongs to the thread 'top'. It is likely, but not certain that a stack overflow in this thread caused the segmentation violation (SEGV). It may also be caused by a different, unknown problem and the quickthread is not related.

The suspected quickthread belongs to the DPI domain. Its stack has a size of 4194300.96 K bytes and is located from address '0x9eb43dc' to '0x9eb5000'.

Its redzone has a size of 4.00 K bytes and is located from address '0x9eb5000' to '0x9eb4000'.

The SEGV happened at address '0x9eb4390' which is 3184 bytes into the redzone.

Increase the stack size for this thread and check whether this solves the problem. See the VCS user guide for more information.

You should give a conservative (less restrictive) estimate about the DPI quickthread runtime memory based on the import DPI routine code. If the default runtime memory of 256 KB is too restrictive, the DPI quickthread runtime memory size can be set with the environment variable `DPI_QSTACK_SIZE` in the following ways:

- Before running the simulation as follows:

```
% setenv DPI_QSTACK_SIZE <number>
```

Or

- From the DPI application using the `setenv` function as follows:

```
setenv ("DPI_QSTACK_SIZE", "<number>", 1);
```

Here, the `<number>` should be provided in bytes.

For example, to set the limit of 8 KB, the value should be  $8 * 1024 = 8192$ .

If the default DPI quickthread runtime memory size is overwritten and a heavy weight DPI routine is invoked, VCS issues the following message at runtime:

Note-[DPI-RTMBSS] DPI runtime memory buffer size set  
The size of the runtime memory buffer for invoking time consuming DPI task(s) has been overwritten from default 256KB to 8192B (8KB) by environment variable `DPI_QSTACK_SIZE`.

## Diagnosing Quickthread Issues in SystemC

VCS reports runtime crashes in the following two scenarios:

- A quickthread overruns its allocated stack
- Simulation runs out of memory due to quickthread stacks

The default stack size of a SystemC thread (either `SC_THREAD` or `SC_CTHREAD`) is 1MB and the default stackguard size is 16KB.

If a quickthread overruns its allocated stack, then it will probably try to read/write into its redzone. This causes an SEGV with the diagnostic message. Here is an example:

```
Error-[SC-VCS-QTHREAD-OVERRUN] Stack of quickthread maybe too small
```

The simulation received a fatal segmentation violation signal SEGV and will end, because it accessed protected stack guard memory. This memory belongs to the thread 'top.ref\_model\_0.cpu.ALU'. It is likely, but not certain that a stack overflow in this thread caused the segmentation violation (SEGV). It may also be caused by a different, unknown problem and the quickthread is not related.

The suspected quickthread belongs to SystemC domain. Its stack has a size of 60 K bytes and is located from address '0x800a0000' to '0x800a0efff'.

Its redzone has a size of 4 K bytes and is located from address '0x800a0f000' to '0x800a0ffff'.

The SEGV happened at address '0x800a0f004' which is 5 bytes into the redzone.

Increase the stack size for this thread and check whether this solves the problem. This can be done by calling the `stack_size()` method within the SC\_CTOR. Alternatively, start the simulation with '`simv -sysc=stacksize:10M`'. See the VCS SystemC user guide for more information.

## Limitations

The SC-VCS-QTHREAD-OVERRUN diagnostic applies only to quickthreads. It is not available if you use POSIX threads in SystemC by defining environment `SYSC_USE_PTHREADS`.

## Simulation Runs Out of Memory Due to Quickthread Stacks

Each quickthread allocates memory for its stack. Simv may run out of memory due to this. When allocation of memory for a SystemC stack of a quickthread fails, a message like the following is printed:

```
Error-[SC-VCS-QTHREAD-ALLOC] Thread memory allocation failed
```

```
The creation of thread 'top.sc_thread_04' in the SystemC
domain failed
because its stack of 64MB could not be allocated. Currently,
149MB stack
memory are allocated by 95 threads.
```

```
Details about stack allocation:
(sorted by size in decreasing order)
32MB total (31.9MB stack + 19.9KB guard) in
SystemC:top.sc_thread_05
16MB total (15.9MB stack + 19.9KB guard) in
SystemC:top.sc_thread_06
8.01MB total (7.99MB stack + 19.9KB guard) in
SystemC:top.sc_thread_07
(~50 lines removed, we show approx. 50..60 stack frames
, ordered by size, largest first)
...(truncated)...
Total: 149MB qthread stack memory used in 95 threads.
```

```
If this was a 32 bit simulation, consider a 64 bit
simulation. You can also
decrease the stack size for other threads. This can be
done by calling the
stack_size() method within the SCCTOR. Alternatively,
start the simulation
with e.g. 'simv -sysc=stacksize:500k'. See the VCS user
guide, chapter Using SystemC for more information.
```

## Reducing or Turning Off Redzones

You can decrease the number of redzones or turn them off altogether if the number of quickthreads you are using is exceedingly large. For instance, if the quickthreads are reaching the limit set in your OS, then some of the operations may fail. To avoid such a situation, you may want to decrease the number of the redzones or turn them off completely. Though the diagnostic is not supported when a particular thread overruns its stack, you would still increase the chances of running your simulation without any issues.

You can use the following environment variable to either decrease the number of redzones or turn them off completely. To decrease the number of redzones, you must set the following environment variable to a value greater than 2000 and less than 30000. For example:

```
setenv SNPS_VCS_SYSC_RESERVED_MAP_COUNT 10000
```

Setting the above environment variable to a value higher than 30000 will turn off the redzones completely.

---

## Post-Processing Diagnostics

This section describes the following topic:

- “[Using the vpdutil Utility to Generate Statistics](#)”

---

### Using the vpdutil Utility to Generate Statistics

The `vpdutil` utility generates statistics of the data in the VPD file. This utility takes a single VPD file as an input. You can specify options to this utility to query at design, module, instance, and node levels.

This utility supports time ranges and input lists for query on more than one object. Output is in ASCII to stdout with option to redirect to an output file.

### The vpdutil Utility Syntax

The syntax of the `vpdutil` utility is as follows:

```
vpdutil <input_vpd_file>
    [-help]
    [-vc_info]
    [-tree [-lvl <level>][-source]]
    [-vc_info_detail]
    [-info]
    [-design]
    [-find_forces]
    [-start <Time> -end <Time>]
    [-find_glitches]
    [output_file_name]
```

## **Options**

`-h/help`

Displays the options to be used with the `vpdutil` application.

`output_file_name`

Writes the output of the `vpdutil` application to a file instead of `stdout`.

### **Options for VPD File Information**

`-info`

Prints the basic information present in the header of the VPD file.

### **Options for Design Information**

`-design`

Prints statistics about static design hierarchy in the VPD file.

`-tree`

Prints the full hierarchy tree in the VCD-like (not vcd compatible) format.

`-lvl <level>`

Prints the tree with the hierarchy depth=level.

`-source`

Prints source file/line data to tree.

### **Options for Value Change Information**

`-vc_info`

Displays value change information with the number of dump off events, force events, glitch events, and repeat count events.

`-vc_info_detail`

Prints the detailed value change summary statistics about the given VPD file.

`-find_forces`

Displays forces on node and the times when forces occurred.

`-start <Time> -end <Time>`

Enables the collection of value change data between start time to end time.

`-find_glitches`

Prints the list of nodes with glitches and the time when glitches occurred, if the glitch capturing is enabled during the simulation.

---

## Sparse Memory Diagnostics

You can use the `-Xkeyopt=sparseDiag` option at compile time to enable sparse memory diagnostics. This option prints detailed diagnostics about the memories that are inferred as sparse and the memories that are not inferred as sparse. This option is also useful for debugging and determining opportunities for sparse memory inferencing. This section describes the following:

- [Compile Time Options](#)
- [Runtime Options](#)

- [Sparse Disable Options](#)
- 

## Compile Time Options

The compile time diagnostics options are as follows:

`-Xkeyopt=sparseDiag`

This is a basic option. All the other options are used along with this option.

This option prints all the memories above the default threshold of 512 MB and mention sparse inferred/non-sparse. For the memories which are not inferred as sparse, it prints the offending construct which inhibits the memory from being inferred as sparse.

**Table 7-3 Options with -Xkeyopt=sparseDiag**

<b>Options with -Xkeyopt=sparseDiag+&lt;...&gt;</b>	<b>Description</b>
print	This option prints information about memory above threshold, whether they are inferred sparse or not. Default threshold value is 512MB.
printDetails	This option prints information about memory above threshold, whether they are inferred sparse or not. For the memories which are not inferred as sparse, it prints the offending construct which inhibits the memory from being inferred as sparse. The default threshold value is 512MB.
elabThres+<int>	<p>This option specifies the threshold above which all memories are considered under <code>sparseDiag</code>. However, if <code>elabThres</code> is not specified, 512MB is considered the default threshold.</p> <p>The size must be specified in MB. This option can be specified alone or at the right end of the option. You can compute the threshold (in MB) using the <i>Elaboration Threshold in MB = P * U * f * (Y? 2: 1) / (8 * 1024 * 1024)</i> formula. Here, P = Packed Dim, U = Unpacked Dim, F = Flat Memory Count and Y = 4State</p>
force	This option enables memories above a specified threshold to be sparse. The default threshold value is 512MB.
noDynUnsparse	This option stops dynamic unsparsing for memories at runtime.
logFile+<FileName>	This option prints all sparse memory diagnostics in specified file name. If this option is not specified, it prints in <code>snps_SparseMemComp&lt;name&gt;.log</code> file which is available at the present working directory. For PC flow, this logfile is created in child partition directories. Here the name is generated using PID, date combination so that the log file name is unique for every compilation. This option can be specified alone or at the right end of the option combination.
noDesignNames	This option does not print module/node names used in a design.

**Note:**

You can use all the above options in conjunction. For example, to print diagnostics for all the memories above 100MB, you can use the `-Xkeyopt=sparseDiag+elabThres+100` option.

## **Example**

Consider the following testcase:

*Example 7-9 test.v*

```
module tb;

    modA A1();
    modA A2();

endmodule

module modA();

    reg [40:0] mda1[40000000:0];
    bit [40:0] mda2[400:10000000];

    initial begin
        mda1[0][1] = 1'b0;
        mda2[500][1] = 1'b1;
        $finish;
    end

endmodule
```

Run the example using the following command:

```
% vcs -sverilog -Xkeyopt=sparseDiag test.v
```

The output generated is as follows:

```
SPMEM_ELAB_THRES_DIAG:: Scope "modA", Node "mda1"
is Inferred Sparse BitSizes=41 ArrSize=40000001
FlatInstCnt=2 State=4STATE
```

In this example, if you want to find the threshold, you can use the *Elaboration Threshold in MB = P \* U \* f \* (Y? 2: 1) / (8 \* 1024 \* 1024)* formula. Here, P= 41, U = 40000001, f = 2, Y= 2. Therefore,

$$\text{Elab Threshold in MB} = 41 * 40000001 * 2 * 2 / (8 * 1024 * 1024) = 782.012 \text{ MB}$$

---

## Runtime Options

The runtime diagnostic options are as follows:

```
-sparse+logFile[+<fileName>] -sparse+stats
```

This option prints all the sparse memory diagnostics in specified file name. If this option is not specified, it prints in `snpS_SparseMemRun<name>.log` available at the present working directory. Here, the name is generated using PID, date combination so that the log file name is unique for every compilation. This option can be specified alone or at the right end of the option combination.

For example, consider the example "[test.v](#)". To print the sparse memory diagnostics in file abc, use the following commands:

```
-% vcs -sverilog -Xkeyopt=sparseDiag test.v
-% simv -sparse+logFile+abc -sparse+stats
```

A file abc is created in your working directory. The following is the format of the abc log file.

*Figure 7-3 abc logfile*

```
-----SPARSE MEMORY SUMMARY-----
test.v:10 :: mda1
Scope : tb.A1
WordSize: 41, ArraySize: 40000001 LaneSize: 64
TotalLanes: 625001      LanesWritten: 1   Occupancy:
0.000160%
4-state, STD/BYTE LAYOUT, vcs-memory:12 units
Entries:1,
test.v:11 :: mda2
Scope : tb.A1
WordSize: 41, ArraySize: 9999601  LaneSize: 64
TotalLanes: 156244      LanesWritten: 1   Occupancy:
0.000640%
2-state, NON-STD/WORD LAYOUT, vcs-memory:8 units
Entries:64,
test.v:10 :: mda1
Scope : tb.A2
WordSize: 41, ArraySize: 40000001 LaneSize: 64
TotalLanes: 625001      LanesWritten: 0   Occupancy:
0.000000%
4-state, STD/BYTE LAYOUT, vcs-memory:12 units
Entries:0,
test.v:11 :: mda2
Scope : tb.A2
WordSize: 41, ArraySize: 9999601  LaneSize: 64
TotalLanes: 156244      LanesWritten: 0   Occupancy:
0.000000%
2-state, NON-STD/WORD LAYOUT, vcs-memory:8 units
Entries:0,
-----DONE-----
```

**-sparse+noDynUnsparse**

This option stops dynamic unsparsing of all the design memories.

**-sparse+stats**

This option reports sparse memory stats at the end of simulation, that is, module name, node name, dynamically unsparsed or not, unpacked dimension, packed dimension and count of memory words written till end of simulation.

Consider the example “[test.v](#)” . Run the example using the following commands:

- % vcs -sverilog -Xkeyopt=sparseDiag test.v
- % simv -sparse+stats

The sparse memory summary report is generated as follows:

*Figure 7-4 Sparse Memory Summary Report*

```
-----SPARSE MEMORY SUMMARY-----
test.v:10 :: mda1
Scope : tb.A1
WordSize: 41, ArraySize: 40000001      LaneSize: 64      TotalLanes: 625001      LanesWritten: 1      Occupancy: 0.000160%
4-state, STD/BYTE LAYOUT, vcs-memory:12 units
Entries:1,
test.v:11 :: mda2
Scope : tb.A1
WordSize: 41, ArraySize: 9999601      LaneSize: 64      TotalLanes: 156244      LanesWritten: 1      Occupancy: 0.000640%
2-state, NON-STD/WORD LAYOUT, vcs-memory:8 units
Entries:64,
test.v:10 :: mda1
Scope : tb.A2
WordSize: 41, ArraySize: 40000001      LaneSize: 64      TotalLanes: 625001      LanesWritten: 0      Occupancy: 0.000000%
4-state, STD/BYTE LAYOUT, vcs-memory:12 units
Entries:0,
test.v:11 :: mda2
Scope : tb.A2
WordSize: 41, ArraySize: 9999601      LaneSize: 64      TotalLanes: 156244      LanesWritten: 0      Occupancy: 0.000000%
2-state, NON-STD/WORD LAYOUT, vcs-memory:8 units
Entries:0,
-----DONE-----
```

-sparse+unsparselThresPercent+<Int>

This option specifies the threshold Integer percentage beyond which the memory is dynamically unsparsed. By default, the threshold is 20%. This option can be specified alone or at the right end of the option.

---

## Sparse Disable Options

The following are the sparse disable options:

`-Xkeyopt=sparseDisable`

This option disables sparse inferencing. It disables the auto inferencing and unsparse pragma specified by you and config file based sparse memories.

`-Xkeyopt=sparseDisable+autoInfer`

This option disables auto inferencing of sparse memories. The memories that are marked as sparse using either pragma or config file is considered for sparsing.



# 8

## VPD, VCD, and EVCD Utilities

---

This chapter describes the following:

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

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

- Value Change Dumping (VCD)

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

- VCDPlus Dumping (VPD)

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

- Extended VCD (EVCD)

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

VCS also provides several post-processing utilities to:

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

## Advantages of VPD

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

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

- Can save source statement execution data.

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

---

## Dumping a VPD File

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

- [Using System Tasks](#) - For Verilog designs.
  - [Using UCLI](#) - For VHDL, Verilog, and mixed designs.
- 

## Using System Tasks

VCS provides Verilog system tasks to:

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

### Enable and Disable Dumping

You can use the `$vcpluson` and `$vcplusoff` Verilog system tasks to enable and disable the dumping of the simulation history in VPD format.

Note:

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

## **\$vcdpluson**

The following is the syntax of the `$vcdpluson` system task:

```
$vcdpluson  
(level| "LVL=integer_variable", scope*, signal*);
```

Usage:

*level| "LVL=integer\_variable"*

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

You can also specify the number of hierarchy scope levels using “`LVL=integer_variable`”. Where, `integer_variable` specifies the level to descend to record signal value changes.

*scope*

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

*signal*

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

**Note:**

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

### **Example 1: Record all signal value changes**

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

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

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

```
'timescale 1ns/1ns
module test ();
...
risc1 risc(...);
initial
$vcndlplus(test.risc1.alureg);
...
endmodule
```

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

## **\$vcplusplusoff**

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

The following is the syntax of the **vcplusplusoff** system task:

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

### **Example 1: Turn recording off**

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

This example enables dumping at 0ns and disables dumping after 5ns.

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

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

This example enables dumping of the entire design. However, `$vcdplusoff` disables the dumping of the instance `alu1` and instances below `alu1`.

**Note:**

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

## Override the VPD Filename

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

```
$vcdplusfile ("filename.vpd");
$vcdpluson();
```

**Note:**

You must use `$vcdpluson` after specifying `$vcdplusfile`, as shown above, to override the default filename.

### Example:

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

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

## Dump Multi-Dimensional Arrays and Memories

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

Following are the two ways to view MDA data:

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

Note:

You should use the compilation/elaboration option `+memcbk` to use these system tasks.

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

## Syntax for Specifying MDAs

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

`system_task(Mda);`

Where,

`system_task`

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

Mda

Name of the MDA to be recorded.

### Example

This section provides example and graphical representation of MDA and memory declaration using the `$vcdplusmemon` system task.

Consider the following example code:

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

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

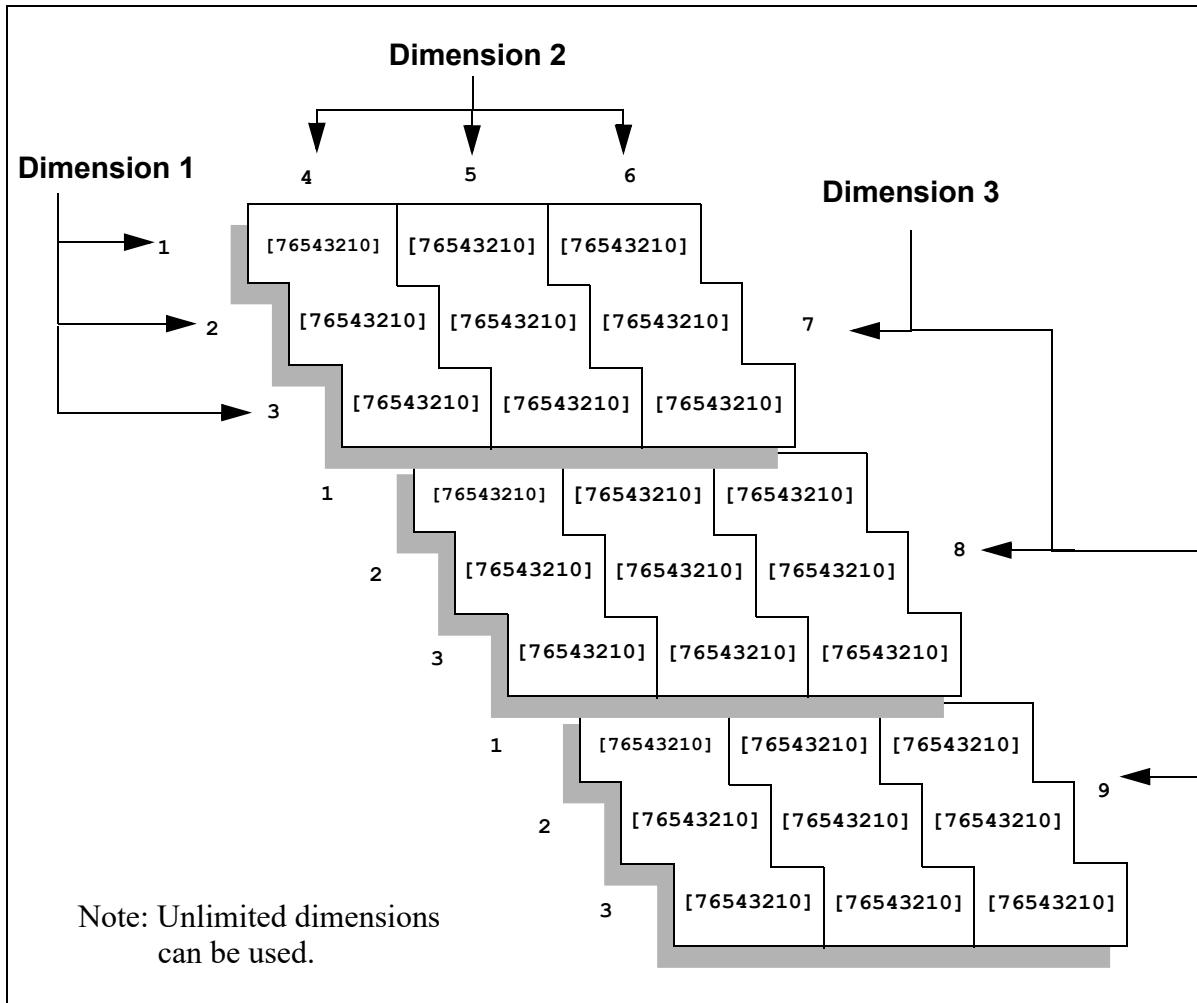
### Example: To dump all elements to the VPD File

Consider the following example code:

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

In this example, `$vcdplusmemon` dumps the entire `mem01` MDA.

**Figure 8-1** `reg [7:0] mem01 [1:3] [4:6] [7:9]`



## Using \$vcplusmemorydump System Task

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

You can specify to dump the complete set of multi-dimensional array elements only once. You can specify multiple element subsets of an array using multiple `$vcplusmemorydump` commands, but they

must occur in the same simulation time. In subsequent simulation times, `$vcdplusmemorydump` commands must use the initial set of array elements or a subset of those elements. Dumping elements outside the initial specifications result in a warning message.

## Capture Delta Cycle Information

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

### **\$vcdplusdeltacycleon**

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

Glitch detection is automatically turned on when VCS executes `$vcdplusdeltacycleon` unless you have previously used `$vcdplusglitchon/off`. Once you use `$vcdplusglitchon/off`.

Syntax:

```
$vcdplusdeltacycleon;
```

Note:

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

### **\$vcdplusdeltacycleoff**

The `$vcdplusdeltacycleoff` system 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.

### Syntax:

```
$vcdplusdeltacycleoff;
```

---

## Dumping an EVCD File

EVCD dumps the signal value changes and the direction of the ports at the specified module instance. You can dump an EVCD file using the following methods:

- [Using \\$dumpports System Task](#)
- [Dumping EVCD File for Mixed Designs Using UCLI dump Command](#)
- [Limitations](#)

---

## Using \$dumpports System Task

The \$dumpports system task creates an EVCD file as specified in IEEE Standard 1364-2001. The EVCD file records the transition times and values of the ports in a module instance. The EVCD file contains more information than the VCD file specified by the \$dumpvars system task. It includes strength levels and information on whether the test bench or the Device Under Test (DUT) is driving the signal's value.

### Syntax:

```
$dumpports(module_instance, [module_instance, ] "file  
ename");
```

Example:

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

---

## Dumping EVCD File for Mixed Designs Using UCLI dump Command

You can either use the `$dumpports` system task or the UCLI `dump` command to dump an Extended Value Change Dump (EVCD) file for mixed designs. However, due to the XMR restriction in VHDL, you may not be able to use `$dumpports` for all mixed design flows.

For pure Verilog design flow, it is recommended to use the `$dumpports` system task to dump an EVCD file, as it does not require any changes at compile time. Also, `$dumpports` allows you to dump multiple EVCD files, which is not possible with UCLI.

For mixed design flows, it is recommended to use the UCLI `dump` command along with the configuration file to dump the EVCD file, as described in the following use model.

### Use Model

To dump an EVCD file for mixed design flows, it is recommended to use the configuration file with the `+optconfigfile` compile-time option to specify all the instances for which the UCLI `dump` command may be used to dump EVCD.

```
% vcs +optconfigfile+file_name.cfg -  
debug_access+r+cbk+drivers file_name.v
```

Where, `file_name.cfg` is the configuration file which allows you to specify the instances that needs to be dumped at compile time. Following is the syntax of the configuration file:

```
instance {list_of_instance_hierarchical_names}  
{enable_evcd};
```

For example,

```
instance {top.dut} {enable_evcd};
```

Note:

- The configuration file only enables EVCD dumping, it does not dump EVCD. To dump an EVCD file, you must use the UCLI `dump` command at runtime, as follows:

```
ucli% dump -file test.evcd -type EVCD  
ucli% dump -add {top.dut}
```

- If the configuration file is not specified, then a warning message is issued for the cases where ports are connected to the bidirectional switches, and EVCD results may not be accurate.
- All forces are considered as TB regardless of where the force is applied from (TB, DUT, or UCLI).

## Use Model for Dumping CCN Driver Through INOUT

EVCD file contains the CCN driver when the CCN is connected through INPUT or OUTPUT ports in Verilog-VHDL or VHDL-Verilog mixed designs. However, if a target VHDL instance lies inside a VHDL connected through INOUT ports, you must use the `+dumpports+mxccn` option at compile time to dump CCN drivers.

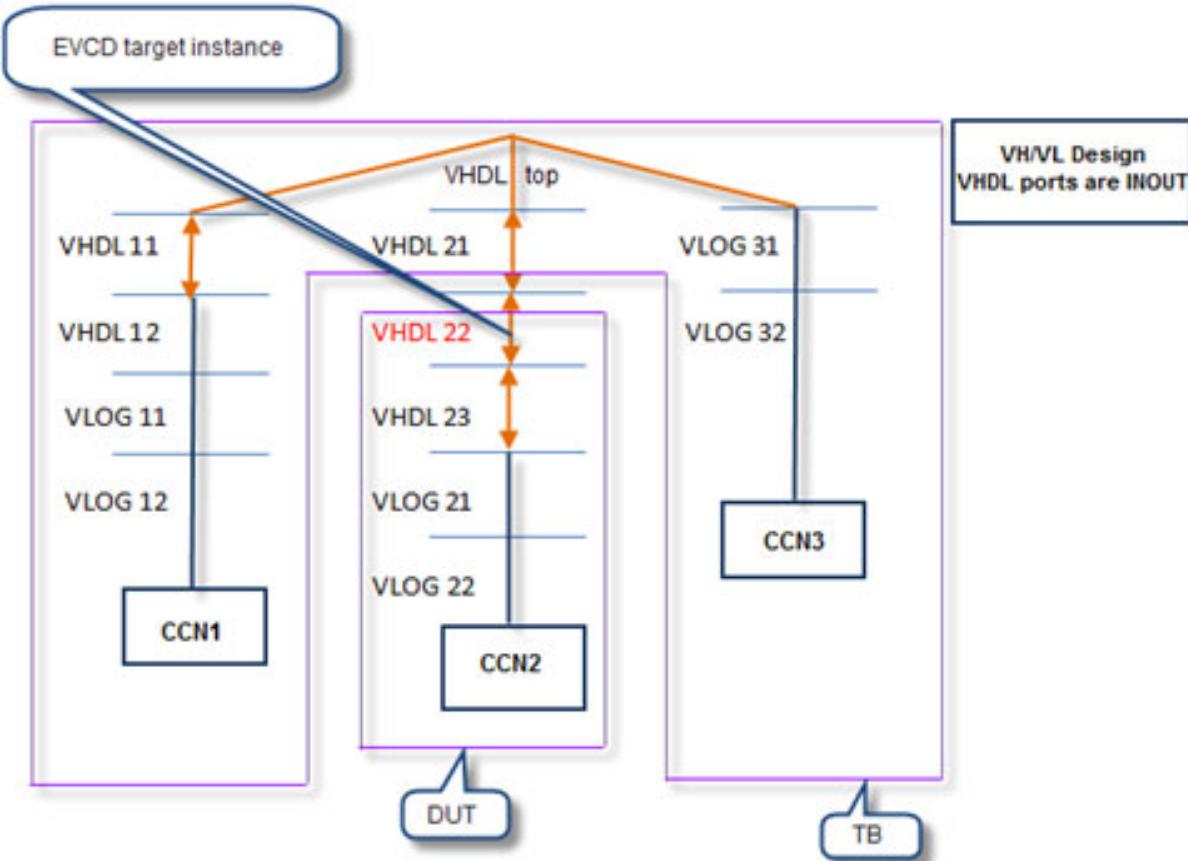
```
% vhdlan <design files>
```

```
% vlogan <design file>

% vcs +optconfigfile+file_name.cfg -
debug_access+r+cbk+drivers top_module
+dumpports+mxccn
```

Following is a sample VHDL-Verilog design which requires  
+dumpports+mxccn:

Both testbench and DUT have VHDL and the design is mixed (VHDL and Verilog). The Verilog design has CCN which is connected to VHDL through the INOUT ports.



---

## Limitations

Following are the limitations of the EVCD dumping using \$dumpports or UCLI command dump -type EVCD:

### Unsupported Port Types

- For Verilog DUT:
  - Ports can only be of type Verilog-2001. SystemVerilog type ports are not allowed. VCS generates a warning message, if it finds any unsupported port type.
  - SystemVerilog complex types (including MDAs, dynamic arrays, associative arrays, queues, and so on) are not supported, and not legal in LRM. Interface or virtual interface is not supported.
- For VHDL DUT:
  - Ports can only be of type STD\_LOGIC, STD\_ULOGIC, STD\_LOGIC\_VECTOR, STD\_ULOGIC\_VECTOR, BIT, BIT\_VECTOR, BOOLEAN. Any user-defined type or sub-type of the above types is supported.
  - Complex types like aggregates, MDA, or enums are not allowed as port or port drivers. A warning message is generated if such constructs are found.
  - Ports having type with user-defined resolution functions in VHDL are not supported.

### Unsupported DUT Types

- DUT cannot be SV program, interface, SystemC, Spice, or Verilog-AMS.

## **SystemC Support**

- Each SystemC module is treated like a Verilog shell, and multiple drivers cannot be detected inside SystemC.
  - SystemC is not supported as a DUT.
- 

## **Post-processing Utilities**

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

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

Note:

All utilities are available in \$VCS\_HOME/bin.

This section describes these utilities in the following sections:

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

- “The vpdutil Utility”
- 

## The vcdpost Utility

You can 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 specify 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 specify the following in your source code:

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

Where, `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 the signal `out1` are for the entire signal, not just for 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. Consider the following example:

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

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

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

Where,

`+scalar`

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

`+unique`

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

`input_VCD_file`

The name of the VCD file created by VCS.

`output_VCD_file`

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

---

## The `vcdiff` Utility

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

### Note:

The `vcdiff` utility cannot compare dump files of different types.

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 first compares the header sections and reports any signals/scopes that are present in one dump file, but absent in other. Then, the `vcdiff` utility compares the value change sections of the dump files for signals that appear in both dump files. This utility determines value change differences based on the final value of the signal in a time step.

## 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] [-ignoreregns]
[ingorereals]
[-ignorefunctaskvars] [-ignoretiming units] [-
ignorestrength]
[-geninclude [filename]] [-spikes]
```

## Options for Specifying Scope/Signal Hierarchy

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

### **-noabsentsig**

Does not report any signals that are present in one dump file, but are absent in 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, `vcdiff` returns an error status upon completion even if it does not 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) for Value Change Comparison

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 compared from the top of the design to the indicated scope. All child scopes/signals of the indicated scope are compared unless modified by the `-level` option (below).

**-level** *N*

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

**-include** [*file*]

Reports value change compares 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 comparison. 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 Comparison

The following options limit when `vcdiff` detects value change differences:

**-min** *time*

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

**-max** *time*

Specifies the stopping time (in simulation units) when value change comparison ends. By default, this occurs at the latest time found in either dump file.

**-strobe** *first\_time delta\_time*

Only checks for differences when the `strobe` is true. The strobe is true at `first_time` (in simulation units) and then every `delta_time` increment thereafter.

#### **-prestrobe**

Used in conjunction with `-strobe`, tells `vcdiff` to look for differences just before the strobe is true.

#### **-when expression**

Reports differences only when the given `when` expression is true. Initially this expression can consist only of scalar signals, combined with `and`, `or`, `xor`, `xnor`, `and` `not` operators, and employ parentheses to group these expressions. You must fully specify the complete path (from root) for all the signals used in expressions.

Note:

Operators may be either Verilog style (`&`, `|`, `^`, `~^`, `~`) or VHDL (`and`, `or`, `xor`, `xnor`, `not`).

#### **-synch signal**

Checks for differences only when the given signal changes value. In effect, the given signal is a “clock” for value change comparison, where the differences are checked only on the transitions (any) of this signal.

#### **-synch0 signal**

As `-synch` (above) except that it checks for differences when the given signal transitions to '0'.

#### **-synch1**

As `-synch` (above) except that it checks for differences only when the given signal transitions to '1'.

Note:

The `-max`, `-min` and `-when` options must be true in order for `vcdiff` to report 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.

**`-ignorereg`**

Does not report value change differences on signals that are of type register.

**`-ignorewire`**

Does not report value change differences on signals that are of type wire.

**`-ignorereal`**

Does not report value change differences on signals that are of type real.

**-ignorefunctaskvars**

Does not report value change differences on signals that are function or task variables.

**-ignorestrength** (EVCD only)

EVCD files contain a richer set of signal strength and directional information than VCD or even VPD files. This option ignores the strength portion of a signal value when checking for differences.

**-compare01xz** (EVCD only)

Converts all signal state information to equivalent 4-state values (0, 1, x, z) before 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 differences for a signal until the maximum number of differences are reported (see `-limitdiffs`).

### **-wrapsize *columns***

Wraps the output of vectors longer than the given size to the next line. The default 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 (first signal defined on a net) when they are different in the two dump files.

### **-limitdiffs *number\_of\_diffs***

By default, `vcdiff` stops after the first 50 differences are reported. This option overrides this default behavior. Setting this value to 0 causes `vcdiff` to report all differences.

### **-geninclude *filename***

Produces a separate file of the given name in addition to the standard `vcdiff` output. This file contains the 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 an input to the `vcat` tool using the `-include` option of `vcat`.

#### **-spikes**

A spike is defined as a signal that changes multiple times in a single time step. This option annotates (with #) the value change differences detected when the signal spikes (glitches). It keeps and reports a total count of such differences.

## **The `vcdiff` Utility Output Example**

The following is an example of the `vcdiff` output:

```
--- top.sig1 --- 200 ---
< 200 0
---
> 100 1

--- top.sig2 --- 200 ---
< 100 1
---
> 200 0
```

In this example, there are two differences between the two compared dump files. The format of a difference is as follows:

```
--- signal_hierarchical_name --- time_of_mismatch ---
< time_of_last_change change_to_this_value
---
> time_of_last_change change_to_this_value
```

Where:

< (line beginning with <)

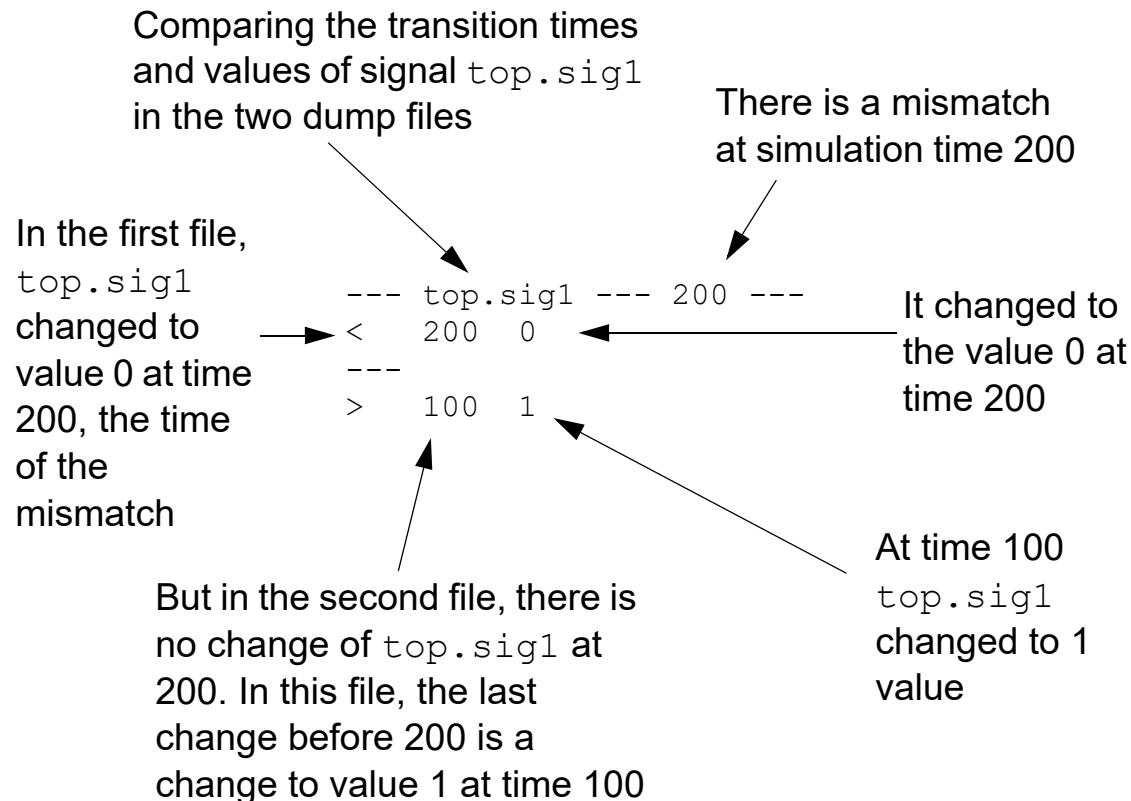
Contains the time of the last value change of the signal (at or before the time of the mismatch) in the first dump file on the `vcdiff` command line.

> (line beginning with >)

Contains the corresponding information in the second dump file on the `vcdiff` command line.

[Figure 8-2](#) shows the Callout notes on the different names and values in an example difference.

*Figure 8-2 An Annotated Difference Example*



You can infer from this example that signal `top.sig1`, in both the first and second dump file, transitioned to 1 at time 100 because there is no mismatch at time 100.

---

## 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]
```

Where,

`-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`, `vcat` output looks as follows:

```
--- 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`, `vcat` output looks as follows:

```
--- 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 the simulation time rather than signal name.

**-min** *time*

Specifies the 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 of an instance and all signals hierarchically under it.

**-level** *level\_number*

Specifies the number of hierarchical levels for which `vcat` displays data. The starting point is either the top-level module or the module instance you specify with the **-scope** option.

**-include** *filename*

Specifies a file that contains a list of module instances and signals. The `vcat` utility only displays data for these signals or the signals in these module instances.

**-ignore** *filename*

Specifies a file that contains a list of module instances and signals. However, the `vcat` utility does NOT display data for these signals or the signals in these module instances.

**-spikes**

Indicates all zero-time transitions with the `>>` symbol in the leftmost column. In addition, prints a summary of the total number of spikes seen at the end of the `vcat` output. The following is an example of the new output:

```
--- DF_test.logic.I_348.N_1 ---
 0      x
 100    0
 120    1
>>120  0
 4000   1
 12000  0
 20000  1

Spikes detected:  5
```

**-noalpha**

By default, `vcat` displays signals within a module instance in alphabetical order. This option disables this ordering.

**-wrapsize size**

Specifies value displays for wide vector signals, how many bits to display on a line before wrapping to the next line.

**-showmasters**

Specifies showing collapsed net masters.

**-showdefs**

Specifies displaying signals, but not their value changes or the simulation time of these value changes.

`-showcodes`

Specifies displaying the signal's VCD file identifier code.

`-stdin`

Enables you to use standard input, such as piping the VCD file into `vcat`, instead of specifying the filename.

`-vgen`

Generates (from a VCD file) two types of source files for a module instance: one that models how the design applies stimulus to the instance, and the other that models how the instance applies stimulus to the rest of the design. See “[Generating Source Files From VCD Files](#)” .

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

```
vcat exp1.vcd

exp1.vcd: scopes:6 signals:12 value-changes:13

--- top.mid1.in1 ---
0 1

--- top.mid1.in2 ---
0 xxxxxxxx
10000 00000000

--- top.mid1.midr1 ---
0 x
2000 1

--- top.mid1.midr2 ---
0 x
```

2000 1

In this output, for example, you see that signal `top.mid1.midr1` at time 0 had a value of `X`, and at simulation time 2000 (as specified by the `$timescale` section of the VCD file, which VCS derives from the time precision argument of the `'timescale` compiler directive) this signal transitioned to 1.

## Generating Source Files From VCD Files

The `vcat` utility can generate Verilog and VHDL source files that are one of the following:

- A module definition that succinctly models how a module instance is driven by a design, that is, a concise testbench module that instantiates the specified instance and applies stimulus to that instance the way the entire design does. This is called testbench generation.
- A module definition that mimics the behavior of the specified instance to the rest of the design, that is, it has the same output ports as the instance and in this module definition the values from the VCD file are directly assigned to these output ports. This is called module generation.

Note:

The `vcat` utility can only generate these source files for instances of module definitions that do not have inout ports.

Testbench generation enables you to focus on a module instance, applying the same stimulus as the design does, but at faster simulation because the testbench is far more concise than the entire design. You can substitute module definitions at different levels of abstraction and use `vcendiff` to compare the results.

Module generation enables you to use much faster simulating “canned” modules for a part of the design to enable the faster simulation of other parts of the design that need investigation.

The name of the generated source file from testbench generation begins with testbench followed by the module and instance names in the hierarchical name of the module instance, separated by underscores. For example `testbench_top_ad1.v`.

Similarly, the name of the generated source file from module generation begins with `moduleGeneration` followed by the module and instance names in the hierarchical name of the module instance, separated by underscores. For example  
`moduleGeneration_top_ad1.v`.

You enable `vcat` to generate these files by doing the following:

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

## Writing the Configuration File

The configuration file is named `vgen.cfg` by default, and `vcat` looks for it in the current directory. This file needs three types of information specified in the following order:

1. The hierarchical name of the module instance.
2. Specification of testbench generation with the keyword `testbench` or specification of module generation with the keyword `moduleGeneration`.
3. The module header and the port declarations from the module definition of the module instance.

You can use Verilog comments in the configuration file.

The following is an example of a configuration file:

*Example 8-1 Configuration File*

```
top.ad1
testbench
//moduleGeneration
module adder (out,in1,in2);
  input in1,in2;
  output [1:0] out;
```

You can use a different name and location for the configuration file. In order to do this, you must enter it as an argument to the `-vgen` option. For example:

```
vcat filename.vcd -vgen /u/design1/vgen2.cfg
```

*Example 8-2 Source Code*

Consider the following source code:

```
module top;
  reg r1,r2;
  wire int1,int2;
  wire [1:0] result;

  initial
  begin
    $dumpfile("exp3.vcd");
    $dumpvars(0,top.pa1,top.ad1);
    #0 r1=0;
    #10 r2=0;
    #10 r1=1;
    #10 r2=1;
    #10 r1=0;
    #10 r2=0;
    #10 r1=1;
    #10 r2=1;
```

```

#10 r1=0;
#10 r2=0;
#10 r1=1;
#10 r2=1;
#10 r1=0;
#10 r2=0;
#100 $finish;
end

passer pa1 (int1,int2,r1,r2);
adder ad1 (result,int1,int2);
endmodule

module passer (out1,out2,in1,in2);
input in1,in2;
output out1,out2;

assign out1=in1;
assign out2=in2;
endmodule

module adder (out,in1,in2);
input in1,in2;
output [1:0] out;

reg r1,r2;
reg [1:0] sum;

always @ (in1 or in2)
begin
r1=in1;
r2=in2;
sum=r1+r2;
end

assign out=sum;
endmodule

```

Notice that the stimulus from the testbench module named `test` propagates through an instance of a module named `passer` before it propagates to an instance of a module named `adder`. The `vcat`

utility can generate a testbench module to stimulate the instance of adder in the same exact way, but in a more concise and therefore faster simulating module.

If you use the sample `vgen.cfg` configuration file in [Example 8-1](#) and enter the following command line:

```
vcat filename.vcd -vgen
```

The generated source file, `testbench_top_ad1.v`, is as follows:

```
module tbench_adder ;
wire [1:0] out ;
reg in2 ;
reg in1 ;
initial #131 $finish;
initial $dumpvars;
initial begin
#0 in2 = 1'bx;
#10 in2 = 1'b0;
#20 in2 = 1'b1;
#20 in2 = 1'b0;
#20 in2 = 1'b1;
#20 in2 = 1'b0;
#20 in2 = 1'b1;
#20 in2 = 1'b0;
end
initial begin
in1 = 1'b0;
forever #20 in1 = ~in1 ;
end
adder ad1 (out,in1,in2);
endmodule
```

This source file uses significantly less code to apply the same stimulus with the instance of module `passer` omitted.

If you revise the `vgen.cfg` file to have `vcat` perform module generation, the generated source file, `moduleGeneration_top_ad1.v`, is as follows:

```
module adder (out,in1,in2) ;
  input in2 ;
  input in1 ;
  output [1:0] out ;
  reg [1:0] out ;
  initial begin
    #0 out = 2'bxx;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
  end
endmodule
```

Notice that the input ports are stubbed, and the values from the VCD files 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

Following is the syntax of the `vcsplit` utility:

```
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 is regarded as all bits are specified.

## The vcd2vpd Utility

The `vcd2vpd` utility converts a VCD file generated using `$dumpvars` or UCLI dump commands to a VPD file.

Following is the syntax of the `vcd2vpd` utility:

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

**Usage:**

`-b<min_buffer_size>`

Minimum buffer size in KB used to store Value Change Data (VCD) 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 VCD.

+bitrangenospace

Support non-standard VCD files that do not have white space between a variable identifier and its bit range.

+vpdnoreadopt

Turn off read optimization format.

## Options for Specifying EVCD Options

+dut+dut\_sufix

Modifies the string identifier for the Device Under Test (DUT) half of the split signal. The default value is \_DUT.

+tf+tf\_sufix

Modifies the string identifier for the Test-Fixture half of the split signal. The default value is \_TF.

+indexlast

Appends the bit index of a vector bit as the last element of the name.

vcd\_file

Specify the vcd filename or use "-" to indicate VCD data to be read from stdin.

vpd\_file

Specify the VPD file name. You can also specify the path and the filename of the VPD file, otherwise, the VPD file is 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 \$vcplusplus or UCLI dump commands to a VCD or EVCD file.

The syntax is as shown below:

```
vpd2vcd [-h] [-q] [-s] [-x] [-xlrn] [+zerodelayglitchfilter]
 [+morevhdl] [+start+value] [+end+value] [+splitpacked] [-f]
```

`cmd_filename] vpd_file vcd_file`

Here:

`-h`

Translate hierarchy information only.

`-q`

Suppress the copyright and other informational messages.

`-s`

Allow sign extension for vectors. Reduces the file size of the generated *vcd\_file*.

`-x`

Expand vector variables to full length when displaying `$dumpoff` value blocks.

`-x1rm`

Convert uppercase VHDL objects to lowercase.

`+zerodelayglitchfilter`

Zero delay glitch filtering for multiple value changes within the same time unit.

`+morevhdl`

Translates the VHDL types of both directly mappable and those that are not directly mappable to Verilog types.

Note:

This option may create a non-standard VCD file.

`+start+time`

Translate the value changes starting after the specified start time.

`+end+time`

Translate the value changes ending before the specified end time.

Note:

Specify both start time and end time to translate the value changes occurring between start and end time.

`-f cmd_filename`

Specify a command file containing commands to limit the design converted to VCD or EVCD. See the “[The Command File Syntax](#)” section for more information.

`+splitpacked`

Use this option to change the way packed structs and arrays are reported in the output VCD file. It does the following:

- Treats a packed structure the same as an unpacked structure and dumps the value changes of each field.

Consider the following example:

```
typedef logic [1:0] t_vec;

typedef struct packed {
    t_vec f_vec_b;
} t_ps_b;

module test();
    t_ps_b var_ps_b;
endmodule
```

The VCD file created in the previous example is as follows:

```
$scope module test $end
$scope fork var_ps_b $end
$var reg 2 ! f_vec_b [1:0] $end
$upscope $end
$upscope $end
```

- Treats a packed MDA as an unpacked MDA except for the inner most dimensions.

Consider the following example:

```
typedef logic [1:0] t_vec;

module test();
    t_vec [3:2] var_vec;
endmodule
```

The VCD file created in the previous example is as follows:

```
$scope module test $end
$var reg      2 %     var_vec[3] [1:0] $end
$var reg      2 &     var_vec[2] [1:0] $end
$upscope $end
```

- Expands all packed arrays defined in a packed struct.

Consider the following example:

```
typedef logic [1:0] t_vec;

typedef struct packed {
    t_vec f_vec;
    t_vec [3:2][1:0] f_vec_array;
} t_ps;

module test();
    t_ps var_ps;
endmodule
```

The VCD file created in the previous example is as follows:

```
$scope module test $end
$scope fork var_ps $end
$var reg      2 '    f_vec [1:0] $end
$var reg      2 (    f_vec_array[3][1] [1:0] $end
$var reg      2 )    f_vec_array[3][0] [1:0] $end
$var reg      2 *    f_vec_array[2][1] [1:0] $end
$var reg      2 +    f_vec_array[2][0] [1:0] $end
$upscope $end
$upscope $end
```

- Expands all dimensions of a packed array defined in a packed struct.

Consider the following example:

```
typedef logic [1:0] t_vec;

typedef struct packed {
    t_vec f_vec;
    t_vec [3:2][1:0] f_vec_array;
} t_ps;

module test();
    t_ps [1:0] var_paps;
endmodule
```

The VCD file created in the previous example is as follows:

```
$scope module test $end
$scope fork var_paps[1] $end
$var reg      2 '    f_vec [1:0] $end
$var reg      2 (    f_vec_array[3][1] [1:0] $end
$var reg      2 )    f_vec_array[3][0] [1:0] $end
$var reg      2 *    f_vec_array[2][1] [1:0] $end
$var reg      2 +    f_vec_array[2][0] [1:0] $end
$upscope $end
$scope fork var_paps[0] $end
```

```

$var reg      2 ,      f_vec [1:0] $end
$var reg      2 -      f_vec_array[3][1] [1:0] $end
$var reg      2 .      f_vec_array[3][0] [1:0] $end
$var reg      2 /      f_vec_array[2][1] [1:0] $end
$var reg      2 0      f_vec_array[2][0] [1:0] $end
$upscope $end
$upscope $end

```

- Expands and prints the value of each member of a packed union.

Consider the following example:

```

module testit;

    typedef logic [1:0] t_vec;

    typedef union packed {
        t_vec f_vec;
        struct packed {
            logic f_a;
            logic f_b;
        } f_ps;
    } t_pu_v;
    typedef union packed {
        struct packed {
            logic f_a;
            logic f_b;
        } f_ps;
        t_vec f_vec;
    } t_pu_s;
    t_pu_v var_pu_v;
    t_pu_s var_pu_s;
endmodule

```

The VCD file created in the previous example is as follows:

```

$scope module testit $end
$scope fork var_pu_v $end
$var reg      2 -      f_vec [1:0] $end

```

```

$scope fork f_ps $end
$var reg      1 .    f_a $end
$var reg      1 /    f_b $end
$upscope $end
$upscope $end
$scope fork var_pu_s $end
$scope fork f_ps $end
$var reg      1 0    f_a $end
$var reg      1 1    f_b $end
$upscope $end
$var reg      2 2    f_vec [1:0] $end
$upscope $end
$upscope $end

```

## The Command File Syntax

Using a command file, you can generate:

- A VCD file for the whole design or for the specified instances.
- Only the port information for the specified instances.
- An EVCD file for the specified instances.

Note the following before writing a command file:

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

```
dumpvars 1 adder4
```

- All comments must start with “//”. For example:
- ```
//Add your comment here
dumpvars 1 adder4
```
- All comments written after a command must be preceded by a space. For example:

```
dumpvars 1 adder4 //can write your comment here
```

A command file can contain the following commands:

```
dumpports instance [instance1 instance2 ...]
```

Specify an instance for which an EVCD file has to be generated. You can generate an EVCD file for more than one instance by specifying the instance names separated by a space. You can also specify multiple `dumpports` commands in the same command file.

```
dumpvars [level] [instance instance1 instance2 ...]
```

Specify an instance for which a VCD file has to be generated. [*level*] is a numeric value indicating the number of levels to traverse down the specified instance. If not specified, or if the value specified is "0", then all the instances under the specified instance are dumped.

You can generate a VCD file for more than one instance by specifying the instance names separated by a space. You can also specify multiple `dumpvars` commands in the same command file.

If this command is not specified or the command has no arguments, then a VCD file is 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 are dumped.

You can generate a dump file for more than one instance by specifying the instance names separated by a space. You can also specify multiple `dumpvcdports` commands in the same command file.

Note:

`dumpvcdports` splits the inout ports of type wire into two separate variables:

- one shows the value change information driven into the port. VCS adds a suffix `_DUT` to the basename of this variable.
- the other variable shows the value change information driven out of the port. VCS adds a suffix `_TB` to the basename of this variable.

`dutsuffix DUT_suffix`

Specify a string to change the suffix added to the variable name that shows the value change data 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 data 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 is the start time of the VCD 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 VCD file.

Note:

Only one `+end` command is allowed in a command file, and must be equal to or greater than the start time.

## Limitations

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

---

## The vpdmerge Utility

Using the `vpdmerge` utility, you can merge different VPD files storing simulation history data for different simulation times, or the parts of the design hierarchy into one large VPD file.

The syntax is as shown below:

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

Usage:

-h

Displays a list of the valid options and their purpose.

-o *merged\_VPD\_filenames*

Specifies the name of the output merged VPD file. This option is required.

-q

Specifies quiet mode. Disables the display of most output to the terminal.

-hier

Specifies merge input based on unique hierarchy. (Default is merge input based on time.)

If -hier option is specified:

- The VPD files being merged must be split on scope boundaries.
- The first VPD file containing signals for a scope is used to create the values for those signals. Values for those signals in all other VPD files are ignored.

-v

Specifies verbose mode. Enables the display of warning and error messages.

## Restrictions

The `vpdmerge` utility includes the following restrictions:

- VCS must have written the input VPD files on the same platform as the `vpdmerge` utility.
- The input VPD files cannot contain delta cycle data (different values for a signal during the same time step).
- The input VPD files cannot contain named events.
- The merged line stepping data does not always accurately replay scope changes within a time step.
- If you are merging VPD files from different parts of the design using the `-hier` option, the VPD files must be used for distinctly different parts of the design, they cannot contain information for the same scope.
- You cannot use the `vpdmerge` option on two VPD files created based on timing, for both timing and hierarchy (using the `-hier` option) based merging.

## 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 `-hier` 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, but 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.

## **The `vpdutil` Utility**

The `vpdutil` utility generates statistics about the data in the `vpd` file. This utility takes a single VPD file as input. You can specify options to this utility to query at design, module, instance, and node levels.

This utility supports time ranges and input lists for query on more than one object. Output is in ASCII to `stdout` with option to redirect to an output file.

For more information, see “[Using the `vpdutil` Utility to Generate Statistics](#)” .



# 9

## Performance Tuning

---

VCS delivers the best performance during both compile-time and runtime by reducing the size of the simulation executable, and the amount of memory consumed for compilation/elaboration and simulation. By default, it is optimized for the following types of designs:

- Designs with many layers of hierarchy
- Gate-level designs
- Structural RTL-level designs - Using libraries where the cells are RTL-level code
- Designs with extensive use of timing such as delays, timing checks, and SDF back annotation, particularly to INTERCONNECT delays

However, depending on the phase of your design cycle, you can fine-tune VCS for a better compile-time and runtime performance.

This chapter describes the following sections:

- Analysis-time Performance

During analysis, you can analyze all of both Verilog and VHDL files in a single command line. For example, perform the following to analyze Verilog files:

```
% vlogan file1.v file2.v file3.v
```

For additional information, see the section entitled, “[Analysis](#)” .

- Compile-time Performance

Compile-time performance plays a very important role when you are in the initial phase of your design development cycle. In this phase, you may want to modify and recompile the design to observe the behavior. Since, this phase involves lot many recompiling cycles, achieving a faster compilation is important. For additional information, see the section entitled, “[Compile-time Performance](#)” .

- Runtime Performance

Runtime performance is important in regression phase or in the final phase of the design development cycle. For additional information, see the section entitled, “[Runtime Performance](#)” .

- Obtaining VCS Consumption of CPU Resources

You can now capture the CPU resource statistics for compilation and simulation using the `-reportstats` option. For more information, see “[Obtaining VCS Consumption of CPU Resources](#)” .

- Dumping Design Statistics

VCS generates design statistics covering various construct usages (such as number of modules, number of instances, number of signals and so on) and the design size. For more information, see “[Dumping Design Statistics](#)” .

---

## Compile-time Performance

You can improve compile-time performance in the following ways:

- “[Incremental Compilation](#)”
  - “[Compile Once and Run Many Times](#)”
  - “[Parallel Compilation](#)”
  - “[Improving VCS Compile Performance and Capacity](#)”
- 

### Incremental Compilation

During compilation/elaboration, VCS builds the design hierarchy. By default, when you recompile the design, VCS compiles only those design units that have changed since the last compilation/elaboration. This is called incremental compilation.

The incremental compilation feature is the default in VCS. It triggers recompilation of design units under the following conditions:

- Changes in the command-line options
- Change in the target of a hierarchical reference
- Change in the ports of a design unit

- Change in the functional behavior of the design
- Change in a compile-time constant such as a parameter/generic

The following conditions do not cause VCS to recompile a module:

- Change of time stamp of any source file
- Change in file name or grouping of modules in any source file
- Unrelated change in the same source file
- Non-functional changes such as comments or white space

## **Compile Once and Run Many Times**

The VCS use model is devised in such a way that you can create a single binary executable and execute it many times avoiding the elaboration step for all but the first run. For information on the VCS use model, see “[Using the Simulator](#)” .

For example, you can use this feature in the following scenarios:

- Use VCS runtime features, like passing values at runtime, to modify the design, and simulate it without re-compiling or re-elaborating. For information on runtime features, see [Chapter - "Simulating the Design"](#).
- Run the same test with different seeds.
- Create a softlink of the executable and the .daidir or .db.dir directory in a different directory, to run multiple simulations in parallel.

---

## Parallel Compilation

You can improve the compile-time performance by specifying the number of parallel processes VCS can launch for the native code generation phase of the compilation/elaboration. You should specify this using the compile-time option `-j<num_of_processes>`, as shown below:

```
% vcs -j<num_of_processes> [options] top_entity/module/config
```

**Note:**

Parallel compilation applies only for the Verilog portion of the design.

For example, the following command line forks off two parallel processes to generate a binary executable:

```
% vcs -j2 top
```

---

## Improving VCS Compile Performance and Capacity

For gate-level simulation (GLS) designs, all connected networks are placed in one elaboration module during VCS elaboration. Because of the strong connectivity in GLS design, it always generates a very large elaboration module. If the size (nodes or gates) of the elaboration module exceeds 32-bit, it results in capacity issues with GLS design that uses the `-hspt=gates` option.

VCS addresses the capacity issues of VCS elaboration by stopping global optimizations and by using partitioning in `vcselab` stage. Partitioning reduces the peak size of the elaboration module and balances the size of the elaboration modules. Thus, it helps in reducing the compile performance overhead.

## Use Model

In GLS design, partitioning is inserted using following options.

- `-hsopt=elabpart` - When you use this option, VCS uses partitioning of modules and connected networks.
- `-hsopt=j` - When you use this option, VCS speeds up the `vcselab` phase by compiling the partitions in parallel using multiple cores on a machine. By default, VCS automatically decides on the number of cores to be used for parallel compilation.

You can also specify the number of cores that VCS uses for parallel compilation using the `-hsopt=j<N>` option, where `N` is the number of cores.

- You can also insert partitioning manually using the `+optconfigfile` configuration file by specifying the module name for which partitioning is required to be inserted. The configuration file contains the following:

```
module {<module name>} {partition} ;
```

---

## Runtime Performance

VCS runtime performance is based on the following:

- Coding Style (see Modeling and Coding Style Guide)

- Access to the internals of your design at runtime, using PLIs, UCLI, debugging using GUI, dumping waveforms, and so on

Note:

Many runtime performance improvements or optimizations in this release are enabled using the `-Xkeyopt=rtopt` option.

Also, many runtime performance improvements for UVM testbenches are enabled using the `-Xkeyopt=tbopt` option. The optimization applies to UVM factory, resource pool and regular expression operations.

This section describes the following to improve the runtime performance:

- “[Using Radiant Technology](#)”
- “[Improving Performance When Using PLIs](#)”
- “[Enabling TAB File Capabilities in UCLI Using -debug\\_access](#)”

---

## Using Radiant Technology

VCS Radiant Technology applies performance optimizations to the Verilog portion of your design while VCS compiles your Verilog source code. These Radiant optimizations improve the simulation performance of all types of designs from behavioral, 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

Radiant Technology optimizations are not enabled by default. You enable them using the compile-time options:

`+rad`

Specifies using Radiant Technology

Note:

These optimizations are also enabled for SystemVerilog part of the design.

`+optconfigfile`

Optional. Specifies applying Radiant Technology optimizations to part of the design using a configuration file as described in the following section.

## Applying Radiant Technology to Parts of the Design

The configuration file enables you to apply Radiant optimizations selectively to different parts of your design. You can enable or disable Radiant optimizations for all instances of a module, specific instances of a module, or specific signals.

You specify the configuration file with the `+optconfigfile` compile-time option. For example:

`+optconfigfile+file_name`

Note:

The configuration file is a general purpose file that has other purposes, such as specifying ACC write capabilities. Therefore, to enable Radiant Technology optimizations with a configuration file, you must also include the `+rad` compile-time option.

## The Configuration File Syntax

The configuration file contains one or more statements that set Radiant optimization attributes, such as enabling or disabling optimization on a type of design object, such as a module definition, a module instance, or a signal.

The syntax of each type of statement is as follows:

```
module {list_of_module_identifiers} {list_of_attributes};  
or
```

```
instance  
{list_of_module_identifiers_and_hierarchical_names}  
{list_of_attributes};  
or
```

```
tree [(depth)] {list_of_module_identifiers}  
{list_of_attributes};
```

Usage:

```
module
```

Keyword that specifies that the attributes in this statement apply to all instances of each module 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 each module in the list specified by module identifier.
- All module instances in the list specified by their hierarchical names.
- 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, hierarchical names of module instances, or signals enclosed in curly braces:

{      }

**Note:**

Follow the Verilog syntax for signal names and hierarchical names of module instances.

tree

Keyword that specifies that the attributes in this statement apply to all instances of the modules in the list, specified by module identifier, and also apply to all module instances hierarchically under these module instances.

depth

An integer that specifies how far down the module hierarchy, from the specified modules, you want to apply Radiant optimization attributes. You can specify a negative value. A negative value specifies descending to the leaf level and counting up levels of the hierarchy to apply these attributes. This specification is optional. Enclose this specification in parentheses: ( )

The valid Radiant optimization attributes are as follows:

noOpt

Disables Radiant optimizations on the module instance or signal.

noPortOpt

Prevents port optimizations such as optimizing away unused ports on a module instance.

Opt

Enables all possible Radiant optimizations on the module instance or signal.

PortOpt

Enables port optimizations such as optimizing away unused ports on a module instance.

Statements can use more than one line and must end with a semicolon.

Verilog style comments characters `/* comment */` and `// comment` can be used in the configuration file.

## Configuration File Statement Examples

The following are examples of statements in a configuration file.

### Module Statement Example

```
module {mod1, mod2, mod3} {noOpt, PortOpt};
```

This module statement example disables Radiant optimizations for all instances of modules `mod1`, `mod2`, and `mod3`, with the exception of port optimizations.

### Multiple Module Statement Example

```
module {mod1, mod2} {noOpt};  
module {mod1} {Opt};
```

In this example, the first module statement disables radiant optimizations for all instances of modules `mod1` and `mod2` and then the second module statement enables Radiant optimizations for all instances of module `mod1`. VCS processes statements in the order in which they appear in the configuration file so the enabling of optimizations for instances of module `mod1` in the second statement overrides the first statement.

### Instance Statement Example

```
instance {mod1} {noOpt};
```

In this example, `mod1` is a module identifier, so the statement disables Radiant optimizations for all instances of `mod1`. This statement is the equivalent of:

```
module {mod1} {noOpt};
```

## Module and Instance Statement Example

```
module {mod1} {noOpt};  
instance {mod1.mod2_inst1.mod3_inst1,  
mod1.mod2_inst1.reg_a} {noOpt};
```

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

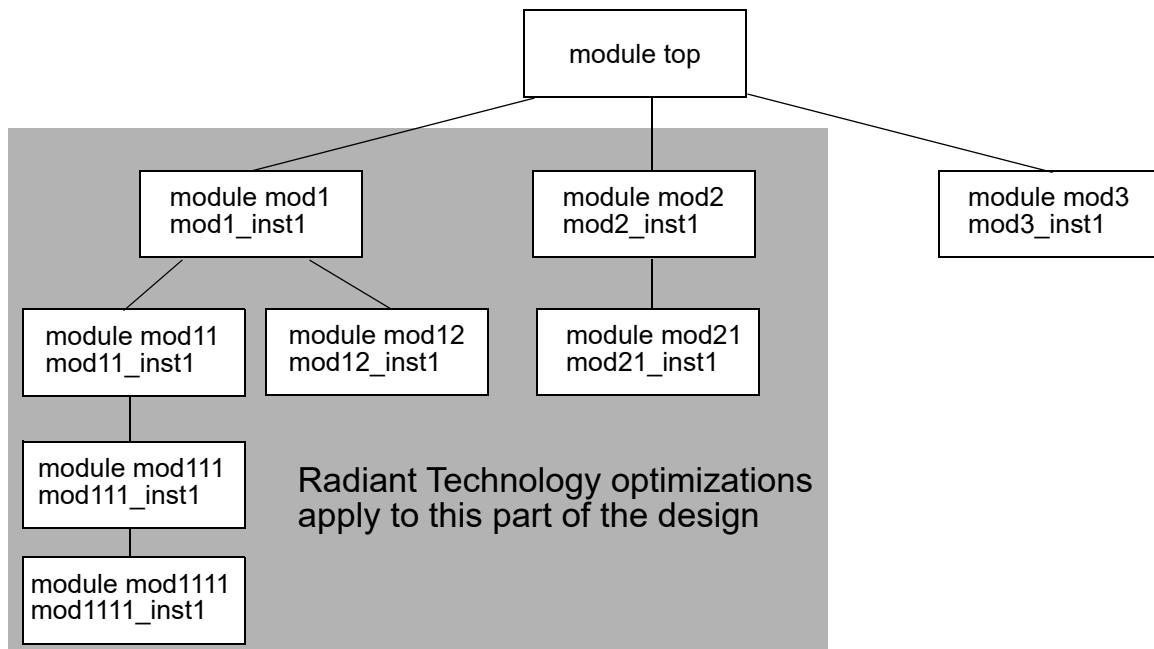
The instance statement disables Radiant optimizations for the following:

- Hierarchical instance mod1.mod2\_inst1.mod3\_inst1
- Hierarchical signal mod1.mod2\_inst1.reg\_a

## First Tree Statement Example

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

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

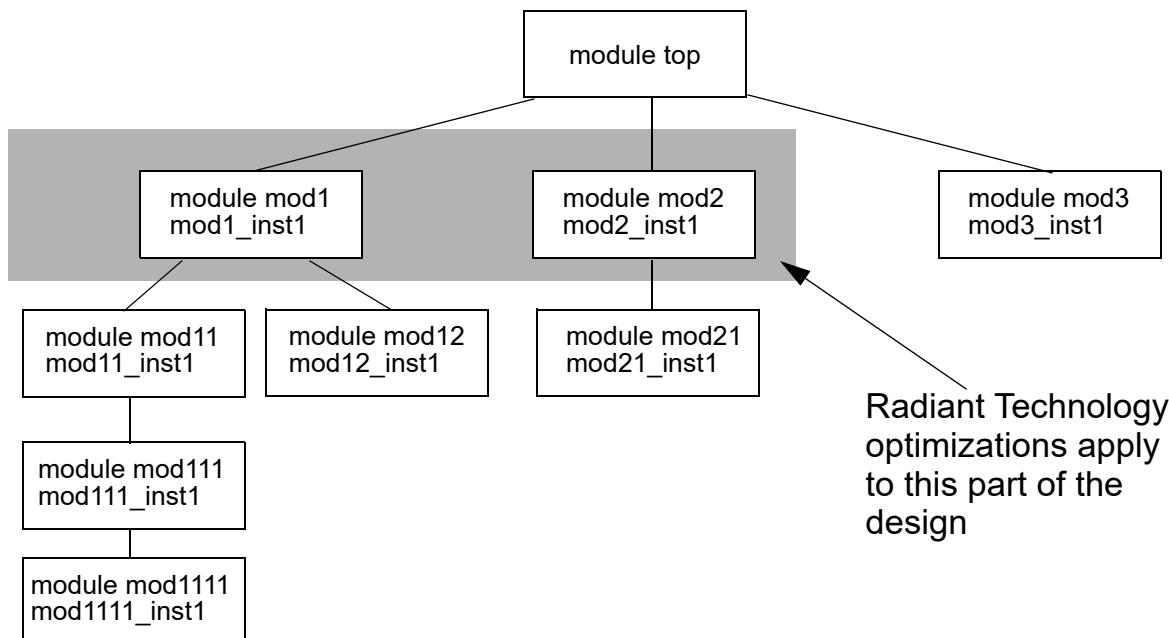


The statement enables Radiant Technology optimizations for the instances of modules `mod1` and `mod2` and for all the module instances hierarchically under these instances.

## Second Tree Statement Example

```
tree (0) {mod1,mod2} {Opt};
```

This modification of the previous tree statement includes a depth specification. A depth of 0 means that the attributes apply no further down the hierarchy than the instances of the specified modules, `mod1` and `mod2`.



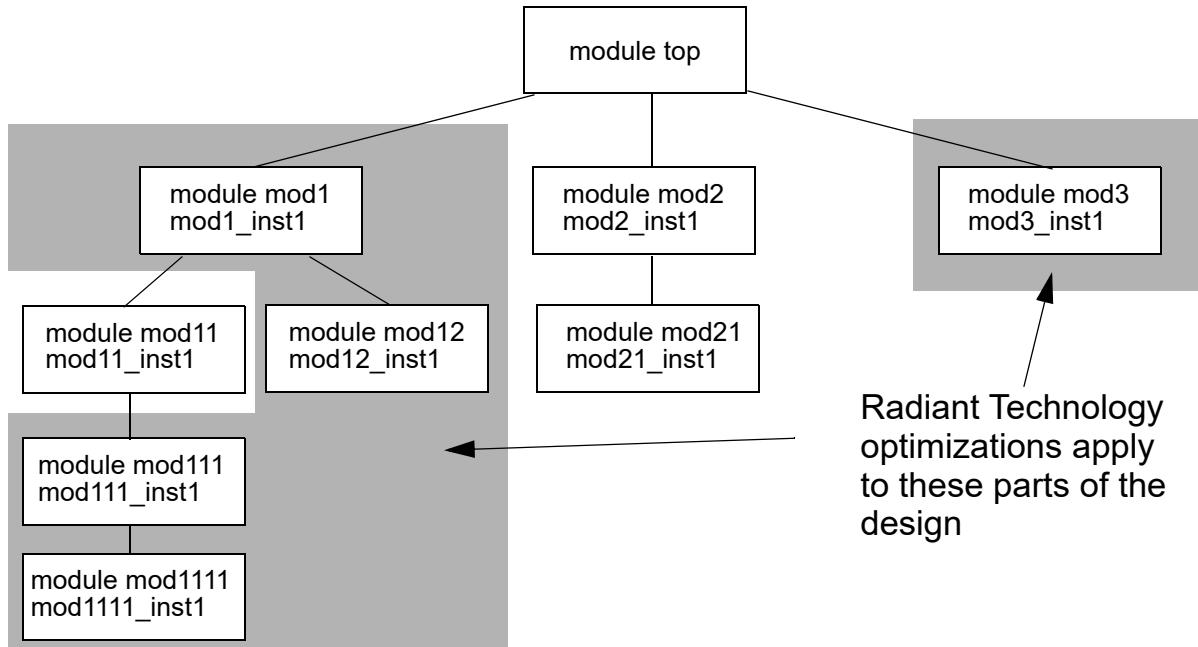
A tree statement with a depth of 0 is the equivalent of a module statement.

## Third Tree Statement Example

You can specify a negative value for the depth value. If you do this, specify ascending the hierarchy from the leaf level. For example:

```
tree (-2) {mod1, mod3} {Opt};
```

This statement specifies looking down the module hierarchy under the instances of modules `mod1` and `mod3` to the leaf level and counting up from there. (Leaf level module instances contain no module instantiation statements.)



In this example, the instances of `mod1111`, `mod12`, and `mod3` are at a depth of -1 and the instances of `mod111` and `mod1` are at a depth of -2. The attributes do not apply to the instance of `mod11` because it is at a depth of -3.

## Fourth Tree Statement Example

You can disable Radiant optimizations at the leaf level under specified modules. For example:

```
tree(-1) {mod1, mod2} {noOpt};
```

This example disables optimizations at the leaf level, the instances of modules `mod1111`, `mod12`, and `mod21`, under the instances of modules `mod1` and `mod2`.

## **Known Limitations**

Radiant Technology is not applicable to all simulation situations. Some features of VCS are not available when you use Radiant Technology.

These limitations are:

- Back-annotating SDF Files

You cannot use Radiant Technology if your design back-annotates delay values from either a compiled or an ASCII SDF file at runtime.

- SystemVerilog

Radiant Technology does not work with SystemVerilog design construct code. For example, structures and unions, new types of always blocks, interfaces, or things defined in \$root.

The only SystemVerilog constructs that work with Radiant Technology are SystemVerilog assertions that refer to signals with Verilog-2001 data types, not the new data types in SystemVerilog.

## **Potential Differences in Coverage Metrics**

VCS supports coverage metrics with Radiant Technology and you can enter both the `+rad` and `-cm` compile-time options. However, Synopsys does not recommend comparing coverage between two simulation runs when only one simulation was compiled for Radiant Technology.

The Radiant Technology optimizations, though not changing the simulation results, can change the coverage results.

## **Compilation Performance With Radiant Technology**

Using Radiant Technology incurs longer incremental compile times because the analysis performed by Radiant Technology occurs every time you recompile the design even when only a few modules have changed. However, VCS only performs the code generation phase on the parts of the design that have actually changed. Therefore, the incremental compile times are longer when you use Radiant Technology, but shorter than a full recompilation of the design.

---

## **Improving Performance When Using PLIs**

As mentioned earlier, the runtime performance is reduced when you have PLIs accessing the design. In some cases, you may have ACC capabilities enabled on all the modules in the design, including those which actually do not require them. These scenarios unnecessarily reduce the runtime performance. Ideally the performance can be improved if you are able to control the access rights of the PLIs. However, this may not be possible in many situations. In this situation, you can use the `+vcs+learn+pli` runtime option.

`+vcs+learn+pli` tells VCS to write a new tab file with the ACC capabilities enabled on the modules/scopes which actually need them during runtime. Now, during recompile, along with your original tab file, you can pass the new tab file using the compile-time option, `+applylearn+[tabfile]`, so that the next simulation will have a better runtime. Therefore, this is a two-step process:

- Using the runtime option `+vcs+learn+pli`
- Using the compilation/elaboration option `+applylearn+[tabfile]` during recompile. You do not have to reanalyze the files in this step.

The use model and an example is shown below:

## Use Model

Step1: Using the runtime option +vcs+learn+pli.

## VCS Two-Step Flow

### Compilation

```
% vcs [vcs_options] Verilog_files
```

### Simulation

```
% simv [sim_options] +vcs+learn+pli
```

## VCS Three-Step Flow

### Analysis

```
% vlogan [vlogan_options] file1.v file2.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

### Note:

Specify the VHDL bottommost entity first, then move up in order.

### Elaboration

```
% vcs [vcs_options] top_cfg/entity/module
```

### Simulation

```
% simv [sim_options] +vcs+learn+pli
```

Step2: Using the compilation/elaboration option  
+applylearn+[tabfile].

## VCS Two-Step Flow

### Compilation

```
% vcs [vcs_options] +applylearn+[tabfile] Verilog_files
```

### Simulation

```
% simv [sim_options]
```

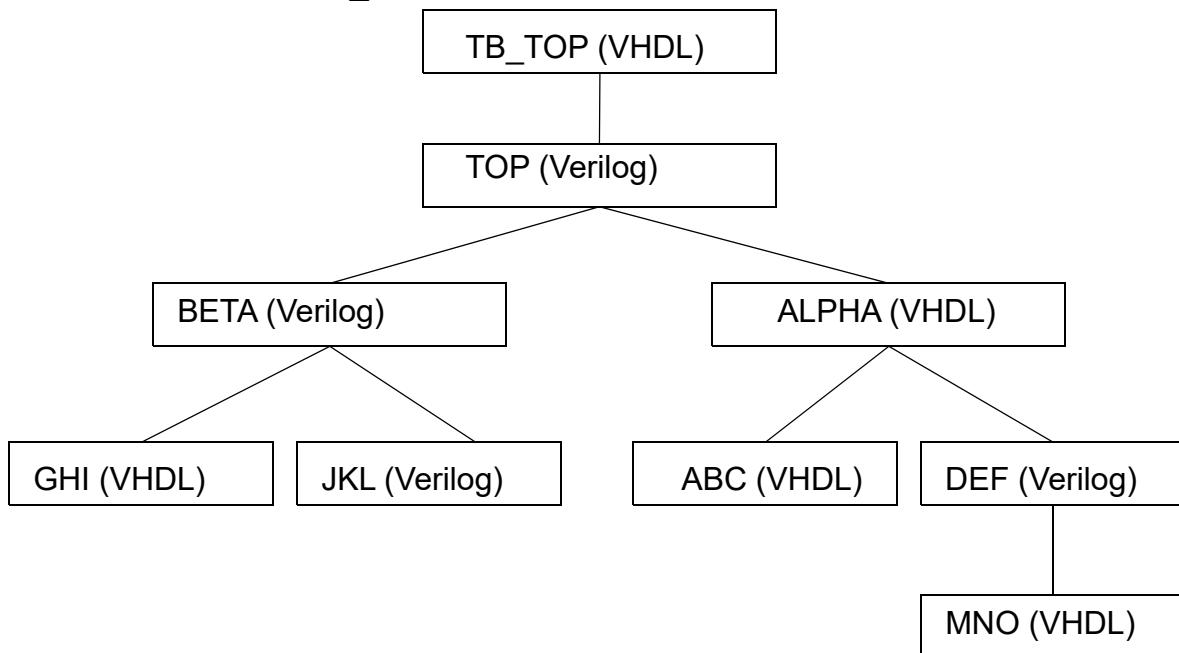
## VCS Three-Step Flow

### Elaboration

```
% vcs [vcs_options] +applylearn+[tabfile] top_cfg/entity/module
```

### Simulation

```
% simv [sim_options]
```



Consider the above example, and your `pli.tab` file is as follows:

```
% cat pli.tab
```

```
////// MY TAB FILE/////
acc=rw:*
```

The above tab file enables ACC read/write capabilities on all the modules in the design. However, in this example, you are only interested in having ACC read/write capabilities on the `jkl` module only.

The use model to invoke `+vcs+learn+pli` is as follows:

Step 1: Using the `+vcs+learn+pli` runtime option in three-step flow.

## Analysis

```
% vlogan def.v jkl.v beta.v top.v
% vhdlan mno.vhd abc.vhd alpha.vhd ghi.vhd tb_top.vhd
```

Note:

Specify the VHDL bottommost entity first, then move up in order.

## Elaboration

```
% vcs TB_TOP -P pli.tab pli.c
```

## Simulation

```
% simv +vcs+learn+pli
```

By default, the use of the `+vcs+learn+pli` option creates a `pli_learn.tab` file in the current working directory. You can see that the `pli_learn.tab` file has ACC capabilities enabled on only the `jkl` module.

```
% cat pli_learn.tab
////////////////// SYNOPSYS INC /////////////////////
// PLI LEARN FILE
```

```
// AUTOMATICALLY GENERATED BY VCS (TM) LEARN MODE
///////////////////////////////
acc=rw:jk1
//SIGNAL string:rw
```

Now, you can use the new tab file during elaboration to achieve a better runtime performance. The use model is as shown below:

Step 2: Using the elaboration `+applylearn+[tabfile]` option in three-step flow.

### **Elaboration**

```
% vcs TB_TOP -P pli.tab +applylearn+pli_learn.tab pli.c
```

### **Simulation**

```
% simv
```

---

## **Enabling TAB File Capabilities in UCLI Using -debug\_access**

UCLI checks for the debug capability of a signal applied through a PLI table file (`pli.tab`), instance/signal based PLI(SIGPLI), PLI learn file, or config file. UCLI enables this capability with the `-debug_access` option, which is the minimum debug option required to enable UCLI.

This feature improves the runtime performance by allowing you to run your design with minimum debug capability.

### **Use Model**

Following is the use model to check for the debug capability of a signal applied through a tab file:

```
% vcs -debug_access -sverilog -P file.tab file.v  
% ./simv -ucli
```

Where, `file.tab` is the tab file that specifies the debug capability for a signal.

## Example

Consider the following test case (`test.v`) and tab file (`test.tab`):

### *Example 9-1 test.v*

```
module top;  
    reg clk, a,b,c,d;  
    dut d1(clk,a,b);  
    dut1 d2(clk,c,d);  
    initial begin  
        clk=0;  
        forever #1 clk =~clk;  
    end  
    initial begin  
        #15 $finish;  
    end  
endmodule;  
  
module dut(input clk,a,output b);  
    initial begin  
        $display("DUT B=%b\n",b);  
    end  
endmodule  
  
module dut1(input clk,a,output b);  
initial begin  
    $display("DUT1 B=%b\n",b);  
end  
endmodule
```

### *Example 9-2 test.tab*

```
acc+=frc:dut.a
```

Compile and run `test.v` as follows:

```
% vcs -nc -sverilog -debug_access -P test.tab test.v
% ./simv -ucli
```

Following is the output:

```
ucli% force top.d1.a 0
ucli% get top.d1.a
'b0
```

Although the above test case is compiled with minimum debug option `-debug_access`, the force capability enabled through the tab file is available in UCLI.

---

## Impact on Performance

Compiling with the `-debug_access` option disables VCS optimizations and this impacts the performance. The `-debug_access` option disables fewer optimizations, whereas the `-`

`debug_access+all` option disables all the optimizations. The following table describes these options and their performance impact:

*Table 9-1 Performance Impact of -debug\_access*

| Options                        | Description                                                                                                                                              |
|--------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-debug_access</code>     | Use this option to generate a dump file. You can also use this option to invoke UCLI and Verdi with some limitations. This has least performance impact. |
| <code>-debug_access+f</code>   | Use this option if you want to use the <code>force</code> command at the UCLI prompt, and for more debug capabilities.                                   |
| <code>-debug_access+all</code> | This option enables all debug capabilities, and therefore will have a huge performance impact.                                                           |

See the section “[Compiling/Elaborating the Design in the Debug Mode](#)” for more information.

Note that using extensive user interface commands, like `force` or `release` at runtime, will have a huge impact on the performance.

To improve the performance, Synopsys recommends you to convert these user interface commands to HDL files and to compile/elaborate and simulate them along with the design.

Contact Synopsys Support Center ([vcs\\_support@synopsys.com](mailto:vcs_support@synopsys.com)) or your Synopsys Application Consultant for further assistance.

---

## Obtaining VCS Consumption of CPU Resources

You can capture the CPU resource statistics for compilation and simulation using the `-reportstats` option.

---

## Use Model

You can specify this option at compile time as well as runtime or both depending on your requirement.

For example:

```
%vcs -reportstats  
or  
%simv -reportstats
```

Note:

This option is supported only on RHEL32, RHEL64, SUSE32, and SUSE64 platforms. If you attempt to use this option on other platforms, VCS issues a warning and then continues.

When you specify this option at compile time, VCS prints out the following information.

## Compile Time

```
Compilation Performance Summary  
=====  
vcs started at      : Sat Nov 12 11:02:38 2011  
Elapsed time        : 4 sec  
CPU Time            : 3.0 sec  
Virtual memory size : 361.7 MB  
Resident set size   : 141.7 MB  
Shared memory size  : 79.7 MB  
Private memory size : 62.1 MB  
Major page faults   : 0  
=====
```

The details of the above report are as follows:

- VCS start time

- Elapsed real time: wall clock time from VCS start to VCS end
- CPU time: Accumulated user time + system time from all processes spawned from VCS
- Peak virtual memory size summarized from all the contributing processes at specific time points
- Sum of resident set size from all the contributing processes at specific time points
- Sum of shared memory from all the contributing processes at specific time points
- Sum of private memory from all the contributing processes at specific time points
- Major fault accumulated from all processes spawned from VCS

## Simulation Time

Specifying this option at compile time and runtime, VCS prints out both the compile time and simulation time data:

Following is the simulation time sample report data:

```
Simulation Performance Summary
=====
Simulation started at : Sat Nov 12 11:02:43 2011
Elapsed Time           : 1 sec
CPU Time               : 0.1 sec
Virtual memory size   : 152.2 MB
Resident set size     : 106.5 MB
Shared memory size    : 21.2 MB
Private memory size   : 85.3 MB
Major page faults     : 0
=====
```

If you specify the option only at runtime and not at compile time, VCS prints only runtime data at runtime.

---

## Dumping Design Statistics

VCS generates design statistics covering various construct usages (such as number of modules, number of instances, number of signals and so on) and the design size. The information helps you to verify or view the design changes done over a period of time and even across releases.

---

### Use Model

To dump user design statistics, use the following compile-time option:

```
-design_stats=<design|module|hier|filename>
[:(top_module_name)|(log_filename)]
```

Where,

<design>

Specify this option to dump statistics for the whole design.

<module>

Specify this option to dump statistics for the specified module in the design.

<hier:(top\_module\_name)>

Specify this option to dump statistics for the hierarchy top specified with <top\_module\_name>.

<filename: (log\_filename)>

Specify this option to dump statistics into a file whose name is specified by log\_filename. If the filename is not specified, VCS dumps the design statistics in the vcs\_design\_stats.log file by default.

---

## Usage Example

Consider the following test case (test.v):

### *Example 9-3 Dumping design statistics*

```
module top;
    reg r;
    sub0 s0();
endmodule
module sub0;
    reg r;
    sub1 s1();
endmodule
module sub1();
    reg r;
endmodule
```

Compile the test case as follows:

```
% vcs -design_stats test.v
```

VCS generates the statistics for the whole design in the vcs\_design\_stats.log file.

Compile the test case as follows:

```
% vcs -design_stats=deisgn,filename:my.log test.v
```

VCS generates the statistics for the whole design in the `my.log` file.

Compile the test case as follows:

```
% vcs -design_stats=module,filename:my.log test.v
```

VCS generates the statistics for each module of the design in the `my.log` file.

Compile the test case as follows:

```
% vcs -design_stats=hier:sub0,filename:my.log test.v
```

VCS generates the statistics for the part of hierarchy of the design starting from module `sub0` in the `my.log` file. In other words, VCS generates statistics only for `sub0` and `sub1` modules.

The content of the generated `vcs_design_stats.log` is as follows:

```
// =====
// DESIGN STATISTICS
// =====
//
Static
//
-----
// No. of all modules (module+interface+package+program) : 5
// No. of module instances:
2
// No. of all processes: 5
// No. of all nodes (variable+net): 452
// No. of all parameters: 44
// No. of class defns: 391
// No. of verilog task defns: 3

// =====
// MODULE "top" STATISTICS
// =====
// Syntax Features: sv, svtb
// No. of all modules (module+interface+package+program) : 1
// No. of module instances: 1
// No. of all processes: 1
// No. of all statements 5
//
```

```
// =====
// MODULE "sub" STATISTICS
// =====
// No. of all modules (module+interface+package+program) : 1
// No. of all nodes (variable+net) : 2
// No. of scalar nets: 2
```

---

## Limitations

The features have the following limitations:

- The design statistics are dumped only for Verilog portion of the design.
- VHDL and SystemC portions are not covered.

# 10

## Using X-Propagation

---

This chapter includes the following sections:

- “Introduction to X-Propagation”
- “Using the X-Propagation Simulator”
- “VHDL Two-State Objects in X-Propagation”
- “X-Propagation Code Examples”
- “Support for Active Drivers in X-Propagation”
- “Support for Ternary Operator”
- “Limitations”

---

## Introduction to X-Propagation

Designers use RTL constructs to describe hardware behaviors. However, certain RTL simulation semantics are insufficient to accurately model the hardware behaviors. Therefore, simulation results are either too optimistic or pessimistic than the actual hardware behaviors.

The simulation semantics of conditional constructs in Verilog and the simulation semantics of the STD\_LOGIC and STD\_LOGIC\_VECTOR types along with the Boolean equality and relational operators are insufficient to accurately model the ambiguity inherent in uninitialized registers and power-on reset values. This is particularly problematic when indeterminate states that are modeled as X values become control expressions.

Standard RTL simulations ignore the uncertainty of X-valued control signals and assign predictable output values. As a result, RTL simulations often fail to detect design problems related to the lack of X-Propagation. However, the same design problems can be detected in gate-level simulations. With X-Propagation support in RTL simulations, engineers can save time and effort in debugging differences between RTL and gate-level simulation results.

The simulation semantics of Verilog and VHDL control constructs is insufficient to account for the ambiguity of statements executed under X control. A more accurate simulation model to handle indeterminate control signals is to execute the design with a 0 and 1 control signal and then merge the results.

Gate-level simulations and pseudo-exhaustive 2-state simulations are techniques used to expose X-Propagation (Xprop) problems. However, as designs grow in size, these techniques become increasingly expensive and time consuming, often covering only a fraction of the overall design space.

The VCS Xprop simulator provides an effective simulation model that allows Xprop problems to be exposed by standard RTL simulations.

The VCS Xprop simulator provides two built-in merge modes that you can choose at either compile time or runtime:

- xmerge mode

This mode is more pessimistic than a standard gate-level simulation.

- tmerge mode

This mode is closer to actual hardware behavior and is the more commonly used mode.

In addition to these two merge modes, you can also select the vmerge mode at runtime to specify the standard RTL semantics, which effectively disables the enhanced Xprop semantics.

- vmerge mode

This mode is the classic Verilog and VHDL (optimistic) behavior.

---

## Guidelines for Running X-Propagation Simulations

Enabling Xprop on an entire design changes the simulation behavior, and may result in simulation failures. To facilitate deployment on existing designs, you can use the following divide-and-conquer approach to debug the failures:

- Enable Xprop on certain blocks at a time.
- Find and fix any design or testbench issues.
- Repeat the steps for the next set of blocks.

Debugging simulation failures is easier when only a small block is enabled for Xprop simulation at a time. However, resolving the Xprop simulation issues in all the small blocks independently does not guarantee that the entire design can simulate without any Xprop problems. Multiple iterations may be required to debug and fix all the issues.

One of the most common sources of simulation differences with Xprop enabled is incorrect initialization sequences. The behavior is typically caused by a reset or clock signal transitioning from 0 to X, 1 to X or vice versa.

If a flip-flop is sensitive to the rising edge of its clock signal, an X to 1 transition triggers the flip-flop and pass the value from input to output when coded using the Verilog `posedge` or VHDL `clk'` event type of usage. Effectively, the RTL constructs in these cases consider the X to 1 transition as `true`. However, in an Xprop simulation, the same clock transition causes the flip-flop to merge the input and output, possibly resulting in an unknown value. Therefore, to effectively load new values onto a flip-flop, you must ensure that clock signals have valid and stable values.

You can specify various Xprop behaviors using a configuration file. In an Xprop configuration file, you can specify the top-level module of a DUT (Design Under Test) and enable Xprop on the DUT instance tree. The Xprop technology is targeted for designs simulating actual hardware (synthesizable RTL code). Non-synthesizable or testbench blocks should be excluded from Xprop simulation using the configuration file.

Debugging a simulation mismatch is easier at the RTL level than at the gate level because RTL descriptions are closer to the actual functional intent of a circuit. There are different methods to debug RTL simulation failures.

A typical debug flow is:

- Identify a regression or test failure.
- Rerun the test with waveform dumping enabled.
- Go to point of test failure (assertion, monitor).
- Trace back a mismatching signal to its origin.
- Identify the root cause of the problem.

One method is to compare the dump file of a passing test and a failing test and search for differences near the point of failure.

Another debugging method is to compare a test when the simulation passes and when the simulation fails. The user identifies potential code modifications between the passing and the failing simulations that may cause simulation failure.

Traditional RTL debug techniques can be used to debug Xprop simulation failures.

However, you should generally not compare waveform files with and without Xprop enabled. This can lead to extraneous and wasted debug cycles. For example, resetting a device may take 10ms in normal RTL mode. In Xprop mode, the reset or clock may take 100ms due to an indeterminate reset signal. If you compare the waveform files of the two simulations, you can find that because the simulations are not cycle accurate with respect to one another, the actual problem is at a point much farther away in the future than the first few simulation mismatches.

Most simulation debug tools automatically trace back signal changes across multiple logic levels to some origin that caused the signal changes. These debug tools are closely tied to RTL behaviors. Since VCS signal update is different when Xprop is enabled, these debug tools may not function accurately in Xprop simulations. Some manual interventions may be required to use these debug tools correctly.

The recommended debug methodology is to implement sufficient number of assertions or testbench monitors. With this methodology any deviation from the correct design functionality triggers one of these checkers and gives a runtime error message. You can debug the simulation problem starting with the error message.

---

## Using the X-Propagation Simulator

The Xprop usage model compiles the design in Xprop mode and execute the simv executable. The `-xprop` compile-time option is used to enable Xprop and to specify the merge mode at run time. By default, VCS uses the `tmerge` merge mode. However, you can specify a different merge mode at either compile time or runtime.

Following is the syntax of the `-xprop` option:

```
vcs -xprop[=tmerge|xmerge|xprop_config_file]
      [-xprop=flowctrl]
      [-xprop=nestLimit=<limit>]
      other_vcs_options
```

Where:

`-xprop`

Use this option without an argument to enable Xprop in the entire design. The default `tmerge` merge mode is used at runtime.

`tmerge`

Use the `tmerge` merge mode in the entire design. The merge result yields `x` when all output values of logic 0 and logic 1 control signal are different, similar to a ternary operator. This mode is closer to actual hardware behavior and is more commonly used.

`xmerge`

Use the `xmerge` merge mode in the entire design. Merge result always yields `x`. This mode is more pessimistic than a standard gate-level simulation.

`xprop_config_file`

Specify a configuration file. You can define the scope of the Xprop instrumentation and select the merge mode in the configuration file. For more information on using the Xprop configuration file, see “[X-Propagation Configuration File](#)” .

flowctrl

The `xprop_vhdl.log/xprop.log` log file records a YES line for the for loop as next/continue, exit/break and return statements are supported. In addition, the parent statement chain is no longer disabled for Xprop with these structures and the logging of these disabled statements is changed from NO to YES.

`nestLimit=<limit>`

Specify the nesting limit for the `case` and `if` statements. Here, `<limit>` is any integral value. If you specify 0 or any negative value, Xprop is disabled completely. By default, the nesting limit for `case` and `if` statements is set to 128.

Note:

Simulation behavior is undefined for multiple specifications of the `-xprop` options, `tmerge`, `xmerge` and `xprop_config_file`. In the following command, the application of the option is undefined and an error is generated.

```
% vcs -xprop -xprop=xp_config_file1  
-xprop=xp_config_file2 design.v
```

You may specify the following options only once in the compilation command:

- Merge mode (`tmerge` or `xmerge`)
- Configuration file (`xprop_config_file`)

Examples:

```
vcs -xprop
```

```
vcs -xprop=xprop.cfg  
vcs -xprop=tmerge top.v
```

**Note:**

The automatic hardware inference of flip-flops in Verilog simulations is enabled by default. Flip-flops with an active reset value of 0 are correctly simulated when the reset signal transition from X to 0. VCS generates a file named `unifiedInference.log` file to record a list of inferred flip-flops

---

## Specifying X-Propagation Merge Mode

Merge mode can be mentioned at both compile time and runtime.

For example,

```
% vcs -xprop=xmerge -sverilog top.v
```

When the `-xprop=xmerge` option is specified, the design is compiled and simulation starts in the `xmerge` mode.

```
% vcs -xprop=tmerge -sverilog top.v
```

When the `-xprop=tmerge` option is specified, the design is compiled and simulation starts in the `tmerge` mode.

To change the merge mode at runtime you can invoke the `$set_x_prop` Verilog system task or VHDL `set_x_prop` procedural calls.

Verilog examples:

```
$$set_x_prop("tmerge");
```

```
$set_x_prop("xmerge");  
$set_x_prop("vmerge");  
$set_x_prop("xprop");
```

### VHDL examples:

```
set_x_prop("tmerge");  
set_x_prop("xmerge");  
set_x_prop("vmerge");  
set_x_prop("xprop");
```

#### Note:

You must use the `use` clause in the file that calls the `set_x_prop` procedures.

```
use SYNOPSYS.XPROP_USER.ALL;
```

When only `-xprop` compile-time option is passed without specifying a merge mode and if you do not use `$set_x_prop` to specify the merge mode, the default `tmerge` mode is used.

The `vmerge` runtime merge mode enables standard RTL simulation behaviors, which effectively disables the Xprop semantics.

#### Note:

When multiple merge modes are specified using runtime options at the same delta time, the last merge mode always wins.

Another method to specify the Xprop merge mode is to use a Xprop configuration file. The `xprop` runtime merge mode reverts the merge scheme to use the merge scheme specified at the compile time. For more information on how to use the Xprop configuration file, see “[X-Propagation Configuration File](#)” .

For example:

```
% vcs -xprop=xp_config cache.v alu.v
```

Use the merge mode and the scope of Xprop instrumentation specified in the Xprop configuration file `xp_config`.

Whenever the merge mode is changed at runtime, VCS generates a message. For example, when the merge mode is changed to `tmerge` mode, the following message is generated:

```
Note-[XPROP_RT_BANNER] Xprop Runtime Banner
Global merge mode is set to tmerge at time 0
```

Note:

Changing the merge mode does not retrigger the statements or blocks. You can view the effective values from the new merge mode only on the next input change.

## Compile Time Diagnostic Report

When you compile a design with Xprop enabled, VCS generate reports that record all the statements considered for Xprop instrumentation, whether or not the statements are instrumented, and the reason for statements not being instrumented. Reports are generated with the name `xprop.log` and `xprop_vhdl.log` for Verilog and VHDL respectively.

Report entries are created for the following HDL constructs:

- `if` statement in Verilog and VHDL
- `case` statement in Verilog and VHDL
- Body of an edge triggered `always` block in Verilog
- Edge sensitive expression in VHDL

Below is the format of a report entry in `xprop.log`/  
`xprop_vhdl.log` files:

```
filename:line_number YES|NO ["reason" (primary_line)]
```

Where:

`filename`

Name of the source file containing a statement being considered for Xprop instrumentation.

`line_number`

Line number that corresponds to the start of the statement.

`YES | NO`

Xprop instrumentation status of the statement.

`reason`

The reason for the statement not being instrumented. This is issued only when the Xprop instrumentation status is `NO`.

`primary_line`

The line number of the statement containing the actual construct not being instrumented. This is issued only when the Xprop instrumentation status is `NO`.

xprop.log example:

```
decode.v:3 YES
decode.v:7 NO "prevented by sub-statement" (12)
decode.v:16 NO "delay statement" (17)
decode.v:18 YES
decode.v:20 NO "a dynamic object" (22)
```

xprop\_vhdl.log example:

```
mux.vhd:22 YES
mux.vhd:35 YES
mux.vhd: 44 NO assignment has non-null delay (41)
```

The Xprop statistics are presented at the end of the xprop.log report. The statistics consist of the number of assignment statements considered for Xprop instrumentation, the number of statements instrumented, and the ratio of those two numbers (instrumented/instrumentable) that represents the percentage of the design instrumented for Xprop.

For example:

```
eth.v:31 YES
eth.v:45 NO "a dynamic type expression" (48)
eth.v:52 YES
=====
X P R O P      S T A T I S T I C S
instrumentable assignments:      7
instrumented assignments:        5
instrumentation success rate:   71%
```

The Xprop instrumentation numbers reported in xprop.log are essentially the same between different compilation flows. In certain cases, subtle internal differences between the compilation flows may affect the calculation of the instrumentation numbers. Even though

the instrumented code is same in both flows, the numbers in `xprop.log` may differ. You should not compare the Xprop instrumentation numbers between different compilation flows.

**Note:**

VCS does not record Xprop instrumentation information on modules that are excluded from Xprop via the configuration file. Instrumentation for instances excluded via the configuration file is recorded unless all instances of a particular module are excluded. The excluded modules do not appear in the `xprop.log` file.

The `xprop_merged.log` file contains merged statistics from both VHDL and Verilog.

## Querying X-Propagation at Runtime

You can use the `$is_xprop_active` Verilog system function or the `is_xprop_active` VHDL procedure to query the X-prop status for a particular module or an entity instance. The function returns an 1 if Xprop is enabled in the current instance.

For example:

```
$set_x_prop("tmerge");
$display( "%m: is Xprop active = %d", $is_xprop_active() ) ;

$set_x_prop("xmerge");
$display( "%m: is Xprop active = %d", $is_xprop_active() ) ;

$set_x_prop("vmerge");
$display( "%m: is Xprop active = %d", $is_xprop_active() ) ;
```

The Xprop configuration file:

```
tree {top}{xpropOn}
```

```
instance {top.dut2}{xpropOff}
```

The query result:

```
top.dut1: is Xprop active = 1  
top.dut1: is Xprop active = 1  
top.dut1: is Xprop active = 0  
top.dut2: is Xprop active = 0  
top.dut2: is Xprop active = 0  
top.dut2: is Xprop active = 0
```

In the above example, Xprop instrumentation in the `top.dut2` instance is disabled at compile time using the configuration file. As a result, the instrumentation cannot be changed at runtime using the `$set_x_prop` system tasks. The Xprop status is shown in the query result.

The example for querying in VHDL is:

```
report boolean'image(is_xprop_active);
```

## X-Propagation Instrumentation Report

You can generate an Xprop instrumentation report with the `-report` runtime option. The report displays the instrumentation status of every module instance in a design.

```
-report=xprop [+exit]
```

Where:

`xprop`

Generate an Xprop instrumentation report named `xprop_config.report` and continue the simulation.

`xprop+exit`

Generate an Xprop instrumentation report named  
`xprop_config.report` and terminate the simulation.

The following are the formats of the statements in the Xprop instrumentation report:

ON:`instance`  
OFF:`instance`

Where:

`instance`

A module instance in a design.

ON

The instance is included in Xprop instrumentation.

OFF

The instance is excluded from Xprop instrumentation.

For example:

`simv -report=xprop`

The `xprop_config.report` file:

ON: `top`  
ON: `top.il`  
ON: `top.p1`  
OFF: `top.p1.ul`

## **Automatic Hardware Inference of Flip-Flops Enabled by Default**

The automatic hardware inference of flip-flops in Verilog simulations is enabled by default. Flip-flops with an active reset value of 0 are correctly simulated when the reset signal transition from X to 0. VCS generates an `unifiedInference.log` file to record a list of inferred flip-flops. The unified inference statistics are presented at the end of the log file.

The following are the formats of the `unifiedInference.log` file entries:

```
filename:line_number YES:SyncFF|AsyncFF  
filename:line_number NO "reason"
```

Where:

`filename`

Name of the source file containing a flip-flop that is being considered for Xprop instrumentation.

`line_number`

Line number that corresponds to the start of a statement describing the flip-flop.

`YES`

The specified flip-flop is inferred.

`SyncFF|AsyncFF`

Type of the specified flip-flop. `SyncFF` indicates a synchronized flip-flop. `AsyncFF` indicates an asynchronous flip-flop.

NO

The specified flip-flop is not inferred.

reason

The reason for the specified flip-flop not being inferred.

For example:

```
lib.v:3 YES:AsyncFF
lib.v:10 YES:SyncFF
lib.v:17 NO "Unable to infer clock for flip-flop"
=====
Unified Inference Statistics
Number of always_ff: 0
Number of always_latch: 0
Number of always @*: 0
Number of always_comb: 0
Number of always: 4
Number of flip-flop candidates: 3
Number of synchronous flops inferred: 1
Number of Asynchronous flops inferred: 1
```

---

## X-Propagation Configuration File

The Xprop configuration file is used to define the scope of Xprop instrumentation in a design. The file allows you to specify the design hierarchies or modules to be excluded or included for Xprop

instrumentation. You can also use the file to specify merge modes. Synopsys recommends that you use a Xprop configuration file when Xprop is enabled.

**Important:**

If you use an Xprop configuration file, by default VCS does not perform Xprop instrumentation. You must use the `xpropOn` attribute to specify the design hierarchies or modules for Xprop instrumentation. For example:

```
module {*}{xpropOn};
```

The Xprop configuration file consists of the following types of statements:

- “X-Propagation Instrumentation Definition”
- “X-Propagation Merge Mode Specification”

The statements are processed in sequential order. Subsequent statements override the previously listed statements. If Xprop merge mode is specified multiple times in the configuration file, the merge mode from the last statement is enabled.

**Note:**

You can add comments to the file using the character types `//` and `/* */`. For example:

```
// This is a single line comment
/* This is
   a multi-line comment*/
```

## X-Propagation Configuration File Syntax

The following is the BNF of the Xprop configuration file.

```

xprop_config_text ::= { xprop_config_item ; }

xprop_config_item :=
| merge = merge_function
| xindex_select_method = select_mode
| disable_xindex = read_write_operation
| module_item
| instance_item
| tree_item

merge_function ::= tmerge | xmerge
select_mode ::= resolution | dimensional | random
read_write_operation ::= read | write

module_item ::= module { module_identifier_list } {
xprop_mode }

tree_item ::= tree { module_identifier_list } { xprop_mode }

instance_item ::= instance { instance_path_list } {
xprop_mode }

xprop_mode ::= xpropOn | xpropOff | tmerge | xmerge

module_identifier_list ::= module { , module }

module ::= module_identifier

instance_path_list ::= instance_path { , instance_path }
instance_path ::= { instance_identifier . }
instance_identifier

```

### Note:

If a merge mode is explicitly specified for a module, or a tree, or an instance, then that mode is used throughout the simulation, whenever Xprop is enabled on that module, or instance, or tree.

For the pure Verilog and the Verilog portion of an mixed HDL design, the following warning message is generated:

```
Warning-[UACF] Unrecognized attribute in Config file  
Unrecognized attribute 'xmerge' found in config statement,  
it will be ignored.
```

## X-Propagation Instrumentation Definition

The following are the BNF rules for the Xprop instrumentation definition in a configuration file.

```
module_item ::= module { module_identifier_list } {  
    xprop_mode }
```

```
tree_item ::= tree { module_identifier_list } {  
    xprop_mode }
```

```
instance_item ::= instance { instance_path_list } {  
    xprop_mode }
```

module\_item

Apply the specified Xprop mode to all modules in the list.

instance\_item

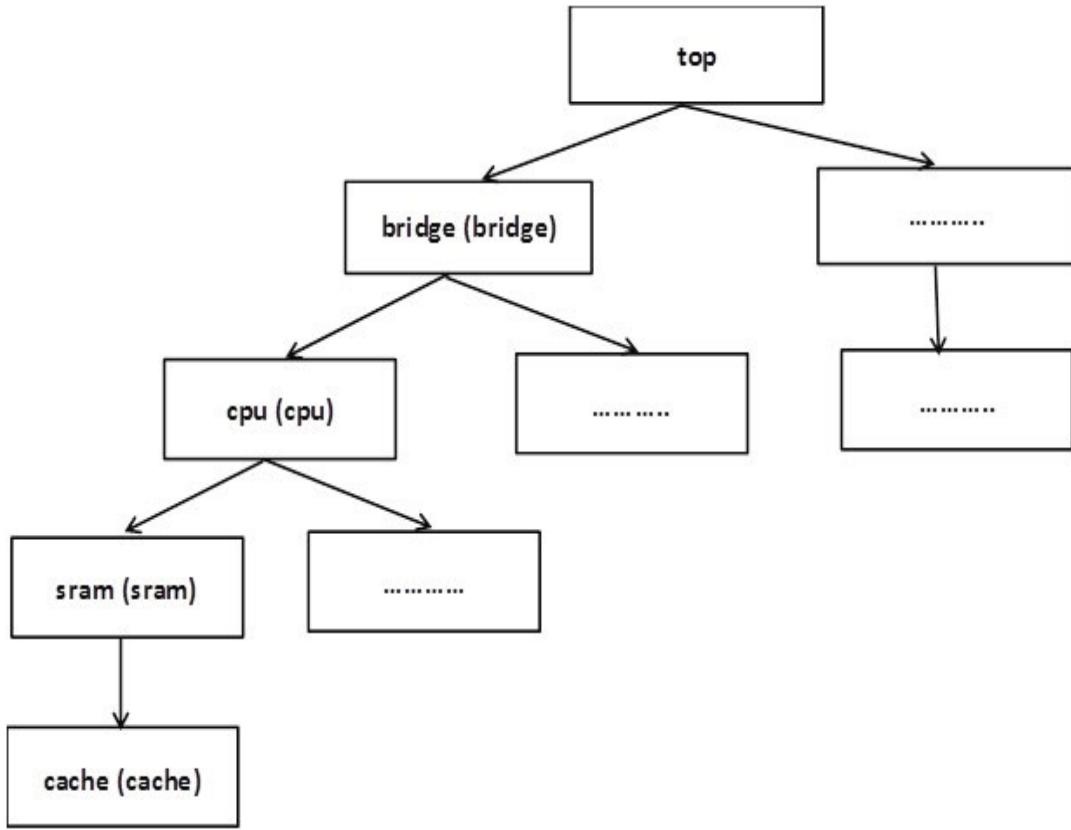
Apply the specified Xprop mode to all instances in the list and recursively to all the sub-instances.

tree\_item

Apply the specified Xprop mode to all modules in the list and recursively to all the sub-instances.

For example,

Consider the following figure, in which Xprop instrumentation is turned off for the tree and the module and is turned on for the instance.



```

tree      { bridge }           { xpropOff } ;
instance { top.bridge.cpu } { xpropOn } ;
module   { sram }             { xpropOff } ;

```

The first line specifies that the entire sub-tree under the module `bridge` should be excluded from Xprop instrumentation. The second line designates that the sub-tree under `top.bridge.cpu` should be included for gate-X-propagation. The third line specifies that all instances of the `sram` module are excluded from Xprop instrumentation and not the sub-instance, `cache`.

A module specification in a configuration file only affects the module instances, but not its sub-instances. For Verilog modules only, a module identifier may include asterisks (\*) to denote a wildcard string.

### X-Propagation Merge Mode Specification

The following are the BNF rules for the Xprop merge mode specification in a configuration file.

```
merge = merge_function  
merge_function ::= tmerge | xmerge
```

`merge_function`

Enable the specified Xprop merge mode.

For example:

```
merge = xmerge ;
```

In the above example, `xmerge` mode is enabled.

---

### Xprop Instrumentation Control

[Table 10-1](#) is a summary of the Xprop instrumentation control in a design hierarchy, when Xprop is enabled using various methods.

**Table 10-1 Xprop Instrumentation Control**

| Xprop Specification                                                                                                                                                                                                                     | Instrumentation Control                              | Runtime Control                                                                                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Compile-time option<br><br>example:<br>vcs -xprop=tmerge                                                                                                                                                                                | Entire hierarchy is instrumented for Xprop semantics | Verilog task call<br>\$set_x_prop or VHDL procedure call set_x_prop can be used to change merge mode at runtime for instances that have been already compiled for Xprop<br><i>Caution: Use sparingly</i> |
| Configuration file<br><br>example:<br>vcs -xprop=cfg                                                                                                                                                                                    | Instance/module specific control                     | Verilog task call<br>\$set_x_prop or VHDL procedure call set_x_prop can be used to change merge mode at runtime for instances that have been already compiled for Xprop<br><i>Caution: Use sparingly</i> |
| Pragma based<br><br>Verilog example:<br>always (*xprop_off*)<br>@ (posedge clk)<br><br>VHDL example:<br>process (s, a, b)<br>-- pragma xprop_off<br>begin<br>if (s = '1') then<br>r <= a;<br>else<br>r <= b;<br>end if;<br>end process; | Disabled Xprop for this process only                 | Process is disabled and cannot be enabled at runtime                                                                                                                                                     |

When a model uses Xprop configuration file, by default nothing is instrumented for Xprop.

To enable Xprop on all modules, packages, and \$unit, start the configuration with a wildcard \* (asterisk).

```
module { * } { xpropOn };
```

As an example, consider the following code that needs to be simulated:

```
%cat t.v:  
`include "unit_functions.vh"  
module top ();  
    alldmtd alldmtd_xPropMe ();  
    alldmtd alldmtd_FreeSpirit ();  
endmodule  
  
module alldmtd ();  
    import my_pkg::*;  
    logic [6:0] a7,b7,c7;  
    logic a,b;  
        assign a7 = f_INC(b7,a ,b );  
        assign c7 = f_PKG(b7,a ,b );  
endmodule  
  
%cat unit_functions.vh :  
function logic [6:0] f_INC(logic [6:0] vec, logic enable45,  
logic enable67);  
    logic [6:0] tmp;  
    unique casez ({vec,enable45,enable67})  
        9'b0000000_11 : tmp = 7'b1111111;  
        9'b1111111_11 : tmp = 7'b1111110;  
        9'b1111110_11 : tmp = 7'b1111100;  
        default         : tmp = 7'b1111111;  
    endcase  
    f_INC = tmp;  
endfunction  
  
%cat pkg_functions.vh :  
package my_pkg;  
function logic [6:0] f_PKG(logic [6:0] vec, logic enable45,  
logic enable67);  
    logic [6:0] tmp;  
    unique casez ({vec,enable45,enable67})  
        9'b0000000_11 : tmp = 7'b1111111;  
        9'b1111111_11 : tmp = 7'b1111110;
```

```

    9'b1111110_11 : tmp = 7'b1111100;
    default        : tmp = 7'b1111111;
    endcase
    f_PKG = tmp;
endfunction

endpackage

```

**To enable Xprop instrumentation on just some hierarchy (without \$unit and without any packages), use the following rule:**

```
instance {top.alldmtxd_xPropMe } {xpropOn};
```

**To enable Xprop instrumentation on \$unit, use the following rule:**

```
module {_vcs_unit* }{xpropOn};
```

**To enable Xprop instrumentation on my\_pkg package, use the following option:**

```
module {my_pkg }{xpropOn};
```

**To turn on xprop on alldmtxd\_xPropMe(), the configuration file looks like this.**

```
%cat xprop.cfg
instance {top} {xpropOff};
instance {top.alldmtxd_xPropMe } {xpropOn};
```

## Limitation

The Xprop Instrumentation configuration has the following limitation:

- Partial wildcard is not supported in Xprop configuration rules.

---

## Process Based X-Propagation Exclusion

The Xprop configuration file allows exclusion of Xprop at module or entity level. For more information, see “[X-Propagation Configuration File](#)” . If you need finer granularity, you can use the `xprop_off` attribute to disable Xprop on specific process.

Verilog example:

```
always (* xprop_off *) @ ( posedge clk ) begin
    if (we) begin
        q <= in;
    end
end
```

VHDL example:

```
process (s, a, b)
-- pragma xprop_off
begin
    if (s = '1') then
        r <= a;
    else
        r <= b;
end if;
end process;
```

Note:

The pragma can be specified between the `process` keyword and the final semicolon after the `end process` statement. The pragma has no effect if it is specified outside of the `process` and `end process` statements. No message is issued in such cases.

The pragma is not required to start on a new line. However, the `pragma` keyword must be the first word of a comment followed by the `xprop_off` keyword with no intervening text.

---

## Support for XIndex Element Merging

VCS supports XIndex element merging in Verilog. This helps you to configure the simulator to propagate `xs` through array indexes. This improves RTL confidence and overall verification productivity by reducing debug effort during gate-level simulations.

The following two operations are considered for XIndex element merging in Verilog:

- Write element merging semantics
- Read element merging semantics

A write operation merges the RHS expression into each potentially written element. A read operation merges values of all the elements potentially read.

When dealing with X-indexes, there are two distinct operations to consider: Index selection and Merge mode.

The Index selection is the mechanism that determines the array locations involved in the indexing operation.

The Merge mode is the semantic associated with the array locations identified by the index selection. There are three different index selection mechanisms and two merge behaviors, which leads to six distinct new behaviors, in addition to the default HDL behavior.

The following are the index selection mechanisms:

- Dimensional index
- Random index
- Index resolution

The three index selection mechanisms exhibit different accuracy and performance characteristics.

Aside from these different index selection mechanisms, Xprop merge modes (`tmerge`, `xmerge`, `xpropOff`) is applied. The merge mode is not independently controllable. Instead, the merge mode of the local context is applied.

XIndex considers all select expressions within an Xprop region as specified by the configuration file, if any, including select expressions in `always` blocks, processes and continuous or concurrent assignments. However, XIndex does not consider select expressions that are in instance port connections.

XIndex does not result in any additional YES entries in the log file. All select expressions within Xprop-enabled regions are considered. However, instance port expressions and XIndex expressions on function calls associated with output or inout expressions are not instrumented. Such unsupported expressions are logged in the `xprop.log` file with a NO entry.

## **X-Index**

An indexing expression that contains one or more X bits (Verilog) or the X value (VHDL).

## **Index BSpace**

Index BSpace is the set of values that are possible in the addressable space of the designated array given a particular X-Index pattern. The addressable space of the array are the index values within the zero-based power-of-2 space that encompasses the bounds of dimensions that are indexed.

For example, the BSpace for the `X0X` pattern in range `[6:1]` is the set of minterms `{001, 100, 101}`. In this case, the zero-based power-of-2 space for the range `6:1` is `7:0` ( $2^{**3}-1:0$ ). For read operations, the index BSpace of an X pattern includes all minterms in the set. For write operations, the index BSpace is further constrained to the subset of minterms that are in-bounds.

A multidimensional BSpace is comprised of a set of n-tuples, where n is the number of dimensions addressed in the multidimensional array select expression. For example, consider the following code snippet:

```
logic array[6:1][7:0][3:0];
logic [2:0] a, b;
logic [1:0] c;
logic d;
...
a = 3'bxx0x;
b = 1;
c = 2'b0x
array[a][b][c] = d;
```

For the assignment in the last statement, BSpace is the set of 3-tuples:

```
{(001, 1, 00), (001, 1, 01), (100, 1, 00), {100, 1, 01}, (101, 1, 00), (101, 1, 01)}
```

For a read operation, BSpace also includes the out-of-bounds tuples within the power-of-2 space:

```
{(000, 1, 00), (000, 1, 01), (001, 1, 00), (001, 1, 01), (100, 1, 00), {100, 1, 01}, (101, 1, 00), (101, 1, 01)}
```

## **Addressing Models**

When considering dimensional indexing in the context of XIndex, the array addressing models the actual hardware.

XIndex element merging considers every dimension to exist within a power-of-2 sized space. High-order bits that lie outside an array's addressing space are ignored.

For ranges that are not a power-of-2, some minterms may be out-of-bounds. For write operations, these out-of-bounds minterms are always ignored, regardless of the element merging method. For read operations, these out-of-bounds minterms result in the undefined value, which is currently  $\text{X}$ .

The section consists of the following subsections:

- “[Unsigned Indexing](#)”
- “[Signed indexing](#)”

### **Unsigned Indexing**

Dimensions where both bounds are positive (including 0) are considered unsigned. For unsigned indexing, the power-of-2 space is  $2^{**n-1} : 0$ , where  $n$  is the smallest integer such that  $2^{**n}$  is greater than the high bound of the dimension. For example, the range  $6 : 1$  results in  $n=3$ .

### **Signed indexing**

Dimensions where either bounds are negative are considered signed indexing. For signed indexing the power-of-2 space is  $2^{** (n-1)} - 1 : -(2^{** (n-1)})$ , where  $n$  is a smallest integer such

that  $2^{**}(n-1)$  is greater than the high bound and  $- (2^{**}(n-1))$  is less than or equal to the low bound. For example, the ranges `8:-8` and `7:-9` result in  $n=5$ , while range `7:-8` results in  $n=4$ .

## Merge Modes

The runtime merge mode for XIndex Element Merging is the same as for the Xprop merge mode. The merge mode is applied to specific instances when specified in the configuration file. Alternatively, the merge mode can be set globally in either the configuration file or via the compile-time command line option.

The merge mode can also be modified at runtime via the `$set_x_prop()` system task.

Note:

The X-Index Element Merging merge mode is not independently controllable.

### TMerge

For write operations, this mode merges the value being written (RHS) with the value of each element in the BSpace and storing the result of the merge into the corresponding array location. For read operations, the value read is the merge of all the values obtained from each index in the BSpace.

### XMerge

When the BSpace has more than one minterm this merge mode always results in an X being read or written. For a write operation, all elements selected by the BSpace are written.

## **VMerge**

The default X-Index merge mode follows standard Verilog semantics for read and write operations. VHDL also follows these same semantics when in Xprop mode.

- X-Index write behavior: The write is ignored.
- X-Index read behavior: The result is always X.

Note:

This merge mode disables both the index selection as well as the Xprop merge mode.

## **Index Selection Methods**

Element merging method is applied within an Xprop instrumentation region and are not implemented in regions where Xprop is disabled.

The section consists of the following subsections:

- “[Dimensional Index](#)”
- “[Random Index](#)”
- “[Index Resolution](#)”

### **Dimensional Index**

The dimensional index method computes the BSpace by first converting an X-Index into all xs in the corresponding dimension. An index that does not contain an x is unaffected. Therefore, an index with value `3'bxx0x` is converted to `3'bxxxx` prior to computing BSpace minterms. Once the BSpace is computed, the RHS of the assignment is merged with the element at each index in the X-Index

BSpace, similar to the resolution model. For read operation, the elements of the BSpace are merged with each other to derive the result.

For VHDL, no conversion is necessary for BSpace computation as an index is either known or it is X.

This index-selection mechanism is specified via the following configuration file directive:

```
xindex_select_method=dimensional
```

### **Random Index**

For the random index method, an in-bound random index is selected from the X-Index BSpace. For a write operation, the selected array cell is written with a value computed according to the merge mode. For a read operation, the value at the (randomly) selected index is returned without merging.

For VHDL, a random index is generated in-bounds of the dimension being indexed.

This index-selection mechanism is specified via the following configuration file directive:

```
xindex_select_method=random
```

### **Index Resolution**

For write operations, the index resolution mode merges the value being written (RHS) with each element addressed by the minterms of X-Index BSpace.

If a BSpace minterm is out-of-bounds for that dimension, the element write is ignored.

For read operations, the values of all the elements in the BSpace are merged to derive the result. If any BSpace index at a dimension is out-of-bounds, the value merged for the element is the default value X.

VHDL does not support the Index Resolution selection mechanism. If the resolution mechanism is specified in VHDL, it is downgraded to the dimensional model.

This index-selection mechanism is specified via the following configuration file directive:

```
xindex_select_method=resolution
```

## **Disabling XIndex Merging for Read or Write Operations**

The index-selection mechanism and the merge mode are applied consistently to all indexing expressions for read and write operations. However, there may be situations where the XIndex element merging might cause undue performance degradation on certain operations. In particular, read operations employing the index resolution mechanism along with the `tmerge` merge mode can incur a high performance cost for relatively small accuracy gain. As read operations are much more common than write operations, and a single XIndex read operation may require the merging of many cells (potentially the entire array), the added accuracy might be unwarranted. To control such situations, you can provide two configuration directives to independently enable or disable the XIndex element merging of either a read or a write operation.

To enable or disable the XIndex element merging of a read operation, use the following configuration directive:

```
disable_xindex = read
```

To enable or disable the XIndex element merging of a write operation, use the following configuration directive:

```
disable_xindex = write
```

When you specify read or write to this directive, it disables the corresponding operation. By default, you can enable both of these operations by specifying the `xindex_select_method` directive.

## Use Model

You can enable XIndex element merging by using the configuration file directive as follows:

```
% vcs -xprop=<xprop_config_file> testcase.v  
<other_vcs_options>
```

Where, `xprop_config_file` specifies the Xprop configuration file.

The configuration file contains the following content:

```
merge = merge_mode  
module {memory_rtl} {xpropOn};  
xindex_select_method=select_method;
```

Where:

`merge_mode`

Specifies the merge mode. The merge mode can be `tmerge`, `xmerge`, or `xpropOff` (`vmerge`).

`select_method`

Specifies the element merging method. The element merging method are dimensional, random, or resolution. If you specify more than one element merging method, the last method specified in the configuration file is used.

## Examples

Consider the following examples:

```
//example.v
=====
module memory_rtl (clk,reset,wr,addr,data_in,data_out);
    input    clk,reset;
    input    wr;
    input    [1:0] addr;
    input    [7:0] data_in;
    output   [7:0] data_out;
    wire     [7:0] data_out;
    reg      [7:0] mem [0:3];
    reg      [7:0] rdata;
    reg out_enable;
    assign   data_out = out_enable ? rdata : 'bz;
    always @ (posedge clk or posedge reset)
    begin
        if (reset) begin
            mem[0] <= 8'b11000011;
            mem[1] <= 8'b11010100;
            mem[2] <= 8'b11110110;
            mem[3] <= 8'b00110011;
        end else if(wr)
            mem[addr] <= data_in;
    end
    always @ (posedge clk)
    begin
        if(wr==0)
        begin
            rdata <= mem[addr];
            out_enable <= 1'b1;
        end
        else out_enable <=1'b0;
    end
endmodule
```

```

end
endmodule

// testcase.sv
=====
module top;
bit clk;
reg reset,wr;
reg [1:0] addr;
reg [7:0] data_in;
wire [7:0] data_out;
always #5 clk++;
memory_rtl inst(clk,reset,wr,addr,data_in,data_out);
initial
begin
    reset = 1'b1;
    #3;
    reset = 1'b0;
    $display("*****reading from memory location*****");
    wr = 1'b0;
    addr = 2'b0x;
    @ (negedge clk);
    $display("addr : %b data_out : %b",addr,data_out);
    addr = 2'b1x;
    @ (negedge clk);
    $display("addr : %b data_out : %b",addr,data_out);
    $display("*****writing to memory location*****");
    wr = 1'b1;
    addr = 2'b0x;
    data_in = 8'b11010100;
    @ (negedge clk);
    $display("addr : %b mem[0] : %b mem[1] : %b mem[2] : %b
mem[3] : %b",addr,top.inst.mem[0],top.inst.mem[1],
top.inst.mem[2],top.inst.mem[3]);
    addr = 2'b1x;
    data_in = 8'b11000100;
    @ (negedge clk);
    $display("addr : %b mem[0] : %b mem[1] : %b mem[2] : %b
mem[3] : %b",addr,top.inst.mem[0],top.inst.mem[1],
top.inst.mem[2],top.inst.mem[3]);
    $finish;
end

```

```
endmodule
```

## Dimensional Mode

To run the examples in the dimensional merging mode, use the following compile-time and runtime commands:

```
% vcs -sverilog example.v testcase.sv -xprop=testcase.cfg  
% ./simv
```

Where, `testcase.cfg` has the following entries:

```
merge=tmerge;  
module {memory_rtl} {xpropOn};  
xindex_select_method=dimensional;
```

It generates the following output:

```
*****reading from memory location*****  
addr : 0x data_out : xxxx0xxx  
addr : 1x data_out : xxxx0xxx  
*****writing to memory location*****  
addr : 0x mem[0] : 110x0xxx mem[1] : 11010100 mem[2] :  
11x101x0 mem[3] : xxx10xxx  
addr : 1x mem[0] : 110x0xxx mem[1] : 110x0100 mem[2] :  
11xx01x0 mem[3] : xxxx0xxx
```

## Random Mode

To run the examples in the random merging mode, use the following compile-time and runtime commands:

```
% vcs -sverilog example.v testcase.sv -xprop=testcase.cfg  
% ./simv
```

Where, `testcase.cfg` has the following entries:

```
merge=tmerge;
```

```
module {memory_rtl} {xpropOn};  
xindex_select_method=random;
```

It generates the following output:

```
*****reading from memory location*****  
addr : 0x data_out : 11000011  
addr : 1x data_out : 00110011  
*****writing to memory location*****  
addr : 0x mem[0] : 11000011 mem[1] : 11010100 mem[2] :  
11110110 mem[3] : 00110011  
addr : 1x mem[0] : 11000011 mem[1] : 11010100 mem[2] :  
11110110 mem[3] : xxxx0xxx
```

## Resolution Mode

To run examples in the resolution merging mode, use the following compile-time and runtime commands:

```
% vcs -sverilog example.v testcase.sv -xprop=testcase.cfg  
% ./simv
```

Where, `testcase.cfg` file has the following entries:

```
merge=tmerge;  
module {memory_rtl} {xpropOn};  
xindex_select_method=resolution;
```

It generates the following output:

```
*****reading from memory location*****  
addr : 0x data_out : 110x0xxx  
addr : 1x data_out : xx110x1x  
*****writing to memory location*****  
addr : 0x mem[0] : 110x0xxx mem[1] : 11010100 mem[2] :  
11110110 mem[3] : 00110011  
addr : 1x mem[0] : 110x0xxx mem[1] : 11010100 mem[2] :  
11xx01x0 mem[3] : xxxx0xxx
```

## Limitations

The feature has the following limitations:

- Only memory of type `reg`, `wire`, and `logic` declared in the module scope are supported.
- Memory declared inside a class and dynamic array types are not supported.

---

## Bounds Checking

In Verilog, the `-boundscheck` compile-time option can be used to catch invalid indices. When the option is specified, the four types of assertions listed in [Table 10-2](#) are enabled:

Table 10-2 Assertions Enabled In Bounds Checking

| Assertion Type | Description                                                   | Default Behavior |
|----------------|---------------------------------------------------------------|------------------|
| xindex-wr      | Write with indeterminate index<br>(index containing X values) | Warning          |
| xindex-rd      | Read with indeterminate index<br>(index containing X values)  | Warning          |
| index-wr       | Write with out-of-bounds index                                | Warning          |
| index-rd       | Read with out-of-bounds index                                 | Warning          |

In Xprop simulations, the behavior and severity of the above four types of assertions can be controlled by a set of runtime Xprop assertion control Verilog system tasks. The system tasks operate on all assertions of the specified type in a design.

**Note:**

For VHDL, these assertion controls are synonymous to each other as VHDL does not distinguish between an out-of-bounds index and an X index (`xindex`). It also does not distinguish between an indexed-read and an indexed-write.

- `$xprop_assert_on(assertion_type)`  
Enable the specified assertion type.
- `$xprop_assert_off(assertion_type)`  
Disable the specified assertion type. Assertion check of the specified assertion type is stopped until a subsequent call of the `$xprop_assert_on` task with the same assertion type is executed.
- `$xprop_assert_fatal(assertion_type)`  
Set the severity of the specified assertion type to fatal. When an assertion of the specified type is triggered, the simulation terminates.
- `$xprop_assert_warn(assertion_type)`  
Set the severity of the specified assertion type to warning. When an assertion of the specified type is triggered, the simulation continues.

**Note:**

The Xprop assertion control system tasks have no effect on Verilog behavior if the `-boundscheck` option is not used. The assertion control system tasks are always effective in VHDL.

---

## Detecting Unknown Values in Type Conversion Functions

VHDL provides type conversion functions to convert different types. For example, the `conv_integer` function is used to convert `std_logic_vector` to `integer` types. If the `std_logic_vector` contains an unknown value, then the type conversion results in a value of 0. This kind of conversion is considered as a lossy conversion because a 4-state value is converted to a 2-state value. Lossy conversion can mask potential design problems due to the lack of Xprop. VCS provide an assertion to detect lossy conversions. You can use the following methods to enable or disable the assertion:

- Specify an `ASSERT_IGNORE_XPROP_LOSSY_CONVERSION` declaration in the `synopsys_sim.setup` file as shown:

```
ASSERT_IGNORE_XPROP_LOSSY_CONVERSION<=[TRUE, FALSE]>
```

- Specify the `set_assert_off(LOSSY_CONV)` procedure as shown:

```
set_assert_on(LOSSY_CONV)
```

---

## Time Zero Initialization

When Xprop is enabled, initialization of a latch or flip-flop at time zero may result in an `x` output instead of the initialized value. At time zero, the SystemVerilog `always_latch` and `always_comb` processes are executed. An `x` value on the clock signal propagates through the device and may cause an indeterminate output.

---

## Handling Non-pure Functions Due to Static Lifetime

VCS provides an easy way to denote the lifetime of all user-defined functions that do not specify an explicit lifetime as automatic.

Functions with a static lifetime (default) often create side-effects that require the compiler to consider those functions as non-pure. The side-effects due to static lifetime sometimes leads to simulation-synthesis mismatches. Moreover, they prevent the code that calls those functions from being instrumented by hardware-accurate simulation features, such as Xprop. To eliminate the side-effects due to static lifetime, VCS provides the `-fauto` compile-time option. When this option is specified, all Verilog functions that do not specify an explicit lifetime are automatically converted to automatic functions.

The default lifetime of Verilog functions defined within modules or interfaces is static (note that functions in program blocks or class methods are already automatic by default). This means that all the arguments, the return value, and all the variables declared within those functions are static and retain their values in between calls. The retention of values across calls may result in side-effects such that the behavior of the function depends not only on the current argument values, but also on the previous invocations. By definition, such functions are considered as non-pure functions.

For example:

```
module foo;
    function crc (input [31:0] data);
        reg tmp;
        tmp = tmp ^ data;
        crc = tmp;
    endfunction
endmodule
```

In the above example, the `crc` function is static by default. Therefore, the state of the `tmp` variable is retained through each invocation of the `crc` function. The result of each `crc` function call does not depend solely on the input argument `data`. Therefore, the `crc` function is a non-pure function.

You can use the `-fauto` compile-time option to change the lifetime of all functions that do not specify an explicit lifetime to automatic. The automatic lifetime eliminates the potential simulation-synthesis mismatches and enable the instrumentation of code that calls such functions in Xprop simulations. The behavior of the `-fauto` option is similar to declaring an automatic lifetime for the functions.

For example:

```
function automatic crc (input [31:0] data);
```

Note:

Functions that do indeed rely on the value retention side-effect for correct simulation need to be modified to specify the intended lifetime.

For example:

```
function static crc (input [31:0] data);
```

---

## Supporting UCLI Commands for X-Propagation Control Tasks

X-Propagation (Xprop) supports UCLI/TCL commands. The commands allow you to control the Xprop behavior at runtime. The commands return a success value or return a resultant value for a query, such as `$is_xprop_active()`.

---

## Use Model

This section describes how to use the UCLI commands.

### UCLI Command to Specify the Merge Mode

The syntax of the new UCLI command to specify the merge mode is as follows:

```
xprop {-is_active [inst_name]  
| -merge_mode {vmerge|tmerge|xmerge|xprop}}
```

This command is equivalent to the Verilog `$set_x_prop()` and `$is_xprop_active()` system task calls, as well as the VHDL built-in package sub-programs `XPROPUSER.set_x_prop()` and `XPROP_USER.is_xprop_active()`.

For example,

```
xprop -is_active top.dut.core0.dff  
xprop -merge_mode vmerge  
xprop -merge_mode xprop
```

Note:

- For a non-Xprop simulation, the command returns `False` and generates a warning message if the `-merge_mode` option is present.
- You must use either `-is_active` option or `-merge_mode` option. If neither or both options are provided, or if the value of the `-merge_mode` option is not valid, a help message is generated.

- The UCLI command allows you to provide both relative (to the current scope) instance name and absolute instance name. If no instance name is provided for the `-is_active` option, the command uses the current scope.
- If the `[inst_name]` option does not exist, a warning message is generated and the UCLI command returns `False`.

## **UCLI Command to Control Error Messages or Warning Messages**

The syntax of the new UCLI command to control error messages or warning messages is as follows:

```
report_violations -type {oob_index_rd | oob_index_wr |
x_index_rd | x_index_wr | lossy_conversion| enum_cast |
ffdcheck} {-severity {warn | error}| -on | -off}
```

The following command is equivalent to

```
$xprop_assert_{on,off,warn,fatal} () or
XPROP_USER.xprop_assert_{on,off,warn,fatal} ()
Verilog system task calls and VHDL XPROP_USER built-in package
sub-programs.
```

Note:

- Multiple options are allowed, however, at least one option must be provided. If no option is provided, or illegal options or option values are provided, a help message is generated.
- If both `-on` and `-off` options are provided, a warning message is generated. The command returns `False` and the violation reporting state is not changed.

- Multiple options are allowed to the (singular) `-type` option, if presented in a TCL list (enclosed in braces and separated by spaces).
- For pure VHDL in non-Xprop mode, this command is not relevant under any circumstances. Hence, this command always generates a warning message and returns `False`.
- For Verilog and VHDL in non-Xprop mode, this command generates a warning message and returns `False` for `lossy_conversion`, `enum_cast` and `ffdcheck` violation types. The VHDL portion of the design is not affected by this command while running in non-Xprop mode.

## VHDL Two-State Objects in X-Propagation

VCS supports following two-state objects in Xprop:

- `Integer` (`XInteger`)
- `Boolean` (`XBoolean`)
- User-defined enumerations less than 256 values (`XEnum`)

In VHDL, integers are frequently used for array indexes as an array constraint is always of discrete type. As these integer values cannot represent `x` values during Xprop simulation, the `x` value is lost.

To propagate the `x` value, VCS promotes objects and object elements from 2-state types to X-state types. These types have a particular value that is defined to be the `XValue`.

---

## Supported XValues

The following XValues are supported:

- `XInteger`: `0x7fffffed`  
You can override this XValue using the `vhdl_intx_override=<value>` directive in the Xprop configuration file. If the RTL simulation computes integer value that matches this `XInteger` value, then a runtime error (`VHDL_XPROP_INT_X`) is issued to notify this overlap.
- For the type `XEnum` and `XBoolean`, `x` value is an out-of-range value.

---

## Supported Objects

All the objects that are declared in the Xprop enabled entity are promoted.

---

## Supported Expressions

Expressions are promoted based on the operand type. The following list shows the list of operators and the corresponding operands:

- LRM Operators  
XBoolean overload functions are created for the following Boolean operators:
  - Logicals -- `not`, `and`, `or`, `xor`, `nand`, `nor`, `xnor`

XInteger overload functions are created for the following Integer operators:

- Arithmetic: +, -, /, \*, mod, rem
- Miscellaneous: \*\*, abs
- Shift: sll, srl, sla, sra, rol, ror

Note: The shift operators are only available on unidimensional arrays of bit and Boolean types. An x value on shift or rotate count of a Boolean vector results in all xs on the Boolean vector. The bit vector values do not change with a indeterminate shift or rotate count.

- Relational: =, /=, <, <=, >, >=
- Std\_logic\_arith Operators
  - For the arithmetic + operator:
    - Function + (L: Xinteger, R signed) return signed
    - Function + (L: signed, R Xinteger) return signed
    - Function + (L: Xinteger, R unsigned) return unsigned
    - Function + (L: unsigned, R Xinteger) return unsigned
    - Function + (L: Xinteger, R signed) return std\_logic\_vector
    - Function + (L: signed, R Xinteger) return std\_logic\_vector
    - Function + (L: Xinteger, R unsigned) return std\_logic\_vector
    - Function + (L: unsigned, R Xinteger) return std\_logic\_vector

For arithmetic + operator, the function returns a vector with all elements set to  $x$  when the integer input is the  $x$  value.

For the other arithmetic operators, – and \*, the function overload is analogous.

- For the relational = operator:
  - Function = (L: Xinteger, R: signed) return Xboolean
  - Function = (L: signed, R: Xinteger) return Xboolean
  - Function = (L: Xinteger, R: unsigned) return Xboolean
  - Function = (L: unsigned, R: XInteger) return Xboolean

For other relational operators /=, <, <=, >, >= the function returns a Xboolean  $x$  value when either the XInteger argument is  $x$  or when the signed or unsigned vector has any  $x$  element.

- For the conversion functions:
  - Function conv\_integer (signed) return XInteger
  - Function conv\_integer (unsigned) return XInteger
  - Function conv\_signed (XInteger) return signed
  - Function conv\_unsigned (XInteger) return unsigned
  - Function conv\_std\_logic\_vector (XInteger) return std\_logic\_vector

Conversion functions return all “ $x$ ” when the argument is  $x$ .

- Std\_logic\_signed Operators
  - For the arithmetic + and – operator:

- **Function + (L: std\_logic\_vector, R XInteger) return std\_logic\_vector**
- **Function + (L: XInteger, R std\_logic\_vector) return std\_logic\_vector**
- **Function - (L: std\_logic\_vector, R XInteger) return std\_logic\_vector**
- **Function - (L: XInteger, R std\_logic\_vector) return std\_logic\_vector**
- For the relational = operator:
  - **Function = (L: std\_logic\_vector, R XInteger) return Xboolean**
  - **Function = (L: XInteger, R std\_logic\_vector) return Xboolean**
- For other relational operators /=, <, <=, >, >= the function overload is analogous.
- For the conversion functions:
  - **Function conv\_integer (std\_logic\_vector) return XInteger**
- Std\_logic\_unsigned Operators
  - For the arithmetic + and - operator:
    - **Function + (L: std\_logic\_vector, R XInteger) return std\_logic\_vector**
    - **Function + (L: XInteger, R std\_logic\_vector) return std\_logic\_vector**

- Function - (L: std\_logic\_vector, R XInteger) return std\_logic\_vector
- Function - (L: XInteger, R std\_logic\_vector) return std\_logic\_vector
- For the relational = operator:
  - Function = (L: std\_logic\_vector, R XInteger) return Xboolean
  - Function = (L: XInteger, R std\_logic\_vector) return Xboolean

For other relational operators /=, <, <=, >, >= the function overload is analogous.

- For the conversion functions:
  - Function conv\_integer(std\_logic\_vector) return XInteger
- Numeric\_std Operators
  - For the arithmetic + and – operator:
    - Function + (L: unsigned, R XNatural) return unsigned
    - Function + (L: XNatural, R Unsigned) return unsigned
    - Function – (L: signed, R XInteger) return signed
    - Function – (L: XInteger, R signed) return signed
  - For other arithmetic operators –, \*, /, rem, mod the function overload is analogous.
  - For the relational = operator:

- Function = (L: XNatural, R unsigned) return Xboolean
- Function = (L: unsigned, R XNatural) return Xboolean
- Function = (L: XInteger, R signed) return Xboolean
- Function = (L: signed, R XInteger) return Xboolean

For other relational operators `/=`, `<`, `<=`, `>`, `>=` the function overload is analogous.

- For the shift or rotate operators:
  - Function `sll` (L: signed, Count XInteger) return signed
  - Function `sll` (L: unsigned, Count XInteger) return unsigned
 For other shift or rotate operators `srl`, `rol`, `ror` the function overload is analogous.
- For the shift or rotate functions:
  - Function `shift_left` (L: signed, Count XNatural) return signed
  - Function `shift_left` (L: unsigned, Count XNatural) return unsigned
 For other shift or rotate functions `shift_right()`, `rotate_left()`, `rotate_right()` the function overload is analogous.
- For the conversion functions:
  - Function `to_integer` (unsigned) return XNatural
  - Function `to_integer` (signed) return XInteger

- Function `to_unsigned` (Arg: XNatural; Size: natural) return unsigned
- Function `to_signed` (Arg: XInteger; Size: natural) return signed
- For the resize functions:
  - Function `resize` (Arg: unsigned; new\_size: XNatural) return unsigned
  - Function `resize` (Arg: signed; new\_size: XNatural) return signed

## Supported Attributes

Attributes need to be extended to support the XValue for XInteger, XEnum and XBoolean:

The following supported attributes result in an `x` if the input parameter is `X`:

- POS
- VAL
- SUCC
- PRED
- LEFTOF
- RIGHTOF

The following supported attributes propagate `x`, if the prefix is promoted to X-type:

- LAST\_VALUE
- DRIVING\_VALUE
- DELAYED

In addition, the following attributes are also supported:

- IMAGE: For `X`, it returns the string `<basetype name>-X`. For example, for `Natural`, it returns `INTEGER-X`.
- VALUE: It returns `X` value when the input is the string returned by `image` on `X` input.

## Limitations

The feature has the following limitations:

- The supported two-state objects for promotion to `X`-state types are `enums` with less than 256 values, `integer` and `boolean`.
- `enums` in standard packages, with the exception of `boolean`, are not promoted.
- The `case` statements support only `enums` with less than 256 values. `enums` ranging 256 values or more disable the indeterminate execution of the `case` statements.
- By default, the initial value of two-state is '`LEFT`'.
- The two-state `X` value that crosses an Xprop -> Non-Xprop boundary or an VHDL Xprop -> Verilog boundary takes the integer value representing `X`. For `enums`, the value is 255 and for `integer` it is the customized value.

- The corruption value in NLP is the `X` value, if the object is declared in an Xprop scope. For objects declared in a non-xprop scope, the corruption value is '`LEFT`'.
- 

## X-Propagation Code Examples

X-Propagation (Xprop) changes the simulation semantics of the standard HDL conditional constructs. This section describes the Xprop simulation behavior of the following code examples.

- “If Statement”
  - “Verilog Example”
  - “VHDL Example”
- “Case Statement”
  - “Verilog Example”
  - “VHDL Example”
- “Edge Sensitive Expression”
  - “Verilog Example”
  - “VHDL Example”
- “Latch”
  - “Verilog Example”
  - “VHDL Example”

---

## If Statement

### Verilog Example

The following Verilog code example of an `if` statement represents a simple multiplexer:

```
always@*
  if(s)
    r = a;
  else
    r = b;
```

Table 10-3 Xprop Merge Mode Truth Table for if Statement

| s | a | b | vmerge | tmerge | xmerge |
|---|---|---|--------|--------|--------|
| X | 0 | 0 | 0      | 0      | X      |
| X | 0 | 1 | 1      | X      | X      |
| X | 1 | 0 | 0      | X      | X      |
| X | 1 | 1 | 1      | 1      | X      |

Table 10-3 describes the truth table of the above code example when the control signal `s` of the `if` statement is unknown with a value of `X`.

Under the `vmerge` mode, the standard HDL simulation semantics is used. When the control signal `s` is unknown, the output signal `r` is always assigned the value of the `else` statement. In this case, the value of `r` is the same as signal `b`.

Under the `tmerge` mode, the simulation semantics is modified when the control signal `s` is unknown. In this scenario, two of the case statements are executed, one considering `s=0` and one considering `s=1`. The output values that result from both statements are then

merged. If the values of the output signal  $r$  are same in both conditions, then the merged value of  $r$  is same as in either condition  $s=0$  and  $s=1$ . If the values of the output signal  $r$  are different then the merged value of  $r$  is  $X$ . Thus, if the input signals  $a$  and  $b$  are same, the value of  $r$  is same as  $a$  (or  $b$ ). If  $a$  and  $b$  are different, the value of  $r$  is  $X$ .

In the xmerge mode, when the control signal  $s$  is unknown, the value of the output signal  $r$  is always  $X$ .

## VHDL Example

The following VHDL code example of an `if` statement represents a simple multiplexer:

```
entity MUX is
    Port (s : IN std_logic;
          a : IN std_logic;
          b : IN std_logic;
          r : OUT std_logic );
end MUX;

architecture BEHAVIORAL of MUX is
begin
    process (s, a, b)
    begin
        if (s = '1') then
            r <= a;
        else
            r <= b;
        end if;
    end process;
end BEHAVIORAL;
```

*Table 10-4 Xprop Merge Mode Truth Table for if Statement*

| s | a | b | r in vmerge | r in tmerge | r in xmerge |
|---|---|---|-------------|-------------|-------------|
| X | 0 | 0 | 0           | 0           | X           |
| X | 0 | 1 | 1           | X           | X           |
| X | 1 | 0 | 0           | X           | X           |
| X | 1 | 1 | 1           | 1           | X           |

**Table 10-4** describes the truth table of the above code example when the control signal `s` of the `if` statement is unknown with a value of `X`.

Under the vmerge mode, the standard HDL simulation semantics is used. When the control signal `s` is unknown, the output signal `r` is always assigned the value of the `else` statement. In this case, the value of `r` is the same as signal `b`.

Under the tmerge mode, the simulation semantics is modified when the control signal `s` is unknown. In this scenario, two of the case statements are executed, one considering `s=0` and one considering `s=1`. The output values that result from both statements are then merged. If the values of the output signal `r` are same in both conditions, then the merged value of `r` is same as in either condition `s=0` and `s=1`. If the values of the output signal `r` are different, then the merged value of `r` is `X`. Thus, if the input signals `a` and `b` are same, the value of `r` is same as `a` (or `b`). If `a` and `b` are different, the value of `r` is `X`.

In the xmerge mode, when the control signal `s` is unknown, the value of the output signal `r` is always `X`.

---

## Case Statement

### Verilog Example

The following Verilog code example of a `case` statement represents a simple multiplexer:

```
case (s)
  1'b0: r = a;
  1'b1: r = b;
endcase
```

Table 10-5 Xprop Merge Mode Truth Table for Case Statement

| s | a | b | vmerge | tmerge | xmerge |
|---|---|---|--------|--------|--------|
| X | 0 | 0 | r(t-1) | 0      | X      |
| X | 0 | 1 | r(t-1) | X      | X      |
| X | 1 | 0 | r(t-1) | X      | X      |
| X | 1 | 1 | r(t-1) | 1      | X      |

Table 10-5 describes the truth table of the above code example when the control signal `s` of the `case` statement is unknown with a value of `X`.

In the `vmerge` mode, the standard HDL simulation semantics is used. When the control signal `s` is unknown, the value of the output signal `r` remains the same as before the `case` statement is executed.

In the `tmerge` mode, when the control signal `s` is unknown, the `case` statement is executed twice, once with `s=0` and once with `s=1`. The output values from both conditions are computed and merged. If the values of the output signal `r` are same in both conditions, then the merged and the final value of `r` is same as in both conditions `s=0` and `s=1`. If the values of the output signal `r` are different in both

conditions, then the merged and the final value of `r` is determined by the values of the input signals `a` and `b`. If `a` and `b` are same, the merged and the final value of `r` is the same as `a` and `b`. If `a` and `b` are different, the merged and the final value of `r` is `X`.

In the `xmerge` mode, the value of the output signal `r` is always `X` when the control signal `s` is unknown.

## VHDL Example

The following VHDL code example of a `case` statement represents a simple multiplexer:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity SIMPLECASE is

Port (s : IN std_logic;
      a : IN std_logic;
      b : IN std_logic;
      r : OUT std_logic );
end SIMPLECASE;

architecture BEHAVIORAL of SIMPLECASE is

begin

process (s, a, b)

begin
  case s is
    when '0' => r <= a;
    when '1' => r <= b;
    when others => null;
  end case;
end process;
```

```
end BEHAVIORAL;
```

*Table 10-6 Xprop Merge Mode Truth Table for Case Statement*

| s | a | b | r in vmerge | r in tmerge | r in xmerge |
|---|---|---|-------------|-------------|-------------|
| X | 0 | 0 | r(t-1)      | 0           | X           |
| X | 0 | 1 | r(t-1)      | X           | X           |
| X | 1 | 0 | r(t-1)      | X           | X           |
| X | 1 | 1 | r(t-1)      | 1           | X           |

**Table 10-6** describes the truth table of the above code example when the control signal `s` of the `case` statement is unknown, with a value of `X`.

In the `vmerge` mode, the standard HDL simulation semantics is used. When the control signal `s` is unknown, the value of the output signal `r` remains the same as before the `case` statement is executed.

In the `tmerge` mode, when the control signal `s` is unknown, the `case` statement is executed twice, once with `s=0` and once with `s=1`. The output values from both conditions are computed and merged. If the values of the output signal `r` are same in both conditions, then the merged and the final value of `r` is same as in both conditions `s=0` and `s=1`. If the values of the output signal `r` are different in both conditions, then the merged and the final value of `r` is determined by the values of the input signals `a` and `b`. If `a` and `b` are same, the merged and the final value of `r` is same as `a` and `b`. If `a` and `b` are different, the merged and the final value of `r` is `X`.

In the `xmerge` mode, the value of the output signal `r` is always `X` when the control signal `s` is unknown.

## Unique/Priority Case Variants

SystemVerilog allows `unique` or `priority` directives to be applied to a `case` statement. Synthesis also allows a `case` to be specified as `full_case` or `parallel_case`.

Handling of these case variants requires that indeterminate bits in the case selector can cause more than one branch to match ambiguously, which in turn alter the condition under which `unique`/`priority` assertions are triggered. The ambiguity due to multiple matches is naturally handled by executing all the matching branches and merging the resulting outputs. This behavior is not modified by the triggering of an assertion. These assertions are triggered as follows:

**Priority:** Assertion is triggered when none of the branches matches deterministically. Therefore, zero matches or multiple ambiguous matches (due to a comparison resulting in X) that do not completely cover the Boolean space of the indeterminate bits involved does not constitute a deterministic match and triggers the assertion.

**Unique:** Assertion is triggered when either none of the branches matches deterministically (as for `priority` above), or when more than one branch matches non-ambiguously. Therefore, multiple exact matches trigger the assertion, whereas a single exact match does not trigger the assertion.

To demonstrate the assertion semantics, consider the following two examples:

```
priority case ( sel )
    2'b00 : a = 1;
    2'b01 : a = 2;
    3'b11 : a = 3;
endcase
```

```

unique case ( 1'b1 )
    b + c : d = 1;
    b - c : d = 2;
endcase

```

When the `sel` condition is `2'b0X`, the first example (priority case) ambiguously matches the first and the second branches. Therefore, the output is `a=merge(1, 2)`. Also, since the Boolean space of the indeterminate selector is completely covered by the two matches, no assertion is triggered. If, however, the `sel` condition is `2'b1X`, then a priority assertion is triggered and the output is `a=merge(3, a)`.

If `b` and `c` have the values 1 and 0 respectively, the second example (unique case) unambiguously matches both branches, thereby triggering an unique assertion. Conversely, if `b` or `c` are X, then the example ambiguously matches the same two branches and triggers a (no match) unique assertion.

## Edge Sensitive Expression

### Verilog Example

In standard Verilog RTL simulations, a positive edge transition is triggered for the following value changes in a clocking signal:

```

0 -> 1
0 -> X
0 -> Z
X -> 1
Z -> 1

```

If X is considered as either a 0 or a 1 value, then in the  $0 \rightarrow X$  transition, X may represent a value of 0, which denotes no transition. And, X may represent a value of 1, which denotes a positive edge transition. An Xprop simulation considers both these behaviors and merges the outcome.

The following Verilog code example of an edge sensitive expression represents a simple D flip-flop with an inactive reset:

```
always@ (posedge clk, negedge rst)
    if (!rst)
        q <= 1'b0;
    else
        q <= d;
```

Table 10-7 Xprop Merge Mode Truth Table for D Flip-Flop

| clk               | vmerge | tmerge          | xmerge |
|-------------------|--------|-----------------|--------|
| $0 \rightarrow 1$ | d      | d               | d      |
| $0 \rightarrow X$ | d      | merge(d,q(t-1)) | X      |
| $0 \rightarrow Z$ | d      | merge(d,q(t-1)) | X      |
| $X \rightarrow 1$ | d      | merge(d,q(t-1)) | X      |
| $Z \rightarrow 1$ | d      | merge(d,q(t-1)) | X      |

Table 10-7 describes the truth table of the above code example with different value transitions of the clocking signal `clk`.

In all merge modes, if the clocking signal `clk` is changing from 0 to 1, then a positive edge transition is triggered. The output signal of the D flip-flop `q` is assigned the value of the input signal `d`.

For all other clocking signal transitions, the output signal `q` is assigned the value of the input signal `d` in the `vmerge` mode. The output signal `q` is assigned the value of `X` in the `xmerge` mode. In the

tmerge mode, the current value of the output signal  $q$  is merged with the input signal  $d$ , as described in the tmerge column of the truth table [Table 10-3](#). Then, the merged value is assigned to  $q$ .

## VHDL Example

In standard VHDL RTL simulations, a positive edge transition is triggered for the following value changes in a clocking signal:

```
0 -> 1  
0 -> X  
0 -> Z  
X -> 1  
Z -> 1
```

If X is considered as either a 0 or a 1 value, then in the  $0 \rightarrow X$  transition, X may represent a value of 0, which denotes no transition. And, X may represent a value of 1, which denotes a positive edge transition. An Xprop simulation considers both these behaviors and merges the outcome.

The following VHDL code example of an edge sensitive expression represents a simple D flip-flop with an inactive reset:

```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity FLOP is  
  
generic (  
    width : integer := 4);  
  
port (  
    d      : in  std_logic_vector(width-1 downto 0);  
    rst   : in  std_logic;  
    clk   : in  std_logic;  
    q      : out std_logic_vector(width-1 downto 0));
```

```

end FLOP;

architecture arc of FLOP is

begin
    process (clk, rst)
    begin
        if rst = '1' then
            q <= "0000";
        elsif (rising_edge (clk) ) then
            q <= d;
        end if;
    end process;

end arc;

```

*Table 10-8 Xprop Merge Mode Truth Table for D Flip-Flop*

| clk    | q in vmerge | q in tmerge     | q in xmerge |
|--------|-------------|-----------------|-------------|
| 0 -> 1 | d           | d               | d           |
| 0 -> X | d           | merge(d,q(t-1)) | X           |
| 0 -> Z | d           | merge(d,q(t-1)) | X           |
| X -> 1 | d           | merge(d,q(t-1)) | X           |
| Z -> 1 | d           | merge(d,q(t-1)) | X           |

**Table 10-8** describes the truth table of the above code example with different value transitions of the clocking signal `clk`.

In all merge modes, if the clocking signal `clk` is changing from 0 to 1, then a positive edge transition is triggered. The output signal of the D flip-flop `q` is assigned the value of the input signal `d`.

For all other clocking signal transitions, the output signal `q` is assigned the value of the input signal `d` in the vmerge mode. The output signal `q` is assigned the value of `x` in the xmerge mode. In the

tmerge mode, the current value of the output signal  $q$  is merged with the input signal  $d$ , as described in the tmerge column of the truth table [Table 10-4](#). Then, the merged value is assigned to  $q$ .

**Note:**

The standard VHDL behavior for the above flip-flop is to not store a new value when the clock transitions from either  $0 \rightarrow X$  or  $X \rightarrow 1$ , whereas the standard behavior of the corresponding Verilog model shown above is to store the new value. Xprop semantics, therefore, normalize the behavior of the two languages.

## Latch

### Verilog Example

The following Verilog code example of an `if` statement without an `else` branch represents a simple latch.

```
always@(*)
  if(g)
    q <= d ;
```

Table 10-9 Xprop Merge Mode Truth Table for Latch

| g | d | vmerge | tmerge          | xmerge |
|---|---|--------|-----------------|--------|
| X | 0 | q(t-1) | merge(d,q(t-1)) | X      |
| X | 1 | q(t-1) | merge(d,q(t-1)) | X      |

[Table 10-9](#) describes the truth table of the above code example when the control signal  $g$  of the `if` statement is unknown with a value of  $X$ .

In the vmerge mode, when the control signal  $g$  is unknown, the value of the output signal  $q$  is unchanged. In the tmerge mode, the current value of the output signal  $q$  is merged with the input signal  $d$ , as described in the tmerge column of the truth table [Table 10-3](#). The merged value assigned to  $q$  when the control signal  $g$  is unknown thus depends on the values of both  $q$  and  $d$ .

In the xmerge mode, the value of the output signal  $q$  is always  $x$  when the control signal  $g$  is unknown.

## VHDL Example

The following VHDL code example represents a simple latch:

```
library ieee;
use ieee.std_logic_1164.all;

entity LATCH is

    generic (
        width : integer := 4);

    port (
        d: in std_logic;
        g: in std_logic;
        q: out std_logic);
end LATCH;

architecture arc of LATCH is
begin
    process (g, d)
    begin
        if g = '1' then
            q <= d;
        end if;
    end process;

end arc;
```

*Table 10-10 Xprop Merge Mode Truth Table for Latch*

| <b>g</b> | <b>d</b> | <b>q in vmerge</b> | <b>q in tmerge</b> | <b>q in xmerge</b> |
|----------|----------|--------------------|--------------------|--------------------|
| X        | 0        | q(t-1)             | merge(d,q(t-1))    | X                  |
| X        | 1        | q(t-1)             | merge(d,q(t-1))    | X                  |

[Table 10-10](#) describes the truth table of the above code example when the control signal  $g$  of the `if` statement is unknown with a value of X.

In the vmerge mode, when the control signal  $g$  is unknown, the value of the output signal  $q$  is unchanged. In the tmerge mode, the current value of the output signal  $q$  is merged with the input signal  $d$ , as described in the tmerge column of the truth table [Table 10-4](#). The merged value assigned to  $q$  when the control signal  $g$  is unknown thus depends on the values of both  $q$  and  $d$ .

In the xmerge mode, the value of the output signal  $q$  is always X when the control signal  $g$  is unknown.

## Support for Active Drivers in X-Propagation

Verdi supports active driver tracing functionality for the designs compiled with X-Propagation (using the `-xprop` option). Verdi supports active drivers tracing for X-Propagation (Xprop) in combinational logic, latches, and flip-flops. This section describes the following topics:

- [Combinational Logic](#)
- [Latches](#)

- [Flip-flops](#)
- [Key points to Note](#)

## Combinational Logic

Verdi supports active drivers tracing for Xprop in combinational logic. For tmerge, all drivers that are used for Xprop merge function are shown as active. For xmerge, all drivers from the branches of control structure with unknown condition value are shown as active.

For example, consider the following test case (`comb_logic.sv`) that contains the combinational logic:

*Example 10-1 comb\_logic.sv*

```
module top();
    reg op;
    reg [2:0] out;
    reg [1:0] a,b;
    dut dut1 (op,a,b,out);
    initial begin
        op = 1'b0; a = 2'b11; b = 2'b10;
        #1 op = 1; a = 2'b00;
        #3 a = 1; b = 2'b01; op = 1'bx;
        #2 op = 1; b = 2'b11;
        #1 op = 1'bx;
        #2 op = 0;
    end
endmodule

module dut(input op,a,b, output out);
    logic op;
    logic [1:0] a,b;
    logic [2:0] out;
    always_comb begin
        if (op)
            out = a + b;
        else

```

```
        out = a - b;  
    end  
endmodule
```

Perform the following steps:

1. Compile the `comb_logic.sv` code shown in [Example 10-1](#) as follows:

```
% vcs -sverilog -debug_access+drivers -kdb comb_logic.sv
```

2. Invoke the Verdi GUI using the following command:

```
% simv -gui
```

3. In console pane, use the following command:

```
verdi> fsdbDumpvars 0 "top" +all +trace_process
```

```
verdi> run
```

4. Perform **Trace Drivers**.

Verdi displays active drivers for the combinational logic signals that cause Xprop in the **DriverLoad** Pane, as shown in [Figure 10-1](#).

*Figure 10-1 Viewing Active Drivers of the Combinational Logic Signals*

The screenshot shows the Verdi interface with the 'OneTrace' tab selected. A table displays the active drivers for the signal 'out'. The table has columns for Value, Time, File(Line), and Scope. The data shows three contributors: 'out = a + b;', 'out = a - b;', and 'out'. All three contributors have a value of '2->x' at time 4, originating from 'comb\_logic.sv(19)', 'comb\_logic.sv(22)', and 'comb\_logic.sv(24)' respectively, within the 'top.dut1' scope.

| Signals/Drivers/Loads |       |      |                   |          |  |
|-----------------------|-------|------|-------------------|----------|--|
|                       | Value | Time | File(Line)        | Scope    |  |
| out                   | 2->x  | 4    | comb_logic.sv(19) | top.dut1 |  |
| + out = a + b;        |       | 4    | comb_logic.sv(22) | top.dut1 |  |
| + out = a - b;        |       | 4    | comb_logic.sv(24) | top.dut1 |  |

Verdi displays both active and inactive contributor signals for the driver. Active contributor is a contributor that has a value change and impacts the value of the traced signal. You can expand the active contributor signal, as shown in [Figure 10-2](#), to further trace the origin of X.

*Figure 10-2 Tracing Origin of X*

The screenshot shows the Verdi interface with the 'OneTrace' tab selected. A table displays the tracing path for the signal 'out'. The table has columns for Value, Time, File(Line), and Scope. The data shows the signal 'out' with a value of '2->x' at time 4, originating from 'comb\_logic.sv(19)'. It then traces through 'out = a + b;' and 'out = a - b;' to 'a' and 'b', and finally to 'op'. The 'op' row shows multiple assignments: 'op = 1'b0; a = 2'b11; b = 2'b10'; '#1 op = 1; a = 2'b00'; '#3 a = 1; b = 2'b01; op = 1'bx'; '#2 op = 1; b = 2'b11'; '#1 op = 1'bx'; '#2 op = 0;'. All these assignments are listed under 'top' scope.

| Signals/Drivers/Loads            |       |      |                   |          |  |
|----------------------------------|-------|------|-------------------|----------|--|
|                                  | Value | Time | File(Line)        | Scope    |  |
| out                              | 2->x  | 4    | comb_logic.sv(19) | top.dut1 |  |
| + out = a + b;                   |       | 4    | comb_logic.sv(22) | top.dut1 |  |
| + out = a - b;                   |       | 4    | comb_logic.sv(24) | top.dut1 |  |
| + a                              | 0->1  | 4    | comb_logic.sv(18) | top.dut1 |  |
| + b                              | 2->1  | 4    | comb_logic.sv(18) | top.dut1 |  |
| + op                             | 1->x  | 4    | comb_logic.sv(17) | top.dut1 |  |
| op = 1'b0; a = 2'b11; b = 2'b10; |       | 4    | comb_logic.sv(8)  | top      |  |
| #1 op = 1; a = 2'b00;            |       | 4    | comb_logic.sv(9)  | top      |  |
| #3 a = 1; b = 2'b01; op = 1'bx;  |       | 4    | comb_logic.sv(10) | top      |  |
| #2 op = 1; b = 2'b11;            |       | 4    | comb_logic.sv(11) | top      |  |
| #1 op = 1'bx;                    |       | 4    | comb_logic.sv(12) | top      |  |
| #2 op = 0;                       |       | 4    | comb_logic.sv(13) | top      |  |

#### Note:

In the xmerge mode, all RHS contributors are displayed as inactive. In tmerge mode, the contributors with value change that can impact RHS value, are displayed as active. Otherwise, they are displayed as inactive.

## Latches

Verdi supports active drivers tracing for Xprop in latches. Verdi displays the control signals as contributors for the drivers.

For example, consider the following test case (`latch.sv`) that contains the latch signals:

*Example 10-2 latch.sv*

```
module top;
    reg clk,y,a;
    dut dut1 (clk,a,y);
    initial begin
        clk = 1'b1;
        a = 1'b0;
        #1 clk = 1'b0;
        #1 a = 1'b1;
        #1 clk = 1'bx;
        #1 clk = 1'b1;
    end
endmodule

module dut(input clk,a, output y);
    logic clk,a,y;
    always_latch begin
        if (clk)
            y = a;
    end
endmodule
```

Perform the following steps:

1. Compile the `latch.sv` code shown in [Example 10-2](#) as follows:

```
% vcs -sverilog -debug_access+drivers -kdb latch.sv
```

2. Invoke the Verdi GUI using the following command:

```
% simv -gui
```

3. In console pane, use the following command:

```
verdi> fsdbDumpvars 0 "top" +all +trace_process
```

```
verdi> run
```

4. Perform **Trace Drivers**.

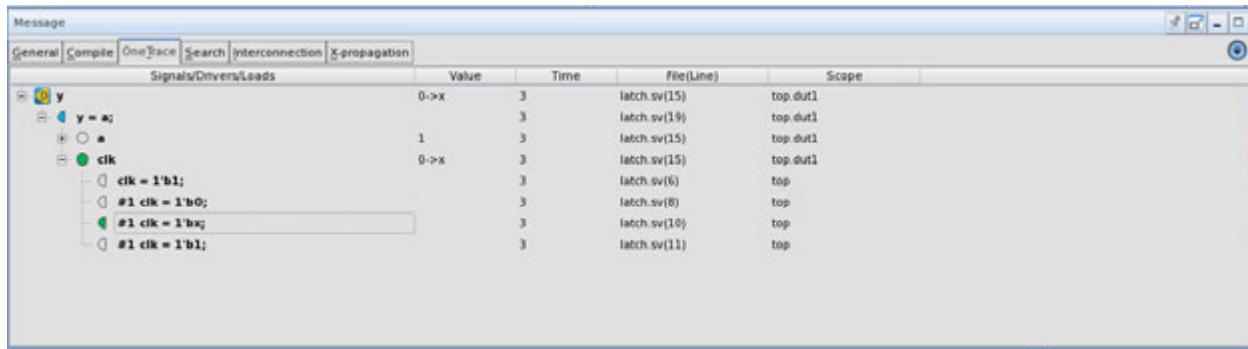
Verdi displays active drivers for the latch signals that cause Xprop in the **DriverLoad** Pane, as shown in [Figure 10-3](#).

*Figure 10-3 Viewing Active Drivers of the Latch Signals*

| Signals/Drivers/Leads |       |      |              |          |
|-----------------------|-------|------|--------------|----------|
|                       | Value | Time | File(Line)   | Scope    |
| y                     | 0->x  | 3    | latch.sv(15) | top.dut1 |
| y = a1                | 3     | 3    | latch.sv(19) | top.dut1 |
| a                     | 1     | 3    | latch.sv(15) | top.dut1 |
| clk                   | 0->x  | 3    | latch.sv(15) | top.dut1 |

Verdi displays both active and inactive contributor signals for the driver. You can expand the active contributor signal, as shown in [Figure 10-4](#), to further trace the origin of X.

*Figure 10-4 Tracing Origin of X*



## Flip-flops

Verdi supports active drivers tracing for Xprop in flip-flops. Verdi also displays signals from the edge-sensitivity list, which has transition to X, as contributors for the drivers.

For example, consider the following test case (`flip_flop.sv`):

*Example 10-3 flip\_flop.sv*

```
module top;
    reg clk;
    reg q,d;
    dut dut1 (clk,d,q);
    always #2 clk = ~clk;
    initial begin
        clk = 1'b0;
        d = 1'b1;
        #4 d = 1'b0;
        #1 clk = 1'bx;
        #1 d = 1'b0;
        #1 clk = 1'b1;
        #1 d = 1'b1;
        #10 $finish();
    end
endmodule

module dut(input clk,d,output q);
```

```
logic clk,q,d;
always_ff@(posedge clk) q <= d;
endmodule
```

Perform the following steps:

1. Compile the `flip_flop.sv` code shown in [Example 10-3](#) as follows:

```
% vcs -sverilog -debug_access+drivers -kdb flip_flop.sv
```

2. Invoke the Verdi GUI using the following command:

```
% simv -gui
```

3. In console pane, use the following command:

```
verdi> fsdbDumpvars 0 "top" +all +trace_process
```

```
verdi> run
```

4. Perform **Trace Drivers**.

Verdi displays active drivers for the flip-flops that cause Xprop in the **DriverLoad** Pane, as shown in [Figure 10-5](#).

*Figure 10-5 Viewing Active Drivers of Flip-Flops*

The screenshot shows the Verdi interface with the X-propagation tab selected. A table lists the active drivers for signal `q`. The columns are: Signals/Drivers/Loads, Value, Time, File(Line), and Scope. There are two entries:

| Signals/Drivers/Loads                           | Value | Time | File(Line)       | Scope    |
|-------------------------------------------------|-------|------|------------------|----------|
| <code>q</code>                                  | 1->x  | 5    | flip_flop.sv(20) | top.dut1 |
| <code>always_ff@(posedge clk) q &lt;= d;</code> |       | 5    | flip_flop.sv(21) | top.dut1 |

Verdi does not display signals from always block sensitivity list as contributor except in this case, where a signal from the process sensitivity list (for example, clock) has an unclean edge and causes Xprop. You can expand the active contributor signal, as shown in [Figure 10-6](#), to further trace the origin of X.

*Figure 10-6 Tracing Origin of X*

The screenshot shows the Verdi interface with the OneTrace tab selected. The table now includes expanded information for the `clk` and `q` signals. The expanded view for `clk` shows its assignment and the sensitivity list of the always block that drives it. The expanded view for `q` shows its assignment and the sensitivity list of the always block that drives it, along with the values of the driving signals `clk` and `d`.

| Signals/Drivers/Loads                           | Value | Time | File(Line)       | Scope    |
|-------------------------------------------------|-------|------|------------------|----------|
| <code>clk</code>                                | 0->x  | 5    | flip_flop.sv(20) | top.dut1 |
| <code>always #2 clk = ~clk;</code>              |       | 5    | flip_flop.sv(5)  | top      |
| <code>clk = 1'b0;</code>                        |       | 5    | flip_flop.sv(7)  | top      |
| <code>#1 clk = 1'bX;</code>                     |       | 5    | flip_flop.sv(10) | top      |
| <code>#1 clk = 1'b1;</code>                     |       | 5    | flip_flop.sv(12) | top      |
| <code>q</code>                                  | 1->x  | 5    | flip_flop.sv(3)  | top      |
| <code>always_ff@(posedge clk) q &lt;= d;</code> |       | 5    | flip_flop.sv(21) | top.dut1 |
| <code>+ clk</code>                              | 0->x  | 5    | flip_flop.sv(20) | top.dut1 |
| <code>+ d</code>                                | 1->0  | 4    | flip_flop.sv(20) | top.dut1 |

## Key points to Note

- The change of the merge function at runtime is ignored. That is, analysis is done using the merge function specified at compile time.

---

## Support for Ternary Operator

The behavior of the ternary operator may be more optimistic than some of its possible gate-level implementations. Therefore, when the `xmerge` function is in effect, the behavior of the ternary operator is modified to be more pessimistic.

Specifically, when `xmerge` is in effect, `(cond ? a : b)` becomes:

```
(|cond === 'X) ? 'X : (cond ? a : b)
```

When the `tmerge` function is in effect, the ternary operator remains unchanged.

---

## Limitations

1. X-Propagation is not supported with the following VCS feature:

- `+vcs+initreg` option described in the section “[Initializing Verilog Variables, Registers, and Memories](#)” .

2. X-Propagation support for the following VCS features have limitations:

- Code coverage in Verilog

Code coverage does not exclude branches that are executed ambiguously, that is, under control of an X. Therefore, code coverage results may appear to overestimate coverage when Xprop is enabled.

- A loop with non-constant bounds

The loops with variable bounds are not synthesizable. Xprop requires loop bounds to be constant or constant expressions for the loops to be instrumented. Along with the use of explicit variables of 4 state types, functions returning non-constant values are also treated as non-constant expressions.

- VHDL specific limitations

Delays are not supported in VHDL code.

3. Limitations that disable Xprop at the call site and setup the parent chain. These limitations at the call site do not prevent instrumentation of the procedure body. If a call is determined as non-xpropable, all the internal structural statements are also disabled for Xprop.

- `wait()` Statements

The `wait()` statement cannot appear in the Xprop region because it is not possible to merge delayed values and normal values

- Signal output parameters

Signal output parameters indicate a signal assignment within the procedure. Merging and signal final assignment are completed at the end of the last conditional statement within the sub-program. As merging is not completed at a call site, the call site cannot be within an Xprop region.

- Up-level references and external names

The references can be made to objects that are undergoing merging in different contexts. It is not possible to know which call site context the merging should be utilized from within the sub-program. Thus, the call site to the sub-program with up-level or external references must exist in a non-xprop region, where no merging is possible.

- Side Effects

It is not possible to merge side effects. Thus, the call must occur in a non-xprop region.

- Un-instrumented sub-program with output arguments

A sub-program that is not instrumented cannot be executed in a non-deterministic method. So, its call site should not be in an Xprop region.

#### 4. Limitations that disables Xprop within the sub-program body:

- Return statements

Return statements execute immediately irrespective of whether the branch is executed in a deterministic way or not.

Note:

This limitation only applies to Verilog methods. For VHDL, return statement is supported when the switch `-xprop=floopctrl` is specified.

- Dynamic for loop bounds

Unlike in a process where the loop bounds as the loop cannot be dynamic, the rule is relaxed for sub-programs because unconstrained types are statically constrained at the call site. The loop can then iterate over the static bounds, however, the bounds appear dynamic from within a sub-program.

- Edges

Clock edges are not detected within a sub-program. Flip-flops in sub-programs are not inferred. A clock edge in a sub-program is treated as a normal condition within the sub-programs.

- Unlike in a process, `translate_on` or `translate_off` around a sub-program does not suppress Xprop.
  - The `xprop_on` or `xprop_off` pragma does not suppress Xprop in a sub-program.
5. For composite types, only `integral`, `std_logic`, `enum`, `boolean` and `record` types are supported.

Note:

For Integer, enum and Boolean types, instrumentation only happens if the `synopsys_sim.setup` file contains an entry `XPROP_ANALYSIS_CHECK=true`.

6. For memories, MDA of `std_logic` base types and vector of vectors are supported. The MDA of `std_logic` include `std_logic`, `std_ulogic`, `std_logic_vector`, and `std_ulogic_vector`.

Note:

For Integer, enum and boolean types, instrumentation only happens if the `synopsys_sim.setup` file contains an entry `XPROP_ANALYSIS_CHECK=true`.

7. Other `non-std_[u]logic` based multidimensional arrays or arrays of arrays are not supported.
8. Record types that contain field of unsupported types are not supported.
9. The `$set_x_prop` Verilog system task or VHDL `set_x_prop` does not trigger process blocks to re-evaluate. Therefore, unless there is a change in the input data, output does not reflect the behavior of the new merge scheme.
10. For unique/priority case or case with default branch, instrumentation is not done if bit-length of non-constant case expressions exceeds 8 bits.

# 11

## Gate-Level Simulation

---

This chapter contains the following sections:

- “SDF Annotation”
- “Precompiling an SDF File”
- “SDF Configuration File”
- “Delays and Timing”
- “Using the Configuration File to Disable Timing”
- “Using the timopt Timing Optimizer”
- “Using Scan Simulation Optimizer”
- “Negative Timing Checks”
- “Using VITAL Models and Netlists”

- “Support for Identifying Non-Annotated Timing Arc and Timing Check Statements”
- 

## SDF Annotation

The Open Verilog International Standard Delay File (OVI SDF) specification provides a standard ASCII file format for representing and applying delay information. VCS supports OVI versions 1.0, 1.1, 2.0, 2.1, and 3.0 of this specification.

In the SDF format, a tool can specify intrinsic delays, interconnect delays, port delays, timing checks, timing constraints, and pulse control (PATHPULSE).

When VCS reads an SDF file, it “back-annotates” delay values to the design, that is, it adds delay values or changes the delay values specified in the source files.

Following are ways to back-annotate the delays specified in the SDF file:

- “Using the Unified SDF Feature”
- “Using the \$sdf\_annotate System Task”
- “Using the -xlm Option for SDF Retain, Gate Pulse Propagation, and Gate Pulse Detection Warning”
- “Enhancing SDF Annotation to Support Nets Through SPICE”

---

## Using the Unified SDF Feature

The Unified SDF feature allows you to back-annotate SDF delays using the following compilation/elaboration option:

```
-sdf min|typ|max:instance_name:file.sdf
```

For two-step flow:

### Compilation

```
% vcs -sdf min|typ|max:instance_name:file.sdf \  
[compile_options]
```

### Simulation

```
% simv [run_options]
```

For two-step flow:

### Analysis

```
% vlogan [vlogan_options] file2.v file3.v  
% vhdlan [vhdlan_options] file4.vhd file5.vhd
```

### Note:

The VHDL bottommost entity first, then move up in order.

### Elaboration

```
% vcs -sdf min|typ|max:instance_name:file.sdf \  
[elab_options] top_cfg/entity/module
```

### Simulation

```
% simv [run_options]
```

For more information on specifying delays and SDF files, see  
[“Options for Specifying Delays and SDF Files”](#)

For an example, see the \$VCS\_HOME/doc/examples/timing/mx\_unified\_sdf directory.

---

## Using the \$sdf\_annotate System Task

You can use the \$sdf\_annotate system task to back-annotate 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 an SDF file.

*module\_instance*

Specifies the scope where back-annotation starts. The default is the scope of the module instance that calls \$sdf\_annotate.

*"sdf\_configfile"*

Specifies the SDF configuration file. For more information on the SDF configuration file, see the [“SDF Configuration File” section](#).

*"sdf\_logfile"*

Specifies an SDF log file to which VCS sends error messages and warnings. By default, VCS displays no more than ten warnings and ten error messages about back-annotation and writes no more than that in the log file you specify with the `-l` option. However, if you specify the SDF log file with this argument, the SDF log file receives all messages about back-annotation. You can also use the `+sdfverbose` runtime option to enable the display of all back-annotation messages.

`"mtm_spec"`

Specifies which delay values of `min:typ:max` triplets VCS back-annotates. Its possible values are "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. Its possible values are "FROM\_TYPICAL", "FROM\_MIMINUM", "FROM\_MAXIMUM", and "FROM\_MTM" (default).

The usage model to simulate a design using `$sdf_annotate` is same as the basic usage model as shown below:

For two-step flow:

## Compilation

```
% vcs [elab_options] top_cfg/entity/module
```

## **Simulation**

```
% simv [run_options]
```

For three-step flow:

## **Analysis**

```
% vlogan [vlogan_options] file2.v file3.v  
% vhdlan [vhdlan_options] file4.vhd file5.vhd
```

### **Note:**

The VHDL bottommost entity first, then move up in order.

## **Elaboration**

```
% vcs [elab_options] top_cfg/entity/module
```

## **Simulation**

```
% simv [run_options]
```

For more details, see “[Options for Specifying Delays and SDF Files](#)”.

---

## **Using the -xIrm Option for SDF Retain, Gate Pulse Propagation, and Gate Pulse Detection Warning**

The following sections explain how to use the new features added under the `-xIrm` option:

- [“Using the Optimistic Mode in SDF”](#)
- [“Using Gate Pulse Propagation”](#)
- [“Generating Warnings During Gate Pulses”](#)

## Using the Optimistic Mode in SDF

Currently, when you use the `-sdfretain` option, SDF retain is visible whenever there is a change in related inputs.

When you specify the `-sdfretain` option with `-xlrn alt_retain`, SDF retain is visible only when there is a change in the output. This new behavior is called optimistic mode. For example, consider the following Verilog code:

```
and u(qout,d1,d2);  
  
specify  
    (d1 => qout) = (10); //RETAIN (6)  
    (d2 => qout) = (10);  
endspecify
```

The corresponding SDF entry is:

```
(IOPATH d1 qout (RETAIN (6))(10) )  
(IOPATH d2 qout (10) )
```

The default output for the above example is:

```
time= 10 , d1=0,d2=0, qout=0  
time= 100 , d1=1,d2=0, qout=0  
time= 106 , d1=1,d2=0, qout=x // since input d1 change at /  
//100, VCS propagate "x" to qout  
time= 110 , d1=1,d2=0, qout=0  
= 200 , d1=0,d2=0, qout=0  
time= 206 , d1=0,d2=0, qout=x // since input d1 change at  
//200, VCS propagate "x" to qout  
time= 210 , d1=0,d2=0, qout=0  
time= 300 , d1=0,d2=1, qout=0  
time= 400 , d1=1,d2=1, qout=0  
time= 406 , d1=1,d2=1, qout=x  
time= 410 , d1=1,d2=1, qout=1
```

The output using the `-x1rm alt_retain` option (new behavior) is as follows:

```
time= 10 , d1=0,d2=0, qout=0
time= 100 , d1=1,d2=0, qout=0 // since there is no logic
                                //change on "qout", no retain "x" seen
time= 200 , d1=0,d2=0, qout=0
time= 300 , d1=0,d2=1, qout=0
time= 400 , d1=1,d2=1, qout=0
time= 406 , d1=1,d2=1, qout=x // since there is logic change
                                //on "qout", retain "x" propagated
time= 410 , d1=1,d2=1, qout=1
```

## Using Gate Pulse Propagation

Using the `-x1rm gd_pulseprop` option, VCS always propagates a gate pulse, even when the pulse width is equal to the gate delay. For example, consider the following Verilog code:

```
module dut(qout,dinA,dinB);
    output qout;
    input dinA;
    input dinB;

    xor #10 inst(qout,dinA,dinB);

endmodule
```

Under the `-x1rm gd_pulseprop` option, if the pulse width on a gate is equal to the gate delay, VCS always propagates the pulse as shown below:

```
0 qout=x, dinA=1 dinB=1
10 qout=0, dinA=0 dinB=1
20 qout=1, dinA=0 dinB=0
30 qout=0, dinA=0 dinB=1
40 qout=1, dinA=0 dinB=0
50 qout=0, dinA=0 dinB=0
```

## Generating Warnings During Gate Pulses

Using the `-x1rm gd_pulsegarn` option, VCS generates a warning when it detects that the width of a pulse is identical to the gate delay. For example, consider the following Verilog code:

```
module dut(qout,dinA,dinB);
output qout;
input dinA;
input dinB;

xor #10 inst(qout,dinA,dinB);
endmodule
```

Under the `-x1rm gd_pulsegarn` option, if the pulse width on a gate is equal to the gate delay, VCS generates the following warning message:

```
0 qout=x, dinA=1 dinB=1
```

```
Warning-[PWIWGD] Pulse Width Identical With Gate Delay
verilogfile.v, 42
top.mid_inst.dut_inst
At time 10, pulse width identical with gate delay "10" is
detected
```

```
10 qout=0, dinA=0 dinB=1
```

```
20 qout=1, dinA=0 dinB=0
```

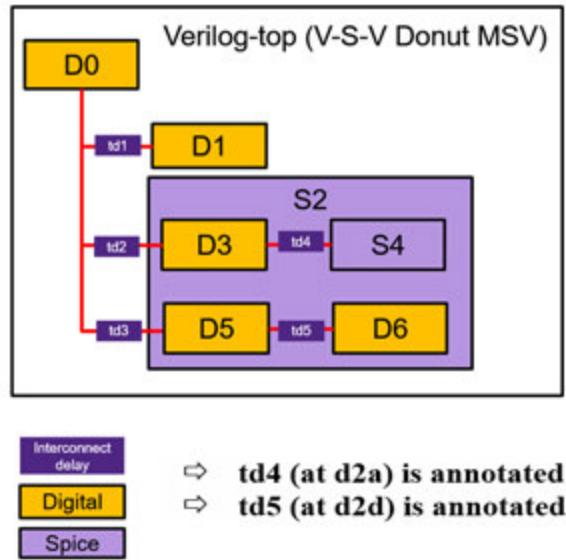
---

## Enhancing SDF Annotation to Support Nets Through SPICE

VCS supports SDF in Mixed Signal Verification (MSV) that covers the INTERCONNECT delay between digital port and analog port. VCS also annotates INTERCONNECT delay when SPICE region is

parent (such as a V-S-V donut configuration). Therefore, you can use the compile time option `+msvsdfext`, to enable SDF annotation with MSV to include digital nets passing through SPICE.

*Figure 11-1 V-S-V Donut Configuration*



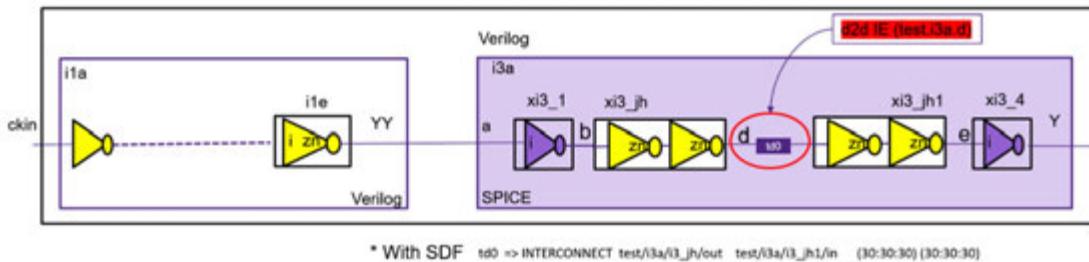
## Use Model

VCS annotates SDF delays in the input, inout and output ports using the `+msvsdfext` option for the following scenarios:

- d2d under V-S-V donut configuration

[Figure 11-2](#) shows a d2d connection where `td0` is the interconnect delay. SDF is annotated in input, output scenarios.

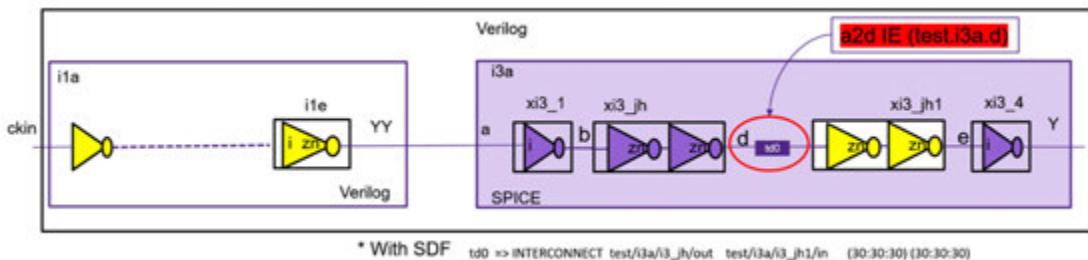
*Figure 11-2 d2d under V-S-V Donut Configuration*



- a2d under V-S-V donut configuration

[Figure 11-3](#) shows a d2d connection where  $td0$  is the interconnect delay. SDF is annotated in input, output scenarios.

*Figure 11-3 a2d under V-S-V Donut Configuration*



## Limitations

The following are the limitations of the current implementation of SDF annotation through SPICE:

- Supports only INTERCONNECT delay.
- VCS identifies the d2a and inout scenario under V-S-V donut configuration and generates a warning message.

- Shows limitation of multiple drivers that are running at the same time. This is a design issue and the designer must resolve this issue.
- Does not support if the destination signal is connected to tristate.
- Behavior of other delay types such as “PORT” (`PORT test/i2a/a (10)`) is not determined.
- Does not support VHDL SPICE boundary in IC delays.
- Does not support real type and user defined type in SPICE boundary with IC delays.

## Precompiling an SDF File

Whenever you compile your design, if your design back-annotates 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. In addition, VCS parses even when you are using incremental compilation and the parts of the design back-annotated by the SDF file are unchanged.

VCS can parse the precompiled SDF file much faster than it can parse the ASCII-text SDF file. Therefore, for large SDF files, it is good 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, 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, has no effect when the precompiled version is newer than the ASCII-text SDF file. However, it creates 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 `$sdf_annotation` system task. You can also specify the precompiled SDF file in the `$sdf_annotation` system task. The `+csdf+precompile` option also supports zipped SDFs.

## **Precompiling SDF Without Compiling Design Files**

You can also precompile the SDF files without compiling the entire set of design files. For this, use the following command option:

```
+csdf+precomp+file+<sdf file>
```

The following is the use model for this option:

```
vcs +csdf+precomp+file+<sdf file>
```

For example:

```
vcs +csdf+precomp+file+test1.sdf
```

### **Writing Precompiled SDF to a Different Directory**

You can write the precompiled SDF to a different directory. To do this, use the following command option:

```
+csdf+precomp+dir+PRE_COMP_SDF/
```

The following is the use model for this option:

```
vcs +csdf+precomp+file+<sdf file>
+csdf+precomp+dir+<DIR>
```

For example:

```
mkdir PRE_COMP_SDF
vcs +csdf+precomp+file+./test.sdf
+csdf+precomp+dir+PRE_COMP_SDF/
```

Note: The precompiled SDF file is generated in the `PRE_COMP_SDF` directory.

---

## SDF Configuration File

You can use the configuration file to control the following on a module-type basis as well as on a global basis:

- The `min:typ:max` selection
- Scaling
- The Module-Input-Port-Delay (MIPD) 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 `INSTANCE` to a specific `IOPATH` or `TIMINGCHECK` in its sub hierarchy 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 11-1](#).

*Table 11-1 VCS Simulation Delay Objects/Constructs*

|               | SDF Constructs  | VCS Simulation Object                |
|---------------|-----------------|--------------------------------------|
| <b>Delays</b> |                 |                                      |
|               | PATHPULSE       | module path pulse delay              |
|               | GLOBALPATHPULSE | module path pulse reject/error delay |
|               | IOPATH          | module path delay                    |
|               | PORT            | module input port delay              |

*Table 11-1 VCS Simulation Delay Objects/Constructs*

| SDF Constructs       | VCS Simulation Object                                                                                 |
|----------------------|-------------------------------------------------------------------------------------------------------|
| INTERCONNECT         | module input port delay or,<br>intermodule path delay when<br>+multisource_int_delays is<br>specified |
| NETDELAY             | module input port delay                                                                               |
| DEVICE               | primitive and module path delay                                                                       |
| <b>Timing-Checks</b> |                                                                                                       |
| SETUP                | \$setup timing-check limit                                                                            |
| HOLD                 | \$hold timing-check limit                                                                             |
| SETUPHOLD            | \$setup and \$hold timing-check<br>limit                                                              |
| RECOVERY             | \$recovery timing-check limit                                                                         |
| SKEW                 | \$skew timing-check limit                                                                             |
| WIDTH                | \$width timing-check limit                                                                            |
| PERIOD               | \$period timing-check limit                                                                           |
| NOCHANGE             | ignored                                                                                               |
| PATHCONSTRAINT       | ignored                                                                                               |
| SUM                  | ignored                                                                                               |
| DIFF                 | ignored                                                                                               |
| SKEWCONSTRAINT       | ignored                                                                                               |

## SDF Configuration File Commands

This section explains the following commands used in SDF configuration files with their syntax and examples:

- [The INTERCONNECT\\_MIPD Command](#)
- [The MTM Command](#)

- [The SCALE Commands](#)

## The INTERCONNECT\_MIPD Command

The `INTERCONNECT_MIPD` command selects INTERCONNECT delays in the SDF file that are mapped to MIPDs in VCS . You can specify one of the following to VCS :

MINIMUM

Selects the shortest delay from all INTERCONNECT delay value entries in the SDF file to MIPD for the input or the inout port instance. The delay specifies the connection to the input or inout port.

MAXIMUM

Selects the longest delay from all INTERCONNECT delay value entries in the SDF file to MIPD for the input or the inout port instance. The delay specifies the connection to the input or inout port.

AVERAGE

Selects the average delay of all INTERCONNECT delay value entries in the SDF file to MIPD for the input or the inout port instance. The delay specifies the connection to the input or inout port.

LAST

Selects the delay in the last INTERCONNECT delay value entries in the SDF file to MIPD for the input or the inout port instance. The delay specifies the connection to the input or inout port.

The default value of the `INTERCONNECT_MIPD` command is MAXIMUM and its syntax is as follows:

```
INTERCONNECT_MIPD = MINIMUM | MAXIMUM | AVERAGE | LAST;
```

For example:

```
INTERCONNECT_MIPD=LAST;
```

## The MTM Command

The command annotates the minimum, typical, or maximum delay value. You can specify 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 the MTM command is TOOL\_CONTROL and its syntax is as follows:

```
MTM = MINIMUM | TYPICAL | MAXIMUM | TOOL_CONTROL;
```

For example:

```
MTM=MAXIMUM;
```

## The SCALE Commands

There are the following two types of SCALE commands:

- **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 back-annotated to the Verilog tool. Each multiplier represents 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 back-annotated 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\_MM**

Scales directly from the minimum, typical, and maximum timing specifications in the SDF file.

The syntax of **SCALE\_FACTORS** and **SCALE\_TYPE** is as follows:

```
SCALE_FACTORS = number : number : number;
SCALE_TYPE = FROM_MINIMUM | FROM_TYPICAL | FROM_MAXIMUM |  
FROM_MM;
```

For example:

```
SCALE_FACTORS=100:0:9;  
SCALE_TYPE=FROM_MM;  
SCALE_FACTORS=1.1:2.1:3.1;  
SCALE_TYPE=FROM_MINIMUM;
```

## An SDF Example With Configuration File

The following example uses the VCS SDF configuration file, `sdf.cfg`:

The content of the `test.v` file is as follows:

```
// 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

```

The content of the `test.sdf` file is as follows:

```

SDF File: (DELAYFILE
(SDFVERSION "3.0")
(DESIGN "sdftest")
(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) (25:26:2
7)))
(COND (i2==1) (IOPATH i2 o2
(10:11:12) (13:14:15) (16:17:18) (19:20:21) (22:23:24) (25:26:2
7)))
)))
)
)
)

```

The content of the `sdf.cfg` file is as follows:

```

PATHPULSE=IGNORE;
INTERCONNECT_MIPD=MAXIMUM;
MTM=TOOL_CONTROL;
SCALE_FACTORS=100:0:9;
SCALE_TYPE=FROM_MTM;
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); }
}
```

## Delays and Timing

This section describes the following topics:

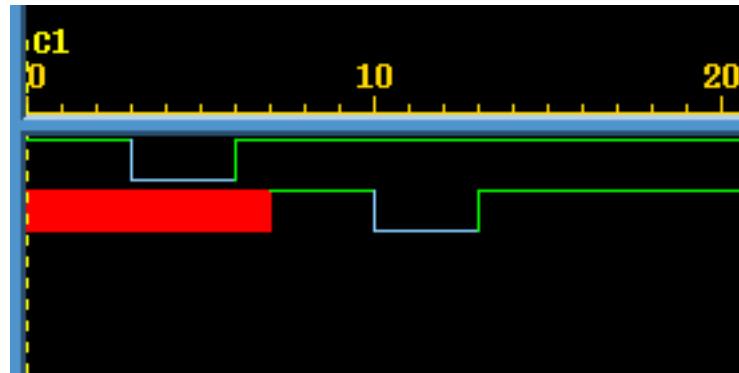
- “Transport and Inertial Delays”
  - “Pulse Control”
  - “Specifying the Delay Mode”
  - “Support for Delayed Annotation During Simultaneous Switching on Inputs”
- 

## 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. For example, [Figure 11-4](#) 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. For an explanation on how this is done, see [“Enabling Transport Delays”](#) and [“Pulse Control”](#).

*Figure 11-4 Transport Delay Waveforms*

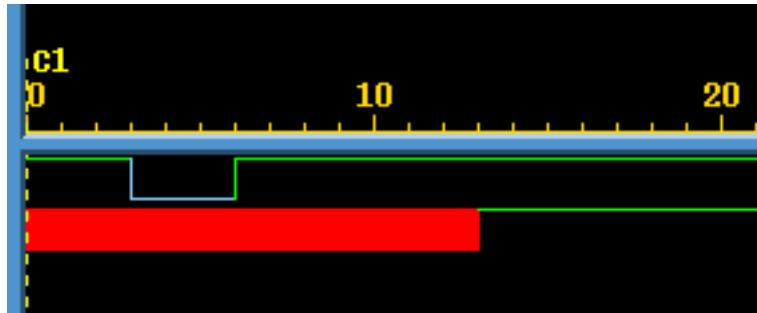


At time 0, a pulse three-time unit wide begins on the input port. This pulse is narrower than the module path delay of seven-time units, but this pulse propagates through the module and appears on the output port after seven-time units. Similarly, another narrow pulse begins on the input port at time 3 and it also appears on the output port seven-time units later.

You can apply transport delays on all module path delays and all SDF INTERCONNECT delays back-annotated to a net from an SDF file. For more information on SDF back-annotation, see “[SDF Annotation](#)”.

Inertial delays, in contrast, filter out all pulses that are narrower than the delay. [Figure 11-5](#) shows the waveforms for the same input and output ports when you have not enabled transport delays for module path delays.

Figure 11-5 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. Also, the pulse that begins at time 3 does not propagate. Note that the wide pulse that begins at time 6 does propagate to the output port.

Gates, switches, MIPDs, and continuous assignments only have inertial delays, which are the default type of delay for module path delays and INTERCONNECT delays back-annotated from an SDF file to a net.

## The Inertial Delay Implementation

The inertial delay implementation is the same for primitives [gates, switches, and User-Defined Primitives (UDP)], continuous assignments, MIPDs, module path delays, and INTERCONNECT delays back-annotated from an SDF file to a net. For more details on SDF back-annotation, see “[SDF Annotation](#)”. There is also a third implementation that is for module path and INTERCONNECT delays and pulse control, see “[Pulse Control](#)”.

The implementation of inertial delays is as follows:

Consider an event that is scheduled by the leading edge of a pulse and is either scheduled for a later simulation time or has not yet occurred. This event is replaced by the event that is scheduled by the trailing edge at the end of the specified delay and at a new simulation time. All narrow pulses are filtered out.

Note:

VCS enables more complex and flexible pulse control processing when you include the `+pulse_e/number` and `+pulse_r/number` options. For details on these options, see “[Pulse Control](#)”.

## Enabling Transport Delays

Transport delays are not the default delays. 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. For details on these options, see “[Pulse Control](#)”.

You can specify transport delays on a net to which you back-annotate SDF INTERCONNECT delays with the `+transport_int_delays` compile-time option. For this option to work, you must also include the `+pulse_int_e/number` and `+pulse_int_r/number` compile-time options. For details on these options, see “[Pulse Control](#)”.

The `+pulse_e/number`, `+pulse_r/number`, `+pulse_int_e/number`, and `+pulse_int_r/number` options define specific thresholds for pulse width, which allow you to tell VCS to filter out only some of the pulses and let the other pulses propagate. For details on these options, see “[Pulse Control](#)”.

---

## Pulse Control

As discussed in previous sections, for pulses narrower than a module path or INTERCONNECT delay, you have two options. One is to filter all pulses by using the default inertial delay. Another is to allow all pulses to propagate by specifying transport delays. VCS also provides a third option - pulse control. Pulse control allows you to do the following:

- Allow pulses that are slightly narrower than the delay to propagate.
- 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.

For module path delays, specify pulse control with the `+pulse_e/number` and `+pulse_r/number` compile-time options. For INTERCONNECT delays, specify pulse control with the `+pulse_int_e/number` and `+pulse_int_r/number` compile-time options.

The `number` argument of the `+pulse_e/number` option 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 `number` argument of the `+pulse_int_e/number` option specifies a percentage of the INTERCONNECT delay. VCS replaces pulses whose widths are narrower than the specified

percentage of the delay with an `X` value pulse on the inout or output port instance that is the load of the net to which you back-annotated the INTERCONNECT delay. It also displays a warning message.

The `number` argument of the `+pulse_r/number` option 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 and VCS ignores these pulses.

Similarly, the `number` argument of the `+pulse_int_r/number` option 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](#)”) or inertial delays (see “[Pulse Control With Inertial Delays](#)”).

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 propagate to the output. This is the default behavior.

For more details, see “[Specifying Pulse on Event or Detect Behavior](#)”.

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. However, you can disable the display of the warning message with the `+no_pulse_msg` runtime option.

---

## Pulse Control With Transport Delays

You can 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.

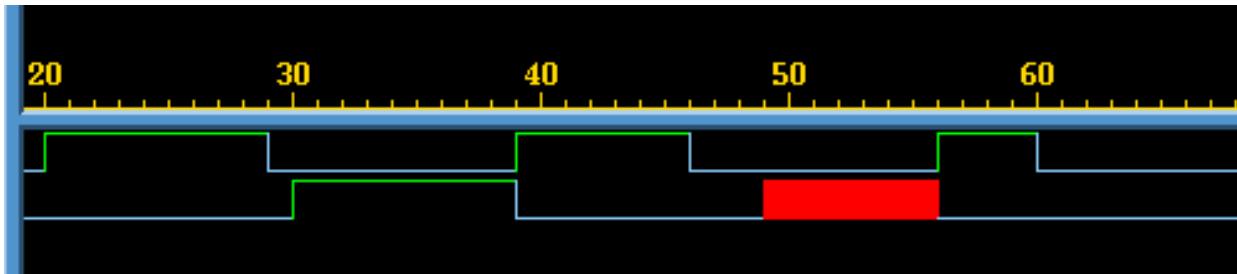
You can specify transport delays for INTERCONNECT delays on nets with the `+transport_int_delays`, `+pulse_int_e/number`, and `+pulse_int_r/number` options. You must include all three of these options.

If you want VCS to propagate all pulses, no matter how narrow, specify a 0 percentage. For example, if you want VCS to replace pulses that are narrower than 80% of the delay with an `x` value pulse (and display a warning message) and filter out pulses that are narrower than 50% of the delay, enter the `+pulse_e/80` and `+pulse_r/50` or `+pulse_int_e/80` and `+pulse_int_r/50` compile-time options.

[Figure 11-6](#) shows the waveforms for input and output ports for an instance of a module that models a buffer with a ten-time unit module path delay. For this, the `vcs` command contains the following compile-time options:

```
+transport_path_delays +pulse_e/80 +pulse_r/50
```

*Figure 11-6 Pulse Control With Transport Delays*



In the example illustrated in [Figure 11-6](#), 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. This is because transport delays are enabled and the nine-time unit 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. However, VCS has replaced it with an X value pulse because seven-time units is less than 80% of the module path delay. VCS issues a warning message in this case.

7. At time 56, the input port toggles to 1 ending a ten-time unit wide value 0 pulse. Also, at time 56, the output port toggles to 0. The ten-time unit wide value 0 pulse that began at time 46 on the input port is propagating to the output port.
8. At time 60, the input port toggles to 0 ending a four-time unit wide value 1 pulse. Four-time units is less than 50% of the module path delay. Therefore, VCS filters out this pulse and no indication of it appears on the output port.

## Pulse Control With Inertial Delays

You can enter the `+pulse_e/number` and `+pulse_r/number` or `+pulse_int_e/number` and `+pulse_int_r/number` options without the `+transport_path_delays` or `+transport_int_delays` options. If you do this, you are specifying pulse control for inertial delays on module path delays and INTERCONNECT delays.

There is a special implementation of inertial delays with pulse control for module path delays and INTERCONNECT delays. In this implementation, value changes on the input can schedule two events on the output.

The first of these two scheduled events always causes a change on the output. The type of value changes on the output is determined by the following:

- The first event is scheduled by the leading edge of a pulse whose width is equal to or wider than the percentage specified by the `+pulse_e/number` option. Then, the value change on the input propagates to the output.

- The pulse is not wider than the percentage specified by the `+pulse_e/number` option, but is wider than the percentage specified by the `+pulse_r/number` option. Then, the value change is replaced by an `X` value.
- The pulse is not wider than the percentage specified by the `+pulse_r/number` option and the pulse is filtered out.

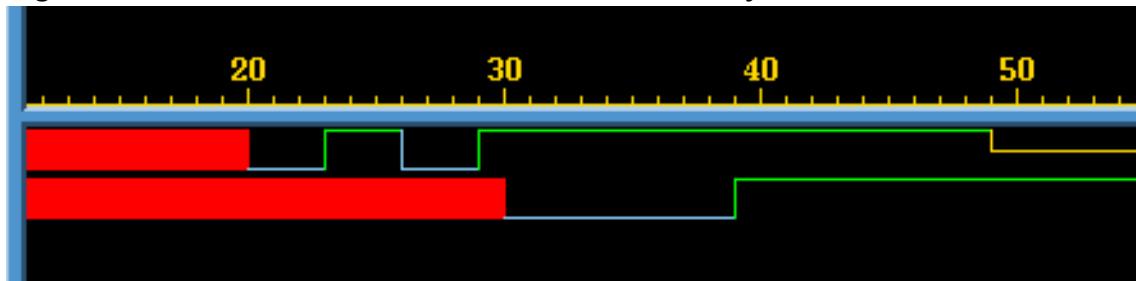
The second scheduled event is always tentative. If another event occurs on the input before the first event occurs on the output, that additional event on the input cancels the second scheduled event and schedules a new second event.

[Figure 11-7](#) shows the waveforms for 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 contains the following compile-time options:

```
+pulse_e/0 +pulse_r/0
```

In this example, specifying 0 percentage means that the trailing edge of all pulses can change the second scheduled event on the output. Specifying 0 does not mean that all pulses propagate to the output because this implementation has its own way of filtering out short pulses.

*Figure 11-7 Pulse Control With Inertial Delays*



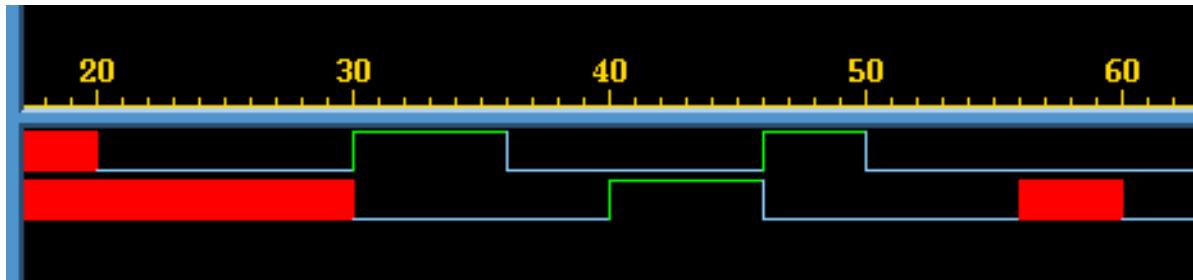
In the example illustrated in [Figure 11-7](#), the following occurs:

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 and it occurs.
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 is not 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.

[Figure 11-8](#) shows the waveforms for input and output ports for an instance of the same module with a ten-time unit module path delay. The `vcs` command contains the following compile-time options:

```
+pulse_e/60 +pulse_r/40
```

*Figure 11-8 Pulse Control With Inertial Delays and Narrow Pulses*



In the example illustrated in Figure 11-8, 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 does not 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. However, it is equal to the width specified with the `+pulse_r/40` option, therefore, VCS changes the first scheduled event from a change to 1 to a change to X at time 56 and schedules a second event on the output, a transition to 0 at time 60.
7. At time 56, the output transitions to X and VCS issues a warning message.
8. At time 60, the output transitions to 0.

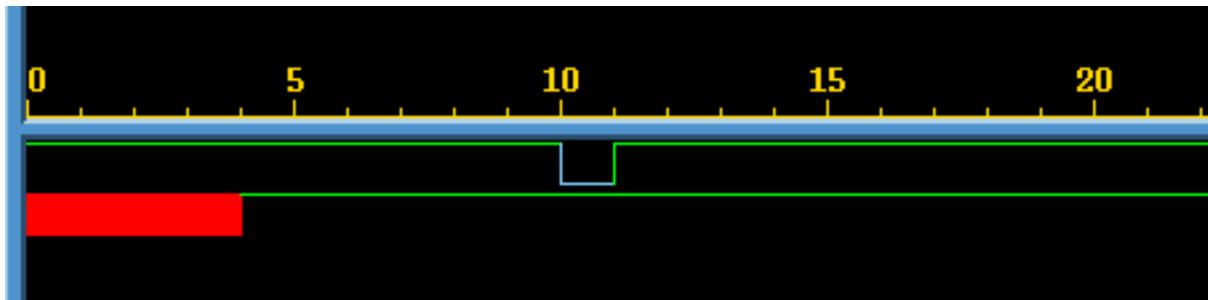
Pulse control sometimes blurs the distinction between inertial and transport delays. In this example, the results are the same if you also included the `+transport_path_delays` option.

## Specifying Pulse on Event or Detect Behavior

Asymmetric delays, such as different rise and fall times for a module path delay, can cause schedule cancellation problems for pulses. These problems persist when you specify transport delays and can persist for a wide range of percentages that you specify for pulse control options.

For example, for a module that models a buffer, if you specify a rise time of 4 and a fall time of 6 for a module path delay, a narrow value 0 pulse can cause scheduling problems, as illustrated in [Figure 11-9](#).

Figure 11-9 Asymmetric Delays and Scheduling Problems



In this example, you include the `+pulse_e/100` and `+pulse_r/0` options. The scheduling problem is that the leading edge of the pulse on the input, at time 10, schedules a transition to 0 on the output at time 16; but the trailing edge, at time 11, schedules a transition to 1 on the output at time 15.

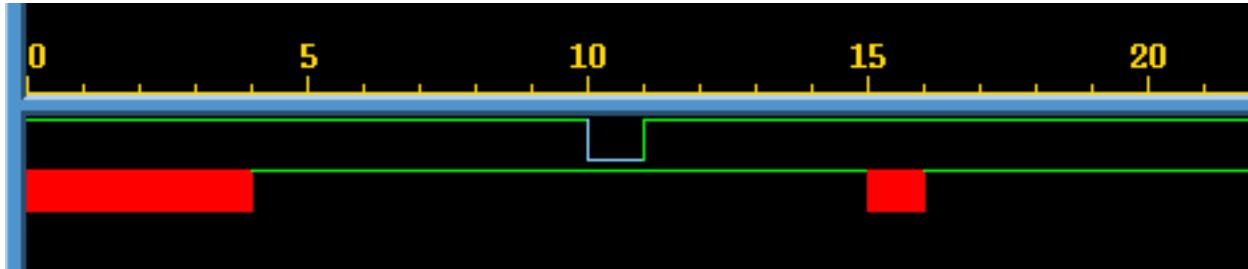
Obviously, the output has to end up with a value of 1 so VCS cannot allow the events scheduled at time 15 and 16 to occur in sequence; if it did, the output ends up with a value of 0. This problem persists when you enable transport delays and whenever the percentage specified in the `+pulse_r/number` 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 does not do is indicate that something happened on the output between times 15 and 16. You might want to see an error message and an X value pulse on the output indicating there was an undefined event on the output between these simulation times. You see this message and the X value pulse, if you include the `+pulse_on_event` compile-time option, specifying pulse on event behavior, as illustrated in [Figure 11-10](#). Pulse on event behavior calls

for an X value pulse on the output after the delay and when there are asymmetrical delays scheduling events on the output that would be canceled by VCS, to output an X value pulse between those events instead.

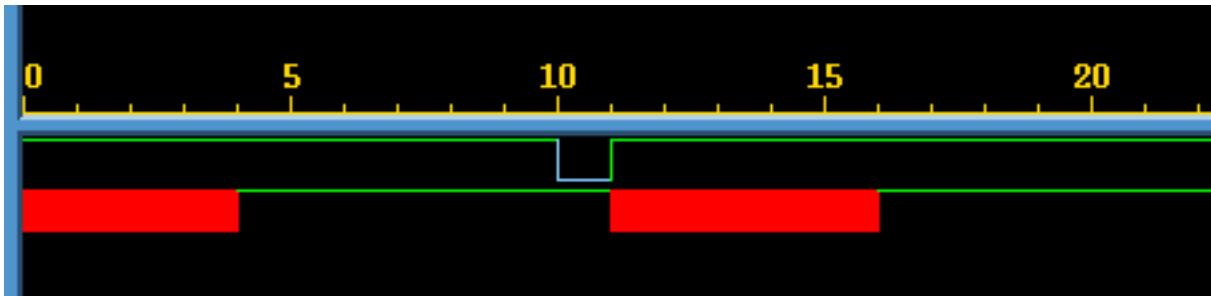
Figure 11-10 Using +pulse\_on\_event



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 makes 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. This is done as soon as events on the input, controlled by the `+pulse_e/number` and `+pulse_r/number` options, schedule an X value pulse to appear on the output. Pulse on detect behavior differs from pulse on event behavior in that it calls for the X value pulse to begin before the delay elapses. [Figure 11-11](#) illustrates pulse on detect behavior.

*Figure 11-11 Using +pulse\_on\_detect*



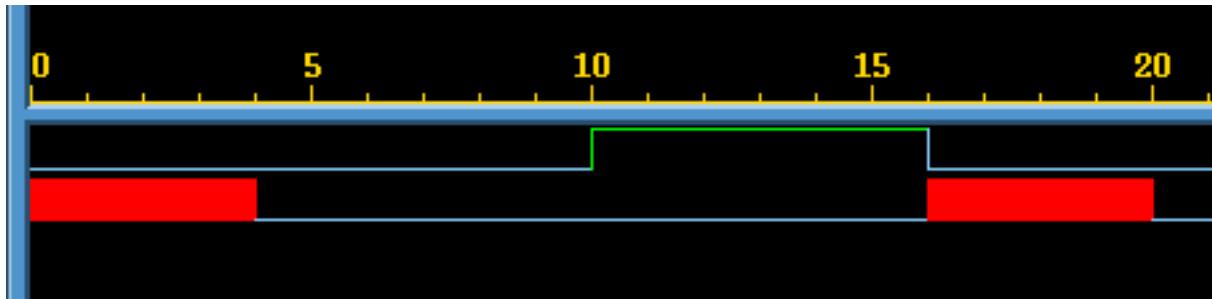
In this example, by including the `+pulse_on_detect` option, VCS causes the leading edge of the X value pulse on the output to begin at time 11. This is because of an unusual event that occurred on the output between times 15 and 16 because of the rise at simulation time 11.

Using pulse on detect behavior can also show you when VCS has scheduled multiple events for the same simulation time on the output. This is done by starting the leading edge of an X value pulse on the output as soon as VCS has scheduled the second event.

For example, a module that models a buffer has a rise time module path delay of 10 time units and a fall time module path delay of 4 time units.

[Figure 11-12](#) shows the waveforms for the input and output port when you include the `+pulse_on_detect` option.

*Figure 11-12 Pulse on Detect Behavior Showing Multiple Transitions*



In the example illustrated in [Figure 11-12](#), 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 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 shows you an X value pulse on the output starting when the second event was scheduled on the output port.

---

## Specifying the Delay Mode

It is possible for a module definition to include module path delay that does not equal the cumulative delay specifications in primitive instances and continuous assignment statements in that path.

[Example 11-1](#) shows such a conflict.

### *Example 11-1 Conflicting Delay Modes*

```
'timescale 1 ns / 1 ns
module design (out,in);
output out;
input in;
wire int1,int2;

assign #4 out=int2;

buf #3 buf2 (int2,int1),
      buf1 (int1,in);

specify
  (in => out) = 7;
endspecify
endmodule
```

In [Example 11-1](#), the module path delay is seven-time units, but the delay specifications distributed along that path add up to ten-time units.

If you include the `+delay_mode_path` compile-time/analysis option, VCS ignores the delay specifications in the primitive instantiation and continuous assignment statements and uses only the module path delay. In [Example 11-1](#), it uses the seven-time unit delay for propagating signal values through the module.

If you include the `+delay_mode_distributed` compile-time/analysis option, VCS ignores the module path delays and uses the delay in the delay specifications in the primitive instantiation and continuous assignment statements. In [Example 11-1](#), it uses the ten-time unit delay for propagating signal values through the module.

There are other modes that you can specify:

- If you include the `+delay_mode_unit` compile-time/analysis option, VCS ignores the module path delays and changes the delay specification in all primitive instantiation and continuous assignment statements to the shortest time precision argument of all the `'timescale` compiler directives in the source code. (The default time unit and time precision argument of the `'timescale` compiler directive is 1 s). In [Example 11-1](#) the `'timescale` compiler directive has a precision argument of 1 ns. VCS might use this 1 ns as the delay, but if the module definition is used in a larger design and there is another `'timescale` compiler directive in the source code with a finer precision argument, then VCS uses the finer precision argument.
- If you include the `+delay_mode_zero` compile-time/analysis option, VCS changes all delay specifications and module path delays to zero.
- If you include none of the compile-time options described in this section, when, as in [Example 11-1](#), the module path delay does not equal the distributed delays along the path, VCS uses the longer of the two.

---

## Support for Delayed Annotation During Simultaneous Switching on Inputs

VCS supports delayed annotations when multiple inputs change simultaneously. It ignores condition checking and inserts the least delay from applicable delays.

When multiple inputs change simultaneously, they create an impact on a specific output signal. If there are no matching conditional arcs that extend from the inputs (which are toggling) to the output, then VCS does not annotate a zero delay.

You can enable delayed annotation using the `+ignorempcond` runtime option.

---

## Usage Example

Consider that at any given instance when both inputs `in1` and `in2` changes from 1 to 0 simultaneously and if there are no matching arcs for either `in1==0` or `in2==0`.

### *Example 11-2 Support for Delayed Annotation*

```
assign out =in1& in2;
specify
  if(in1==1'b1)  (in2=>out)=3;
  if(in2==1'b1)  (in1=>out)=4;
```

In the default behavior, the output changes from 1 to 0 without any delay. Using the `+ignorempcond` option, the output changes from 1 to 0 with a delay of 3 units.

---

## 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. For details on how to do this, see “[The Configuration File Syntax](#)”. The attribute keywords for timing are as follows:

`noIopath`

Disables module path delays in the specified module instances.

`Iopath`

Enables module path delays in the specified module instances.

`noSpecify`

Disables the specify blocks in the specified module instances.

`Specify`

Enables specify blocks in the specified module instances.

`noTiming`

Disables timing checks in the specified module instances.

`Timing`

Enables timing checks in the specified module instances.

---

## Using the `timopt` Timing Optimizer

The `timopt` timing optimizer can yield large speedups for full-timing gate-level designs. The `timopt` timing optimizer makes its optimizations based on clock signals and sequential devices that it identifies in the design. `timopt` is particularly useful when you use SDF files because SDF files cannot be used with Radian Technology (+rad).

You enable `timopt` with the `+timopt+clock_period` compile-time option, where the argument is the shortest clock period (or clock cycle) of the clock signals in your design. For example:

```
+timopt+100ns
```

This options specifies that the shortest clock period is 100ns.

`timopt` first displays the number of sequential devices that it finds in the design and the number of these sequential devices to which it might be able to apply optimizations. For example:

```
Total Sequential Elements : 2001
Total Sequential Elements 2001, Optimizable 2001
```

`timopt` then displays the percentage of identified sequential devices to which it can actually apply optimizations followed by messages about the optimization process.

```
TIMOPT optimized 75 percent of the design
Starting TIMOPT Delay optimizations
Done TIMOPT Delay Optimizations
DONE TIMOPT
```

The next step is to simulate the design and see if the optimizations applied by `timopt` produce a satisfactory increase in performance. If you are not satisfied, there are additional steps that you can take to get more optimizations from `timopt`.

If `timopt` is 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 the number of clock signals and sequential devices, 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 signals and sequential devices that it finds questionable.
2. You review and edit this file, validate 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` makes additional optimizations that it did not make, because it was unsure of the signals and sequential devices in the `timopt.cfg` file that it wrote during the first compilation.

4. Look at the `timopt.cfg` file again:
  - If `timopt` wrote no new entries for potential clock signals or sequential devices, go to step 5.
  - If `timopt` wrote new entries, but you make no changes to the new entries, go to step 5.

- If you make modifications to the new entries, return to step 3.
5. `timopt` does not need to look for any more clock signals and it can assume that the `timopt.cfg` file correctly specifies clock signal and sequential devices. At this point, it just needs to apply the latest optimizations. Compile your design one more time, including the `+timopt` compile-time option, but without its `+clock_period` argument.
  6. You now simulate your design using `timopt` optimizations. `timopt` monitors the simulation and makes its optimizations based on its analysis of the design and information in the `timopt.cfg` file. During simulation, if it finds that its assumptions are incorrect, for example, the clock period for a clock signal is incorrect, or there is a port for asynchronous control on a module for a sequential device, `timopt` displays a warning message similar to the following:

```
+ Timopt Warning: for clock testbench.clockgen..clk:
TimePeriod 50ns Expected 100ns
```

## Editing the `timopt.cfg` File

When editing the `timopt.cfg` file, first edit the potential sequential device entries. Edit the potential clock signal only when you have made no changes to the entries for sequential devices.

## Editing Potential Sequential Device Entries

The following is an example of potential sequential devices:

```
// 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 { tfnpsc } { CP, true, CLN};
```

In this example, `CP` is the clock port and the `true` keyword 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` writes new clock signal entries in the `timopt.cfg` file.

## Editing Clock Signal Entries

The following is an example of the clock signal entries:

```
clock {  
    // test.badClock , // 1  
    test.goodClock // 2000  
} {100ns};
```

These clock signals have a period of 100ns or longer. This time value comes from the `+clock_period` argument that you added to the `+timopt` compile-time option when you first compiled the design. The entry for the signal `test.badClock` is commented out because it connects to a small percentage of the sequential devices in the design. In this instance, it is only 1 of the 2001 sequential devices that it identified in the design. The entry for the signal `test.goodClock` is not commented out because it connects to a large percentage of the sequential devices. In this instance, it is 2000 of the 2001 sequential devices in the design.

To make `timopt` use a commented out clock signal when it optimizes the design in a subsequent compilation, remove the comment characters preceding the signal's hierarchical name.

---

## Using Scan Simulation Optimizer

Scan Simulation Optimizer (`scanopt`) yields large speed-ups when used with Serial Scan DFT simulations. The optimizations are done based on the scan cells that are identified in the design. This optimization is applicable only on the Serial Scan DFT designs, using scan flops built with the MUX-FLOP combination.

This optimization can be enabled by using the `-scanopt=<clock_period>` compile-time option, where the `clock_period` argument is the shortest clock period (or clock cycle) of the clock signals in the design. For example, you must use `-scanopt=100ns` for a shortest clock period of 100ns.

The optimizer applies its optimization after scan flops in the design are identified. There is an option for providing all the scan flops in the design through a configuration file, `scanopt.cfg`, in the current

directory. This can be used if the optimizer fails to identify the scan flops, thereby, not producing a satisfactory performance improvement.

For example, for a design with shortest clock period of 100ns, you can supply the list of scan flops in the file, `scanopt.cfg` using the format specified in the following section, and then use the following compile-time option.

```
-scanopt=100ns, cfg
```

This enables the optimizer to pick up the scan flops specified in the configuration file and use for its optimization.

The optimizer also determines the length of the scan chain(s) on its own. If there are multiple scan chains, the minimal scan length is chosen for optimizations.

---

## ScanOpt Configuration File Format

The following format must be used for specifying a scan flop:

```
BEGIN_FLOP      <scan_cell_name>
    BEGIN_PORT
        Q_PORT   <q_port_name>
        [QN_PORT <qn_port_name>]
        D_PORT   <d_port_name>
        TI_PORT  <ti_port_name>
        TE_PORT  <te_port_name>
    END_PORT
END_FLOP
```

The section between BEGIN\_FLOP and END\_FLOP corresponds to one particular scan flop. The <scan\_cell\_name> field corresponds to the name of scan flop (scan cell). Multiple sections can be used to specify multiple scan flops.

The section between BEGIN\_PORT and END\_PORT also corresponds to ports of the scan flop. Specifying Q\_PORT, D\_PORT, TI\_PORT, and TE\_PORT are mandatory, whereas QN\_PORT could be optional.

---

## ScanOpt Assumptions

### Combinational Path Delays

By default, the optimizer assumes that the worst case delay for any combinational path in the design is not more than *five times* the shortest clock period and applies the optimizations. The following banner is printed at the compile time to indicate this assumption to you:

*"ScanOpt assumes that no combinational path has worst-case delay more than 5 clock period. Please use,  
"-scanopt=<clock\_period>,cdel=<overriding\_value>" to override the assumed value"*

For example, for a design with shortest clock period of 100ns, if the default value of 5 is to be overridden with a value of 10, you can use the following compile-time option.

```
-scanopt=100ns,cdel=10
```

## **Length of Test Cycles**

The optimizer assumes that the simulation remains in the test mode for at least the scan chain length times the shortest clock period. Any violation of this assumption is automatically detected during the simulation, and the following error message is displayed quitting the simulation.

*“Error: Simulation has been aborted due to fatal violation of ScanOpt assumptions. Please refer to the documentation for more details. To get around this error, please rerun simulation with “-noscanopt” switch”*

For example, if the inferred length of scan chain in the design is 5000 and the short clock period is 100ns, then the Test enable signal(s) should remain in test mode for at least 500000ns (that is, 5000 \* 100ns).

**Note:**

The `-noscanopt` option can be used at runtime, thereby avoiding re-compilation of the design.

---

## **Improving the ScanOpt for Debug Support**

VCS enhances the `scanopt` performance even in the presence of debug capabilities using the  
`-scanopt=nodumpcombo` option and speeds up the design execution when you run the design in `scanopt` mode with debug capabilities.

## **Use Model**

The feature has the following use model:

```
% vcs <file> <other options>
-scanopt=relaxdbg, nodumpcombo (with or without the -
-hsopt=gates option)
```

```
% simv
```

The following table describes the options:

|                         |                                                                                                                                                                                                                                                                      |
|-------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                         | -scanopt=nodumpcombo                                                                                                                                                                                                                                                 |
| With<br>-hsopt=gates    | <ul style="list-style-type: none"><li>• Displays a warning message if you access any signal in scan optimized hierarchy using the PLI/VPI/UCLI call for read/write/force values or for callback.</li><li>• Skips the signals from the generated FSDB file.</li></ul> |
| Without<br>-hsopt=gates | <ul style="list-style-type: none"><li>• Displays a warning message during compilation.</li></ul>                                                                                                                                                                     |

Note:

The -scanopt=nodumpcombo option is useful only when the FSDB file is dumped along with the -hsopt=gates option. When the -scanopt=nodumpcombo option is specified, VCS generates the FSDB file that do not contain any signal in scan optimized hierarchy. Therefore, the nCompare Waveform Comparison module does not display any difference between the FSDB file generated with and without the scanopt optimization.

## Usage Example

Consider the following examples:

**test.v**

```
module top;
    reg E,SE,CP,SI,SI_1,D;
    reg CDN;
    wire Q, Q_1, QN;
    wire w_1_4 = Q ^ 1;
    wire w_2_4 = Q_1 ^ 1;

    initial begin
        #100;
```

```

SE = 1'b0;
$display("===== Performing FORCE, READ and CALLBACK when SE = 0
=====");
$forceData();
#1000;
SE = 1'b1;
#300;
$display("===== Performing FORCE, READ and CALLBACK when SE = 1
=====");
$forceData();
#2000;
$finish;
end
always
begin
    SE <= 1;
    D <= generateLogicValue();
    .....
    #10;
end

```

```

always
begin
    SI <= generateLogicValue();
    SI_1 <= generateLogicValue();
    #100;
end

```

```

scan_cell inst_1(.E(E), .SE(SE), .CP(CP), .SI(SI_1), .D(w_1_4), .CDN(CDN),
.Q(Q_1), .QN(QN));
endmodule

```

```

`celldefine
module scan_cell(E, SE, CP, SI, D, CDN, Q, QN);
    input E, SE, CP, SI;
    input D;
    input CDN;
    output Q;
    output QN;
    mux_udp (D1, Q_buf, D, E);
    dff_udp (Q_buf, D2, CP, CDN_i, SDN, notifier);
    buf_udp (xCP_check, CP_check, 1'b1);
    buf_(CDN_i, CDN);
endmodule
`endcelldefine

```

### **test.c**

```

#include "stdio.h"
#include "sv_vpi_user.h"

int forceData ( char *data)
{
    vpiHandle frcH, frcI;
    vpiHandle GH;
    s_vpi_value valG;
    GH = vpi_handle(vpiSysTfCall, NULL);
    frcI = vpi_iterate(vpiArgument, GH);
    frcH = vpi_handle_by_name("top.inst_1.D", NULL);
    valG.format = vpiScalarVal;
    valG.value.scalar = 0;
    vpi_put_value(frcH, &valG, NULL, vpiForceFlag);
}

```

### **test.tab**

```
$forceData call=forceData check=forceData acc+=frc:*
```

Compile and simulate the examples as follows:

```
% vcs -sverilog test.v -P test.tab test.c -debug_access+all  
-scanopt=relaxdbg,nodumpcombo -hsopt=gates  
  
% simv
```

VCS generates the following warning message with the  
-scanopt=relaxdbg, nodumpcombo option:

```
Warning-[VPI-FORCE-SCAN] VPI force on scanopt hierarchy  
At time 1410, in PLI routine called from test.v, 20  
VPI force on scan optimized signal 'top.inst_1.D' may not have any effect.
```

Note:

The above message indicates that, at time 1410, the signal from the combo cloud connected to D pin of the scan cell is accessed through PLI/VPI/UCLI call when the Scan Enable (SE) is high. When you use the nodumpcombo option, a warning message is generated and the simulation continues as scan optimized hierarchy is accessed. The values that are dumped after SE is high and are valid values.

---

## Negative Timing Checks

Negative timing checks are either \$setuphold timing checks with negative setup or hold limits, or \$recrrem timing checks with negative recovery or removal limits.

This following sections describe their purpose, how they work, and how to use them:

- “The Need for Negative Value Timing Checks”

- “The \$setuphold Timing Check Extended Syntax”
  - “The \$recrem Timing Check Syntax”
  - “Enabling Negative Timing Checks”
  - “Checking Conditions”
  - “Toggling the Notifier Register”
  - “SDF Back-Annotation to Negative Timing Checks”
  - “How VCS Calculates Delays”
- 

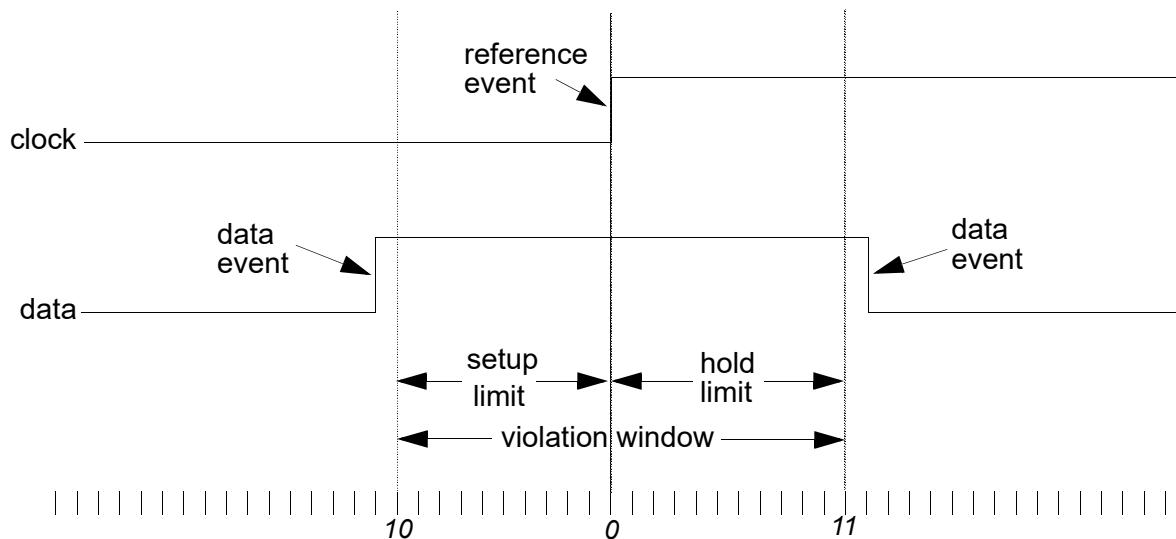
## The Need for Negative Value Timing Checks

The `$setuphold` timing check defines a timing violation window of a specified amount of simulation time before and after a reference event. For example, a transition on a clock signal, in which a data signal must remain constant. A transition on the data signal, called a data event, during the specified window is a timing violation. For example:

```
$setuphold (posedge clock, data, 10, 11, notifyreg);
```

In this example, VCS reports the timing violation if there is a transition on signal `data` less than 10 time units before, or less than 11 time units after, a rising edge on signal `clock`. When there is a timing violation, VCS toggles a notify register, in this example, `notifyreg`. You could use this toggling of a notify register to output an `x` value from a device, such as a sequential flop, when there is a timing violation.

*Figure 11-13 Positive Setup and Hold Limits*



In this example, both the setup and hold limits have positive values. When this occurs, the violation window straddles the reference event.

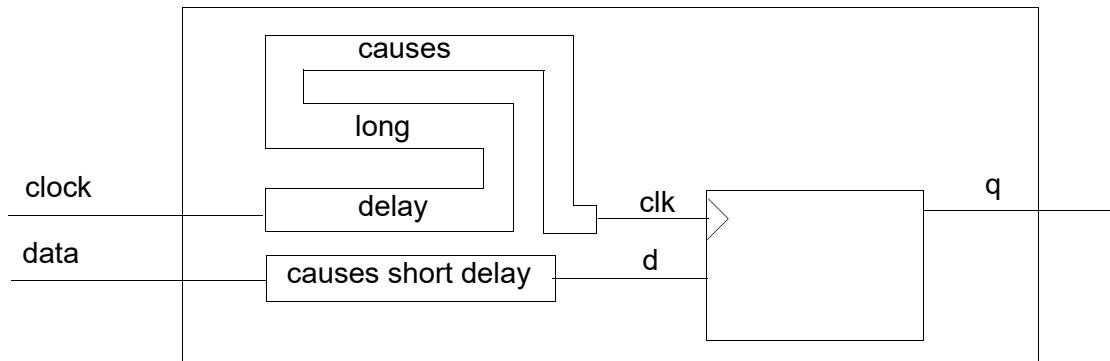
There are cases where the violation window cannot straddle the reference event at the inputs of an ASIC cell. Such a case occurs when:

- The data event takes longer than the reference event to propagate to a sequential device in the cell.
- Timing must be accurate at the sequential device.
- You need to check for timing violations at the cell boundary.

It also occurs when the opposite is true, that is, when the reference event takes longer than the data event to propagate to the sequential device.

When this happens, use the `$setuphold` timing check in the top-level module of the cell to look for timing violations when signal values propagate to that sequential device. In this case, you need to use negative setup or hold limits in the `$setuphold` timing check.

*Figure 11-14 ASIC Cell With Long Propagation Delays on Reference Events*

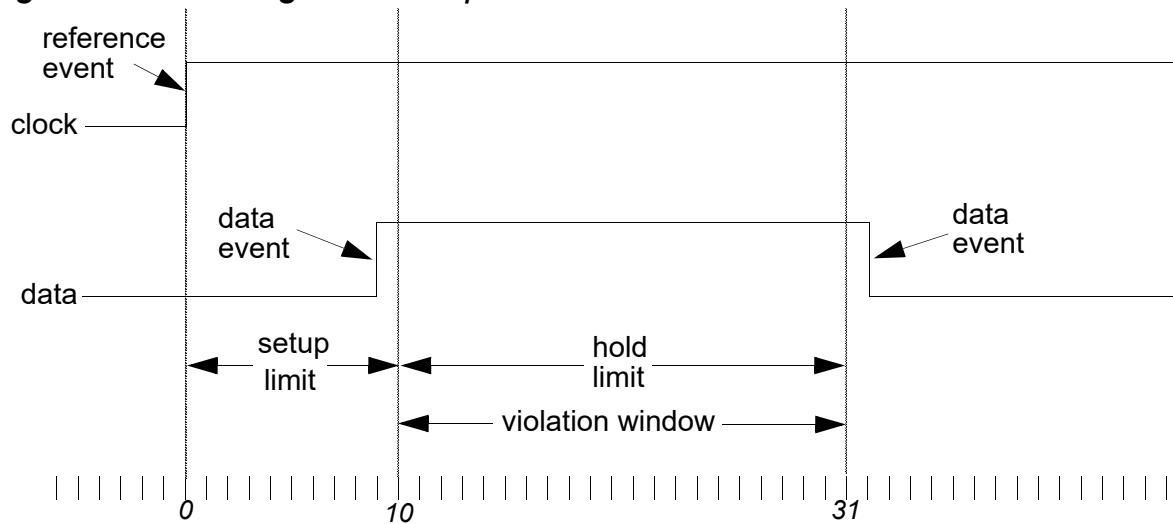


When this occurs, the violation window shifts at the cell boundary so that it no longer straddles the reference event. It shifts to the right when there are longer propagation delays on the reference event. This right shift requires a negative setup limit:

```
$setuphold (posedge clock, data, -10, 31, notifyreg);
```

[Figure 11-15](#) illustrates this scenario.

**Figure 11-15 Negative Setup Limit**



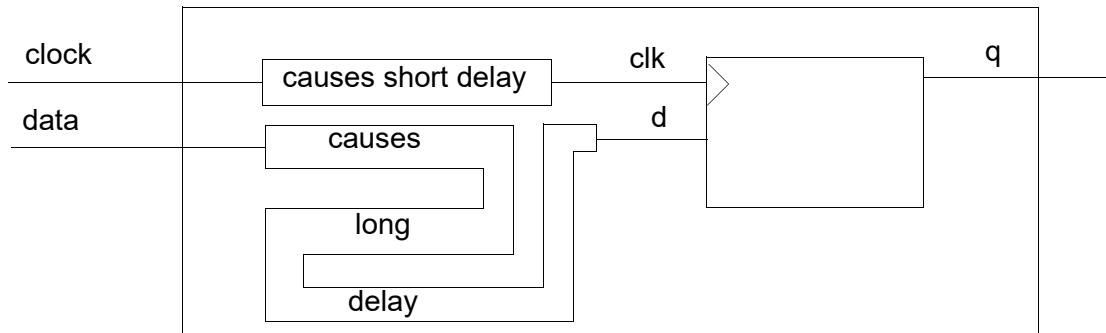
In this example, the `$setuphold` timing check is in the specify block of the top-level module of the cell. It specifies that there is a timing violation if there is a data event between 10 and 31 time units after the reference event on the cell boundary.

This is giving the reference event a “head start” at the cell boundary, anticipating that the delays on the reference event allow the data events to “catch up” at the sequential device inside the cell.

**Note:**

When you specify a negative setup limit, its value must be less than the hold limit.

*Figure 11-16 ASIC Cell With Long Propagation Delays on Data Events*

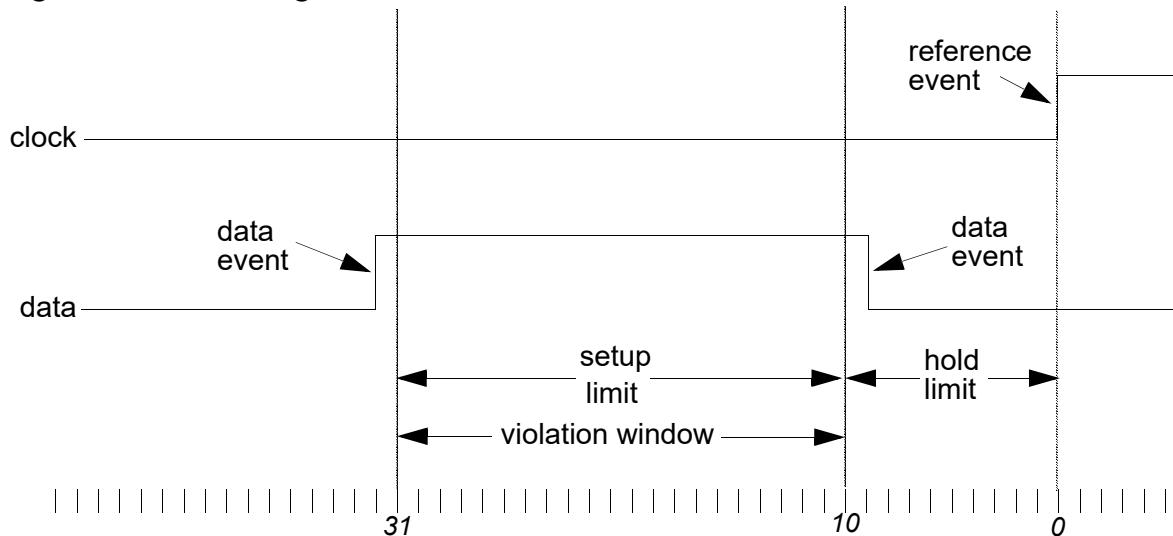


The violation window shifts to the left when there are longer propagation delays on the data event. This left shift requires a negative hold limit:

```
$setuphold (posedge clock, data, 31, -10, notifyreg);
```

[Figure 11-17](#) illustrates this scenario.

*Figure 11-17 Negative Hold Limit*



In this example, the `$setuphold` timing check is in the specify block of the top-level module of the cell. It specifies that there is a timing violation if there is a data event between 31 and 10 time units before the reference event on the cell boundary.

This is giving the data events a “head start” at the cell boundary, anticipating that the delays on the data events allow the reference event to “catch up” at the sequential device inside the cell.

**Note:**

When you specify a negative hold limit, its value must be less than the setup limit.

To implement negative timing checks, VCS creates delayed versions of the signals that carry the reference and data events and an alternative violation window where the window straddles the delayed reference event.

You can specify the names of the delayed versions by using the extended syntax of the `$setuphold` system task, or by allowing VCS to name them internally.

The extended syntax also allows you to specify expressions for additional conditions that must be true for a timing violation to occur.

## **The `$setuphold` Timing Check Extended Syntax**

The `$setuphold` timing check has the following extended syntax:

```
$setuphold(reference_event, data_event, setup_limit,  
hold_limit, notifier, [timestamp_cond, timecheck_cond,  
delayed_reference_signal, delayed_data_signal]);
```

The following additional arguments are optional:

`timestamp_cond`

This argument specifies the condition which determines whether or not VCS reports a timing violation.

In the setup phase of a `$setuphold` timing check, VCS records or “stamps” the time of a data event internally. 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. When a data event occurs, it can compare the times of these events to see if there is a hold timing violation. If the condition specified by this argument is false, VCS does not record or “stamp” the reference event, so there cannot be a hold timing violation.

#### `timecheck_cond`

This argument specifies the condition which determines whether or not VCS reports a timing violation.

In the setup phase of a `$setuphold` timing check, VCS compares or “checks” the time of the reference event with the time of the data event to see if there is a setup timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no setup timing violation.

Similarly, in the hold phase of a `$setuphold` timing check, VCS compares or “checks” the time of a data event with the time of a reference event to see if there is a hold timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no hold timing violation.

#### `delayed_reference_signal`

The name of the delayed version of the reference signal.

```
delayed_data_signal
```

The name of the delayed version of the data signal.

The following example demonstrates how to use the extended syntax:

```
$setuphold(ref, data, -4, 10, notifrl, stampreg==1, , d_ref,  
          d_data);
```

In this example, the *timestamp\_cond* argument specifies that *reg stampreg* must equal 1 for VCS to “stamp” or record the times of data events in the setup phase or “stamp” the times of reference events in the hold phase. If this condition is not met, and stamping does not occur, VCS does not find timing violations no matter what the time is for these events. Also in the example, the delayed versions of the reference and data signals are named *d\_ref* and *d\_data*.

You can use these delayed signal versions of the signals to drive sequential devices in your cell model. For example:

```
module DFF(D,RST,CLK,Q);  
  input D,RST,CLK;  
  output Q;  
  reg notifier;  
  DFF_UDP d2(Q,dCLK,dD,dRST,notifier);  
  specify  
    (D => Q) = 20;  
    (CLK => Q) = 20;  
    $setuphold(posedge CLK,D,-5,10,notifier,,,dCLK,dD);  
    $setuphold(posedge CLK,RST,-8,12,notifier,,,dCLK,  
              dRST);  
  endspecify  
endmodule  
  
primitive DFF_UDP(q,clk,data,rst,notifier);  
  output q; reg q;
```

```

input data,clk,rst,notifier;

table
// clock  data  rst   notifier  state  q
// -----
r    0     0      ?      : ? : 0 ;
r    1     0      ?      : ? : 1 ;
f    ?     0      ?      : ? : - ;
?    ?     r      ?      : ? : 0 ;
?    *     ?      ?      : ? : - ;
?    ?     ?      *      : ? : x ;
endtable
endprimitive

```

In this example, the `DFF_UDP` user-defined primitive is driven by the delayed signals `dClk`, `dD`, `dRST`, and the notifier `reg`.

## Negative Timing Checks for Asynchronous Controls

The `$recrem` timing check is used for checking how close asynchronous control signal transitions are to clock signals. Similar to the setup and hold limits in `$setuphold` timing checks, the `$recrem` timing check has recovery and removal limits. The recovery limit specifies how much time must elapse after a control signal toggles from its active state before there is an active clock edge. The removal limit specifies how much time must elapse after an active clock edge before the control signal can toggle from its active state.

In the same way a reference signal, such as a clock signal and data signal can have different propagation delays from the cell boundary to a sequential device inside the cell, there can be different

propagation delays between the clock signal and the control signal. For this reason, there can be negative recovery and removal limits in the \$recrem timing check.

## The \$recrem Timing Check Syntax

The \$recrem timing check syntax is very similar to the extended syntax for \$setuphold:

```
$recrem(reference_event, data_event, recovery_limit,  
removal_limit, notifier, [timestamp_cond, timecheck_cond,  
delayed_reference_signal, delayed_data_signal]);
```

`reference_event`

Typically, the reference event is the active edge on a control signal, such as a clear signal. Specify the active edge with the posedge or negedge keyword.

`data_event`

Typically, the data event occurs on a clock signal. Specify the active edge on this signal with the posedge or negedge keyword.

`recovery_limit`

Specifies how much time must elapse after a control signal, such as a clear signal toggles from its active state (the reference event), before there is an active clock edge (the data event).

`removal_limit`

Specifies how much time must elapse after an active clock edge (the data event), before the control signal can toggle from its active state (the reference event).

notifier

A register whose value VCS toggles when there is a timing violation.

timestamp\_cond

This argument specifies the condition which determines whether or not VCS reports a timing violation.

In the recovery phase of a `$recrem` timing check, VCS records or “stamps” the time of a reference event internally. 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. 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 \$recrm timing check, VCS compares or “checks” the time of a reference event with the time of a data event to see if there is a removal timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no removal timing violation.

`delayed_reference_signal`

The name of the delayed version of the reference signal, typically a control signal.

`delayed_data_signal`

The name of the delayed version of the data signal, typically a clock signal.

---

## Enabling Negative Timing Checks

To use a negative timing check you must include the `+neg_tchk` compile-time option when you compile your design. If you omit this option, VCS changes all negative limits to 0.

If you include the `+no_notifier` compile-time option with the `+neg_tchk` option, you only disable notifier toggling. VCS still creates the delayed versions of the reference and data signals and displays timing violation messages.

Conversely, if you include the `+no_tchk_msg` compile-time option with the `+neg_tchk` option, you only disable timing violation messages. VCS still creates the delayed versions of the reference and data signals and toggles notifier regs when there are timing violations.

If you include the `+neg_tchk` compile-time option but also include the `+notimingcheck` or `+nospecify` compile-time options, VCS does not compile the `$setuphold` and `$recrem` timing checks into the `simv` executable. However, it does create the signals that you specified in the `delayed_reference_signal` and `delayed_data_signal` arguments, and you can use these to drive sequential devices in the cell. Note that there is no delay on these "delayed" arguments and they have the same transition times as the signals specified in the `reference_event` and `data_event` arguments.

Similarly, if you include the `+neg_tchk` compile-time option and then include the `+notimingcheck` runtime option instead of the compile-time option, you disable the `$setuphold` and `$recrem` timing checks that VCS compiled into the executable. At compile time, VCS creates the signals that you specified in the `delayed_reference_signal` and `delayed_data_signal` arguments, and you can use them to drive sequential devices in the cell, but the `+notimingcheck` runtime option disables the delay on these "delayed" versions.

---

## Other Timing Checks Using the Delayed Signals

When you enable negative timing limits in the `$setuphold` and `$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 11-3 Notifier Register Example for Delayed Reference and Data Signals**

```
'timescale 1ns/1ns

module top;
    reg clk, d;
    reg rst;
    wire q;

    dff dff1(q, clk, d, rst);

    initial begin
        $monitor($time,,clk,,d,,q);
        rst = 0; clk = 0; d = 0;
        #100 clk = 1;
        #100 clk = 0;
        #10 d = 1;
        #90 clk = 1;
        #1 clk = 0; // width violation
        #100 $finish;
    end
endmodule

module dff(q, clk, d, rst);
    output q;
    input clk, d, rst;
    reg notif;

    DFF_UDP(q, d_clk, d_d, d_rst, notif);

    specify
        $setuphold(posedge clk, d, -10, 20, notif, , , d_clk,
                   d_d);
        $setuphold(posedge clk, rst, 10, 10, notif, , , d_clk,
                   d_rst);
        $width(posedge clk, 5, 0, notif);
    endspecify
endmodule
```

```

primitive DFF_UDP(q,data,clk,rst,notifier);
output q; reg q;
input data,clk,rst,notifier;

table
// clock  data  rst   notifier  state  q
// -----
    r      0      0      ?      :      ?      :      0  ;
    r      1      0      ?      :      ?      :      1  ;
    f      ?      0      ?      :      ?      :      -  ;
    ?      ?      r      ?      :      ?      :      0  ;
    ?      *      ?      ?      :      ?      :      -  ;
    ?      ?      ?      *      :      ?      :      x  ;
endtable
endprimitive

```

In this example, if you include the `+neg_tchk` compile-time option, the `$width` timing check uses the delayed version of signal `clk`, named `d_clk`, and the following sequence of events occurs:

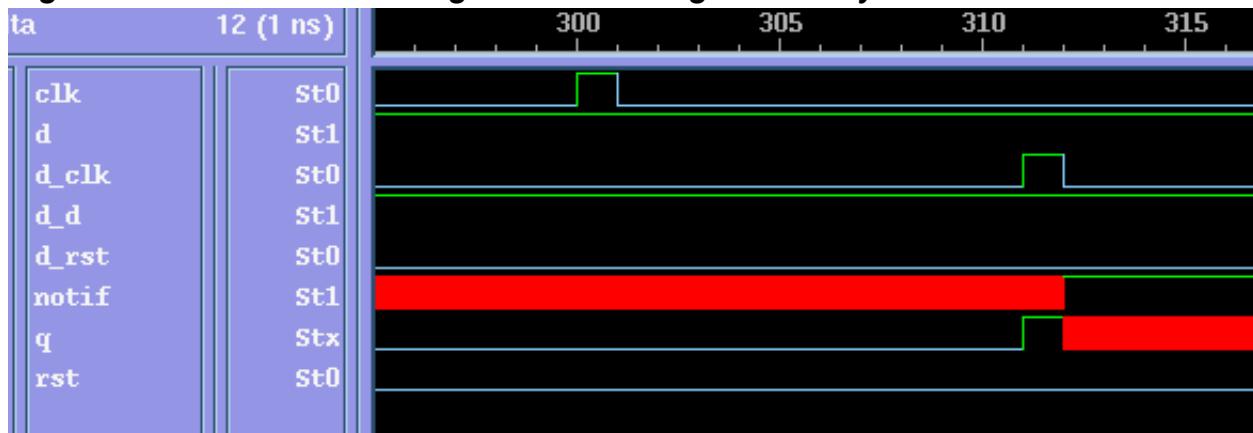
1. At time 311, the delayed version of the clock transitions to 1, causing output `q` to toggle to 1.
2. At time 312, the narrow pulse on the clock causes a width violation:

```
"test1.v", 31: Timing violation in top.dff1
$width( posedge clk:300,      : 301, limit: 5);
```

The timing violation message looks like it occurs at time 301, but you do not see it until time 312.

3. Also at time 312, `reg notif` toggles from `x` to 1. This changes output `q` from 1 to `x`. There are no subsequent changes on output `q`.

*Figure 11-18 Other Timing Checks Using the Delayed Versions*

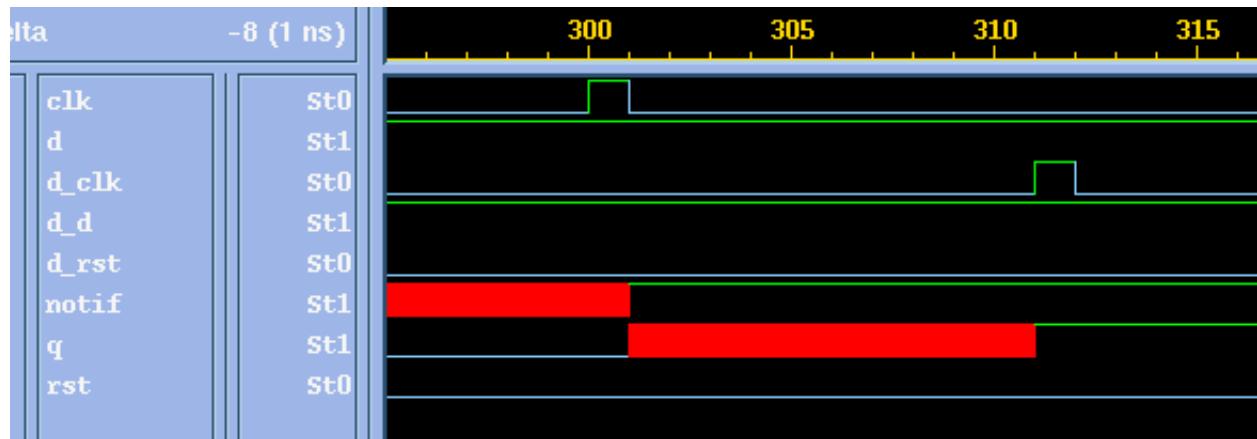


If you include both the `+neg_tchk` and `+old_ntc` compile-time options, the `$width` timing check does not use the delayed version of signal `clk`, causing the following sequence of events to occur:

1. At time 301, the narrow pulse on signal `clk` causes a width violation:  

```
"test1.v", 31: Timing violation in top.dff1
$width( posedge clk:300,    : 301, limit: 5);
```
2. Also at time 301, the notifier reg named `notif` toggles from `X` to `1`. In turn, this changes the output `q` of the user-defined primitive `DFF_UDP` and module instance `dff1` from `0` to `X`.
3. At time 311, the delayed version of signal `clk`, named `d_clk`, reaches the user-defined primitive `DFF_UDP`, thereby changing the output `q` to `1`, erasing the `X` value on this output.

Figure 11-19 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 in either of the following cases:

- When there is a value change on the original reference and data signals at the cell boundary
- 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:

- It does not matter when VCS evaluates these expressions:
  - If the operands in these expressions are neither the original nor the delayed versions of the reference or data signals
  - If these operands are signals that do not change value between value changes on the original reference and data signals and their delayed versions
- If the operands in these expressions are delayed versions of the original reference and data signals, then you want VCS to evaluate these expressions when there are value changes on the delayed versions of the reference and data signals. VCS does this by default.
- If the operands in these expressions are the original reference and data signals and not the delayed versions, then you want VCS to evaluate these expressions when there are value changes on the original reference and data signals. To specify evaluating these expressions when the original reference and data signals change value, include the `+NTC2` compile-time option.

## Toggling the Notifier Register

VCS waits for a timing violation to occur on the delayed versions of the reference and data signals before toggling the notifier register. Toggling means the following value changes:

- X to 0
- 0 to 1
- 1 to 0

VCS does not change the value of the notifier register if you have assigned a Z value to it.

---

## SDF Back-Annotation to Negative Timing Checks

You can back-annotate negative setup and hold limits from SDF files to `$setuphold` timing checks and negative recovery and removal limits from SDF files to `$recrem` timing checks, if the following conditions are met:

- You included the arguments for the names of the delayed reference and data signals in the timing checks.
- You compiled your design with the `+neg_tchk` compile-time option.
- For all `$setuphold` timing checks, the positive setup or hold limit is greater than the negative setup or hold limit.
- For all `$recrem` timing checks, the positive recovery or removal limit is greater than the negative recovery or removal limit.

As documented in the OVI SDF3.0 specification:

- **TIMINGCHECK** statements in the SDF file back-annotate timing checks in the model which match the edge and condition arguments in the SDF statement.
- If the SDF statement specifies `SCOND` or `CCOND` expressions, they must match the corresponding `timestamp_cond` or `timecheck_cond` in the timing check declaration for back-annotation to occur.
- If there is no `SCOND` or `CCOND` expressions in the SDF statement, all timing checks that otherwise match are back-annotated.

---

## How VCS Calculates Delays

This section describes how VCS calculates the delays of the delayed versions of reference and data signals. It does not describe how you use negative timing checks; it is supplemental material intended for users who would like to read more about how negative timing checks work in VCS.

VCS uses the limits you specify in the `$setuphold` or `$recrem` timing check to calculate the delays on the delayed versions of the reference and data signals. For example:

```
$setuphold(posedge clock,data,-10,20, , , , del_clock,  
          del_data);
```

This specifies that the propagation delays on the reference event (a rising edge on signal clock), are more than 10 but less than 20 time units more than the propagation delays on the data event (any transition on signal data).

So when VCS creates the delayed signals, `del_clock` and `del_data`, and the alternative violation window that straddles a rising edge on `del_clock`, VCS uses the following relationship:

$$20 > (\text{delay on } \text{del\_clock} - \text{delay on } \text{del\_data}) > 10$$

There is no reason to make the delays on either of these delayed signals any longer than they have to be so the delay on `del_data` is 0 and the delay on `del_clock` is 11. Any delay on `del_clock` between 11 and 19 time units would report a timing violation for the `$setuphold` timing check.

Multiple timing checks, that share reference or data events, and specified delayed signal names, can define a set of delay relationships. For example:

```
$setuphold(posedge CP,D,-10,20, notifier, , ,  
           del_CP, del_D);  
$setuphold(posedge CP,TI,20,-10, notifier, , ,  
           del_CP, del_TI);  
$setuphold(posedge CP,TE,-4,8,  notifier, , ,  
           del_CP, del_TE);
```

In this example:

- The first `$setuphold` timing check specifies the delay on `del_CP` is more than 10 but less than 20 time units more than the delay on `del_D`.
- The second `$setuphold` timing check specifies the delay on `del_TI` is more than 10 but less than 20 time units more than the delay on `del_CP`.
- The third `$setuphold` timing check specifies the delay on `del_CP` is more than 4 but less than 8 time units more than the delay on `del_TE`.

Therefore:

- The delay on `del_D` is 0 because its delay does not have to be more than any other delayed signal.
- The delay on `del_CP` is 11 because it must be more than 10 time units more than the 0 delay on `del_D`.

- The delay on `del_TE` is 4 because the delay on `del_CP` is 11. The 11 makes the possible delay on `del_TE` larger than 3, but less than 7. The delay cannot be 3 or less, because the delay on `del_CP` is less than 8 time units more than the delay on `del_TE`. VCS makes the delay 4 because it always uses the shortest possible delay.
- The delay on `del_TI` is 22 because it must be more than 10 time units more than the 11 delay on `del_CP`.

In unusual and rare circumstances, multiple `$setuphold` and `$recrem` timing checks, including those that have no negative limits, can make the delays on the delayed versions of these signals mutually exclusive. When this happens, VCS repeats the following procedure until the signals are no longer mutually exclusive:

1. Sets one negative limit to 0.
2. Recalculates the delays of the delayed signals.

## Using VITAL Models and Netlists

You use VCS to validate and optimize a VHDL initiative toward ASIC libraries (VITAL) model and to simulate a VITAL-based netlist. Typically, library developers optimize the VITAL model, and designers simulate the VITAL-based netlist.

The library developer uses a single ASIC cell from the system, verifies its correctness, and optimizes that single cell. The designer simulates large numbers of cells, organized in a netlist, by applying test vectors and timing information.

This section describes how to validate and optimize a VITAL model and how to simulate a VITAL netlist. It contains the following sections:

- “[Validating and Optimizing a VITAL Model](#)”
  - “[Simulating a VITAL Netlist](#)”
  - “[Understanding VITAL Timing Delays and Error Messages](#)”
- 

## **Validating and Optimizing a VITAL Model**

The library developer performs the following tasks:

- Validates the model for VITAL conformance
- Verifies the model for functionality
- Optimizes the model for performance and capacity
- Re-verifies the model for functionality

The following sections describe each of these tasks in detail.

### **Validating the Model for VITAL Conformance**

Library developers can use the `vhdlan` utility to validate the conformance of the VHDL design units to VITAL 95 IEEE specifications, according to level 0 or level 1, as specified in the model.

The `vhdlan` utility checks the VITAL design units for conformance when you set the VITAL attribute on the entity (VITAL\_Level0) and architecture (VITAL\_Level1) to TRUE. The `vhdlan` utility does not check the design unit for VITAL conformance if the attribute is set to FALSE.

## Verifying the Model for Functionality

After validating the model for VITAL conformance, library developers use the binary executable to verify the model's functions. The functional verification includes checking the following:

- Timing values for the cell, including hazard detection
- Correct operation of the timing constraints and violation detection
- Other behavioral aspects of the cell according to specifications

## Optimizing the Model for Performance and Capacity

Library developers use `vhdlan` to analyze the VHDL design units to optimize the model for simulation. The `vhdlan` utility checks the design unit for VITAL conformance before performing any optimization.

To optimize the design units, perform the following steps:

1. Set the VITAL attribute on the entity (VITAL\_Level0) and on the architecture (VITAL\_Level1) to TRUE.

When you optimize architectures that have the VITAL\_Level1 attribute set to TRUE, visibility into the cell is lost and the cell is marked as PRIVATE. Ports and generics remain visible.

2. Use either the OPTIMIZE variable in the setup file or the -optimize option on the vhdlan command line as follows:
  - Set the OPTIMIZE variable in the setup file.

[Table 11-2](#) lists the legal values of the variable, the design unit type, and the results of each setting.

*Table 11-2 Optimize Variable Values*

| Variable | Values | Design Unit Type | Result                                                                                                                |
|----------|--------|------------------|-----------------------------------------------------------------------------------------------------------------------|
| OPTIMIZE | TRUE   | Non-VITAL        | The vhdlan utility does not perform any optimization.                                                                 |
| OPTIMIZE | TRUE   | VITAL            | The vhdlan utility performs the optimization on design units that are VITAL conformant.                               |
| OPTIMIZE | FALSE  | Non-VITAL        | The vhdlan utility does not perform any optimization.                                                                 |
| OPTIMIZE | FALSE  | VITAL            | The vhdlan utility does not perform optimization on design unit regardless of its VITAL conformance status (default). |

- Use the -optimize option on the vhdlan command line. The command-line option overrides the setting in the synopsys\_sim.setup file.

## Re-Verifying the Model for Functionality

After validating and then optimizing the cell, library developers reverify the results against expected results. The optimizations performed by VCS typically result in correct code.

## Understanding Error and Warning Messages

If the VITAL conformance checks for a design unit fail, VCS issues an error message and stops the optimization of the design unit. Simulation files (`.sim` and `.o` files) are not created, and simulation is not possible for this design unit until the model is changed to conform to VITAL specifications.

If VCS reports a warning message, the optimization stops only if the message is related to the VITAL architecture, otherwise the optimization continues. Simulation files are generated, and you can simulate the design units.

[Table 11-3](#) lists the status of optimization and simulation file generation based on the type of messages that VCS issues.

*Table 11-3 Analyzer Status Messages*

| VITAL Attribute        | Message Types | Optimization | Simulation Files |
|------------------------|---------------|--------------|------------------|
| Level 0 (entity)       | error         | stops        | not created      |
| Level 1 (architecture) | error         | stops        | not created      |
| Level 0 (entity)       | warning       | continues    | created          |
| Level 1 (architecture) | warning       | stops        | created          |

For a complete list of conformance checking error messages, see [“VITAL Error Messages for Level 0 Conformance Issues” on page 92](#) and [“VITAL Error Messages for Level 1 Conformance Issues” on page 94](#).

When analyzing VITAL models, you can relax VITAL conformance violation errors to a warnings, by setting `RELAX_CONFORMANCE` variable in `synopsys_sim.setup` file to TRUE. This value of this variable by default is FALSE.

## Distributing a VITAL Model

VITAL library developers (usually, ASIC vendors) can distribute models (ASIC library) to designers in any of the following formats:

- A VHDL source file

After conformance checking and verification, you can distribute the cell library in source format. The library is unprotected, but it is portable.

- An encrypted VHDL source file

You can distribute the encrypted file similar to the VHDL source file. Because the encryption algorithms are generally not public and the code is protected, models are not portable to other simulators.

- Simulation files (the `.sim` and `.o` files)

The cell is analyzed and optimized by the ASIC vendor. The library is protected and is not portable to other simulators or simulator versions.

For the VHDL file and the encrypted VHDL source file formats, the designer can perform the final compilation to optimize the library object codes by using the `-optimize` option. ASIC vendors can provide designers with a script specifying the correct compilation procedure.

---

## Simulating a VITAL Netlist

A VITAL-based netlist consists of instances of VITAL cells. There are no VITAL specific or other restrictions on the location of such cells in the netlist, nor are there restrictions regarding the quantity or ratio of such cells in relation to other VHDL descriptions.

To simulate a VITAL netlist, simply invoke the binary executable.

## Applying Stimulus

You apply the input stimulus for the VITAL netlist using the same method and format that you use to apply it for any other netlist. For example, you can use WIF, text input/output, or a testbench.

## Overriding Generic Parameter Values

You can override the VITAL generic values in the following ways:

- Using `synopsys_sim.setup` file variables
- Using the elaboration option `-gv generic_name=value`

The following table describes the `SYNOPSYS_SIM.SETUP` variables and the corresponding generic and values allowed:

**Table 11-4 Timing Constraint and Hazard Flags**

| <b>synopsys_sim.setup Variables</b> | <b>Generics</b> | <b>Legal Values</b> | <b>Result</b>                                                             |
|-------------------------------------|-----------------|---------------------|---------------------------------------------------------------------------|
| Force_TimingChecksOn_TO             | TimingChecksOn  | TRUE                | Timing checks are performed.                                              |
|                                     |                 | FALSE               | Timing checks are disabled for that cell.                                 |
|                                     |                 | AsIs                | User-specified value of the generic is not modified. This is the default. |
| Force_XOn_TO                        | XOn             | TRUE                | X's are generated with violations.                                        |
|                                     |                 | FALSE               | X generation is disabled for that cell.                                   |
|                                     |                 | AsIs                | User-specified value of the generic is not modified. This is the default. |
| Force_MsgOn_TO                      | MsgOn           | TRUE                | Messages are reported on violations.                                      |
|                                     |                 | FALSE               | Timing messages are disabled for that cell.                               |
|                                     |                 | AsIs                | User-specified value of the generic is not modified. This is the default. |

For example:

The following setting in your `synopsys_sim.setup` file performs timing checks:

```
Force_TimingChecksOn_To = TRUE
```

Use the corresponding command line to set the generic:

```
% vcs top -gv TimingChecksOn=TRUE
```

These flags override the value of VITAL generic parameters. The flags have no effect if the model does not use the generic parameter. The generics XOn and MsgOn are parameters to VITAL timing and path delay subprograms.

## **Understanding VCS Error Messages**

VCS reports two types of errors: system errors and model/netlist errors.

### **System Errors**

VCS reports a system error if any of the following conditions occur:

- If there are any negative timing values after all timing values are imported and negative constraint calculations (NCC) are performed.

All the adjusted timing values must be positive or zero ( $\geq 0$ ) after all timing values are imported and NCC is performed. If an adjusted value is negative, NCC issues a warning message and uses zero instead.

Use the `man vss-297` and `man vss-298` command to get more information about NCC error messages.

- If you try to “look-into” the parts of the model that are invisible.

This is because the visibility is limited in VITAL cells that have been optimized and the cells are marked as PRIVATE.

### **Model and Netlist Errors**

A VITAL model in a VITAL netlist can generate several kinds of errors. The most important are hazard and constraint violations, both of which are associated with a violation of the timing model. The format of such errors is defined by the VITAL standard (in VHDL packages).

## **Viewing VITAL Subprograms**

You cannot view or access VITAL subprograms. The VITAL packages are built-in. Any reference to a VITAL subprogram (functions or procedures) or any other item in the VITAL packages is converted by VCS to a built-in representation.

## **Timing Back-annotation**

A VITAL netlist can import timing information from a VHDL configuration or an SDF file.

- A VHDL configuration
  - VHDL allows the use of a configuration block to override the values of generics specified in the entity declaration. This is done during analysis of the design.
- SDF file
  - VITAL netlist can import an SDF 3.0 version file. The VITAL standard defines the mapping for SDF 3.0 and the subset supported.

## **VCS Naming Styles**

VCS automatically determines what naming style is used according to the cell:

- For conformance checked VITAL cells (that is, VITAL entities with the VITAL\_Level0 attribute set to TRUE), VCS uses VITAL naming styles.
- For non-VITAL conformance checked cells, VCS uses the Synopsys naming style (or the style described in SDF naming file).

Note:

VCS ignores the SDFNAMINGSTYLE variable in the setup file when determining the naming style.

## Negative Constraints Calculation (NCC)

Adjusting the cell timing values and converting the negative values follows the elaboration and back-annotation phases. VCS follows these steps to prepare the design units for simulation:

1. Design Elaboration

Elaboration is a VHDL step, the design is created and is ready for the simulation run.

2. Back-annotation of timing delay values

Timing values are imported, and the value of generic parameters are updated. VITAL models that support NCC accept back-annotation information as in any other cell.

3. Conversion of the negative constraint values

The value of generic parameters is modified to conform to the NCC algorithm, and negative constraint values are converted to zero or positive.

VCS automatically performs NCC only when the VITAL\_Level0 attribute is set to TRUE for the VITAL entity and the internal clock delay generic (ticd) or internal signal delay generic (tisd) is set.

VCS does not run NCC on design units that have a non-VITAL design type, but you can simulate them.

4. Running the simulation.

## Simulating in Functional Mode

By default, VCS generates code that provides the flexibility of choosing functional or regular VITAL simulation when simulation is run. You can use the `-novitaltiming` runtime option to get functional VITAL simulation; otherwise, you get regular, full-timing VITAL simulation. You can also use `-functional_vital` with `vhdlan` to get full functional VITAL simulation.

Choosing the VITAL simulation mode at analysis time provides a better performance than choosing the mode at runtime, because it eliminates the runtime check for the functional VITAL simulation mode. The trade-off is that you must reanalyze your VITAL sources if you want to switch between functional and timing simulation. Therefore, you should add the appropriate option to the `vhdlan` command line after you determine which simulation mode gives the best performance while preserving correct simulation results.

Using the `-novitaltiming` runtime option eliminates all timing-related aspects from the simulation of VITAL components. With this option, VCS eliminates the following timing-related aspects: wire delays, path delays, and timing checks, and assigns 0-delay to all outputs. The elimination of timing from the simulation of the VITAL components significantly improves the performance of event simulations.

By specifying `-no_functional_vital` for `vhdlan`, you get full timing VITAL simulation without the ability to use functional VITAL at runtime.

However, if your design depends on one or more of the timing-related aspects, you can try reanalyzing the VITAL source files with one or more of the following options, depending on the timing-related or functional capabilities that you need to preserve:

`-keep_vital_ifs`

This option turns off some of the aggressive novitaltiming optimizations related to `if` statements in Level 0 VITAL cells.

`-keep_vital_path_delay`

This option preserves the calls to `VitalPathDelay`. Use this switch to preserve correct functionality of non-zero assignments to the outputs.

`-keep_vital_wire_delay`

This option preserves the calls to `VitalWireDelay`. Use this switch to preserve correct functionality of delays on the inputs.

`-keep_vital_signal_delay`

This option preserves the calls to `VitalSignalDelay`. Use this switch to preserve correct functionality of delays on signals.

`-keep_vital_timing_checks`

This option preserves the timing checks within the VITAL cell.

`-keep_vital_primitives`

This option preserves calls to VITAL primitive subprograms.

---

## Understanding VITAL Timing Delays and Error Messages

This section describes how VCS calculates negative timing constraints during elaboration. This section also lists the error messages that the `vhdlan` utility generates while checking design units for VITAL conformance.

## **Negative Constraint Calculation (NCC)**

VITAL defines the special generics ticd, tisd, tbpd, SignalDelay Block, and equations to adjust the negative setup and hold time and related IOPATH delays.

For VITAL models, NCC adjusts the timing generics for the ticd or tisd generic. The ticd delay is calculated based on SETUP and RECOVERY time. Therefore, NCC resets the original ticd delay in VITAL cells.

## **Conformance Checks**

For VITAL conformance, VCS checks the design units that have the VITAL\_Level0 or VITAL\_Level1 attribute set to TRUE (if the attributes are set to FALSE, VCS issues a warning). The only result of the conformance checking from VCS is the error messages.

VCS performs the following checks:

- Type checking
- Syntactic and semantic checks

### **Type Checks**

VCS checks and verifies the type for generics, restricted variables, timing constraints, delays, and ports.

VITAL\_Level0 timing generics are checked for type and name. The decoded name can only belong to a finite predefined set { tpd, tsetup, thold, trecovery, ...}.

[Table 11-5](#) shows the VITAL delay type names for the generics and the corresponding class for VITAL\_Level0 design units.

**Table 11-5 Delay Type Name and Corresponding Design Unit Class**

| Generic Type Name       | Class                       |
|-------------------------|-----------------------------|
| Time                    | VITAL simple delay type     |
| VitalDelayType          | VITAL simple delay type     |
| VitalDelayArrayType     | VITAL simple delay type     |
| VitalDelayType01        | VITAL transition delay type |
| VitalDelayType01Z       | VITAL transition delay type |
| VitalDelayType01ZX      | VITAL transition delay type |
| VitalDelayArrayType01   | VITAL transition delay type |
| VitalDelayArrayType01Z  | VITAL transition delay type |
| VitalDelayArrayType01ZX | VITAL transition delay type |

VCS checks for the existence of the ports to which the generic refers. For vector subtypes, it checks the index dimensionally.

[Table 11-6](#) contains a list of the predefined timing generics. When VCS finds any port names while checking the generic names, it verifies the type of the generic name.

**Table 11-6 Predefined Timing Generics**

| Prefix Name | Ports                        | VITAL type        |
|-------------|------------------------------|-------------------|
| tpd         | <InPort><OutPort>            | VITAL delay type  |
| tsetup      | <TestPort><RefPort>          | simple delay type |
| thold       | <TestPort><RefPort>          | simple delay type |
| trecovery   | <TestPort><RefPort>          | simple delay type |
| tremoval    | <TestPort><RefPort>          | simple delay type |
| tperiod     | <InPort>                     | simple delay type |
| tpw         | <InPort>                     | simple delay type |
| tskew       | <Port1><Port2>               | simple delay type |
| tncsetup    | <TestPort><RefPort>          | simple delay type |
| tnchold     | <TestPort><RefPort>          | simple delay type |
| tipd        | <InPort>                     | VITAL delay type  |
| tdevice     | <InstanceName>[OutPort]      | VITAL delay type  |
| ticd        | <ClockPort>                  | simple delay type |
| tisd        | <InPort><ClockPort>          | simple delay type |
| tbpd        | <InPort><OutPort><ClockPort> | VITAL delay type  |

VITAL\_level0 control generics are only checked for type as shown in [Table 11-7](#).

*Table 11-7 Type Checks for Control Generics*

| Name           | Type    |
|----------------|---------|
| InstancePath   | String  |
| TimingChecksOn | Boolean |
| Xon            | Boolean |
| MsgOn          | Boolean |

### Syntactic and Semantic Checks

Before conformance checking, VHDL grammar checks are performed. VITAL is a subset of VHDL, so any further checks are actually semantic checks.

### Error Messages

The error messages are grouped into different classes according to the type of error or the hierarchy of error as shown in [Table 11-8](#).

*Table 11-8 Error Message Classes*

| Error Class          | Error Prefix |
|----------------------|--------------|
| Syntax               | VITAL error  |
| Type                 | VITAL error  |
| Context              | VITAL error  |
| Parameter            | VITAL error  |
| Illegal Value        | VITAL error  |
| Entity Error         |              |
| Package              |              |
| Usage                |              |
| Architecture Level 0 |              |
| Architecture Level 1 |              |
| 1. Constraints       |              |
| 2. Delay             |              |

Error messages have the following features:

- Display the description and location information separately.
- Display an error prefix with entity and architecture, type of error, severity level, file name, line number and the offending line from the source.
- Display only user-helpful information.
- Denote the name of the preceding reference as %s. For example, port %s means that the name of the port should appear at the output.
- Are one-liners for grep/awk retrieval from the log file
- Are numbered as follows: E-VTL001, W-VTL002, ...

[Table 11-9](#) and [Table 11-10](#) list all the VITAL error messages. Every message is prefixed with an error class specific message and sufficient context for you to find the problem object. For example, if a port is the offending object, the name of the port and entity are provided. For type violation, the offending type is shown. When there is no indication of what was found, it means that the negation of the statement was found. For example, the error message “The actual part of ... MUST be static” indicates that the type found is not static.

*Table 11-9 VITAL Error Messages for Level 0 Conformance Issues*

| # | Error Class | VITAL Reference<br>Manual section<br>number | Error Message                                                                                                      |
|---|-------------|---------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| 1 | type        | 4.1                                         | The attribute %s { VITAL_Level0, VITAL_Level1 } MUST be declared in package VITAL_Timing and it is declared in %s. |
| 2 | type        | 4.1                                         | The type of the attribute %s { VITAL_Level0, VITAL_Level1 } MUST be Boolean and it is %s.                          |

| #  | Error Class     | VITAL Reference Manual section number | Error Message                                                                                                                          |
|----|-----------------|---------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| 3  | warning         | 4.1                                   | The value of the attribute %s { VITAL_Level0, VITAL_Level1 } MUST be True and it is %s.                                                |
| 4  | scope           | 4.2                                   | %s declared in VITAL package %s cannot have an overloaded outside the package.                                                         |
| 5  | scope           | 4.2.1                                 | Use of foreign architecture body %s for entity %s is prohibited.                                                                       |
| 6  | Not implemented | 4.2.1                                 | The syntactic rule %s, removed in IEEE Std 1076-1993 is illegal in VITAL.                                                              |
| 7  | syntax          | 4.3                                   | The only declaration allowed inside an entity's %s declarative part is VITAL_Level0 attribute declaration.                             |
| 8  | syntax          | 4.3                                   | No statements allowed inside a VITAL entity's %s statement part.                                                                       |
| 9  | semantic        | 4.3.1                                 | Entity %s port %s name CAN NOT contain underscore character(s).                                                                        |
| 10 | semantic        | 4.3.1                                 | Entity %s port %s CAN NOT be of mode LINKAGE.                                                                                          |
| 11 | semantic        | 4.3.1                                 | Entity %s: The type of the scalar port %s MUST be a subtype of Std_Logic. Type is %s.                                                  |
| 12 | semantic        | 4.3.1                                 | Entity %s: The type of vector port %s MUST be Std_Logic_Vector. Type is %s.                                                            |
| 13 | syntax          | 4.3.1                                 | Entity %s port %s CAN NOT be a guarded signal.                                                                                         |
| 14 | semantic        | 4.3.1                                 | Entity %s: a range constraint is not allowed on port %s.                                                                               |
| 15 | semantic        | 4.3.1                                 | Entity %s port %s CAN NOT specify a user defined resolution function.                                                                  |
| 16 | warning         | 4.3.2.1.1                             | Entity %s: No port associated with the timing generic %s. Generic %s unused by VITAL and no check will be performed on it.             |
| 17 | type            | 4.3.2.1.2                             | Entity %s: The type of the scalar generic timing parameter %s does not match the type of associated with a vector port %s.             |
| 18 | type            | 4.3.2.1.2                             | Entity %s: the dimension(s) of the vector timing generic %s does not match that of the associated port %s.                             |
| 19 | type            | 4.3.all                               | The type of the timing generic %s MUST be one of { %s, ... } and it is %s.                                                             |
| 20 | semantic        | 4.3.2.1.3.14                          | Biased propagation delay timing generic %s needs a propagation delay timing generic associated with the same port, condition and edge. |

| #  | Error Class | VITAL Reference Manual section number | Error Message                                                                                                                                                                      |
|----|-------------|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 21 | semantic    | 4.3.2.1.3.14                          | The type %s of biased propagation delay timing generic %s does not match the type %s of the propagation delay timing generic %s associated with the same port, condition and edge. |
| 22 | semantic    | 4.3.3                                 | The type %s of the control generic %s is illegal. Type MUST be %s.                                                                                                                 |
| 23 | semantic    | 4.4.1                                 | Entity %s: Timing generic %s value used before simulation.                                                                                                                         |
| 24 | semantic    | 4.4                                   | Architecture %s { VITAL_Level0, VITAL_Level1 } %s must be associated with a VITAL_Level0 entity.                                                                                   |

Table 11-10 VITAL Error Messages for Level 1 Conformance Issues

| # | Error Class | VITAL Reference Manual section number | Error Message                                                                                                                         |
|---|-------------|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| 1 | semantic    | 6.2                                   | VITAL_GLOBSIG, VERR_USER, MARK<br>Signal '%s' MUST be an entity port or an internal signal.                                           |
| 2 | semantic    | 6.2                                   | VITAL_GLOBSIG, VERR_USER, MARK<br>Signal-valued attribute '%s' is not allowed in a VITAL Level 1 architecture.                        |
| 3 | semantic    | 6.2                                   | It is illegal for a signal %s in architecture %s to have multiple drivers. The drivers are { %s, ... }                                |
| 4 | semantic    | 6.2                                   | Internal signal %s of type %s in architecture %s is illegal. Type can be only of type { Std_ULogic, StdLogic_Vector }. Type is %s.    |
| 5 | semantic    | 6.2                                   | Operators used in a VITAL_Level1 architecture MUST be defined in Std_Logic_1164 . Operator %s is defined in %s.                       |
| 6 | semantic    | 6.2                                   | Subprogram invoked in a VITAL_Level1 architecture MUST be defined in Std_Logic_1164 or VITAL package. Subprogram %s is defined in %s. |

| #  | Error Class | VITAL Reference Manual section number | Error Message                                                                                                                                                                      |
|----|-------------|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 7  | semantic    | 6.2                                   | Formal sub-element association %s in a subprogram call %s is not allowed.                                                                                                          |
| 8  | semantic    | 6.2                                   | Type conversion %s in a subprogram call %s is not allowed.                                                                                                                         |
| 9  | semantic    | 6.4                                   | Multiple wire delay blocks in architecture %s are not allowed. Offending blocks are labeled { %s, ... }. At most one block with a label "WireDelay" is allowed.                    |
| 10 | syntax      | 6.4                                   | Architecture %s body is allowed at most one negative constraint block to compute the internal signal delays declared in entity %s.                                                 |
| 11 | syntax      | 6.4                                   | Architecture %s needs at least one process statement or a concurrent procedure call.                                                                                               |
| 12 | semantic    | 6.4.1                                 | Illegal block label %s. It MUST be "WireDelay."                                                                                                                                    |
| 13 | context     | 6.4.1                                 | Procedure VitalWireDelay MUST be declared in package VITAL_Timing and it is declared in %s.                                                                                        |
| 14 | semantic    | 6.4.1                                 | A call to a VitalWireDelay procedure outside a wire delay block is not allowed.                                                                                                    |
| 15 | semantic    | 6.4.1                                 | At most one wire delay per port of mode IN or INOUT and associated with a wire delay concurrent procedure is allowed inside a wire delay block. Offending signals are { %s, ... }. |
| 16 | semantic    | -                                     | A VITAL predefined name %s CAN NOT be overloaded outside the VITAL package %s.                                                                                                     |
| 17 | semantic    | 6.4.1                                 | Internal wire delayed signal %s representing the wire delay of port %s MUST be the same type as the port.                                                                          |
| 18 | semantic    | 6.4.1                                 | The value of port %s can be read only as an actual part to a wire delay concurrent procedure call.                                                                                 |
| 19 | semantic    | 6.4.1                                 | No range attribute specified for generate statement of a wire delay port %s.                                                                                                       |
| 20 | semantic    | 6.4.1                                 | Only a concurrent procedure call allowed inside an array port %s generate statement.                                                                                               |
| 21 | usage       | 6.4.1                                 | The index for the generate statement %s for the array port %s MUST be the name of the generate parameter %s.                                                                       |

| #  | Error Class | VITAL Reference Manual section number | Error Message                                                                                                                                                                           |
|----|-------------|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 22 | semantic    | 6.4.1                                 | The actual part associated with the input parameter InSig for a wire delay concurrent procedure call MUST be a name of a port of mode IN or INOUT. Offending port %s is of mode %s.     |
| 23 | semantic    | 6.4.1                                 | The actual part associated with the output parameter OutSig for a wire delay concurrent procedure call MUST be a name of an internal signal. The actual part is %s of type %s.          |
| 24 | semantic    | 6.4.1                                 | TWire delay value parameter does not take negative values. Value is %s.                                                                                                                 |
| 25 | semantic    | 6.4.1                                 | The actual part associated with wire delay parameter TWire MUST be locally static or a name of an interconnect delay parameter. Actual part is %s.                                      |
| 26 | semantic    | 6.4.2                                 | VITAL negative constraint block MUST have a label named "SignalDelay." Label is %s.                                                                                                     |
| 27 | semantic    | 6.4.2                                 | Negative constraint %s has no procedure call associated with it and therefore is unused by VITAL.                                                                                       |
| 28 | semantic    | 6.4.2                                 | Negative constraint %s has more than one procedure call { %s, ... } associated with it. Only one procedure call per generic timing parameter is allowed.                                |
| 29 | context     | 6.4.2                                 | Procedure VitalSignalDelay MUST be declared in package VITAL_Timing and it is declared in %s.                                                                                           |
| 30 | semantic    | 6.4.2                                 | A call to VitalSignalDelay is not allowed outside a negative constraint block.                                                                                                          |
| 31 | semantic    | 6.4.2                                 | The actual part associated with the delay value parameter Dly in VitalSignalDelay MUST be a timing generic representing internal signal or internal clock delay. The actual part is %s. |
| 32 | semantic    | 6.4.2                                 | The actual part associated with the input signal parameter S in VitalSignalDelay MUST be a static name denoting an input port or the corresponding wire delay signal (if it exists).    |
| 33 | semantic    | 6.4.2                                 | The actual part associated with the output signal parameter DelayedS MUST be an internal signal.                                                                                        |
| 34 | syntax      | 6.4.3                                 | A VITAL process statement %s MUST have sensitivity list.                                                                                                                                |

| #  | Error Class | VITAL Reference Manual section number | Error Message                                                                                                                                                                                                                                                                         |
|----|-------------|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 35 | context     | 6.4.3                                 | Signal %s CAN NOT appear in the sensitivity list of process %s.                                                                                                                                                                                                                       |
| 36 | semantic    | 6.4.3.1.1                             | Vital <i>unrestricted</i> variable %s MUST be of type { Std_ulogic, Std_logic_vector, Boolean } only. Type is %s.                                                                                                                                                                     |
| 37 | semantic    | 6.4.3.1.1.1                           | The actual part %s of a <i>restricted</i> formal parameter %s MUST be a simple name.                                                                                                                                                                                                  |
| 38 | semantic    | 6.4.3.1.1.1                           | The initial value of the <i>restricted</i> variable %s associated with the <i>restricted</i> formal parameter GlitchData in procedure VitalPathDelay MUST be a VITAL constant or VITAL function with a locally static parameter, but it is %s.                                        |
| 39 | semantic    | 6.4.3.1.1.1                           | The initial value of the <i>restricted</i> variable %s associated with the <i>restricted</i> formal parameter TimingData in procedure %s { VitalSetupHoldCheck, VitalRecoveryRemovalCheck } MUST be a VITAL constant or VITAL function with a locally static parameter, but it is %s. |
| 40 | semantic    | 6.4.3.1.1.1                           | The initial value of the <i>restricted</i> variable %s associated with the <i>restricted</i> formal parameter PeriodPulseData in procedure VitalPeriodPulseCheck MUST be a VITAL constant or VITAL function with a locally static parameter, but it is %s.                            |
| 41 | semantic    | 6.4.3.1.1.1                           | The initial value of the <i>restricted</i> variable %s associated with the <i>restricted</i> formal parameter PreviousDataIn in procedure VitalStateTable can be only a VITAL constant or a VITAL function with a locally static parameter, but it is %s.                             |
| 42 | syntax      | 6.4.3.2                               | A VITAL process statement cannot be empty.                                                                                                                                                                                                                                            |
| 43 | syntax      | 6.4.3.2.1                             | The condition in timing check IF statement MUST be the simple name TimingCheckOn defined in entity %s as a control generic.                                                                                                                                                           |
| 44 | semantic    | 6.4.3.2.1                             | A VITAL timing check statement can be only a call to one of { VITAL_Timing, VITALSetupHoldCheck, VITALRecoveryRemovalCheck, VITALPeriodPulseCheck }.                                                                                                                                  |

| #  | Error Class | VITAL Reference Manual section number | Error Message                                                                                                                                                                                       |
|----|-------------|---------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 45 | semantic    | 6.4.3.2.1                             | The procedure %s { VITAL_Timing,<br>VITALSetupHoldCheck,<br>VITALRecoveryRemovalCheck,<br>VITALPeriodPulseCheck } MUST be declared in package VITAL_Timing, but it is declared in %s.               |
| 46 | semantic    | 6.4.3.2.1                             | A call to %s ( One of { VITAL_Timing(),<br>VITALSetupHoldCheck(),<br>VITALRecoveryRemovalCheck(),<br>VITALPeriodPulseCheck() } ) occurred outside a timing check section.                           |
| 47 | semantic    | 6.4.3.2.1                             | The actual part %s associated with the formal parameter %s (representing a signal name %s) MUST be locally static.                                                                                  |
| 48 | semantic    | 6.4.3.2.1                             | The actual %s associated with the formal parameter HeaderMsg MUST be a globally static expression.                                                                                                  |
| 49 | semantic    | 6.4.3.2.1                             | The actual %s of the timing check procedure %s associated with a formal parameter %s of type Time MUST be a locally static expression or simple name denoting the control generic of the same name. |
| 50 | semantic    | 6.4.3.2.1                             | The actual %s associated with a formal parameter %s { XOn, MsgOn } MUST be a globally static expression.                                                                                            |
| 51 | semantic    | 6.4.3.2.1                             | A function %s call or an operator %s invocation in the actual part to a formal parameter %s MUST be a function/operator defined in one of packages { Standard, Std_logic_1164, VITAL_Timing }.      |
| 52 | semantic    | 6.4.3.2.1                             | The actual %s associated with the formal parameter %s { TestSignalName } MUST be locally static expression.                                                                                         |
| 53 | context     | 6.4.3.2.1                             | variable %s associated with a timing check violation parameter %s could not be used in another timing check statement. It appears in timing check %s.                                               |
| 54 | context     | 6.4.3.2.2                             | procedure VitalStateTable() MUST be declared in the package VITAL_Primitives, but it is declared in %s.                                                                                             |
| 55 | semantic    | 6.4.3.2.2                             | Only a call to the predefined procedure VitalStateTable() is allowed inside a VITAL functionality section.                                                                                          |

| #  | Error Class | VITAL Reference Manual section number | Error Message                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|----|-------------|---------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 56 | semantic    | 6.4.3.2.2                             | The actual %s associated with the StateTable parameter to procedure VitalStateTable MUST be globally static expression.                                                                                                                                                                                                                                                                                                                           |
| 57 | semantic    | 6.4.3.2.2                             | The index constraint on the variable %s associated with the PreviousDataIn parameter MUST match the constraint on the actual associated with the DataIn parameter.                                                                                                                                                                                                                                                                                |
| 58 | semantic    | 6.4.3.2.2                             | The target of a VITAL variable assignment MUST be <i>unrestricted</i> variable denoted by a locally static name, but it is %s.                                                                                                                                                                                                                                                                                                                    |
| 59 | type        | 6.4.3.2.2                             | The target of an assignment statement of a standard logic type inside a functionality section requires a primary on the right side to be one of the following:<br>1. A globally static expression<br>2. A name of a port or an internal signal<br>3. A function call to a standard logic function, a VITAL primitive or VITALTruthTable()<br>4. An aggregate or a qualified expression with an aggregate operand<br>5. A parenthesized expression |
| 60 | semantic    | 6.4.3.2.2                             | A call to function VITALTruthTable CAN NOT occur outside VITAL functionality section.                                                                                                                                                                                                                                                                                                                                                             |
| 61 | semantic    | 6.4.3.2.3                             | The procedure %s { VITALPathDelay, VITALPathDelay01, VITALPathDelay01Z } MUST be defined in package VITAL_Timing, but it is defined in %s.                                                                                                                                                                                                                                                                                                        |
| 62 | semantic    | 6.4.3.2.3                             | A call to procedure %s { VITALPathDelay, VITALPathDelay01, VITALPathDelay01Z } CAN NOT occur outside a path delay section.                                                                                                                                                                                                                                                                                                                        |
| 63 | semantic    | 6.4.3.2.3                             | The actual part associated with the formal parameter OutSignal of a path delay procedure %s { VITALPathDelay, VITALPathDelay01, VITALPathDelay01Z } MUST be a locally static signal name, but it is %s.                                                                                                                                                                                                                                           |
| 64 | semantic    | 6.4.3.2.3                             | The actual part associated with the formal parameter Paths of a path delay procedure %s { VITALPathDelay, VITALPathDelay01, VITALPathDelay01Z } MUST be an aggregate, but it is %s.                                                                                                                                                                                                                                                               |

| #  | Error Class | VITAL Reference Manual section number | Error Message                                                                                                                                                                                                                        |
|----|-------------|---------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 65 | semantic    | 6.4.3.2.3                             | The sub-element PathDelay of the actual part associated with the formal parameter Paths to a path delay procedure %s { VITALPathDelay, VITALPathDelay01, VITALPathDelay01Z } MUST be globally static, but it is %s.                  |
| 66 | semantic    | 6.4.3.2.3                             | The sub-element InputChangeTime of the actual part associated with the formal parameter Paths %s { VITALPathDelay, VITALPathDelay01, VITALPathDelay01Z } MUST be a LastEvent attribute or a locally static expression, but it is %s. |
| 67 | semantic    | 6.4.3.2.3                             | The actual part associated with the formal parameter GlitchMode to a path delay procedure %s MUST be a literal, but it is %s.                                                                                                        |
| 68 | semantic    | 6.4.3.2.3                             | The actual part associated with the formal parameter GlitchData MUST be a locally static name, but it is %s.                                                                                                                         |
| 69 | semantic    | 6.4.3.2.3                             | The actual part associated with the formal parameter %s { Xon, MsgOn } MUST be a locally static expression or a simple name denoting control generic of the same name, but it is %s.                                                 |
| 70 | semantic    | 6.4.3.2.3                             | The actual part associated with the formal parameter %s { OutSignalName, DefaultDelay, OutputMap } MUST be a locally static expression.                                                                                              |
| 71 | No Check    | 6.4.3.2.3                             | Port of type %s { OUT, INOUT, BUFFER } has to be driven by a VITAL primitive procedure call or a path delay procedure, but the driver is %s.                                                                                         |
| 72 | semantic    | 6.4.4                                 | The actual part associated with the formal parameter %s of class VARIABLE or SIGNAL on VITAL primitive %s MUST be a static name, but it is %s.                                                                                       |
| 73 | semantic    | 6.4.4                                 | The actual part associated with the formal parameter %s of class CONSTANT to a procedure call %s MUST be a locally static expression, but it is %s.                                                                                  |
| 74 | semantic    | 6.4.4                                 | The actual part associated with the formal parameter ResultMap to a procedure call %s MUST be a locally static expression, but it is %s.                                                                                             |

| #  | Error Class | VITAL Reference Manual section number | Error Message                                                                                                                                                                                                                      |
|----|-------------|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 75 | semantic    | 6.4.4                                 | The actual part associated with the formal parameter %s { TruthTable, StateTable } on table primitive procedure call %s MUST be a constant whose value expression is an aggregate with fields that are locally static expressions. |
| 76 | No Check    | 7.1.1                                 | VITAL logic primitive %s MUST be defined in package %s.                                                                                                                                                                            |
| 77 | No Check    | 7.3.1                                 | Symbol %s CAN NOT appear in Table %s.                                                                                                                                                                                              |
| 78 | No Check    | 7.3.3.1                               | Wrong number of inputs to an object %s of type VitalTruthTable. The number MUST equal to the value of the DataIn parameter VitalTruthTable.                                                                                        |
| 79 | No Check    | 7.3.3.1                               | Wrong dimensions for table %s of type %s { VitalTruthTable, VitalStateTable }.                                                                                                                                                     |
| 80 | Package     | 7.4.3.2.2                             | procedure VitalStateTable() MUST be declared in VitalPrimitives, but it is declared in %s.                                                                                                                                         |
| 81 | Package     | 7.4.3.2.3                             | procedure %s { VITALPathDelay, VITALPathDelay01, VITALPathDelay01Z } MUST be defined in package VITAL_Timing, but it is defined in %s.                                                                                             |

---

## Support for Identifying Non-Annotated Timing Arc and Timing Check Statements

VCS provides additional diagnostics under the `-diag=sdf` sub-option that lists down all the non-annotated timing arcs and paths, mentioned in the specify block of cell library by SDF file.

The syntax is as follows:

```
%vcs -diag=sdf:verbose <.....>
```

---

## Usage Example

The following example illustrates the usage of timing check:

```
`timescale 1ns/1ns
module test();
wire q;
reg d, clk;
mydff u(q,clk,d);
mydff u1(q,clk,d);

initial begin
    $sdf_annotate("example.sdf",,,,) ;
    $monitor("%4d",$time," q=%b d=%b clk=%b",q,d,clk) ;
    d=0;
    clk=0;
#20 d = 1;
#40 d = 1'bx;
    #200 $finish;
end
always
#20 clk = ~clk;

endmodule

(DELAYFILE
 (SDFVERSION "OVI 2.1")
 (DESIGN "test")
 (DIVIDER /)
 (TIMESCALE 1ns)
 (CELL
 (CELLTYPE "test")
 (INSTANCE )
 (DELAY
 (ABSOLUTE
 (IOPATH test/u/din test/u/qout (5)(3))
 (IOPATH test/u/clk test/u/qout (5)(3))
 )
 )
 )
```

```

        (TIMINGCHECK
            (SETUPHOLD test/u1/din (posedge test/u1/clk) (5:10:15)
(5:10:15))
        )
    )
)

```

To run the example, use the following commands:

```

% vcs -diag=sdf:verbose example.v
% simv
% cat sdfAnnotateInfo

```

The following is the output:

```

Static entries in elaborated design under "test":
    Annotated by SDF "example.sdf":

        No. of Pathdelays      = 4  Annotated = 50.00%
        No. of Tchecks         = 4  Annotated = 50.00%

        Total          Annotated Percentage
IOPATH           4                  2      50.00%

        Path Delays Summary of above

        SETUPHOLD           4                  2      50.00%

        Timing checks Summary of above

OverAll Static entries in elaborated design:
    No. of Pathdelays      = 4  Annotated = 50.00%
    No. of Tchecks         = 4  Annotated = 50.00%

        Total          Annotated Percentage
IOPATH           4                  2      50.00%

        Path Delays Summary of above

```

|           |   |   |        |
|-----------|---|---|--------|
| SETUPHOLD | 4 | 2 | 50.00% |
|-----------|---|---|--------|

Timing checks Summary of above

OverAll Static entries in elaborated design:

|                   |     |                    |
|-------------------|-----|--------------------|
| No. of Pathdelays | = 4 | Annotated = 50.00% |
| No. of Tchecks    | = 4 | Annotated = 50.00% |

| IOPATH | Total | Annotated | Percentage |
|--------|-------|-----------|------------|
|        | 4     | 2         | 50.00%     |

Path Delays Summary of above

|           |   |   |        |
|-----------|---|---|--------|
| SETUPHOLD | 4 | 2 | 50.00% |
|-----------|---|---|--------|

Timing checks Summary of above

```
# ***** Design Not Annotated ***** #
```

```
Module Name: mydff, Instance Name: test.u
$setuphold(posedge clk, din, 3, 3);
```

```
Module Name: mydff, Instance Name: test.u1
(din => qout) = (4);
(clk => qout) = (4);
```

In this example there are two instances of mydff, which are u, u1 respectively.

In SDF file, IOPATH is specified only for instance u and not for u1.

```
(IOPATH test/u/din test/u/qout (5) (3))
(IOPATH test/u/clk test/u/qout (5) (3))
```

Hence in sdfAnnotateInfo file, you can see that IOPATH is 50% annotated.

No. of Pathdelays = 4 Annotated = 50.00%

No. of Tchecks = 4 Annotated = 50.00%

Under the section, "\*\*\*\* Design Not Annotated \*\*\*", you can find more details.

```
Module Name: mydff,      Instance Name: test.ul
              (din => qout) = (4);
              (clk => qout) = (4);
```

Note:

If the minimum value is not mentioned in SDF and when VCS takes a default minimum value, then there are entries `sdfAnnotateInfo` file in the case of typical delay.



# 12

## Coverage

---

VCS monitors the execution of the HDL code during simulation. Verification engineers can determine which part of the code has not been tested yet so that they can focus their efforts on those areas to achieve 100% coverage. VCS offers two coverage techniques to test your HDL code: Code coverage and Functional coverage.

This chapter consists of the following sections:

- “Code Coverage”
- “Functional Coverage”
- “Options For Coverage Metrics”

---

## Code Coverage

The following coverage metrics are classified as code coverage:

- Line Coverage — This metric measures statements in your HDL code that have been executed in the simulation.
- Toggle Coverage — This metric measures the bits of logic that have toggled during simulation. A toggle simply means that a bit changes from 0 to 1 or from 1 to 0. It is one of the oldest metrics of coverage in hardware designs and can be used at both the register transfer level (RTL) and gate level.
- Condition Coverage — This metric measures how the variables or sub-expressions in the conditional statements are evaluated during simulation. It can find errors in the conditional statements that cannot be found by other coverage analysis.
- Branch Coverage — This metric measures the coverage of expressions and case statements that affect the control flow (such as if-statement and while-statement) of the HDL. It focuses on the decision points that affect the control flow of the HDL execution.
- FSM Coverage — This metric verifies that every legal state of the state machine has been visited and that every transition between states has been covered.

For more information about coverage technology and how you can generate the coverage information for your design, see the *Coverage Technology User Guide* in the VCS Online Documentation.

---

## Functional Coverage

Functional coverage checks the overall functionality of the implementation. To perform functional coverage, you must define coverage points for the functions to be covered in the DUT. VCS supports both Native Testbench (NTB) and SystemVerilog covergroup models. Covergroups are specified by users. They allow the system to monitor values and transitions for variables and signals. They also enable cross coverage between variables and signals.

For more information about NTB or SystemVerilog functional coverage models, see the Testbench category in the VCS Online Documentation and see Chapter 19, “Functional Coverage” in the *SystemVerilog LRM IEEE Std. 1800 - 2012* respectively.

---

## Options For Coverage Metrics

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

Specifies elaborating for the specified type or types of coverage. The argument specifies the types of coverage:

`line`

Elaborate for line or statement coverage.

`cond`

Elaborate for condition coverage.

`fsm`

Elaborate for FSM coverage.

tgl

Elaborate for toggle coverage.

branch

Elaborate for branch coverage

assert

Elaborate for SystemVerilog assertion coverage.

For more information on Coverage options, see the *Coverage Technology User Guide* in the VCS *Online Documentation*.

# 13

## Using OpenVera Native Testbench

---

OpenVera Native Testbench is a high-performance, single-kernel technology in VCS that enables:

- Native compilation of testbenches written in OpenVera and in SystemVerilog.
- Simulation of these testbenches along with the designs.

This technology provides a unified design and verification environment in VCS for significantly improving overall design and verification productivity. Native Testbench is uniquely geared towards efficiently catching hard-to-find bugs early in the design cycle, enabling not only completing functional validation of designs with the desired degree of confidence, but also achieving this goal in the shortest time possible.

Native Testbench is built around the preferred methodology of keeping the testbench and its development separate from the design. This approach facilitates development, debug, maintenance and reusability of the testbench, as well as ensuring a smooth synthesis flow for your design by keeping it clean of all testbench code. Further, you have the choice of either compiling your testbench along with your design or separate from it. The latter choice not only saves you from unnecessary recompilations of your design, it also enables you to develop and maintain multiple testbenches for your design.

This chapter describes the high-level, object-oriented verification language of OpenVera, which enables you to write your testbench in a straightforward, elegant and clear manner and at a high level essential for a better understanding of and control over the design validation process. Further, OpenVera assimilates and extends the best features found in C++ and Java along with syntax that is a natural extension of the hardware description languages (Verilog and VHDL). Adopting and using OpenVera, therefore, means a disciplined and systematic testbench structure that is easy to develop, debug, understand, maintain and reuse.

Thus, the high-performance of Native Testbench technology, together with the unique combination of the features and strengths of OpenVera, can yield a dramatic improvement in your productivity, especially when your designs become very large and complex.

This chapter includes the following topics:

- “Usage Model”
- “Key Features”

---

# Usage Model

As any other VCS applications, the usage model to simulate OpenVera testbench includes the following steps:

For two-step flow:

## Compilation

```
% vcs [ntb_options] [compile_options] file1.vr file2.vr  
      file3.v file4.v
```

## Simulation

```
% simv [run_options]
```

For three-step flow:

## Analysis

Always analyze Verilog before VHDL.

```
% vlogan -ntb [vlogan_options] file1.vr file2.vr file3.v  
% vhdlan [vhdlan_options] file3.vhd file4.vhd
```

## Note:

Specify the VHDL bottommost entity first, and then move up in order.

## Elaboration

```
% vcs [other_ntb_options] [compile_options] design_unit
```

## Simulation

```
% simv [run_options]
```

---

## Example

In this example, you have an interface file, a Verilog design `arb.v`, OpenVera testbench `arb.vr`, all instantiated in a Verilog top file, `arb.test_top.v`.

```
//Interface
#ifndef INC_ARB_IF_VRH
#define INC_ARB_IF_VRH

interface arb {
    input clk CLOCK;
    output [1:0]      request OUTPUT_EDGE OUTPUT_SKew;
    output          reset OUTPUT_EDGE OUTPUT_SKew;
    input [1:0] grant  INPUT_EDGE  INPUT_SKew;
} // end of interface arb

#endif

//Verilog module: arb.v
module arb ( clk, reset, request, grant) ;
    input [1:0] request ;
    output [1:0] grant ;
    input reset ;
    input clk ;

parameter IDLE = 2, GRANT0 = 0, GRANT1 = 1;

reg last_winner ;
reg winner ;
reg [1:0] grant ;
reg [1:0] next_grant ;

reg [1:0] state, nxState;

...
endmodule
```

```

//OpenVera Testbench: arb.vr

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

#include "arb.if.vrh"

program arb_test
{ // start of top block

    ...

} // end of program arb_test

```

**Note:**

You can find the complete example in the following path:

\$VCS\_HOME/doc/examples/testbench/ov/Tutorial/  
arb

In this example, we have an interface file, a Verilog design, `design.v` instantiated in a VHDL `top.vhd`. Testbench is in OpenVera.

```

//Interface: verilog_mod.if.vrh
interface verilog_mod {
    input      clk      CLOCK ;
    output din_single  PHOLD #1 ;
    output [7:0] din_vector PHOLD #1 ;
    input   dout_wire_single PSAMPLE #-1;
    input   idout_wire_single PSAMPLE #-1 hdl_node "/top/
dout_wire_single" ;
    input [7:0] dout_wire_vector PSAMPLE #-1;
    input [7:0] idout_wire_vector PSAMPLE #-1 hdl_node

```

```

"/top/dout_wire_vector" ;
    input dout_reg_single PSAMPLE #-1;
    input idout_reg_single PSAMPLE #-1 hdl_node "/top/
dout_reg_single";
    input [7:0] dout_reg_vector PSAMPLE #-1;
    input [7:0] idout_reg_vector PSAMPLE #-1 hdl_node "/
top/dout_reg_vector" ;
} // end of interface verilog_mod

//Verilog module: design.v

module verilog_mod1 (
clk,din_single,din_vector,dout_wire_single,dout_reg_single
,dout_wire_vector,dout_reg_vector
) ;
input clk;
input din_single;
input [7:0] din_vector;
output dout_wire_single ;
output dout_reg_single ;
output dout_wire_vector ;
output [7:0] dout_reg_vector ;
...
endmodule

-- VHDL Top: top.vhd
...
entity top is
    generic (
        EMU : boolean := false);
end top;

architecture vhdl_top of top is

component verilog_mod1
port (
    clk : IN std_logic ;
    din_single : IN std_logic ;
    din_vector : IN std_logic_vector(7 downto 0) ;
    dout_wire_single : OUT std_logic ;
    dout_wire_vector : OUT std_logic_vector(7 downto 0) ;
    dout_reg_single : OUT std_logic ;

```

```

        dout_reg_vector : OUT std_logic_vector(7 downto 0)
    );
end component;
...

begin -- ntbmx_test
    ...
    vshell: test
        port map (SystemClock => SystemClock,
                   \verilog_mod.clk\ => clk,
                   \verilog_mod.din_single\ => din_single,
                   ...
        );
    ...
end vhdl_top;

//OpenVera Testbench: test.vr

#include <veraDefines.vrh>
#define MAX_COUNT 10
#include "interface.if"
...

program test {
    integer i ;
    bit b ;
    integer n ;

    force_it_p fp ;
    ...
}

}

```

**Note:**

You can find the complete example in \$VCS\_HOME/doc/examples/nativetestbench/mixedhdl/testcase\_2

## Usage Model

For two-step flow:

### Compilation

```
% vcs -ntb arb.v arb.vr arb.test_top.v
```

### Simulation

```
% simv
```

For three-step flow

### Analysis

```
% vlogan -ntb test.vr design.v  
% vhdlan top.vhd
```

### Note:

Specify the VHDL bottom-most entity first, and then move up in order.

### Elaboration

```
% vcs top
```

### Simulation

```
% simv
```

## Importing VHDL Procedures

VHDL procedures can be imported into the NTB domain using the `hdl_task` statement:

```
hdl_task OpenVera_name ([parameters])  
"vhdl_task [lib].[package].[VHDL_name]"
```

The only difference to the OpenVera `hdl_task` syntax is that NTB requires the `vhdl_task` keyword. This keyword is required because NTB must be able to distinguish between Verilog and VHDL procedures at analysis time (`vlogan`). The `[lib]`, `[package]` and `[VHDL_name]` entries must point to the VHDL library and package where the `[VHDL_name]` procedure are described. The VHDL procedures are best described in packages so that they can be accessed globally.

The parameters of the VHDL procedure can be of `in`, `out` or `inout` type and are mapped between the OpenVera and VHDL type by use of the global `-ntb_opts sigtype=[type]` command-line option to `vlogan`:

*Table 13-1 Mapping OpenVera and VHDL Datatypes*

| OpenVera<br>data type | VHDL data type    | sigtype    |
|-----------------------|-------------------|------------|
| bit                   | STD_LOGIC         | STD_LOGIC  |
| bit[N-1:0]            | STD_LOGIC_VECTOR  |            |
| bit                   | STD_ULOGIC        | STD_ULOGIC |
| bit[N-1:0]            | STD_ULOGIC_VECTOR | (default)  |
| bit                   | BIT               | BIT        |
| bit[N-1:0]            | BIT_VECTOR        |            |
| bit[N-1:0]            | SIGNED            | SIGNED     |
| bit[N-1:0]            | UNSIGNED          | UNSIGNED   |
| bit[N-1:0]            | INTEGER           | INTEGER    |
| bit                   | BOOLEAN           | BOOLEAN    |
| integer               | INTEGER           | any        |

Note that this flow is limited to one global signal type, so all parameters of all imported and exported type must be the same base `ntb_sigtype`, for example, `STD_LOGIC` and `STD_LOGIC_VECTOR`.

If two or more concurrent calls to an imported procedure can occur, the later one is queued and executed when the procedure is free again. Although this matches OpenVera behavior, the timing shift is probably not what you intended. The solution to this problem is the `-ntb_opts task_import_poolsize=[size]` option to `vlogan`. Here you can define the maximum number of imported tasks or procedures that can be called in parallel without blocking.

## Exporting OpenVera Tasks

OpenVera tasks can be exported into the VHDL and Verilog domains using the `export` keyword in the task definition.

For using the function in VHDL, `vlogan` creates a VHDL wrapper package named `[OpenVera program name]_pkg`. This package is automatically compiled into the `WORK` library. The VHDL part of the design can thus call the OpenVera task in any process that has no sensitivity list. As a prerequisite, the calling entity only needs to include the corresponding “use” statement:

```
use work.[OpenVera program name]_pkg.all;
```

The mapping of the OpenVera and VHDL data types is defined by the `-ntb_opts sigtype=[type]` command-line option as described earlier. The `-ntb_opts task_export_poolsize` command-line option can be used to increase the maximum number of concurrent calls to exported tasks. Note, however that in contrast to the imported tasks, exceeding this limit can cause a runtime error of the simulation.

## Example:

```
---- start OpenVera code fragment ----
export task vera_decrement (var bit[31:0] count)
{
    count = count - 1;
}

program my_testbench
{ ...
    ---- end OpenVera code fragment ----

task automatic vera_decrement ( inout reg [31:0] count) ...
```

The corresponding VHDL procedure named `vera_decrement` is created in `my_testbench_pkg` package and analyzed into the WORK library.

## Using Template Generator

To ease the process of writing a testbench in OpenVera, VCS provides you with a testbench template generator. The template generator supports both a Verilog and a VHDL top design.

Use the following command to invoke the template generator on a Verilog or VHDL design unit:

```
% ntb_template -t design_module_name [-c clock] design_file\  
[-vcs vcs_compile-time_options]
```

Where:

`-t design_module_name`

Specifies the top-level design module name.

`design_file`

Name of the design file.

-c

Specifies the clock input of the design. Use this option only if the specified *design\_file* is a Verilog file.

-template

Can be omitted.

-program

Optional. Use it to specify program name.

-simcycle

Optional. Use this to override the default cycle value of 100.

-vcs *vcs\_compile-time\_options*

Optional. Use it to supply a VCS compile-time option. Multiple *-vcs vcs\_compile-time\_options* options can be used to specify multiple options. Use this option only for Verilog on top designs.

## Example

An example SRAM model is used in this demonstration of using the template generator to develop a testbench environment.

For details on the OpenVera verification language, refer to the *OpenVera Language Reference Manual: Native Testbench*.

## Design Description

The design is an SRAM whose RTL Verilog model is in the file *sram.v*. It has four ports:

- ce\_N (chip enable)
- rdWr\_N (read/write enable)
- ramAddr (address)
- ramData (data)

**Example 13-1 RTL Verilog Model of SRAM in sram.v**

```
module sram(ce_N, rdWr_N, ramAddr, ramData);

    input ce_N, rdWr_N;
    input [5:0] ramAddr;
    inout [7:0] ramData;
    wire [7:0] ramData;
    reg [7:0] chip[63:0];

    assign #5 ramData = (~ce_N & rdWr_N) ? chip[ramAddr] :
8'bzzzzzzz;

    always @ (ce_N or rdWr_N)
    begin
        if(~ce_N && ~rdWr_N)
            #3 chip[ramAddr] = ramData;
    end
endmodule
```

During a read operation, when `ce_N` is driven low and `rdWr_N` is driven high, `ramData` is continuously driven from inside the SRAM with the value stored in the SRAM memory element specified by `ramAddr`. During a write operation, when both `ce_N` and `rdWr_N` are driven low, the value driven on `ramData` from outside the SRAM is stored in the SRAM memory element specified by `ramAddr`. At all other times, `ce_N` is driven high, and as a result, `ramData` gets continuously driven from inside the SRAM with the high-impedance value `Z`.

## **Generating the Testbench Template, the Interface, and the Top-level Verilog Module from the Design**

As previously mentioned, Native Testbench provides a template generator to start the process of constructing a testbench. The template generator is invoked on `sram.v` as shown below:

```
% ntb_template -t sram sram.v
```

Where:

- The `-t` option is followed with the top-level design module name, which is `sram`, in this case.
- `sram` is the name of the module.
- `sram.v` is the name of the file containing the top-level design module.
- If the design uses a clock input, then the `-c` option is to be used and followed with the name of the clock input. Doing so provides a clock input derived from the system-clock for the interface and the design. In this example, there is no clock input required by the design.

Template generator generates the following files:

- `sram.vr.tmp`
- `sram.if.vrh`
- `sram.test_top.v`

### **`sram.vr.tmp`**

This is the template for testbench development. The following is an example, based on the `sram.v` file of the output of the previous command line:

```

//sram.vr.tmp
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_SKEW #-1
#define INPUT_EDGE PSAMPLE
#include <veraDefines.vrh>

// define interfaces, and verilog_node here if necessary

#include "sram.if.vrh"

// define ports, binds here if necessary

// declare external tasks/classes/functions here if
//necessary

// declare verilog_tasks here if necessary

// declare class typedefs here if necessary

program sram_test
{ // start of top block

    // define global variables here if necessary

    // Start of sram_test

    // Type your test program here:

    //

    // Example of drive:
    // @1 sram.ce_N = 0 ;
    //
    //
    // Example of expect:
    // @1,100 sram.example_output == 0 ;
    //

} // end of program sram_test

// define tasks/classes/functions here if necessary

```

### **sram.if.vrh**

This is the interface file which provides the basic connectivity between your testbench signals and your design's ports and/or internal nodes. All signals going back and forth between the testbench and the design go through this interface. The following is the `sram.if.vrh` file which results from the previous command line:

```
//sram.if.vrh
#ifndef INC_SRAM_IF_VRH
#define INC_SRAM_IF_VRH
    interface sram {
        output           ce_N      OUTPUT_EDGE OUTPUT_SKew;
        output           rdWr_N   OUTPUT_EDGE OUTPUT_SKew;
        output [5:0]     ramAddr  OUTPUT_EDGE OUTPUT_SKew;
        inout [7:0]      ramData  INPUT_EDGE  INPUT_SKew
OUTPUT_EDGE OUTPUT_SKew;
    } // end of interface sram

#endif
```

Notice that, for example, the direction of `ce_N` is now "output" instead of "input". The signal direction specified in the interface is from the point of view of the testbench and not the DUT.

This file must be modified to include the clock input.

### **sram.test\_top.v**

This is the top-level Verilog module that contains the testbench instance, the design instance, and the system-clock. The system clock can also provide the clock input for both the interface and the design. The following is the `sram.test_top.v` file that results from the previous command line:

```
//sram.test_top.v
module sram_test_top;
```

```

parameter simulation_cycle = 100;

reg SystemClock;

wire ce_N;
wire rdWr_N;
wire [5:0] ramAddr;
wire [7:0] ramData;

`ifdef SYNOPSYS_NTB
sram_test vshell(
    .SystemClock (SystemClock),
    .\sram.ce_N (ce_N),
    .\sram.rdWr_N (rdWr_N),
    .\sram.ramAddr (ramAddr),
    .\sram.ramData (ramData)
);
`else

vera_shell vshell(
    .SystemClock (SystemClock),
    .sram_ce_N (ce_N),
    .sram_rdWr_N (rdWr_N),
    .sram_ramAddr (ramAddr),
    .sram_ramData (ramData)
);
`endif

`ifndef emu
/* DUT is in emulator, so not instantiated here */
`else
sram dut(
    .ce_N (ce_N),
    .rdWr_N (rdWr_N),
    .ramAddr (ramAddr),
    .ramData (ramData)
);
`endif

initial begin
    SystemClock = 0;
    forever begin
        #(simulation_cycle/2)

```

```

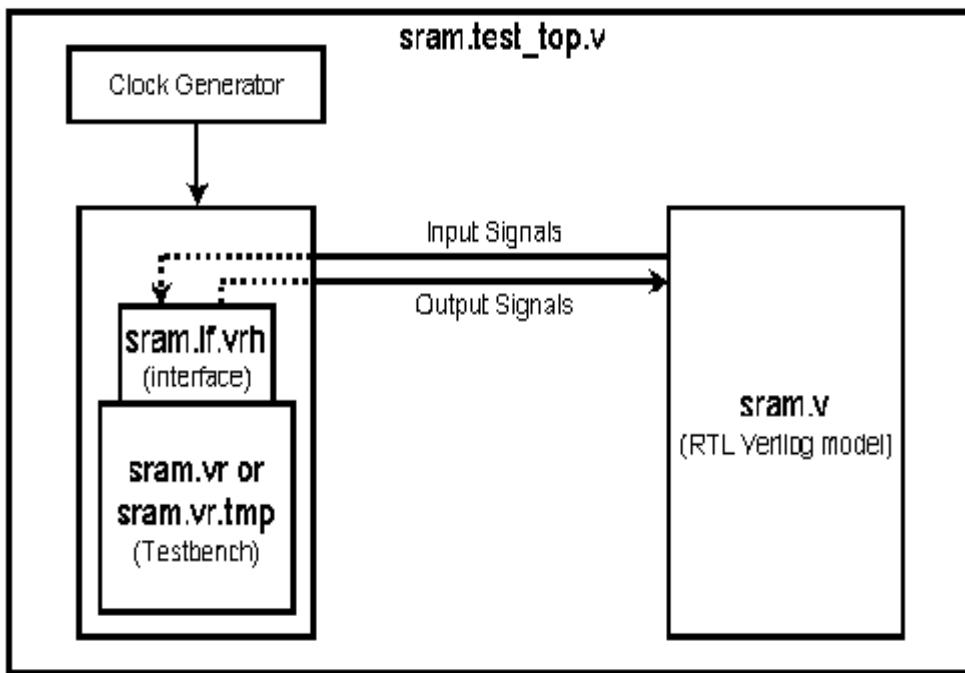
        SystemClock = ~SystemClock;
    end
end

endmodule

```

[Figure 13-1](#) shows how the three generated files and the design connect and fit in with each other in the final configuration.

*Figure 13-1 Testbench and Design Configuration*



### Testbench Development and Description

Your generated testbench template, `sram.vr.tmp`, contains a list of macro definitions for the interface, include statements for the interface file and the library containing predefined tasks and functions, comments indicating where to define or declare the various parts of the testbench, and the skeleton program shell that

will contain the main testbench constructs. Starting with this template, you can develop a testbench for the SRAM and rename it `sram.vr`. An example testbench is shown in [Example 13-2](#).

### **Example 13-2 Example testbench for SRAM, sram.vr**

```
// macro definitions for Interface signal types and skews
#define OUTPUT_EDGE PHOLD      // for specifying posedge-drive type
#define OUTPUT_SKEW #1        // for specifying drive skew
#define INPUT_SKEW #-1        // for specifying sample skew
#define INPUT_EDGE PSAMPLE    // for specifying posedge-sample type

#include <vera_defines.vrh>      // include the library of predefined
                                  // functions and tasks
#include "sram.if.vrh"           // include the Interface file

program sram_test {               // start of program sram_test

    reg [5:0] address = 6'b000_0001; // declare, initialize address (for
                                      // driving ramAddr during Write and
                                      // Read)
    reg [7:0] rand_bits;           // declare rand_bits (for driving
                                  // ramData during Write)
    reg [7:0] data_result;         // declare data_result (for receiving
                                  // ramData during Read)

    @(posedge sram.clk);
    rand_bits = random();          // move to the first posedge of clock
                                   // initialize rand_bits with a random
                                   // value using the random() function

    @1 sram.ramAddr = address;     // move to the next posedge of clock,
                                   // drive ramAddr with the value of
                                   // address
    sram.ce_N = 1'b1;              // disable SRAM by driving ce_N high
    sram.ramData = rand_bits;       // drive ramData with rand_bits and
                                   // keep it ready for a Write
    sram.rdWr_N = 1'b0;            // drive rdWr_N low and keep it ready
                                   // for a Write

    @1 sram.ce_N = 1'b0;           // move to the next posedge of clock,
                                   // and enable a SRAM Write by driving
                                   // ce_N low
    printf("Cycle: %d Time: %d \n", get_cycle(), get_time(0));
    printf("The SRAM is being written at ramAddr: %b Data written: %b \n", address,
          sram.ramData);

    @1 sram.ce_N = 1'b1;           // move to the next posedge of clock,
```

```

        // disable SRAM by driving ce_N high
sram.rdWr_N = 1'b1;
        // drive rdWr_N high and keep it ready
        // for a Read
sram.ramData = 8'bzzzz_zzzz;
        // drive a high-impedance value on
        // ramData

@1 sram.ce_N = 1'b0;                      // move to the next posedge of clock,
  // enable a SRAM Read by driving ce_N
  // low

@1 sram.ce_N = 1'b1;                      // move to the next posedge of clock,
  // disable SRAM by driving ce_N high
data_result = sram.ramData;                // sample ramData and receive the data
  // from SRAM in data_result
printf("Cycle: %d Time: %d\n", get_cycle(), get_time(0));
printf("The SRAM is being read    at ramAddr: %b Data read : %b \n", address,
data_result);

} // end of program sram_test

```

The main body of the testbench is the program, which is named `sram_test`. The program contains three data declarations of type `reg` in the beginning. It then moves execution through a Write operation first and then a Read operation. The memory element of the SRAM written to and read from is `6'b 00_0001`. The correct functioning of the SRAM implies data that is stored in a memory element during a Write operation must be the same as that which is received from the memory element during a Read operation later. The example testbench only demonstrates how any memory element can be functionally validated. For complete functional validation of the SRAM, the testbench would need further development to cover all memory elements from `6'b00_0000` to `6b'11_1111`.

## Interface Description

The generated `if.vrh` file has to be modified to include the clock input. The modified interface is shown in [Example 13-3](#).

## Interface for SRAM, sram.if.vrh

### Example 13-3

```
#ifndef INC_SRAM_IF_VRH
#define INC_SRAM_IF_VRH

interface sram {
    input             clk      CLOCK; // add clock
    output            ce_N    OUTPUT_EDGE OUTPUT_SKew;
    output            rdWr_N  OUTPUT_EDGE OUTPUT_SKew;
    output [5:0]       ramAddr OUTPUT_EDGE OUTPUT_SKew;
    inout [7:0]       ramData INPUT_EDGE  OUTPUT_EDGE OUTPUT_SKew;
} // end of interface sram
#endif
```

The interface consists of signals that are either driven as outputs into the design or sampled as inputs from the design. The clock input, `clk`, is derived from the system clock in the top-level Verilog module.

## Top-level Verilog Module Description

The generated top-level module has been modified to include the clock input for the interface and eliminate code that was not relevant. The clock input is derived from the system clock. [Example 13-4](#) shows the modified top-level Verilog module for the SRAM.

### Example 13-4 Top-level Verilog Module, sram.test\_top.v

```
module sram_test_top;
    parameter simulation_cycle = 100;
    reg             SystemClock;
    wire            ce_N;
    wire            rdWr_N;
    wire [5:0]      ramAddr;
    wire [7:0]      ramData;
    wire            clk = SystemClock; /* Add this line. Interface
   clock input derived from the system clock*/
`ifdef SYNOPSYS_NTB
sram_test vshell(
    .SystemClock (SystemClock),
    .\sram.clk(clk),
```

```

    .\sram.ce_N (ce_N),
    .\sram.rdWr_N      (rdWr_N),
    .\sram.ramAddr     (ramAddr),
    .\sram.ramData     (ramData)
);
`else

    vera_shell vshell(
        .SystemClock (SystemClock),
        .sram_ce_N   (ce_N),
        .sram_rdWr_N (rdWr_N),
        .sram_ramAddr (ramAddr),
        .sram_ramData (ramData)
);
`endif

// design instance
sram dut(
    .ce_N       (ce_N),
    .rdWr_N     (rdWr_N),
    .ramAddr    (ramAddr),
    .ramData    (ramData)
);

// system-clock generator
initial begin
    SystemClock = 0;
    forever begin
        #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
end
end

endmodule

```

The top-level Verilog module contains the following:

- The system clock, `SystemClock`. The system clock is contained in the port list of the testbench instance.

- The declaration of the interface clock input, `clk`, and its derivation from the system clock.
- The testbench instance, `vshell`. The module name for the instance must be the name of the testbench program, `sram_test`. The instance name can be something you choose. The ports of the testbench instance, other than the system clock, refer to the interface signals. The period in the port names separates the interface name from the signal name. A backslash is appended to the period in each port name because periods are not normally allowed in port names.
- The design instance, `dut`.

## **Compiling Testbench With the Design And Running**

The VCS command line for compiling both your example testbench and design is the following:

For two-step flow:

### **Compilation**

```
% vcs -ntb sram.v sram.test_top.v sram.vr
```

### **Simulation**

```
% simv
```

For three-step flow

### **Analysis**

```
% vlogan -ntb sram.v sram.test_top.v sram.vr
```

### **ElaborationCompilation**

```
% vcs top
```

## Simulation

```
% simv
```

You will find the simulation output to be the following:

```
Cycle: 3 Time: 250
The SRAM is being written at ramAddr: 000001 with ramData:
10101100
Cycle: 6 Time: 550
The SRAM is being read at ramAddr: 000001 its ramData is:
10101100
$finish at simulation time           550
V C S   S i m u l a t i o n   R e p o r t
```

---

## Key Features

VCS supports the following features for OpenVera testbench:

- “Multiple Program Support”
- “Class Dependency Source File Reordering”
- “Using Encrypted Files”
- “Functional Coverage”
- “Using Reference Verification Methodology”

---

## Multiple Program Support

Multiple program support enables multiple testbenches to run in parallel. This is useful when testbenches model stand-alone components (for example, Verification IP (VIP) or work from a previous project). Because components are independent, direct communication between them except through signals is undesirable.

For example, UART and CPU models would communicate only through their respective interfaces, and not via the testbench. Thus, multiple program support allows the use of stand-alone components without requiring knowledge of the code for each component, or requiring modifications to your own testbench.

## Configuration File Model

The configuration file that you create, specifies file dependencies for OpenVera programs.

Specify the configuration file as an argument to `-ntb_opts` as shown in the following usage model:

For two-step flow:

```
% vcs -ntb -ntb -ntb_opts  
config=configfileVerilog_and_OV_files
```

For three-step flow:

```
% vlogan -ntb -ntb_opts config=configfile
```

## Configuration File

The configuration file contains the program construct.

The program keyword is followed by the OpenVera program file (`.vr` file) containing the testbench program and all the OpenVera program files needed for this program. For example:

```
//configuration file  
program  
    main1.vr  
    main1_dep1.vr  
    main1_dep2.vr
```

```

...
main1_depN.vr
[NTB_options ]

program
    main2.vr
    main2_dep1.vr
    main2_dep2.vr
    ...
main2_depN.vr
[NTB_options ]

program
    mainN.vr
    mainN_dep1.vr
    mainN_dep2.vr
    ...
mainN_depN.vr
[NTB_options ]

```

In this example, `main1.vr`, `main2.vr` and `mainN` files each contain a program. The other files contain items such as definitions of functions, classes, tasks and so on needed by the program files. For example, the `main1_dep1.vr`, `main1_dep2.vr` .... `main1_depN.vr` files contain definitions relevant to `main1.vr`. Files `main2_dep1.vr`, `main2_dep2.vr` ... `main2_depN.vr` contain definitions relevant to `main2.vr`, and so forth.

## Usage Model for Multiple Programs

You can specify programs and related support files with multiple programs in two different ways:

1. Specifying all OpenVera programs in the configuration file
2. Specifying one OpenVera program on the command line, and the rest in the configuration file

Note:

- Specifying multiple OpenVera files containing the program construct at the VCS command prompt is an error.
- If you specify one program at the VCS command line and if any support files are missing from the command line, VCS issues an error.

### **Specifying all OpenVera programs in the configuration file**

When there are two or more program files listed in the configuration file, the VCS command line is:

```
% vcs -ntb -ntb_opts config=configfile
```

Or

```
% vlogan -ntb -ntb_opts config=configfile
```

The configuration file, could be:

```
program main1.vr -ntb_define ONE  
program main2.vr -ntb_incdir /usr/vera/include
```

### **Specifying one OpenVera program on the command line, and the rest in the configuration file**

You can specify one program in the configuration file and the other program file at the command prompt.

```
% vcs -ntb -ntb_opts config=configfile main2.vr
```

Or

```
% vlogan -ntb -ntb_opts config=configfile main2.vr
```

The configuration file used in this example is:

```
program main1.vr
```

In the previous example, `main1.vr` is specified in the configuration file and `main2.vr` is specified on the command line along with the files needed by `main2.vr`.

## NTB Options and the Configuration File

The configuration file supports different OpenVera programs with different NTB options such as '`include`', '`define`', or '`timescale`'. For example, if there are three OpenVera programs `p1.vr`, `p2.vr` and `p3.vr`, and `p1.vr` requires the `-ntb_define VERA1` runtime option, and `p2.vr` should run with `-ntb_incdir /usr/vera/include` option, specify these options in the configuration file:

```
program p1.vr -ntb_define VERA1  
program p2.vr -ntb_incdir /usr/vera/include
```

and specify the command line as follows.

```
% vcs -ntb -ntb_opts config=configfile p3.vr
```

Or

```
% vlogan -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
```

The following options are allowed for multiple program use.

- `-ntb_define macro`
- `-ntb_incdir directory`
- `-ntb_opts no_file_by_file_pp`
- `-ntb_opts tb_timescale=value`
- `-ntb_opts dep_check`
- `-ntb_opts print_deps`
- `-ntb_opts use_sigprop`
- `-ntb_opts vera_portname`

See the appendix on “Compile-time Options” or “Elaboration Options” for descriptions of these options.

---

## Class Dependency Source File Reordering

In order to ease transitioning of legacy code from Vera’s make-based single-file compilation scheme to VCS NTB, where all source files have to be specified on the command line, VCS provides a way of instructing the compiler to reorder Vera files in such a way that class declarations are in topological order (that is, base classes precede derived classes).

In Vera, where files are compiled one-by-one, and extensive use of header files is a must, the structure of file inclusions makes it very likely that the combined source text has class declarations in topological order.

If specifying a command line like the following leads to problems (error messages related to classes), adding the analysis option `-ntb_opts dep_check` to the command line directs the compiler to activate analysis of Vera files and process them in topological order with regard to class derivation relationships.

```
% vcs -ntb *.vr
```

Or

```
% vlogan -ntb *.vr
```

By default, files are processed in the order specified (or wildcard-expanded by the shell). This is a global option, and affects all Vera input files, including those preceding it, and those named in `-f file.list`.

When using the option `-ntb_opts print_deps` in addition to `-ntb_opts dep_check` with `vlogan/vcs`, the reordered list of source files is printed on standard output. This could be used, for example, to establish a baseline for further testbench development.

For example, assume the following files and declarations:

```
b.vr: class Base {integer i;}
d.vr: class Derived extends Base {integer j;}
p.vr: program test {Derived d = new;}
```

File `d.vr` depends on file `b.vr`, since it contains a class derived from a class in `b.vr`, whereas `p.vr` depends on neither, despite containing a reference to a class declared in the former. The `p.vr` file does not participate in inheritance relationships. The effect of dependency ordering is to properly order the files `b.vr` and `d.vr`, while leaving files without class inheritance relationships alone.

The following command lines result in reordered sequences.

```
% vcs -ntb -ntb_opts dep_check d.vr b.vr p.vr
```

Or

```
% vlogan -ntb -ntb_opts dep_check d.vr b.vr p.vr
```

```
% vcs -ntb -ntb_opts dep_check p.vr d.vr b.vr
```

Or

```
% vlogan -ntb -ntb_opts dep_check p.vr d.vr b.vr
```

The first command line yields the order `b.vr d.vr p.vr`, while the second line yields, `p.vr b.vr d.vr`.

## Circular Dependencies

With some programming styles, source files can appear to have circular inheritance dependencies in spite of correct inheritance trees being cycle-free. This can happen, for example, in the following scenario:

```
a.vr: class Base_A {...}  
      class Derived_B extends Base_B {...}  
b.vr: class Base_B {...}  
      class Derived_A extends Base_A {...}
```

In this example, classes are derived from base classes that are in the other file, respectively, or more generally, when the inheritance relationships project onto a loop among the files. This is, however, an abnormality that should not occur in good programming styles. VCS detects and reports the loop, and will use a heuristic to break it. This may not lead to successful compilation, in which case you can

use the `-ntb_opts print_deps` option to generate a starting point for manual resolution; however, if possible, the code should be rewritten.

## Dependency-based Ordering in Encrypted Files

As encrypted files are intended to be mostly self-contained library modules that the testbench builds upon, they are excluded from reordering regardless of dependencies (these files should not exist in unencrypted code). VCS splits Vera input files into those that are encrypted or declared as such by having the `.vrp` or `.vrhp` file extension or as specified using the `-ntb_vipext` option, and others. Only the latter unencrypted files are subject to dependency-based reordering, and encrypted files are prefixed to them.

Note:

The `-ntb_opts dep_check` compilation/analysis option specifically resolves dependencies involving classes and enums. That is, you only consider definitions and declarations of classes and enums. Other constructs such as ports, interfaces, tasks and functions are not currently supported for dependency check.

---

## Using Encrypted Files

VCS NTB allows distributors of Verification IP (Intellectual Property) to make testbench modules available in encrypted form. This enables the IP vendors to protect their source code from reverse-engineering. Encrypted testbench IP is regular OpenVera code, and is not subject to special processing other than to protect the source code from inspection in the debugger, through the PLI, or otherwise.

Encrypted code files provided on the command line are detected by VCS, and are combined into one preprocessing unit that is preprocessed separately from unencrypted files, and is for itself, always preprocessed in `-ntb_opts no_file_by_file_pp` mode. The preprocessed result of encrypted code is prefixed to preprocessed unencrypted code.

VCS only detects encrypted files on the command line (including `-f` option files), and does not descend into include hierarchies. While the generally recommended usage methodology is to separate encrypted from unencrypted code, and not include encrypted files in unencrypted files, encrypted files can be included in unencrypted files if the latter are marked as encrypted-mode by naming them with extensions `.vrp`, `.vrhp`, or additional extensions specified using the `-ntb_vipext` option. This implies that the extensions are considered OpenVera extensions similar to using `-ntb_fileext` for unencrypted files. This causes those files and everything they include to be preprocessed in encrypted mode.

---

## Functional Coverage

The VCS implementation of OpenVera supports the `covergroup` construct. For more information about the covergroup and other functional coverage model, see the section "Functional Coverage Groups" in the VCS OpenVera Language Reference Manual.

---

## Using Reference Verification Methodology

VCS supports the use of Reference Verification Methodology (RVM) for implementing testbenches as part of a scalable verification architecture.

The usage model for using RVM with VCS is:

For two-step flow:

### **Compilation**

```
% vcs -ntb -ntb_opts rvm [ntb_options] [compile_options]  
file1.vr file2.vr file3.v file4.v
```

### **Simulation**

```
% simv [run_options]
```

For three-step flow

### **Analysis**

Always analyze Verilog before VHDL.

```
% vlogan -ntb -ntb_opts rvm [vlogan_options] file1.vr  
file2.vr file3.v  
% vhdlan [vhdlan_options] file3.vhd file4.vhd
```

Note:

Specify the VHDL bottom-most entity first, and then move up in order.

### **Elaboration**

```
% vcs [other_ntb_options] [compile_options] design_unit
```

### **Simulation**

```
% simv [run_options]
```

For details on the use of RVM, see the *Reference Verification Methodology User Guide*. Though the manual descriptions refer to Vera, NTB uses a subset of the OpenVera language and all language specific descriptions apply to NTB.

Differences between the usage of NTB and Vera are:

- NTB does not require header files (`.vrh`) as described in the *Reference Verification Methodology User Guide* chapter “Coding and Compilation.”
- NTB parses all testbench files in a single compilation.
- The VCS command-line option `-ntb_opts rvm` must be used with NTB.

## Limitations

- The handshake configuration of notifier is not supported (since there is no handshake for triggers/syncs in NTB).
- RVM enhancements for assertion support in Vera 6.2.10 and later are not supported for NTB.
- If there are multiple consumers and producers, there is no guarantee of fairness in reads from channels, etc.



# 14

## Using SystemVerilog

---

VCS supports the SystemVerilog (SV) language (with some exceptions) as defined in the *Standard for SystemVerilog -- Unified Hardware Design, Specification, and Verification Language* (IEEE SystemVerilog LRM Std 1800™-2012).

This chapter describes the following:

- “[Use Model](#)”
- “[Using UVM With VCS](#)”
- “[Using VMM with VCS](#)”
- “[Using OVM with VCS](#)”
- “[Debugging SystemVerilog Designs](#)”
- “[Functional Coverage](#)”
- “[SystemVerilog Constructs](#)”

- “[Extensions to SystemVerilog](#)”

For SystemVerilog assertions, see [Chapter - "Using SystemVerilog Assertions"](#).

---

## Use Model

The use model to analyze, elaborate/compile and simulate your design with SystemVerilog files is as follows:

For two-step flow:

### Compilation

```
% vcs -sverilog [compile_options] Verilog_files
```

### Simulation

```
% simv [simv_options]
```

For three-step flow:

### Analysis

```
% vlogan -sverilog [vlogan_options] file4.sv file5.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

### Note:

Specify the VHDL bottommost entity first and then move up in order.

### Elaboration

```
% vcs [elab_options] top_cfg/entity/module
```

### Simulation

```
% simv [simv_options]
```

To analyze SV files, use the `-sverilog` option with `vlogan/vcs`, as shown in the above use model.

---

## Using UVM With VCS

This version of VCS provides native support for UVM-1.1d. These libraries are located in:

- `$VCS_HOME/etc/uvm-1.1`

UVM 1.1 is now replaced with UVM 1.1d, which is the default. You can load UVM 1.1d by:

- Using the `-ntb_opts uvm` option
- Explicitly specifying the `-ntb_opts uvm-1.1` option

The following sections explain your options for using UVM with VCS:

- [“Update on UVM-1.2”](#)
- [“Natively Compiling and Elaborating UVM-1.1d”](#)
- [“Natively Compiling and Elaborating UVM-1.2”](#)
- [“Compiling the External UVM Library”](#)
- [“Accessing HDL Registers Through UVM Backdoor”](#)
- [“Generating UVM Register Abstraction Layer Code”](#)
- [“Recording UVM Transactions”](#)
- [“Recording UVM Phases”](#)
- [“UVM Template Generator”](#)

- “Using Mixed VMM/UVM Libraries”
  - “Migrating from OVM to UVM”
  - “Where to Find UVM Examples”
  - “Where to Find UVM Documentation”
- 

## Update on UVM-1.2

You can load UVM-1.2 using the `-ntb_opts uvm-1.2` option.

Note:

You may see some backward compatibility issues while migrating to UVM-1.2. For the changes required, refer to the UVM-1.2 release notes

---

## Natively Compiling and Elaborating UVM-1.1d

You can compile and elaborate SystemVerilog code, which extends from UVM-1.1d base classes by using the following command:

```
% vcs -sverilog -ntb_opts uvm [compile_options] \
user_source_files_using_UVM
```

For a mixed-HDL or UVM environment, compile UVM-1.1d with vlogan using the following command:

```
% vlogan -ntb_opts uvm [compile_options]
// no source files here!
```

```
% vlogan -ntb_opts uvm [compile_options] \
<user source files using UVM>
```

Note:

- Complete the first step that compiles the UVM library before using the subsequent commands. The first `vlogan` call compiles the UVM library without any need of specifying user source files.
- In specific cases, the subsequent `vlogan` command might error out with `Error-[UM] Undefined Macro`. In this scenario you must explicitly add ``include uvm_macros.svh` to the file getting this error.

Elaborate the design as follows:

```
% vcs -ntb_opts uvm [elab_options] <top module>
```

Using the `-ntb_opts uvm` option is same as specifying the version explicitly using the `-ntb_opts uvm-1.1` option. However, it is best to specify the version explicitly, because later versions of UVM might carry the default UVM library.

---

## Natively Compiling and Elaborating UVM-1.2

You can compile and elaborate SystemVerilog code, which extends from UVM-1.2 base classes using the following command:

```
% vcs -sverilog -ntb_opts uvm-1.2 [compile_options] \
user_source_files_using_UVM
```

For a mixed-HDL or UVM environment, compile UVM-1.2 with `vlogan` using the following command:

```
% vlogan -ntb_opts uvm-1.2 [compile_options]
// no source files here!

% vlogan -ntb_opts uvm-1.2 [compile_options] \
```

*<user source files using UVM>*

---

## Compiling the External UVM Library

If you want to use a UVM version from Accellera in place of the UVM-1.1d version shipped with VCS, follow either of these procedures:

- “[Using the -ntb\\_opts uvm Option](#)”
- “[Explicitly Specifying UVM Files and Arguments](#)”

### Using the **-ntb\_opts uvm** Option

When you set the `VCS_UVM_HOME` environment variable to specify a UVM library directory, VCS uses this location even if the `-ntb_opts uvm` option is used. For example,

```
% setenv VCS_UVM_HOME <path_to_uvm_library>
```

Here, `<path_to_uvm_library>` is the absolute path to the directory that contains the `uvm_pkg.sv` file. Typically, the `uvm_pkg.sv` file is present in the `src` directory inside the Accellera distribution for UVM.

```
% vcs -sverilog -ntb_opts uvm [compile_options] \
user_source_files_using_UVM
```

This is also supported for the UUM flow and for using vlogan.

## Specifying External uvm\_dpi.cc Source

While using `-ntb_opts uvm`, the `uvm_dpi.cc` is picked up from the UVM installation directory inside the VCS installation directory. However, you might want to use the custom UVM DPI files instead of the ones shipped with the UVM library.

## Explicitly Specifying UVM Files and Arguments

The following example shows how to compile and elaborate the UVM extended code by explicitly specifying the UVM files and arguments:

```
% vcs -sverilog +incdir+${UVM_HOME}/src \
    ${UVM_HOME}/src/uvm_pkg.sv \
    ${UVM_HOME}/src/dpi/uvm_dpi.cc \
    -CFLAGS -DVCS \
    [compile_options] \
    user_source_files_using_UVM
```

For a mixed-HDL or UVM environment, compile with `vlogan` using the following command:

```
% vlogan -sverilog +incdir+${UVM_HOME}/src \
    ${UVM_HOME}/src/uvm_pkg.sv

% vlogan -sverilog +incdir+${UVM_HOME}/src \
    <user source files using UVM>
```

Elaborate the design as follows:

```
% vcs [elab_options] \
    ${UVM_HOME}/src/dpi/uvm_dpi.cc <top module> \
    -CFLAGS -DVCS
```

Note:

`${UVM_HOME}` should point to your UVM release path. It can also point to  `${VCS_HOME}/etc/uvm-1.1`.

---

## Accessing HDL Registers Through UVM Backdoor

If you are using tests that need to access HDL registers through the default UVM register backdoor mechanism, add the `-debug_access+r` option to your command line:

```
% vcs -sverilog -debug_access+r -ntb_opts uvm  
[compile_options] \ user_source_files_using_UVM
```

Note:

The `-debug_access+r` option may affect simulation performance. Therefore, you should use the `+vcs+learn+pli` option to improve the HDL access.

To simulate, use the following command:

```
% simv +UVM_TESTNAME=your_uvm_test [simv_options]
```

If you use the `-b` option with `ralgen`, the `-debug_access+r` option is not required and the HDL backdoor is enabled through cross-module references instead of the VPI. This provides better performance.

Note:

If you want to access VHDL nodes when using UVM-1.1 library, you must specify the `-CFLAGS -DVCS` option during elaboration. With UVM-1.2 library, VHDL nodes access is enabled by default.

---

## Generating UVM Register Abstraction Layer Code

VCS ships a utility called `ralgen`. Given a description of the available registers and memories in a design, `ralgen` automatically generates the UVM RAL abstraction model for these registers and memories. The description of these registers and memories can be in RALF format or in the IPXACT schema.

To generate a register model from a RALF file, use the following command:

```
% ralgen [options] -t <topname> -uvm <filename.ralf>
```

Here, `filename.ralf` is the name of the RALF input file and `topname` is the top block or system name in the RALF file.

To generate a register model from an IPXACT file, you use a two-step flow. The first step is to generate RALF from IPXACT as follows:

```
% ralgen -ipxact2ralf <input_file>
```

The second step is same as the one described above. For more information, see the *UVM Register Abstraction Layer Generator User Guide*.

---

## Recording UVM Transactions

UVM has additional features that allow you to take advantage of VCS transaction recording and Verdi's transaction debugging capabilities. These features are available with both the UVM-1.1d and UVM-1.2 libraries.

No compile-time option is needed for UVM-1.1d and UVM-1.2. You can enable recording by using a runtime option. The transaction and the report recordings are stored in the simulation VPD file.

## Compiling and Simulating UVM-1.1d and UVM-1.2

To compile and simulate your UVM-1.1d or UVM-1.2 code, see [“Enabling FSDB Transaction Recording”](#).

---

### Recording UVM Phases

In addition to UVM transaction recording capabilities, VCS allows you to record the UVM phases and enables the phase debugging capabilities. With this phase recording, you can see the start time and the end time for each component in each phase and the connectivity information for ports in `end_of_elab`. This feature is available with UVM-1.1 libraries in VCS.

To turn on UVM phase recording, use `+UVM_PHASE_RECORD` at runtime and pass the `-debug_access` option during compilation. The phase recordings are stored in the simulation VPD file.

You can enable UVM phase recording in your UVM-1.1 by running the VCS 3-step flow:

1. Define `+UVM_VCS_RECORD` at first step while compiling UVM library:

```
% vlogan -ntb_opts uvm +define+UVM_VCS_RECORD  
% vlogan -ntb_opts uvm [compile_options] \  
<user source files here>
```

2. Pass the `-debug_access` option at the compilation command line, as shown:

```
% vcs -sverilog -ntb_opts -debug_access  
[compile_options]
```

3. Enable phase recording during simulation by adding +UVM\_PHASE\_RECORD.

```
% simv +UVM_TESTNAME=<your_uvm_test>  
\+UVM_PHASE_RECORD [simv_options]
```

You can then use Verdi to debug the UVM phases in **UVM Phase View**. This is supported for interactive debug.

---

## UVM Template Generator

UVM template generator (`uvmgen`) is a template generator for creating robust and extensible UVM-compliant environments. The primary purpose of `uvmgen` is to minimize the VIP and environment development cycle by providing detailed templates for developing UVM-compliant verification environments. You can also use `uvmgen` to quickly understand how different UVM base classes can be used in different contexts. This is possible because the templates use a rich set of the latest UVM features to ensure the appropriate base classes and their features are picked up optimally.

In addition, `uvmgen` can be used to generate both individual templates and complete UVM environments.

`uvmgen` is a part of the VCS installation. It can be invoked by using the following command:

```
% uvmgen [-L libdir] [-X] [-o fname] [-O]
```

where,

`-L` Takes user-defined library for template generation

- X Excludes the standard template library
- o Generates templates in the specified file
- O Overwrites if the file already exists
- q Quick mode to generate the complete environment

For more information, see the *UVM Template Generator (uvmgen) User Guide*.

---

## Using Mixed VMM/UVM Libraries

For interoperability reasons (using UVM components in a VMM environment and vice versa), VCS allows you to load the VMM and UVM libraries simultaneously along with the VMM/UVM interop kit.

The VMM/UVM interop kit is located in the following directory:

- \$VCS\_HOME/etc/uvm-1.1/uvm\_vmm\_pkg.sv (for UVM-1.1 and later UVM releases)

You can load mixed VMM-1.2 and UVM by using a combination of the following VCS options:

- -ntb\_opts uvm[1.1]+rvm

**-or-**

- -ntb\_opts rvm+uvm[1.1]

**-ntb\_opts uvm[1.1]+rvm** is supported for both the mixed-HDL and UVM flows.

For example, in the two-step flow:

```
%vcs -ntb_opts uvm+rvm compile-time_options source_files
```

### In the UUM three-step flow:

```
%vlogan -ntb_opts uvm <no_other_files or options>  
%vlogan -ntb_opts uvm+rvm <compile_options> <compile_files>  
%vcs -ntb_opts uvm+rvm <compile_options> <top_module_name>
```

You can turn off the automatic inclusion of `uvm_vmm_pkg.sv` using the `+define+NO_VMM_UVM_INTEROP` option.

### For example, in the two-step flow:

```
%vcs -ntb_opts uvm+rvm +define+NO_VMM_UVM_INTEROP  
compile-time_options source_files
```

### In the UUM flow:

```
vlogan -ntb_opts uvm  
  
vlogan -ntb_opts uvm+rvm +define+NO_UVM_VMM_INTEROP  
other_options source_files  
  
vcs -ntb_opts uvm+rvm compile-time_options source_files
```

By default, the mixed environment is driven by a VMM top timeline. However, you can define a UVM top using the `+define+UVM_ON_TOP` option, as shown:

```
%vcs -ntb_opts uvm+rvm +define+ UVM_ON_TOP  
compile-time_options source_files
```

### In UUM flow:

```
vlogan -ntb_opts uvm  
  
vlogan -ntb_opts uvm+rvm +define+UVM_ON_TOP other_options
```

```
source_files  
vcs -ntb_opts uvm+rvm compile-time_options top_module_name
```

The UVM/VMM interop kit examples are located in the `$VCS_HOME/doc/examples/uvm_vmm_interop_kit` directory.

For details on how to use the native VMM/UVM interop kit, refer to the Class Reference section available in the VCS *Online documentation > UVM\_VMM documentation*.

Note:

In this version of VCS, the UVM-EA and VMM-1.2 interop kit is no longer included. If you need either one of these kits, contact [vcs\\_support@synopsys.com](mailto:vcs_support@synopsys.com).

---

## Migrating from OVM to UVM

To convert your OVM code to UVM, you can use a script stored in  `${VCS_HOME}/bin/OVM_UVM_Rename.pl`. This script makes the migration process easy.

Note:

This process is simple for SystemVerilog code that extends from OVM 2.1.1 onward.

Use the following command to convert your OVM code to UVM code:

```
% OVM_UVM_Rename.pl
```

This script hierarchically changes all occurrences of `ovm_` to `uvm_` for files with `.v`, `.vh`, `.sv`, and `.svh` extensions.

Change the simulation command line by replacing `OVM_TESTNAME` with `UVM_TESTNAME`.

Note:

Some additional work is required for the base classes that differ between OVM and UVM. For example, you may need to modify callbacks, some global function names, arguments, and so on.

---

## Where to Find UVM Examples

The UVM-1.1d interop examples are located in the following directory:

`$VCS_HOME/doc/examples/uvm`

The UVM-VMM interop examples are located in the following directory:

`$VCS_HOME/doc/examples/uvm_vmm_interop_kit`

---

## Where to Find UVM Documentation

The UVM-1.1d and UVM-VMM interop documentation is available in the following locations.

### UVM-1.1d Documentation

The PDF version of the *UVM-1.1d User Guide* (`uvm_users_guide_1.1.pdf`) is located under VCS documentation in SolvNet.

The PDF version of the *UVM-1.1d Reference Guide* (*UVM\_Class\_Reference\_1.1.pdf*) is located under VCS documentation in SolvNet.

## UVM-VMM Interop Documentation

The unified HTML version of the *UVM-VMM Interop Reference Guide* is accessible in [VCS Online documentation > UVM\\_VMM documentation](#).

---

## Using VMM with VCS

The use model to use VMM with VCS is as follows:

For two-step flow:

### Compilation

```
% vcs -sverilog -ntb_opts rvm [compile-time_options]  
Verilog_files
```

### Simulation

```
% simv [simv_options]
```

For three-step flow

### Analysis

```
% vlogan -sverilog -ntb_opts rvm [vlogan_options] file4.sv  
file5.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

### Note:

Specify the VHDL bottommost entity first, then move up in order.

## **Elaboration**

```
% vcs [elab_options] top_cfg/entity/module
```

## **Simulation**

```
% simv [simv_options]
```

To analyze SystemVerilog files using VMM, use the options `-sverilog` and `-ntb_opts rvm` with `vlogan/vcs`, as shown in the above use model.

For more information on VMM, refer to the *Verification Methodology Manual for SystemVerilog*.

---

## **Using OVM with VCS**

VCS provides native support for OVM 2.1.2. The libraries are located in:

```
$VCS_HOME/etc/ovm
```

---

## **Native Compilation and Elaboration of OVM 2.1.2**

You can compile and elaborate SystemVerilog code that extends from OVM 2.1.2 base classes by using the following command:

```
% vcs -sverilog -ntb_opts ovm [compile_options] \  
<user source files using OVM>
```

When you natively compile and elaborate the OVM code, you do not have to explicitly include OVM source files in the user code as they get parsed by default.

For a mixed-HDL or UVM environment, compile OVM 2.1.2 with vlogan using the following command:

```
% vlogan -ntb_opts ovm [compile_options]
// no source files here!
% vlogan -ntb_opts ovm -sverilog [compile_options] \
<user source files using OVM>
```

**Note:**

- Complete the first step that compiles the OVM library before using the subsequent command. The first vlogan call compiles the OVM library, which does not contain any user source files.
- In specific cases, the subsequent vlogan command might error out with Error-[UM] Undefined macro. In this scenario, explicitly add `include "ovm\_macros.svh" to the file encountering this error.

Elaborate the design as follows:

```
% vcs -ntb_opts ovm [elab_options] <top module>
% simv +OVM_TESTNAME=<ovm testname> <simv options>
```

Using the -ntb\_opts ovm option is same as specifying the version by explicitly using the -ntb\_opts ovm-2.1.2 option.

In some cases, if you have explicitly included ovm.svh, then the OVM source code is recompiled in subsequent vlogan command. To avoid re-compilation, you need to add the +define+OVM\_SVH option in the subsequent vlogan commands.

```
% vlogan -ntb_opts ovm [compile_options]
// no source files here!

% vlogan -ntb_opts ovm -sverilog +define+OVM_SVH
[compile_options] \<user source files using OVM>
```

In cases where `include "ovm\_pkg.sv" is present in the user code, recompilation of the OVM source code is required. To avoid this, you need to pass the +define+OVM\_PKG\_SV option in the subsequent vlogan commands.

```
% vlogan -ntb_opts ovm [compile-time_options]
// no source files here!

% vlogan -ntb_opts ovm -sverilog +define+OVM_PKG_SV
[compile-time_options] user_source_files_using_OVM
```

---

## Compiling the External OVM Library

If you want to use an OVM version from Accellera in place of the OVM 2.1.2 version shipped with VCS, use one of the following procedures:

- “[Using the -ntb\\_opts ovm Option](#)”
- “[Explicitly Specifying OVM Files and Arguments](#)”

## Using the -ntb\_opts ovm Option

When you set the `VCS_OVM_HOME` environment variable to specify a OVM library directory, VCS uses this location even if the `-ntb_opts ovm` option is used. For example,

```
% setenv VCS_OVM_HOME /<path_to_ovm_library>/myOVM-2.1.2
% vcs -sverilog -ntb_opts ovm [compile_options] \
<user source files using OVM>
```

This is also supported for the UUM flow and for using vlogan.

## Explicitly Specifying OVM Files and Arguments

The following example shows how to compile and elaborate the OVM extended code by explicitly specifying the OVM files and arguments:

```
% vcs -sverilog +incdir+${OVM_HOME} \
${OVM_HOME}/ovm_pkg.sv \
[compile_options] \
<user source files using OVM>
```

For a mixed-HDL or UUM environment, compile with vlogan using the following command:

```
% vlogan -sverilog +incdir+${OVM_HOME} \
${OVM_HOME}/ovm_pkg.sv

% vlogan -sverilog +incdir+${OVM_HOME} \
user_source_files_using_OVM
```

**Note:** \${OVM\_HOME} should point to your OVM release path. It can also point to \${VCS\_HOME}/etc/ovm-2.1.2

---

## Recording OVM Transactions

The OVM version shipped with VCS has additional features that allows you to take advantage of VCS and Verdi's transaction recording and debugging capabilities.

To turn on OVM transaction recording, you need to use a specific compile-time option for OVM or use the `-debug_access` option with VCS in the two-step flow and then enable recording using a different runtime option. The transaction and report recordings are stored in the simulation VPD file.

To compile your OVM code, add the `-debug_access` option to your vcs command line.

For the three-step flow, you need to provide the `+define+OVM_VCS_RECORD` option to the first `vlogan` command line, as shown along with any of the `-debug` options.

For example, in the two step flow:

```
% vcs -sverilog -ntb_opts ovm -debug_access\  
[compile-time_options]
```

In the UUM flow:

```
% vlogan -ntb_opts ovm  
+define+OVM_VCS_RECORD[compile_options]  
// no source files here!  
  
% vlogan -ntb_opts ovm [compile_options] \  
<user source files using OVM>
```

Note:

- Complete the first step that compiles the OVM library before using the subsequent commands. The first `vlogan` call compiles the OVM library. Define `OVM_VCS_RECORD` at this step to enable transaction recording that is without any specified user source files.
- In specific cases, the subsequent `vlogan` commands might error out with Error- [UM] Undefined macro. In this scenario, you must explicitly add ``include "ovm_macros.svh"` to the file getting this error.

Elaborate the design as follows:

```
% vcs -ntb_opts ovm [elab_options] top_module -debug_access
```

To simulate, use `+OVM_TR_RECORD` to turn on the transaction recording and use `+OVM_LOG_RECORD` to turn on the recording of OVM report log messages:

```
% simv +OVM_TESTNAME=<my_ovm_testname> +OVM_TR_RECORD \
+OVM_LOG_RECORD [simv_options]
```

You can then use Verdi to debug the transactions and log messages. This is supported for interactive debug. The recorded streams with transactions and report logs are available in the VMM/OVM folder of the transaction browser.

---

## Debugging SystemVerilog Designs

VCS provides UCLI commands to perform the following tasks to debug a design:

| Task                          | Related UCLI commands are... |
|-------------------------------|------------------------------|
| Line stepping                 | step<br>next<br>run          |
| Thread debugging              | step<br>thread               |
| Setting breakpoints           | stop<br>run                  |
| Mailbox related information   | show                         |
| Semaphore related information | show                         |

For detailed information on the UCLI commands, see the *UCLI User Guide*.

---

## Functional Coverage

The VCS implementation of SystemVerilog supports the `covergroup` construct, which you specify as the user. These constructs allow the system to monitor values and transitions for variables and signals. They also enable cross-coverage between variables and signals.

If you have covergroups in your design, VCS collects the coverage data during simulation and generates a database, `simv.vdb`. Once you have `simv.vdb`, you can use the Unified Report Generator to generate text or HTML reports. For more information about covergroups, see the *VCS SystemVerilog LRM*. For more information about functional coverage generated in VCS, see the *Coverage Technology User Guide*.

---

## SystemVerilog Constructs

VCS has implemented the following SystemVerilog constructs in recent releases:

- “Extern Task and Function Calls through Virtual Interfaces”
- “Modport Expressions in an Interface”
- “Interface Classes”
- “Package Exports”
- “Severity System Tasks as Procedural Statements”
- “Width Casting Using Parameters”

- “The std::randomize() Function”
- “SystemVerilog Bounded Queues”
- “wait() Statement with a Static Class Member Variable”
- “Support for Consistent Behavior of Class Static Properties”
- “Parameters and Local Parameters in Classes”
- “SystemVerilog Math Functions”
- “Streaming Operators”
- “Constant Functions in Generate Blocks”
- “Support for Aggregate Methods in Constraints Using the “with” Construct”
- “Debugging During Initialization SystemVerilog Static Functions and Tasks in Module Definitions”
- “Explicit External Constraint Blocks”
- “Generate Constructs in Program Blocks”
- “Error Condition for Using a Genvar Variable Outside of its Generate Block” on page 72
- “Randomizing Unpacked Structs”
- “Making wait fork Statements Compliant with the SV LRM”
- “Making disable fork Statements Compliant with the SV LRM”
- “Using a Package in a SystemVerilog Module, Program, and Interface Header”
- “Disabling DPI Tasks and Functions”
- “Support for Overriding Parameter Values through Configuration”

- “Support for Inclusion of Dynamic Types in Sensitivity List”
- “Support for Assignment Pattern Expression in Non-Assignment Like Context”
- “User-Defined Nettypes”
- “Generic Interconnect Nets”
- “Support for Associative Array With Unpacked Structure as Key”
- “Specifying a SystemVerilog Keyword Set by LRM Version at Command Line”
- “Support for .triggered Property with Clocking Block Name”
- “Support for Intra Assignment Delay With Non-Blocking Assignments in Program Block”

## **Extern Task and Function Calls through Virtual Interfaces**

You can define tasks and functions in an interface with one or more of the modules connected by the interface. You declare them as export in a modport or as extern in the interface. When they are called through virtual interfaces, the actual task or function that VCS executes depends on the interface instance of the virtual interface.

### **Example of exporting tasks in modports**

```
interface simple_bus ; // Define the interface
modport slave (export task Read);
endinterface: simple_bus

module memMod (simple_bus sb_intf);
task sb_intf.Read; // Read method
...
endtask
```

```

endmodule

module top;
simple_bus sb_intf(); // Instantiate the interface
memMod mem(sb_intf.slave); // exports the Read tasks
endmodule

```

## **Example of extern tasks in interfaces**

```

interface intf;
extern task T1();
extern task T2();
endinterface

module top;
intf i1();
intf i2();

virtual intf vi;
M1 m1(i1);
M2 m2(i1);
M3 m3(i2);
M4 m4(i2);

initial begin
    vi = i1;
    vi.T1();    // Task i1.T1 in M1
    vi.T2();    // Task i1.T2 in M2
    vi = i2;
    vi.T1();    // Task i2.T1 in M3
    vi.T2();    // Task i2.T2 in M4
end
endmodule

module M1(intf i1);
task i1.T1;
...
endtask
endmodule

module M2(intf i1);
task i1.T2;

```

```

...
endtask
endmodule

module M3(intf i2);
task i2.T1;
...
endtask
endmodule

module M4(intf i2);
task i2.T2;
...
endtask
endmodule

```

The definition of **extern** subroutines within an interface shall observe the following rules:

- Each interface instance may have different implementations of its **extern** subroutines.
  - The same **extern** subroutine of different interface instances can be defined in different modules.
  - Different **extern** subroutines of the same interface instance can be defined in different modules.
- Every interface instance must have one and only definition of its **extern** subroutines.
  - If an interface instance containing an **extern** subroutine, then one of the modules connected must define that subroutine.
  - Any **extern** subroutine of an interface instance cannot be defined in more than one module.
  - The module implementing any **extern** subroutine can be instantiated only once.

- These rules apply for exported subroutines in modports as well.

## **Limitations**

The feature has the following limitations:

- External task and function calls through virtual interface are not supported in constraints.
- The interface containing an external task or function can only be passed as port to module and program scopes. It cannot be instantiated inside the module defining the external task/function of interface and doing so, it gives an error.

## **Modport Expressions in an Interface**

As described in the IEEE SystemVerilog LRM Std 1800™-2012 Section 25.5.4 “Modport expressions,” a modport expression allows you to include the following in the modport list in an interface:

- elements of an array or structure
- concatenation of elements
- assignment pattern expressions

VCS has implemented modport expressions. For assignment pattern expressions, constant expressions are allowed only with input ports.

The following interface example includes modport expressions in a modport connection list.

### *Example 14-1 Modport Expressions*

```
interface interoceter_intf;
    logic [63:0] intrctr_bus;
    const longint flag=1879048192;
    wire [3:0] w1,w2;
    logic [7:0] log1,log2;
    modport mp1 (output .out(intrctr_bus[31:0]),
                 input .in(flag),.pt({log1,log2}));
    modport mp2 (output .out(intrctr_bus[63:32]),
                 input .in(7));
endinterface
```

part-select of a vector  
concatenating elements

The modport expressions in Example 14-1 are as follows:

```
.out(intrctr_bus[31:0])  
.in(flag)  
.pt({log1,log2})  
.out(intrctr_bus[63:32])  
.in(7)
```

Modport expressions consist of the following:

1. a port name preceded by a period, for example:`.out` or `.pt`.
2. The expression enclosed in parentheses, for example:

```
(intrctr_bus[64:32])
```

## Limitations

In addition to the assignment pattern expressions of elements declared in the interface being limited to the input ports, the feature has the following limitations:

- `ref` ports with modport expressions are not supported.
- Cross-module references in modport expressions are not supported.
- Modport expressions containing a mix of nets and variables are not supported.
- Modport expressions connected to OpenVera ports are not supported.
- Multiple driver checks are not yet supported.

---

## Interface Classes

An interface class can be seen more as a virtual class whose methods have to be pure virtual (see IEEE SystemVerilog LRM Std 1800™-2012 Section 8.22 “Polymorphism: dynamic method lookup”). In addition to the pure virtual methods, interface classes can have type declarations (see the IEEE SystemVerilog LRM 1800™-2012 Section 8.24 “Out-of-block declarations”) and parameter declarations (see IEEE SystemVerilog LRM Std 1800™-2012 Section 6.20 “Constants” and Section 8.27 “Typedef class”). Constraint blocks, nested classes and covergroups are not allowed inside an interface class. An interface class cannot be nested inside any other class.

Just like a virtual class, an interface class cannot be instantiated. However, its handle can be created and all the pure virtual methods declared inside the interface class or its base classes can be accessed through its class handle.

At runtime, a virtual class handle needs to point to a concrete class, which is derived from the virtual class directly or indirectly and provides a concrete implementation of all its pure virtual methods. This approach applies a restriction that all the common methods should go into their virtual class, so that they can be accessed using a virtual class handle and all the objects that can be used to be pointed to by such virtual class handles must be descendants of the virtual class. Interface classes do away with the restriction of being descendant of it in inheritance hierarchy to be able to be pointed to by its handle, thus mitigating the effect of not having multiple inheritance.

Interface classes can have multiple inheritance, that is they can inherit zero or more interface classes using the `extends` keyword. In case of multiple inheritance, name conflicts should be resolved.

An interface class handle can point to class objects that implement the interface class. A class implements an interface class using the keyword `implements`. A class can implement zero or more interface classes using the `implements` keyword. The `implements` keyword is not inheritance. It is a condition. So, classes implementing interface classes do not inherit any type or parameters from them. However, implementing classes can refer to types and parameters inside an interface using the scope resolution `::` operator. A class implements an interface class. If it itself implements the interface class or if any of its ancestor implements the interface class.

For a non-interface class to implement an interface class, it must provide implementations for the set of methods declared in interface class as pure virtual that satisfy the requirements of a virtual class method override (see the IEEE SystemVerilog LRM Std 1800™-2012 Section 8.21 “Abstract classes and pure virtual methods”).

```
interface class I;
    pure virtual function string getName();
endclass

class Foo implements I;
    virtual function string getName();
        return "Foo";
    endfunction
endclass

class Bar extends Foo;
    virtual function string getName();
        return "Bar";
    endfunction
endclass

module test;
Bar bar = new;
Foo foo = new;
I i;
//I i1 = new;
// Like virtual class, interface class can't be instantiated.
initial begin
    i = bar;
    $display(i.getName());
    i = foo;
    $display(i.getName());
end
endmodule
```

In the above example, interface class `I` has a pure virtual method `getName`, and a non-interface class `Foo` implements interface class `I`. Because `Bar` class extends `Foo`, so, `Bar` also implicitly implements interface class `I`. class `Foo` provides implementation for pure virtual function declared inside interface class `I`.

A class that implements an interface class must have a virtual method for every pure virtual method in its interface class. A class that is implementing an interface class can provide implementation for an interface class pure virtual method either by inheriting/ overriding a virtual method or by defining its own virtual method.

A virtual class can also implement an interface class. When a virtual class implements an interface class it must either provide a method implementation for the pure virtual method of interface class or re-declare the method prototype with the pure qualifier.

An interface class handle can point to objects of only those classes that either directly or indirectly implement an interface class using the `implements` keyword.

Methods declared inside the interface classes are allowed to have default arguments. The default argument should be a constant expression and it should be same for all the implementing classes. The default argument should be evaluated in the scope containing the method declaration.

## Difference Between Extends and Implements

When a class extends another class, it inherits all the members and methods of its superclass that are accessible to it based upon the access type. However, when a class implements an interface class, nothing is inherited. If the class needs to access a type or parameter

of the interface class, then the class should use the scope resolution `::` operator to access a member of the interface class, just like the way the static members of a class are accessed.

- An interface class can extend zero or more interface classes using the `extends` keyword.
- An interface class cannot extend a non-interface class.
- An interface class cannot implement another interface or non-interface class.
- An interface class cannot extend a type parameter.
- An interface class cannot extend a forward declared interface class.
- A non-interface class can extend zero or one non-interface class using the `extends` keyword.
- A non-interface class cannot extend an interface class.
- A non-interface class can implement zero or more interface classes using the `implements` keyword.
- A non-interface class cannot implement non-interface class.
- A non-interface class can extend a class and implement interface classes simultaneously.
- A non-interface class cannot implement a type parameter.
- A non-interface class cannot implement a forward declared interface class.

```
interface class Shape;  
    typedef string NAME;  
    pure virtual function NAME getShape();
```

```

endclass

interface class Area;
    pure virtual function int getArea();
endclass

class Rectangle implements Shape, Area;
    int x;
    int y;
    //virtual function NAME getShape();
    //illegal NAME is not accessible

    virtual function Shape::NAME getShape(); // legal
        return "Rectangle";
    endfunction

    virtual function int getArea();
        return (x*y);
    endfunction
endclass

class Square extends Rectangle;
    virtual function string getShape();
        return "Square";
    endfunction
endclass

module test;
Shape s;
Rectangle r = new;
Square sq = new;
initial begin
    s = r;
    $display(s.getShape());
    s = sq;
    $display(s.getShape());
end
endmodule

```

In the above example, the `Rectangle` class implements the `Shape` and `Area` interface classes using the `implements` keyword and provides implementation for both the pure virtual methods declared inside these interface classes. The `Square` class indirectly/ implicitly implements the `Shape` and `Area` interface classes.

The above example also shows that types and parameters inside the interface class cannot be accessed directly by the class implementing the interface class. Because, when a class implements an interface class, it does not inherit anything from interface class. Types and parameters inside an interface class are static and can be accessed using the class scope resolution `::` operator (see the IEEE SystemVerilog LRM Std 1800™-2012 Section 8.24 “Out-of-block declarations”)

## Cast and Interface Class

If a class implements an interface class, then the class' object can be assigned to be the handle of that interface class.

```
interface class I1;
endclass
interface class I2;
endclass

interface class I3 extends I1, I2;
endclass

interface class I4;
endclass

class A implements I3, I4;
endclass

class B extends A;
endclass
```

```

A a = new;
B b = new;
I1 i1 = a; // legal, as class A implements interface
           // class I3 which extends I1;
I2 i2;
I3 i3 = a; // legal, as class A implements interface class I3;
$cast(i2, b); // casting is not required, as class B
               // extends A which implements interface class
               // I3 which in turn extends I2;
$cast(a, i2); // valid, casting is must here.

```

Interface class handles can be cast dynamically if the actual object assigned to destination is valid.

```

$cast(i4, i3); // valid, as i3 is pointing to object of
                // class "A", which implements interface
                // class I4;

```

## Name Conflicts and Resolution

A class can implement multiple interface classes and interface classes can extend multiple interface classes. In such cases, identifiers from multiple name spaces may become visible in the single name space leading to name conflicts. Such name conflicts must be resolved, even when there is no usage of the identifier in the current scope.

### Name Conflicts During Implementation

A class can implement multiple interface classes and when the same method name appears in more than one interface classes, a method name conflict happens that must be resolved by providing an implementation of the method that simultaneously provides implementation for all the implemented interface classes' method with the same name.

```
interface class I1;
```

```

    pure virtual task t();
endclass

interface class I2;
    pure virtual task t();
endclass

class A implements I1, I2;
    virtual task t();
        $display("A::t");
    endtask
endclass

```

In this example class `A` is implementing two interface classes `I1` and `I2` both of which has method with name `t`. The class `A` resolves the conflict by providing an implementation of the virtual task `A::t` that simultaneously provides implementation for both `I1::t` and `I2::t`. However, it may not be always possible to resolve such conflict and thus it results in an error.

```

interface class I1;
    pure virtual task t(int i);
endclass

interface class I2;
    pure virtual task t();
endclass

class A implements I1, I2;
    virtual task t();
        $display("A::t");
    endtask
endclass

```

In this example, although `A::t` provides a valid implementation for `I2::t`, but it does not provide a valid implementation for `I1::t`, so, it is an error.

## Name Conflicts During Inheritance

An interface class can inherit type, parameters, and pure virtual methods from multiple base classes. If the same name is inherited from multiple base interface classes, then a name conflict occurs and it must be resolved. Types and parameters name conflict should be resolved by providing a declaration of type/parameter that overrides all such name collisions. For methods it should provide a single prototype that overrides all the name collisions. The method prototype must also be a valid virtual method override (see the IEEE SystemVerilog LRM Std 1800™-2012 Section 8.21 “Abstract classes and pure virtual methods”) for any inherited method of the same name.

```
interface class I1;
    pure virtual task t(int i);
endclass

interface class I2;
    pure virtual task t(int i);
endclass

interface class I3 extends I1, I2;
    pure virtual task t(int i);
endclass
```

In this example, `I3` inherits method `t` from both `I1` and `I2`. So, it provides a prototype for method `t` which resolves the conflict and the prototype is also a valid virtual method override.

```
interface class I1;
    pure virtual task t(int i);
endclass

interface class I2 extends I1;
endclass

interface class I3 extends I1;
```

```
endclass

interface class I4 extends I2, I3;
endclass
```

In the case of diamond relationship, name conflict does not occur if the originating class for the method is same. So, in the above example, there is no conflict for method name `t` in class `I4` because `I2::t` and `I3::t`, which `I4` inherits, are actually coming from the same interface class `I1`. Therefore, there is only a single copy of `t` in `I4` leading to no name conflict.

```
interface class I1;
    typedef int INT;
    pure virtual task t(INT i);
endclass

interface class I2;
    typedef int INT;
    pure virtual task t1(INT j);
endclass

interface class I3 extends I1, I2;
    typedef int INT;
endclass
```

In the above example, even though `INT` is of the same type in both the interface classes `I1` and `I2`, still `I3` needs to resolve the conflict by redefining the type.

## Interface Class and Randomization

It is legal to call the `randomize()` method on an interface class handle. An inline constraint is also legal on an interface class handle. However, it is of little use because interface class cannot have any members. The `rand_mode()` and `constraint_mode()` methods are illegal interface class handle.

Unlike non-interface classes, interface classes have virtual and empty `pre_randomize()` and `post_randomize()` built-in methods,.So, any direct call to these methods using interface class handle leads to call to these empty methods. However, when `randomize` is called on interface class handle, it leads to the call of `randomize` method on the class object that are pointed to by the interface class handle. This in turn internally calls `pre_randomize()` and `post_randomize()` methods of the actual object that is pointed to by the interface class handle.

```
interface class I;
endclass

class A implements I;
    rand int i;
    function void pre_randomize();
        $display("A::pre_randomize", i);
    endfunction

    function void post_randomize();
        $display("A::post_randomize", i);
    endfunction
endclass

A a = new;
I i = a;

i.randomize(); // it would call A::pre_randomize()
               // and A::post_randomize() internally.
i.pre_randomize(); // built-in empty body I::pre_randomize
```

```
// would be called.
```

---

## Package Exports

Declarations imported into a package are not visible by way of subsequent imports of that package by default. Package export declarations allow a package to specify those imported declarations to be made visible in subsequent imports.

There are three forms of export directives:

```
export pkg::name;
```

This directive both imports and exports *name* from the specified package named *pkg*.

```
export pkg::*;
```

This directive exports all names imported from the *pkg* package into the current package. The imports can be by name reference or by named export directive.

```
export *::*;
```

Exports all names imported from any packages into the current package. The imports can be by name reference or by named export directive. An export directive `* :: *` must match at least one of the import directive.

Unlike package import directives, package export directives *can only* occur at package scope and cannot occur in \$unit.

[Example 14-2](#) illustrates the package export functionality:

### *Example 14-2 The Package Import Functionality Example 1*

```
package p1;
    int x, y;
endpackage
package p2;
    import p1::x;
    export p1::*;

endpackage
```

exports p1::x as the variable named x

p1::x and p2::x are the same declaration

```
package p3;
    import p1::*;
    import p2::*;
    export p2::*;
    int q = x;
endpackage
```

p1::x and q are made available from p3

Although p1::y is a candidate for import, it is not actually imported since it is not referenced.

Since p1::y is not imported, it is not made available by the export

---

## **Severity System Tasks as Procedural Statements**

The severity system tasks can be included as procedural statements in user-defined tasks and functions, in `initial`, `final` and any `always` blocks.

[Example 14-3](#) shows the usage of these system tasks in an `initial` block

### *Example 14-3 Severity Statements in Procedural Blocks*

```
initial
if (Verilog_simulator == "VCS")
    $display("\n\t Smart User! \n");
else
begin
```

```

#10 $warning(2,"\\n\\t Expect a performance cost \\n\\n");
    if (Verilog_simulator == "Questa_questionable")
        #10 $info (3,"\\n\\t you paid too much \\n\\n");
    if (Verilog_simulator == "Indecisive")
        #10 $fatal(1,"\\n\\t give up now\\n");

```

In [Example 14-3](#), because the conditional expression  
`(Verilog_simulator == "VCS")` is true, VCS displays the  
following when it compiles and simulates [Example 14-3](#):

Smart User!

For conditional expression `(Verilog_simulator == "Indecisive")`, VCS displays the following when it compiles and simulates [Example 14-3](#):

```

Warning: "expl.sv", 21: mod: at time 10
          2
          Expect a performance cost

```

```

Fatal: "expl.sv", 25: mod: at time 20
          give up now

```

Note:

The severity system tasks can be used as elaboration system tasks. Elaboration system tasks require the `-assert svaext` compile-time option and the keyword argument.

## Width Casting Using Parameters

VCS used to support width casting using integers only, such as `4'(x)`. However, according to the IEEE SystemVerilog LRM Std 1800™-2012 Section 6.24.1 “Cast operator”:

“If the casting type is a constant expression with a positive integral value, the expression in parentheses shall be padded or truncated to the size specified. It shall be an error if the size specified is zero or negative.”

VCS now supports width casting for any constant expression also.  
For example:

```
parameter p = 16;  
(p+1)'(x-2) // This is now supported
```

According to the syntax:

```
casting_type ::= simple_type | constant_primary | signing  
| string | const  
constant_primary ::= // from A.8.4  
primary_literal | ps_parameter_identifier  
constant_select  
| specparam_identifier [ [ constant_range_expression ] ]  
| genvar_identifier35 | [ package_scope | class_scope ]  
enum_identifier  
| constant_concatenation |  
constant_multiple_concatenation  
| constant_function_call | (   
constant_mintypmax_expression )  
| constant_cast | constant_assignment_pattern_expression  
| type_reference36
```

The constant expressions could also include parameter cross-module references. So, the following examples are legal and are now supported.

#### *Example 14-4 Casting for a Parameter with an Expression*

```
module test (input clk);  
    parameter integer signed XYZ_NUMBER_VL = 3;  
  
    logic [3:0] next_vls_in_use_reg;  
    int next_vl_to_use_reg, next_vl_to_use_re2;
```

```

int XYZ_VL_SIZE, vl_index;

always @ (posedge clk) begin
    vl_index = 5;

    next_vl_to_use_re2 = 4'(3); // ok
    next_vl_to_use_reg = XYZ_NUMBER_VL'(vl_index) ;
        // the line above with an expression now supported
    end
endmodule

```

**Example 14-5 Casting for a Parameter with a Localparam**

```

program p1;
    localparam int aa=4;
    localparam int bb = 10;

    logic[aa-1:0] mytime;
    initial begin
        mytime = aa'(bb); // line now supported
    end
endprogram

```

Casting type can be any positive constant expression. The expression in the parenthesis can be padded or truncated based on the casting type. Cast type can also be parameter cross-module references (which are constant expressions) that can include concatenation as well as assignment patterns.

**Example 14-6 Casting type a positive constant expression**

```

module m #(p = 0);
endmodule

module test;
    localparam int P1=4;
    localparam int P2 = 10;

    logic[P1-1:0] mytime;

    m #(2) u1();

```

```
    initial begin
        mytime = (u1.P1+u1.P1)'(bb);
// line above now supported
    end
endmodule
```

---

## The std::randomize() Function

The `randomize()` function randomizes variables that are not class members.

### Syntax

```
[std::]randomize(variable-identifier-list)
[with constraint-block]
```

### Description

SystemVerilog defines extensive randomization methods and operators for class members. Most modeling methodologies recommend the use of classes for randomization. However, there are situations where the data to be randomized is not available in a class. SystemVerilog provides the `std::randomize()` function to randomize variables that are not class members.

The `std::randomize()` function can be used in the following scopes:

- module
- function
- task
- class method

Arguments to `std::randomize()` can be of integral types including:

- integer
- bit vector
- enumerated type

Object handles and strings cannot be used as arguments to `std::randomize()`.

The variables passed to `std::randomize()` must be visible in the scope where the function is called. Cross-module references are not allowed as arguments to the `std::randomize()` function.

All constraint expressions currently available with `obj.randomize()` in VCS can be used as constraints in the *constraint-block*.

Only constraints specified in the constraint block are honored. Any rand mode specified on the class members is ignored when `std::randomize()` is called with the given class member.

The `pre_randomize()` and `post-randomize()` tasks are not called when `std::randomize()` is used within a class member function.

The `std::` prefix must be explicitly specified for the `randomize()` call.

The `std::randomize()` function is supported in VCS. Files containing `std::randomize()` calls can be compiled with vlogan.

The function using `std::randomize()` can be declared in a task inside a package that can be imported into modules and programs.

## Example

```
module M;
    bit[11:0] addr;
    integer data;

    function bit genAddrData();
        bit success;
        success = std::randomize(addr, data);
        return success;
    endfunction

    function bit genConstrainedAddrData();
        bit success;
        success = std::randomize(addr, data)
            with {addr > 1000; addr + data < 20000;};
        return success;
    endfunction

endmodule
```

The `genAddrData` function uses `std::randomize(addr, data)` to assign random values to `addr` and `data` variables. The `std::randomize()` function randomizes any variables that are visible in the scope.

The `getConstrainedAddrData()` function uses `std::randomize(addr, data)` to assign random values to `addr` and `data` variables.

---

## SystemVerilog Bounded Queues

A bounded queue is a queue limited to a fixed number of items. For example:

```
bit q[$:255];
```

a bit queue whose maximum size is 257 bits

```
int q[$:5000];
```

an int queue whose maximum size is 50001

This section explains how bounded queues work in certain operations.

```
q1 = q2;
```

This is a bounded queue assignment. VCS copies the items in q2 into q1 until q1 is full or until all the items in q2 are copied into q1. The bound number of items in the queues remain the same as declared.

```
q.push_front(new_item)
```

If you add a new item to the front of a full bounded queue, VCS deletes the last item in the back of the queue.

```
q.push_back(new_item)
```

If the bounded queue is full, a new item cannot be added to the back of the queue and the queue remains the same.

```
q1 === q2
```

The behavior of a bounded queue comparison is same as an unbounded queue, that is, the bound sizes should be the same when the two bounded queues are equal.

### **Limitation for SystemVerilog Bounded Queues**

Bounded queues are not supported in constraints.

---

### **wait() Statement with a Static Class Member Variable**

A wait statement with a static class member variable is now supported. Consider the following example:

```
class foo;
    static bit is_true = 0;
    task my_task();
        fork
            begin
                #20;
                is_true = 1;
            end
            begin
                wait(is_true == 1);
                $display("%0d: is_true is now %0d", $time, is_true);
            end
        join
    endtask: my_task
endclass: foo

program automatic main;
    foo foo_i;
    initial begin
        foo_i = new();
        foo_i.my_task();
    end
endprogram: main
```

---

## Support for Consistent Behavior of Class Static Properties

VCS supports access and usage of class static members in structural contexts including continuous assign and force.

Consider the following `test.sv` test case that uses class static member in a continuous assign statement:

```
class mymode;
    static int mode;
endclass

class testbench
    mymode p;
    function new();
        p = new();
    endfunction
endclass

module test()

    testbench tb = new();
    int local_mode;

    assign local_mode = tb.p.mode; //class static variable
                                  in continuous assign
    always@ (local_mode) $display($time, " local_mode
changed: module tb.p = %d", local_mode);
    always@ (tb.p.mode) $display($time, " tb.p.mode changed:
module tb.p = %d", tb.p.mode);
    initial begin
        #1;
        $display($time, " local_mode = %d, tb.p.mode = %d",
local_mode, tb.p.mode);
        #10;
        $display($time, " local_mode = %d, tb.p.mode = %d",
local_mode, tb.p.mode);
    end
endmodule
```

```

module top;

    test t();
    testbench tb = new();
    initial begin
        tb.p.mode = 0;
        #10;
        tb.p.mode = 10;
        #10;
    end

endmodule

```

The expected results after simulating this test case is that the `local_mode` local variable must be equal to `tb.p.mode` at all times.

With previous releases, the results are as follows:

```

1 local_mode = 0, tb.p.mode = 0
10 tb.p.mode changed: module tb.p = 0
11 local_mode = 0, tb.p.mode = 10

```

With this release, the results are as follows:

```

1 local_mode = 0, tb.p.mode = 0
10 tb.p.mode changed: module tb.p = 10
10 local_mode changed: module tb.p = 10
11 local_mode = 10, tb.p.mode = 10

```

## Parameters and Local Parameters in Classes

You can include parameters and local parameters (`localparams`) in classes. For example:

```
class cls;
```

```

localparam int Lp = 10;
parameter int P = 5;
endclass

```

---

## SystemVerilog Math Functions

Verilog defines math functions that behave the same as their corresponding math functions in C. These functions are as follows:

|                           |                       |
|---------------------------|-----------------------|
| <code>\$ln(x)</code>      | Natural logarithm     |
| <code>\$log10(x)</code>   | Decimal logarithm     |
| <code>\$exp(x)</code>     | Exponential           |
| <code>\$sqrt(x)</code>    | Square root           |
| <code>\$pow(x,y)</code>   | $x^{**}y$             |
| <code>\$floor(x)</code>   | Floor                 |
| <code>\$ceil(x)</code>    | Ceiling               |
| <code>\$sin(x)</code>     | Sine                  |
| <code>\$cos(x)</code>     | Cosine                |
| <code>\$tan(x)</code>     | Tangent               |
| <code>\$asin(x)</code>    | Arc-sine              |
| <code>\$acos(x)</code>    | Arc-cosine            |
| <code>\$atan(x)</code>    | Arc-tangent           |
| <code>\$atan2(x,y)</code> | Arc-tangent of $x/y$  |
| <code>\$hypot(x,y)</code> | $\sqrt{x^2+y^2}$      |
| <code>\$sinh(x)</code>    | Hyperbolic sine       |
| <code>\$cosh(x)</code>    | Hyperbolic cosine     |
| <code>\$tanh(x)</code>    | Hyperbolic tangent    |
| <code>\$asinh(x)</code>   | Arc-hyperbolic sine   |
| <code>\$acosh(x)</code>   | Arc-hyperbolic cosine |

|                          |                                         |
|--------------------------|-----------------------------------------|
| <code>\$atanh (x)</code> | Arc-hyperbolic tangent                  |
| <code>\$clog2 (n)</code> | Ceiling of log base 2 of n (as integer) |

## Streaming Operators

Streaming operators that can be applied to any bit-stream data types consists of the following:

- Any integral, packed, or string type
- Unpacked arrays, structures, or class of the above types
- Dynamically sized arrays (dynamic, associative, or queues) of any of the above types

## Packing (Used on RHS)

### Primitive Operation

```
expr_target = {>>|<< slice{expr_1, expr_2, ..., expr_n }}
```

The `expr_target` and `expr_i` can be any primary expressions of any streamed data types.

The slice determines the size of each block measured in bits. If specified, it may be either a constant integral expression, or a simple type.

The `<<` or `>>` determines the order in which blocks of data are streamed.

### Streaming Concatenation

```
expr_target = {>>slice1 {expr1, expr2, {<< slice2{expr3,  
expr4}}}}
```

## Unpacking (Used on LHS)

### Primitive operation

```
{>>|<< slice{expr_1, expr_2, ..., expr_n } } = expr_src;
```

If the unpacked operation includes unbounded dynamically sized types, the process is greedy. The first dynamically sized item is resized to accept all the available data (excluding subsequent fixed sized items) in the stream. Any remaining dynamically sized items are left empty.

### Streaming Concatenation

```
{>>slice1 {expr1, expr2, {<< slice2{expr3, expr4} } } = expr_src;
```

## Packing and Unpacking

```
{>>|<< slice_target{target_1, target_2, ..., target_n } } =  
{>>|<< slice_src{src_1, src_2, ..., src_n } };
```

## Propagation and force Statement

Any operand (either dynamic or not) in the stream can be propagated and forced/released correctly.

## Error Conditions

It has the following error conditions:

- Compile-time error for associative arrays as assignment target

- Runtime error for any null class handles in packing and unpacking operations

## Structures with Streaming Operators

Although the whole structure is not allowed in the stream, any structure members and excluded sub-structures could be used as an operand of both packing and unpacking operations.

For example:

```
s1 = {>>{expr_1, expr_2, ..., expr_n}} //invalid
s1.data = {>>{expr_1, expr_2, expr_n}}//valid
```

## Support for with Expression

VCS supports the `with` expression with streaming operator.

The syntax of the `with` expression defined in the IEEE SystemVerilog LRM Std 1800™-2012 is as follows:

```
stream_expression ::= expression [ with [
array_range_expression ] ]
array_range_expression ::= expression
| expression : expression
| expression +: expression
| expression -: expression
```

## Semantics

The feature has the following semantics:

1. You can set the array expression range within the `with` construct to an integral type or to an expression that evaluates to an integral type. You cannot use other types.

For example,

```
function int eval(int a);
    eval = a;
endfunction

{ >> { target with [0 : 7]} } = 31;//case 1
{ >> { target with [0 : eval(31)]} } = 31;//case 2
```

For case 2, the function evaluates to an integral type. Hence, this case is allowed.

2. You can set the expression before the `with` construct to any single unpacked dimensional array that includes a queue.

For example,

```
bit target[][];
```

Usage:

```
{ >> { target with [0:2]} } = data;
```

This case is illegal, as the expression before the `with` expression has multiple unpacked dimension.

Usage:

```
{ >> { target[3] with [0:2]} } = data;
```

However, this case is legal as the expression before the `with` expression has a single unpacked dimension.

3. The expression within the `with` construct is evaluated immediately before its corresponding array is streamed (packed or unpacked). Thus, the expression can refer to data that are unpacked by the same operator but before the array.

For example,

```
{ >> { a, target with [a:0], b} } = data;
```

In this case, assuming target is a single dimension unpacked array and the value of a is 2 before streaming and the value of a is 4 after streaming, the value of the index should be 4.

4. When you use the `with` expression within the context of an unpack operation and the array is a variable-sized array, the `with` expression must be resized to accommodate the expression range.

For example,

```
Size of data = 32, size of a = 8, sizeof b = 8  
bit target[];  
{ >> { a, target, b} } = data;
```

In this case, a total of 16 bits are allocated to the target.

```
{ >> { a, target with [0: 7], b} } = data;
```

However, in this case only 8 bits are allocated to the target. Therefore, you can limit the data allocation to any dynamic type using the `with` expression.

5. If the array is a fixed-size array and the expression range evaluates to a range outside the extent of the array, only the range that lies within the array is unpacked and an error is issued.

For example,

```
bit target[0:3];  
{ >> {target with [0:7]} } = data;
```

This is illegal, as it exceeds the array bound.

6. If the range expression evaluates to a range smaller than the extent of the array (fixed or variable size), only the specified items are unpacked into the designated array locations. The rest of the array is unmodified.
7. When you use the `with` expression within the context of a pack (on RHS), it behaves in the same way as an array slice.

For example,

```
data = { >> {target with [a:0]} };
```

The above example can be interpreted directly as `{>> {target[value_of_a:0]}}`. In array slice, variable index might not be allowed in all the cases, but with the `with` expression, you can provide a variable index.

8. You can set the array range expression within the `with` construct to be of integral type and it evaluates to values that lie within the bounds of a fixed-size array or to a positive value for dynamic arrays or queues.

## Constant Functions in Generate Blocks

Calls to constant user-defined functions can be included in generate blocks.

As stated in the IEEE SystemVerilog LRM Std 1800™-2012, you can use these constant functions to build complex calculations. The standard also establishes things that cannot be in a user-defined function for it to operate as a constant function. For example, a constant function cannot have an `output`, `inout`, or `ref` argument, they cannot contain statements that schedule events after the function has returned its value, or contain the `fork` construct. There are more than a few other requirements.

A call to a constant function can occur in a `generate` block but the `generate` block cannot contain a definition or declaration of a constant function.

[Example 14-7](#) contains a module definition that includes a `generate` block and a user-defined function that qualifies as a constant function. The `generate` block contains a call to this constant function.

### *Example 14-7 Calling a Constant Function in a Generate Block*

```
module interoceter ( );
parameter adrs_width = 4;
generate
    genvar dim1;
    genvar dim2;
    for (dim1 = 1; dim1 <= adrs_width; dim1 = dim1 + 1)
        begin : outer_floop
            for (dim2 = 0; dim2 < adrs_width; dim2 = dim2 + const_func(dim1))
                begin : inner_floop
                    reg [2:0] P;
                end : inner_floop
            end : outer_floop
        endgenerate

function integer const_func;
    input [31:0] lwrdf; integer intgr0;
begin : func_main
    if (lwrdf > 0)
        begin : ifloop
            lwrdf = {lwrdf >> 1};
            intgr0 = 1;
            while (lwrdf > 0)
                begin : wloop
                    lwrdf = {lwrdf >> 1};
                    intgr0 = {intgr0 << 1};
                end : wloop
            end : ifloop
        end
    else
        intgr0 = 0; // return 0 when lwrdf <= 0
    const_func = intgr0;
end : func_main
endfunction
endmodule
```

The diagram shows a code snippet from Example 14-7. A call to the function `const_func(dim1)` is highlighted with a red box. An arrow labeled "constant function call" points downwards from this box to the function definition below. Another arrow labeled "constant function" points upwards from the function definition back to the call site.

---

## **Support for Aggregate Methods in Constraints Using the “with” Construct**

Aggregate methods in constraint blocks using the `with` construct have two variants, as shown in the following code example:

```

byte arr[3] = { 10, 20, 30 };
class C;
    rand int x1;
    rand int x2;
    rand int x3;
    rand int x4;

constraint cons {
    // Newly implemented variant
    x1 == arr.sum() with (item * item);
    x2 == arr.sum(x) with (x + x);

    // Previously implemented variant
    // Supported in older releases
    x3 == arr.sum() with (arr[item.index] * arr[item.index]);
    x4 == arr.sum(x) with (arr[x.index] + arr[x.index]);
}
endclass

```

The first variant is implemented. For a discussion and examples of aggregate methods in constraints using the `with` construct, see IEEE SystemVerilog LRM Std 1800™-2012 Section 7.12.4 “Iterator index querying”.

As specified in the standard, the entire `with` expression must be in parentheses.

## **Debugging During Initialization SystemVerilog Static Functions and Tasks in Module Definitions**

You can tell VCS to enable UCLI debugging when initialization begins for static SystemVerilog tasks and functions in module definitions with the `-ucli=init` runtime option and keyword argument.

This debugging capability enables you also to set breakpoints during initialization.

If you omit the `=init` keyword argument and just enter the `-ucli` runtime option, the UCLI begins after initialization and you cannot debug inside static initialization routines during initialization.

Note:

- Debugging static SystemVerilog tasks and functions in program blocks during initialization does not require the `=init` keyword argument.
- This feature does not apply to VHDL or SystemC code.

When you enable this debugging, VCS displays the following prompt indicating that the UCLI is in the initialization phase:

```
init%
```

When initialization ends, the UCLI returns to its usual prompt:

```
ucli%
```

During the initialization, the `run` UCLI command with the `0` argument (`run 0`), or the `-nba` or `-delta` options runs VCS until initialization ends. As usual, after initialization, the `run 0` command and argument runs the simulation until the end of the current simulation time.

During initialization the following restrictions apply:

- UCLI commands that alter the simulation state, such as a `force` command create an error condition.
- Attaching or configuring Cbug, or in other ways enabling C, C++, or SystemC debugging during initialization is an error condition.

- The following UCLI commands are not allowed during initialization:
  - session management commands:** save and restore
  - signal and variable commands:** force, release, and call
  - The signal value and memory dump specification commands:** memory -read/-write and dump
  - The coverage commands:** coverage and assertion

## Example

Consider the following code example:

```
module mod1;
  class C;
    static int I=F();
    static function int F();
      logic log1;
      begin
        log1 = 1;
        $display("%m log1=%0b",log1);
        $display("In function F");
        F = 10;
      end
    endfunction
  endclass
endmodule
```

If you simulate this example, with the `-ucli` runtime option, you see the following:

```
Command: simv =ucli
Chronologic VCS simulator copyright 1991-year
Contains Synopsys proprietary information.
Compiler version version-number; Runtime version version-
```

```
number; simulation-start-date-time
mod1.\C::F log1=1
In function F
      V C S      S i m u l a t i o n      R e p o r t
Time: 0
CPU Time:      0.510 seconds;      Data structure size:  0.0Mb
simulation-ends-day-date-time
```

VCS executed the \$display tasks right away and the simulation immediately ran to completion.

If you simulate this example, with the -ucli=init runtime option and keyword argument, you see the following:

```
Command: simv -ucli=init
Chronologic VCS simulator copyright 1991-year
Contains Synopsys proprietary information.
Compiler version version-number; Runtime version version-
number; simulation-start-date-time
init%
```

Note that VCS has not executed the \$display system tasks yet and the prompt is init%.

You can set a breakpoint, for example:

```
init% stop -in \C::F
1
```

To run through the initialization phase:

```
init% run 0
Stop point #1 @ 0 s;
init%
```

The breakpoint halts VCS.

If you run the simulation up to the end of the initialization phase with the `run 0` UCLI command again, you see the following:

```
init% run 0
mod1.\C::F log1=1
In function F
ucli%
```

Now VCS executes the `$display` system tasks and changes the prompt to `ucli%`.

---

## Explicit External Constraint Blocks

External constraint blocks are constraint blocks, also called the constraint bodies, that are outside of a class and at the same hierarchical level of that class. You enable them with external constraint prototypes in the class.

There are two forms of external constraint prototypes:

- explicit — where you include the `extern` keyword in the prototype.
- implicit — where you omit the `extern` keyword in the prototype.

The explicit form is implemented in this release.

The following code example shows these two forms of external constraint prototypes.

```
class Class1;
    rand int int1,int2;
    constraint imp_ext_cnstr_proto1;           // implicit form
    extern constraint exp_ext_cnstr_proto2;     // explicit form
    ...
endclass
```

The external constraint block, or body for these prototypes must be at the same hierarchical level as the class and follow the class definition.

The following are external constraint blocks or bodies for these external constraint prototypes:

```
constraint Class1::imp_ext_cnstr_proto1 {
    int1 inside {0, [3:5], [7:31]};}
constraint Class1::exp_ext_cnstr_proto2 {
    int2 dist {100 := 1, 101 := 2};}
```

Besides the `extern` keyword, the difference between the implicit and explicit forms is how VCS responds when the external constraint block or body for a prototype is missing:

- With the implicit form, VCS handles a missing external constraint block as an empty constraint block. This is not an error condition and VCS just outputs a warning message. For example:

```
Warning-[BCNACMBP] Missing constraint definition
doc_example.sv, 6
prog, "constraint imp_ext_cnstr_proto1;"
The constraint imp_ext_cnstr_proto1 declared in the
class Class1 is not defined.
Provide a definition of the constraint body
imp_ext_cnstr_proto1 or remove the constraint declaration
imp_ext_cnstr_proto1 from the class declaration Class1.
```

An empty constraint block is same as the following:

```
constraint imp_ext_cnstr_proto1 { };
```

With a missing external constraint block for the implicit form, VCS continues to compile or elaborate and generates the simv executable because it is not an error condition. If you do not notice the warning message you might expect to see the missing constraint block constraining the values of the random variables.

- With the explicit form, a missing external constraint block is an error condition. For example:

```
Error-[SV_MEECD] Missing explicit external constraint def
doc_example.sv, 7
prog, "constraint exp_ext_cnstr_proto2;"
The explicit external constraint 'exp_ext_cnstr_proto2'
declared in the class 'Class1' is not defined.
Provide a definition of the constraint body
'exp_ext_cnstr_proto2' or remove the explicit external
constraint declaration 'exp_ext_cnstr_proto2' from the
class declaration 'Class1'.
```

With a missing external constraint block for the explicit form, VCS does not compile or elaborate because it is an error condition.

## Using an Empty Constraint Block

You can use the implicit form of a constraint prototype, without the corresponding constraint block in a subclass to remove a constraint from a base class. For example:

```
module top;
class C;
rand int x;
constraint protoc_1 { x < 5; }
constraint protoc_2 { x > 3; }
endclass

class CD extends C;
rand int y;
constraint protoc_1; // removing this constraint in
// this subclass
```

```

constraint protoCD_1 { x < 6; } // applying a new constraint
                                // on x
endclass

C ci = new;
CD cdi = new;
int res1;
int res2;

initial begin
    repeat (20) begin
        res1 = ci.randomize();      // here x can have value 4 only
        res2 = cdi.randomize();    // here x can have values 4 and 5
        if ((res1 == 1) && (res2 == 1))
            $display("niru>> ci.x=%d  cdi.x=%d",ci.x, cdi.x);
    end
end

endmodule

```

---

## Generate Constructs in Program Blocks

Generate constructs are now supported not just in modules but also in program blocks.

These constructs are described in The IEEE Verilog LRM Std 1364-2005 in the following sections:

### 12.4 Generate constructs

#### 12.4.1 Loop generate constructs

#### 12.4.2 Conditional generate constructs

The following are examples of these constructs in a program block:

```
program prog;
```

```

...
generate
    reg reg1;
endgenerate

if (1) logic log1;

genvar gv1;
for(gv1=1; gv1<10; gv1++) logic log2;

case (param1)
    0 : logic log3;
    ...
endcase

endprogram

```

The first is a generate region, specified with the `generate` and `endgenerate` keywords inside a program block:

```

generate
    reg reg1;
endgenerate

```

The second is a conditional generate construct with the `if` keyword:

```
if (1) logic log1;
```

The third is a generate loop variable declared with the `genvar` keyword, followed by a `for` loop for that variable:

```

genvar gv1;
for(gv1=1; gv1<10; gv1++) logic log2;

```

The fourth is a generate case construct:

```

case (param1)
    0 : logic log3;

```

```
...
endcase
```

---

## Error Condition for Using a Genvar Variable Outside of its Generate Block

A genvar variable declared in local scope of a generate block that is used outside that block is an error condition. The following code example shows this error condition:

```
module test;
generate
    for (genvar i = 0; i < 1; i++)
        begin
            a1: assert final (1);
        end
endgenerate
generate
    for (i = 0; i < 1; i++)
        begin
            a1: assert final (1);
        end
endgenerate
endmodule
```

Compiling/elaborating this example with the following command line:

```
% vcs generate.sv -sverilog -assert svaext
```

Results in the following error message:

```
Error-[IND] Identifier not declared
generate.sv, 9
    Identifier 'i' has not been declared yet. If this error
is not expected,
    please check if you have set `default_nettype to none.

1 error
```

To fix this error, declare `genvar i` in module scope.

---

## Randomizing Unpacked Structs

You can now randomize members of an unpacked struct. You can do this in the following ways:

- use the scope randomize method `std::randomize()`
- use the class randomize method `randomize()`

You can also:

- disable and re-enable randomization in an unpacked struct with the `rand_mode()` method.
- use in-line random variable control to specify the randomized variables with an argument to the `randomize()` method.

## Using the Scope Randomize Method `std::randomize()`

The following example illustrates using this method:

*Example 14-8 First Example of the Scope Randomize Method  
`std::randomize()`*

```
module test();

typedef struct {
    bit [1:0] b1;
    integer i1;
} ST1;

ST1 st1;

initial
repeat (4)
```

```

begin
    std::randomize(st1);
#10 $display("\n\n\t at %0t",$time);
    $display("\t st1.b1 is %0d",st1.b1);
    $display("\t st1.i1 is %0d",st1.i1);
end

endmodule

```

This example randomizes struct instance st1. The \$display system tasks display the following:

```

at 10
st1.b1 is 2
st1.i1 is 1474208060

```

```

at 20
st1.b1 is 1
st1.i1 is 816460770

```

```

at 30
st1.b1 is 3
st1.i1 is -1179418145

```

```

at 40
st1.b1 is 0
st1.i1 is -719881993

```

The following is another code example that randomizes members of an unpacked struct and uses constraints:

*Example 14-9 Second Example of the Scope Randomize Method  
std::randomize()*

```

module test;
    typedef struct {
        rand byte aa;

```

```

        byte bb;
    } ST;

ST st;
bit [3:0] c;

initial begin
    std::randomize(st.bb);      // std randomization on a
                                // struct member
    std::randomize(st) with { st.aa > 10; };
                                // support st.aa in with block
    std::randomize(c,st) with { st.aa > c; };
    $display("\n\n\t at %0t",$time);
    $display("\t st.aa is %0d",st.aa);
    $display("\t st.bb is %0d",st.bb);
    $display("\t bit c is %0d",c);
end
endmodule

```

The \$display system task displays the following:

```

at 0
st.aa is 121
st.bb is -9
bit c is 0

```

*Example 14-10 Third Example of the Scope Randomize Method  
std::randomize()*

```

module test;
    typedef struct {
        byte a0;
        byte b0;
    } ST0;
    typedef struct {
        byte aa;
        ST0 st0;
    } ST_NONE;

    typedef struct {
        rand byte aa;

```

```

        byte bb;
} ST_PART;

typedef struct {
    rand byte aa;
    randc byte bb;
} ST_ALL;

ST_NONE st;
ST_PART st1;
ST_ALL st2;

initial begin
    repeat (5) begin
        // random variables: st.aa st.st0.a0 st.st0.b0
        std::randomize(st);

        // random variables: st1.aa st.bb
        std::randomize(st1) with {st1.aa>st1.bb;};

        // random variables: st2.aa st2.bb
        std::randomize(st2);

        $display("st %p",st);
        $display("st1 %p",st1);
        $display("st2 %p",st2);
    end
end

endmodule

```

**This example randomizes unpacked struct instance st1. The \$display system task displays the following:**

```

st '{aa:54, st0:{a0:60, b0:125}}
st1 '{aa:-125, bb:-126}
st2 '{aa:-9, bb:-90}
st '{aa:27, st0:{a0:-75, b0:-6}}
st1 '{aa:-37, bb:-47}
st2 '{aa:-106, bb:49}
st '{aa:-60, st0:{a0:-86, b0:-60}}

```

```

st1 '{aa:-71, bb:-103}
st2 '{aa:-120, bb:-15}
st '{aa:44, st0:'{a0:-50, b0:5}}
st1 '{aa:-69, bb:-96}
st2 '{aa:96, bb:95}
st '{aa:122, st0:'{a0:-94, b0:-16}}
st1 '{aa:-2, bb:-63}
st2 '{aa:18, bb:-12}

```

## Using the Class Randomize Method `randomize()`

The following example illustrates using this method.

*Example 14-11 The Class Randomize Method `randomize()`*

```

module test();

typedef struct {
    rand bit [1:0] b1;
    rand integer i1;
} ST1;

class CC;
    rand ST1 st1;
endclass

CC cc = new;

initial
repeat (4)
begin
    cc.randomize();
#10 $display("\n\n\t at %0t",$time);
    $display("\t cc.st1.b1 is %0d",cc.st1.b1);
    $display("\t cc.st1.i1 is %0d",cc.st1.i1);
end

endmodule

```

This example randomizes instance `cc` of class `CC` that contains unpacked struct `ST`. The `$display` system task displays the following:

```
at 10
cc.st1.b1 is 3
cc.st1.il is -1241023056
```

```
at 20
cc.st1.b1 is 3
cc.st1.il is -1877783293
```

```
at 30
cc.st1.b1 is 1
cc.st1.il is 629780255
```

```
at 40
cc.st1.b1 is 3
cc.st1.il is 469272579
```

Here is another code example:

*Example 14-12 Another Example of the Class Randomize Method  
randomize()*

```
module test;

typedef struct {
    bit[3:0] c;
    randc bit[1:0] d;
} ST0;

typedef struct {
    rand bit[5:0] a;
    rand bit[5:0] b;
    rand ST0 st0;
    bit [5:0] e;
```

```

}ST;

class CC;
    rand ST st;
endclass

CC cc = new;

initial begin
repeat (10) begin
    // random variables: cc.st.a cc.st.b and cc.st.st0.d
    // state variables: cc.st.e and cc.st.st0.c
    cc.randomize() with { st.a<10 ; st.b>10; st.a+st.b==64;};

    $display("st %p",cc.st);
end
end

endmodule

```

This example randomizes class instance `cc` according to the constraint that follows the `with` keyword. The `$display` system task displays the following:

```

st '{a:'h7, b:'h39, st0:{c:'h0, d:'h0}, e:'h0}
st '{a:'h8, b:'h38, st0:{c:'h0, d:'h1}, e:'h0}
st '{a:'h1, b:'h3f, st0:{c:'h0, d:'h3}, e:'h0}
st '{a:'h1, b:'h3f, st0:{c:'h0, d:'h2}, e:'h0}
st '{a:'h1, b:'h3f, st0:{c:'h0, d:'h0}, e:'h0}
st '{a:'h8, b:'h38, st0:{c:'h0, d:'h1}, e:'h0}
st '{a:'h9, b:'h37, st0:{c:'h0, d:'h2}, e:'h0}
st '{a:'h9, b:'h37, st0:{c:'h0, d:'h3}, e:'h0}
st '{a:'h7, b:'h39, st0:{c:'h0, d:'h3}, e:'h0}
st '{a:'h8, b:'h38, st0:{c:'h0, d:'h1}, e:'h0}

```

## Disabling and Re-enabling Randomization

You can disable and re-enable randomization in an unpacked struct with the `rand_mode()` method.

**Example 14-13 Disabling and Re-enabling Randomization with the rand\_mode() Method**

```
module test();

typedef struct {
    rand integer i1;
} ST1;

class CC;
    rand ST1 st1;
endclass

CC cc = new;

initial
repeat (10)
begin
    cc.randomize();
    #10 $display("\n\t at %0t", $time);
    $display("\t cc.st1.i1 is %0d", cc.st1.i1);
end

initial
begin
    #55 cc.rand_mode(0);
    #20 cc.rand_mode(1);
end

endmodule
```

In this example, the `rand_mode()` method, with its arguments, disables and re-enables randomization in class instance `cc`. The `$display` system task displays the following:

```
at 10
cc.st1.i1 is -902462825

at 20
cc.st1.i1 is -1241023056
```

```
at 30
cc.st1.i1 is 69704603

at 40
cc.st1.i1 is -1877783293

at 50
cc.st1.i1 is -795611063

at 60
cc.st1.i1 is 629780255

at 70
cc.st1.i1 is 629780255

at 80
cc.st1.i1 is 629780255

at 90
cc.st1.i1 is 1347943271

at 100
cc.st1.i1 is 469272579
```

In this example randomization is disabled at simulation time 55 and re-enabled at simulation time 75, enabling new random values at simulation time 90.

In the previous version of VCS , this example results in the following error messages at compile-time:

```
Error-[SV-RISNYI] Rand in Struct Not Yet Implemented
doc_ex3.sv, 4
The qualifier 'rand' was seen in a struct. This is not yet
supported.
Please remove the 'rand' declaration.

1 error
```

The following is another code example:

*Example 14-14 Another Example of Disabling and Re-enabling Randomization with the rand\_mode() Method*

```
module test;

typedef struct {
    bit[3:0] c;
    randc bit[1:0] d;
} ST0;

typedef struct {
    rand bit[5:0] a;
    rand bit[5:0] b;
    rand ST0 st0;
    bit [5:0] e;
} ST;

class CC;
    rand ST st;
    rand bit[2:0] n1;
endclass

CC cc = new;

initial
begin
    cc.st.rand_mode(0);
    repeat (10)
        begin
            // random variables: cc.n1
            // state variables: all members of cc.st
            cc.randomize();
            $display("turn off st %p , cc.n1 %b",
                     cc.st,cc.n1);
        end
    cc.st.rand_mode(1);
    cc.st.st0.rand_mode(0);
    repeat (10)
        begin
            // random variables: cc.n1 cc.st.a cc.st.b
```

```

// state variables: cc.st.e cc.st.st0.c cc.st.st0.d
cc.randomize();
$display("turn off st.st0 %p , cc.n1 %b",
cc.st,cc.n1);
end
cc.st.st0.rand_mode(1);
end

endmodule

```

In this example the `rand_mode()` method disables randomization in unpacked struct instance `cc.st.st0` and then re-enables it. The `$display` system task displays the following:

```

turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 111
turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 000
turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 011
turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 011
turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 001
turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 111
turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 111
turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 011
turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 001
turn off st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 100
turn off st.st0 '{a:'h39, b:'h17, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 010
turn off st.st0 '{a:'h26, b:'h1f, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 001
turn off st.st0 '{a:'h9, b:'h3, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 010
turn off st.st0 '{a:'h23, b:'he, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 101
turn off st.st0 '{a:'h21, b:'h18, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 000
turn off st.st0 '{a:'h34, b:'h1d, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 001
turn off st.st0 '{a:'h2f, b:'h27, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 011
turn off st.st0 '{a:'h2f, b:'h17, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 100
turn off st.st0 '{a:'hd, b:'h34, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 010
turn off st.st0 '{a:'h27, b:'h11, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 010

```

## Using In-Line Random Variable Control

The following example illustrates the usage of in-line random variable control to specify the randomized variables with an argument to the `randomize()` method.

## Example 14-15 In-line Random Variable Control

```
module test();

typedef struct {
    rand integer i1;
} ST1;

typedef struct {
    rand integer i1;
} ST2;

class CC;
    rand ST1 st1;
    rand ST2 st2;
endclass

CC cc = new;

initial
begin
#10 cc.randomize();
    $display("\n\t at sim time %0t",$time);
    $display("\t cc.st1.i1 is %0d",cc.st1.i1);
    $display("\t cc.st2.i1 is %0d",cc.st2.i1);
#10 cc.randomize(st1);
    $display("\n\t at sim time %0t",$time);
    $display("\t cc.st1.i1 is %0d",cc.st1.i1);
    $display("\t cc.st2.i1 is %0d",cc.st2.i1);
#10 cc.randomize(null);
    $display("\n\t at sim time %0t",$time);
    $display("\t cc.st1.i1 is %0d",cc.st1.i1);
    $display("\t cc.st2.i1 is %0d",cc.st2.i1);
#10 cc.randomize(st2);
    $display("\n\t at sim time %0t",$time);
    $display("\t cc.st1.i1 is %0d",cc.st1.i1);
    $display("\t cc.st2.i1 is %0d",cc.st2.i1);
end

endmodule
```

This example supplies the `randomize()` method with arguments for unpacked struct instances `st1` and `st2` and the `null` keyword.

1. At simulation time 20, randomization is limited to `st1`.
2. At simulation time 30, randomization is turned off.
3. At simulation time 40, randomization is limited to `st2`.

The `$display` system task displays the following:

```
at sim time 10
cc.st1.i1 is -902462825
cc.st2.i1 is -1241023056

at sim time 20
cc.st1.i1 is 69704603
cc.st2.i1 is -1241023056

at sim time 30
cc.st1.i1 is 69704603
cc.st2.i1 is -1241023056

at sim time 40
cc.st1.i1 is 69704603
cc.st2.i1 is -1877783293
```

At simulation 20, a new random value is in `st1` but not `st2`.

At simulation time 30, there are no new random values.

At simulation time 40, a new random value is in `st2` but not `st1`.

In the previous version of VCS, this example result in the following error messages at compile-time:

```
Error-[SV-RISNYI] Rand in Struct Not Yet Implemented
doc_ex4.sv, 4
The qualifier 'rand' was seen in a struct. This is not yet
```

supported.

Please remove the 'rand' declaration.

```
Error-[SV-RISNYI] Rand in Struct Not Yet Implemented
doc_ex4.sv, 8
The qualifier 'rand' was seen in a struct. This is not yet
supported.
Please remove the 'rand' declaration.
```

2 errors

Here is another code example:

*Example 14-16 Another Example of In-line Random Variable Control*

```
module test;

typedef struct {
    bit[3:0] c;
    randc bit[1:0] d;
} ST0;

typedef struct {
    rand bit[5:0] a;
    rand bit[5:0] b;
    rand ST0 st0;
    bit [5:0] e;
} ST;

class CC;
    ST st;
    rand bit[2:0] n1;
endclass

CC cc = new;

initial begin
    // random variables: cc.n1
    // state variables: all members of cc.st
repeat (5) begin
    cc.randomize();
```

```

$display("default st %p , cc.n1 %b",cc.st,cc.n1);
end

// random variables: cc.st.a cc.st.b cc.st.st0.d
// state variables: cc.n1 cc.st.e cc.st.st0.c
repeat (5) begin
    cc.randomize(st);
    $display("inline st %p , cc.n1 %b",cc.st,cc.n1);
end

end
endmodule

```

**In this example the `randomize()` method is called without an argument and then with the `st` struct instance argument. The `$display` system tasks display the following:**

```

default st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 111
default st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 000
default st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 011
default st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 011
default st '{a:'h0, b:'h0, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 001
inline st '{a:'h1f, b:'h27, st0:{c:'h0, d:'h0}, e:'h0} , cc.n1 001
inline st '{a:'h11, b:'h34, st0:{c:'h0, d:'h1}, e:'h0} , cc.n1 001
inline st '{a:'h17, b:'h2a, st0:{c:'h0, d:'h2}, e:'h0} , cc.n1 001
inline st '{a:'h1f, b:'h9, st0:{c:'h0, d:'h3}, e:'h0} , cc.n1 001
inline st '{a:'h3, b:'h12, st0:{c:'h0, d:'h3}, e:'h0} , cc.n1 001

```

**VCS executes the second `$display` system task after it executes the `randomize()` method with the `st` argument.**

## Limitation

Random class objects as members of an unpacked struct are not yet implemented (NYI). For example:

```

module test;

class CC0;
    rand int a;

```

```

endclass

typedef struct {
    rand bit[5:0] a;
    rand bit[5:0] b;
    rand CC0 cc0;      // this is not allowed in this release
} ST;

endmodule

```

---

## Making wait fork Statements Compliant with the SV LRM

You can specify making the `wait fork` statements compliant with the SystemVerilog LRM with the `-ntb_opts sv_dwfork` compile-time option and the keyword argument.

The IEEE SystemVerilog LRM Std 1800™-2012 states the following about `wait fork` statements:

“The `wait fork` statement blocks process execution flow until all immediate child sub-processes (processes created by the current process, excluding their descendants) have completed their execution.”

For backward compatibility reasons, by default the VCS blocks the process execution flow until all child sub-processes, not just the immediate child sub-processes, have completed their execution. It also waits only for those processes that are created by the current task or process that contains the `wait fork` statement.

You can specify that VCS be compliant with the standard and block process execution flow only for immediate child sub-processes and wait for processes created by the current process (even if the `wait fork` is contained within a task) with the `-ntb_opts sv_dwfork` compile-time option and the keyword argument.

The following code example shows the difference in behavior for wait fork.

```
program A;
task t1();
    #1 $display($time,, " T1_1 \n");
endtask
task t2();
    fork
        #1 $display($time,, " T2_1 \n");
        #9 $display($time,, " T2_2 \n");
    join_any
endtask
task disp();
    fork
        t1();
        t2();
    join_any
    wait fork;
    $display($time,, "After Wait fork");
endtask
initial begin
    fork
        #1 $display($time,, " Initial Thread 1 \n");
        #5 $display($time,, " Initial Thread 2 \n");
    join_any
    disp();
end
endprogram
```

By default, VCS waits for the execution of:

```
#9 $display($time,, " T2_2 \n");
```

It executes this line at simulation time 10, even though the fork for this \$display system task is not an immediate child sub-process of task disp().

By default, the \$display system task displays the following:

```
1 Initial Thread 1
2 T1_1
2 T2_1
5 Initial Thread 2
10 T2_2
10 After Wait fork
```

If you include the `-ntb_opts sv_dwfork` compile-time option and the keyword argument, the `$display` system task displays the following output:

```
1 Initial Thread 1
2 T1_1
2 T2_1
5 Initial Thread 2
5 After Wait fork
```

---

## Making disable fork Statements Compliant with the SV LRM

You can also specify making `disable fork` statements compliant with the SystemVerilog LRM with the `-ntb_opts sv_dwfork` compile-time option and keyword argument.

The IEEE SystemVerilog LRM Std 1800™-2012 states the following about `disable fork` statements:

“The `disable fork` statement terminates all active descendants (sub-processes) of the calling process.”

For backward compatibility reasons, by default, VCS terminates only those processes that are created by the current task or process that contains the `disable fork`.

You can specify that VCS be compliant with the standard and terminate all the processes that are created by the process that contains the  `disable fork` (even if the  `disable fork` is contained within a task) with the `-ntb_opts sv_dwfork` compile-time option and the keyword argument.

The following code example shows the difference in behavior for  `disable fork`.

```
program A;
task disp();
    fork
        #1 $display($time,, "disp_T1");
        #2 $display($time,, "disp_T2");
    join_any
    disable fork;
    $display($time,, "After disable fork");
endtask
initial begin
fork
    #1 $display($time,, " Initial Thread 1 \n");
    #5 $display($time,, " Initial Thread 2 \n");
join_any
disp();
#10 $display($time,, "End");
end
endprogram
```

By default,  `disable fork` does not disable the fork in the process, but only disables the fork in the task in which it is present to give the following output:

```
1 Initial Thread 1
2 disp_T1
2 After disable fork
5 Initial Thread 2
12 End
```

With the `-ntb_opts sv_dwfork` option, disable fork also disables the fork in the process, giving the following output:

```
1 Initial Thread 1
2 disp_T1
2 After disable fork
12 End
```

---

## Using a Package in a SystemVerilog Module, Program, and Interface Header

Importing from a package to a module, program, or interface by including the package in the module, program, or interface header is now implemented.

This technique of importing from a package is described in the IEEE SystemVerilog LRM Std 1800™-2012 Section 26.4 “Using packages in module headers”.

The primary purpose of this syntax and usage is to enable you to import names in the parameter list or port list, without importing the package into the enclosing scope (`$unit`).

To illustrate this technique, you should import from a package into a module definition and then into a program definition, as shown in [Example 14-17](#) and [Example 14-8](#). This technique is also implemented for importing from a package to an interface.

### *Example 14-17 Importing a Package in a Module Header*

```
package my_pkg;
    typedef reg [3:0] my_type1;
    typedef int my_type2;
endpackage

module my_module import my_pkg::*;
```

```

        (input my_type1 a, output my_type2 z);
:
endmodule

```

In [Example 14-7](#), the design objects declared in the `my_pkg` package are imported into the `my_module` module with the `import` keyword followed by the name of the package. Use the wildcard `*` (asterisk) to specify importing all design objects in the package.

[Example 14-18](#) shows importing from packages in a program header.

#### *Example 14-18 Importing Packages in a Program Header*

```

package pack1;
    typedef struct {
        real r1;
    } struct1;
    typedef enum bit {H,T} bool_sds;
endpackage:pack1

package pack3;
    integer int1=0;
endpackage: pack3

program prog1 import pack1::struct1,pack3::*;
    (output out1,out2);
:
endprogram: prog1

```

The header of the `prog1` program includes the `import` keyword followed by the `pack1` and `pack3` packages. Import the `struct1` structure from `pack1` into the `prog1` program. Then import all the design objects in `pack3` into the program using the wildcard `*` (asterisk).

Note:

VCS does not support using the same name for a package and a module within a compilation unit.

---

## Disabling DPI Tasks and Functions

Based on the IEEE Standard for SystemVerilog 1800™-2012 section 35.9, VCS now provides the support for disable statement to terminate the activity of a task or a named block that is currently executing a DPI call.

When a DPI import subroutine is disabled, the DPI-C code is made to follow a simple disable protocol. The protocol allows the C code to perform any necessary resource cleanup. An import DPI routine is in the disabled state when a disable statement targets either the import DPI routine or a parent of the calling chain to the import DPI routine for disabling. An imported subroutine can enter the disabled state immediately after the return of a call to an exported subroutine, and it adheres to the following disable protocol:

- A disabled import DPI routine shall acknowledge within a program that they have been disabled by calling the `svAckDisabledState()` API function.
- A subroutine can determine that it is in the disabled state by calling the API function `svIsDisabledState()`.
- When an imported task returns due to a disable, it shall return a value of 1. Otherwise, it shall return 0.
- Once an imported subroutine enters the disabled state, it is illegal for the current function call to make any further calls to exported subroutines.

If any of the protocol items are not correctly followed, a fatal simulation error is issued.

## Use Model

The following switch must be passed to VCS to enable this functionality:

```
-dpi_lrm_task
```

In the case of UUM flow, the switch must be passed both at the `vlogan` and `vcs` steps.

---

## Support for Overriding Parameter Values through Configuration

VCS supports overriding the parameter value through a configuration as defined in the SystemVerilog LRM. Configurations can be used either to override parameter values that are declared within a design or to override parameter values for a specific instance of a design.

---

## Example

The following example illustrates overriding of parameter values through configuration:

### *Example 14-19 Example of parameter overriding through configuration*

```
config cfg;
  design rtlLib.top;
  default liblist rtlLib;
  localparam LP = 19;
  instance top.B1 use #(.P(LP)); // assign 19 to top.B1.P
  instance top.B2 use #(.P(3)); // assign 3 to top.B2.P
```

```

instance top.B3 use #(.P()); // assign its default value to
top.B3.P
cell bot use #(.P(10)); // assign 10 to rest of instances of bot
endconfig : cfg

module top;
bot #(11) B1(); // instance parameter value being override
// inside configuration
bot B2();
bot B3();
bot B4();
bot B5();
bot B6();
defparam top.B4.P = 20; // defparam specified parameter value
//being override inside configuration
endmodule

module bot;
parameter P = 9;
initial $display("%m", P);
endmodule

```

---

## Precedence Override Rules

Parameter overriding during elaboration is determined in the following order of priority (highest to lowest):

1. Parameter overriding from VCS elaboration command line (-pvalue)
2. Parameter overriding through a configuration using instance and cell rules
3. defparam using hierarchical path names
4. Instance based overriding

**Note:**

If multiple instance and cell rules are used, VCS applies the rule that appears first in configuration. It ignores multiple rules and generates a warning message.

---

## Limitations

This feature has the following limitations:

- Cross-module references (XMRs) for parameter overriding is not supported.
- For VCS, parameter overriding rules are not supported if the design hierarchy crosses the VHDL boundary.

---

## Support for Inclusion of Dynamic Types in Sensitivity List

VCS ignores the dynamic types while processing the implicit sensitivity lists of `always @*`, `always_comb`, and `always_latch` procedural blocks. Therefore, it issues the following warning:

Warning-[IDTS] Ignoring dynamic type sensitivity

`<source_file>, <line_no> "obj"`

Dynamic types used in `always_comb`, `always_latch`, `always (@*)` will be ignored for the inferred sensitivity list.

VCS supports the dynamic types in the implicit sensitivity list of `always @*` block as specified in the Section 9.2 of the *IEEE Standard SystemVerilog Specification 1800-2012*.

The following usages of arrays would be processed for implicit sensitivity list of `always@*`:

- Selects of dynamic arrays, queues, and associative arrays

- Built-in method calls of:
  - Dynamic array method size()
  - Queue built-in method size()
  - Associative array methods num(), size(), exists()
- If the base type of array is an unpacked structure, then the array would be ignored for the sensitivity list.

This support is enabled under the following compile-time option:

`-ntb_opts sensitive_dyn.`

### Usage Example

The following example shows the case of usage of dynamic array in an always @\* procedural block:

#### *Example 14-20 Usage of Dynamic Array*

```
module top;
  int da1[];
  int da2[];
  int i = 0;
  always @(*) begin
    $display($stime,, "triggered, da1 size is %d", da1.size());
    da2[i] = da1[i];
  end
  initial begin
    da1 = new[2]; //trigger the always block
    da2 = new[2];
    #1;
    da1[0] = 10; //trigger the always block
    #1;
    i++; //trigger the always block
    #1;
    da1[1] = 20; //trigger the always block
    #0 $display($stime,, "da = %p", da2);
  end
end
```

```
endmodule
```

Execute the following command:

```
% vcs -sverilog -ntb_opts sensitive_dyn test.v -R
```

Output:

```
0 triggered, da1 size is      2
1 triggered, da1 size is      2
2 triggered, da1 size is      2
3 triggered, da1 size is      2
3 da = '{10, 20}
```

---

## Support for Assignment Pattern Expression in Non-Assignment Like Context

VCS supports assignment pattern expression that is used in aggregate data types such as unpacked arrays or unpacked structures in non-assignment like contexts, as specified in the Section 10.9 of the *IEEE Standard SystemVerilog LRM Std. 1800-2012*.

An assignment pattern can be used to construct or deconstruct a structure or an array by prefixing the pattern with the name of a data type to form an assignment pattern expression. An assignment pattern expression has a self-determined data type.

---

## Usage Example

The following example shows the case of usage of assignment pattern expression:

### *Example 14-21 Usage of Assignment Pattern Expression*

```
module test;
    byte payload[];
    typedef byte byte_array[];
    initial begin
        payload = '{8'hab, 8'hcd, 8'hcd, 8'hef};
        if (payload[1:2] == byte_array'{8'hcd, 8'hcd})begin
            $display("Matches");
        end
        else
            $display("Failed");
    end
endmodule
```

## **Limitations**

The following are the limitations for this feature:

- Assignment pattern expressions are not supported for associative arrays.
- Multidimensional arrays of unpacked structure which do not have dynamic types as its members are not supported.
- Assignment pattern expressions that are used in left side expression are not supported.
- Selects of multi-dimensional arrays which result in to another multi-dimensional array are not supported as a member of assignment pattern expression.
- Assignment pattern expressions to a nested structure can have only constants, assignment pattern, and objects/variables of other data types as its members. Any other complex expressions such as function calls, expressions involving operators and so on are not supported.

---

## User-Defined Nettypes

SystemVerilog 1800-2012 supports a user-defined net datatype. You declare it with the `nettype` keyword. You can write a user-defined function to be the resolution function for the new net datatype. This resolution function is usually optional.

See section 6.6.7 “User-defined nettypes” in the SystemVerilog (1800-2012) LRM.

### The Resolution Function

The user-defined resolution function has the following characteristics:

- The return type of the function must match the datatype of the nettype.
- The function uses a dynamic array as a single input argument, but the function cannot modify (resize or write) a dynamic array input argument.
- The function is automatic and preserves no state information.
- Parameterized variants of class methods are possible.
- The SystemVerilog (1800-2012) LRM specifies that the resolution function can be a static class method, but there is a limitation against this in the I-2014.03 release.

#### **Important:**

A user-defined nettype that has multiple drivers must have a resolution function.

## Limitations

The following features are not supported in the declaration of user-defined nettypes:

- NLP activities on nettype nets or interconnect are ignored
- Nettypes across language boundaries (SystemC) and partial elaboration boundaries
- A static class method cannot be used as a resolution function.
- Struct members with unpacked dimensions
- Active driver analysis and VCD or EVCD dumping
- Resolution function that is an interface function or a function within a class scope
- \$deposit system task or VPI for deposit with nettypes

## Example of User-Defined Nettype

In the following code example package `n pkg` includes user-defined nettype `myNet` with a resolution function `resFn`.

```
package npkg;
typedef struct {
    real I;
    real V;
    logic active;
}wShape;
function automatic wShape resFn(wShape dr[]);
    int count = 0;
    begin
        resFn.I = 0.0;
        resFn.V = 0.0;
        resFn.active = 0;
        foreach (dr[i]) begin
```

```

        resFn.I += dr[i].I;
        resFn.V += dr[i].V;
        resFn.active = 1;
        count++;
    end
    resFn.I = resFn.I/count;
    resFn.V = resFn.V/count;
end
endfunction
nettype wShape myNet with resFn;
endpackage

module model;
import npkg::*;
myNet w;
wShape v1, v2;
assign w = v1;
assign w = v2;
initial begin
    v1.active = 1;
    v2.active = 1;
    v1.I = 2.5;
    v1.V = 1.5;
    v2.V = 3.5;
    v2.I = 4.5;
end
endmodule

```

## Example of User-Defined Nettype in Arrays

In the following code example package `npkg` includes user-defined nettype `myNet` with a resolution function `resFn`.

```

`timescale 1 ns / 1 fs

package elec_pkg;

typedef real resistance_type;
typedef real voltage_type;

```

```

typedef struct {
    voltage_type      v = 0;
    resistance_type  r = 1e21;
} electrical_type;

function automatic electrical_type Kirchhoff_Law (input
    electrical_type s[]);
    Kirchhoff_Law = '{0.0, 0.0};
    foreach (s[i]) begin
        Kirchhoff_Law.v += s[i].v;
        Kirchhoff_Law.r += s[i].r;
    end
endfunction

nettype electrical_type electrical with Kirchhoff_Law;
nettype voltage_type      voltage;
endpackage

module nfc_g2_bias_reader(o_ibg40u_tx3_2v5,o_atst_2v5);
    import elec_pkg::*;

    output voltage      o_ibg40u_tx3_2v5;
    input  electrical   o_atst_2v5[3:0];

    electrical_type e1,e2;

    assign o_ibg40u_tx3_2v5 = o_atst_2v5[0].v;

    assign o_atst_2v5[0] = e2;
    assign o_atst_2v5[0] = e1;

    initial begin
        e1 = '{0.0,0.0};
        e2 = '{0.0,0.0};
        #5 e1 = '{1.2,1.2};
        e2 = '{1.4,1.4};
        #5 e1 = '{5.0,5.0};
        e2 = '{5.0,5.0};
        #5 $finish();
    end

    initial $monitor($time,"Voltage is %g V

```

```

",o_ibg40u_tx3_2v5);

endmodule

```

## Example of Nettype MDAs of Type Real

The following code example includes nettype MDAs of type `real`.

```

typedef real T;
function automatic real resfn(input real drv[]);
    resfn=0;
    foreach(drv[i])
        resfn = resfn + drv[i];
endfunction

nettype T myNet with resfn;
module top;
    myNet mem[4][5];
    T v;
    assign mem[1][4] = v;
    initial $display ($time,mem[1][4]);
endmodule

```

## Example of Nettype MDAs of Type Unpacked Struct

The following code example includes nettype MDAs of type `unpacked struct`.

```

typedef struct {logic r; real v;} T;
function automatic T resfn(input T drv[]);
    resfn='{0,0};
    foreach(drv[i]) begin
        resfn.r = resfn.r + drv[i].r;
        resfn.v = resfn.v + drv[i].v;
    end
endfunction

nettype T myNet with resfn;
module top;

```

```

myNet mem[4][5];
T v;
assign mem[1][4] = v;
initial $display ($time,,mem[1][4].r);
initial begin
    //mem[0][3]= v;
    #10 v = '{1, 1.1};
    #10 v = '{1, 2.1};
    #10 v = '{1, 3.1};
end
endmodule

```

## Support for Connecting Nettypes through Tranif Gates

VCS also allows you to connect user-defined nettypes through tranif gates. Therefore, you can have a back-to-back connection in a transistor network.

### *Example 14-22 Example of Nettypes Connected through Tranif Gates*

```

function automatic real resfn (input real drv[]);
    real temp;
    resfn=0;
    temp=drv[0];
    foreach(drv[i]) begin
        if(drv[i] > temp)
            temp=drv[i];
    end
    resfn=temp;
endfunction

nettype real real_net with resfn;

module test;
real_net a[3:0];
assign a[0] = 1;
assign a[2] = 2;

```

```
tranif0 trf(a[1],a[0],ctrl);  
endmodule
```

### Key Points to Note

- Nettypes that are connected through tranif gates must have a resolution function.
- The control signal of tranif gates must be of 2-state data type.
- Inout terminals connected to tranif gates must be of the same nettype.

### Limitations

The feature has the following limitations:

- Nettypes of data types other than real and unpacked structs are not supported.
- Force or release on nettypes connected through tran gates are not supported.

---

## Generic Interconnect Nets

SystemVerilog 1800-2012 supports interconnect nets. Interconnects are very generic without any type associated with them and can be connected to any datatype through a port. VCS treats them as connection points and automatically associates an appropriate type to this connection point, depending on what it's connected to.

You declare them with the `interconnect` keyword. Interconnect nets are generic in that they don't have a datatype. They are limited to connecting ports (on modules and interfaces) and terminals (on

primitives including UDPs). You can't use them on the left or right side of procedural statements or in continuous assignment statements or procedural continuous assignment statements.

See section 6.6.8 “Generic interconnect” in the SystemVerilog (1800-2012) LRM.

You enable generic interconnect nets with the `-sv_interconnect` compile-time option.

## Converting Wire Nets and Logic Variables to Interconnect Nets

You can use the `-xlmr coerce_nettpe` compile-time option and keyword argument to tell VCS to internally convert wire datatype nets and logic datatype variables to interconnect nets.

This internal conversion does not change the datatype you see when debugging, for example, with Verdi.

## Limitations

The following features are not supported in the declaration of interconnect nets:

- Interconnects across language boundaries (SystemC) and partial elaboration boundaries
- Interconnect as part of interface or virtual interfaces
- Verilog-AMS is not supported in presence of interconnect

---

## Support for Associative Array With Unpacked Structure as Key

VCS enables you to use unpacked structure as key to associative array as specified in the Section 7.8.5 of the *IEEE Standards for System Verilog LRM 1800-2012*.

The following example shows how you can use associative array with an index of unpacked structure:

### *Example 14-23 Associative Array With Unpacked Structure as Key*

```
module test;
    typedef struct {byte B; int I[*];} Unpkt;
    int arr[Unpkt];
    Unpkt st0,st1;
    initial begin
        st0 = '{1,'{0:1,1:1}};
        st1 = '{2,'{0:2,1:2}};
        #1 arr[st0] = 32'd11;
        arr[st1] = 32'd22;
        $display("%p",arr);
    end
endmodule
```

### Limitation

The following is the limitation of this feature:

Assignment pattern expression for key is not supported.

---

## Specifying a SystemVerilog Keyword Set by LRM Version at Command Line

You can use the `-sv` compile-time option to specify a version of the IEEE Standard for SystemVerilog (also referred as LRM) in a simulation. This version is the language version used by the VCS and VCS compiler and the keyword set version applied to the design. Furthermore, the `-sv` option enables the SystemVerilog mode that is currently specified by the `-sverilog` option.

In every new version of the LRM, new language keywords can be introduced. These new keywords create potential compilation errors in designs or IPs that use the keywords as identifiers. VCS support the SystemVerilog `'begin_keywords` and `'end_keywords` compiler directives as defined in the IEEE Standard for SystemVerilog (IEEE Std P1800-2012), section 22.14

`'begin_keywords`, `'end_keywords`. You can use these directives to specify a version of the LRM and apply that version's keyword set to the source code encapsulated by the directives. You can use the directives to resolve compilation errors caused by keywords introduced in newer versions of the LRM.

In addition, you can use the `-sv` compile-time option to specify a default LRM version of keyword set to be used in a compilation. This default LRM version applies to the source code that is not encapsulated by the SystemVerilog `'begin_keywords` and `'end_keywords` compiler directives.

## The -sv Compile-Time Option

You can use the `-sv` compile-time option to specify a default LRM version of keyword set to be used in a compilation. This default LRM version applies to source code that is not encapsulated by the SystemVerilog `'begin_keywords` and `'end_keywords` compiler directives,

The syntax of the `-sv` option is:

`-sv=SystemVerilog_LRM_Version`

where,

`SystemVerilog_LRM_Version`

Specifies a version of *IEEE Standard for SystemVerilog* and applies that version's keyword set to the design. The supported LRM versions are:

- 2012
- 2009
- 2005

Note:

An error message is issued if the `-sv` option is specified with `-sverilog`.

For example,

**test.v**

```
class C;  
    rand int implements;
```

```

constraint c1 {
    implements inside {2,3,5};
}
endclass

module m;
    C c=new();
    initial begin
        c.implements = 10;
        $display("implements = %0d\n", c.implements);
    end
endmodule

```

## Compilation command

```
%vcs -sv=2009 test.v
```

The keyword `implements` was introduced in the *IEEE Standard for SystemVerilog* (IEEE Std P1800-2012). In the above example, the `-sv` option specifies the LRM version IEEE Std P1800-2009 at compile time. As a result, `implements` is handled as a regular identifier instead of a keyword. With the `-sv=2012` option, a compilation error is issued.

## The `begin\_keywords and `end\_keywords Compiler Directives

VCS supports the SystemVerilog `'begin_keywords` and `'end_keywords` compiler directives as defined in the *IEEE Standard for SystemVerilog* (IEEE Std P1800-2012), section 22.14 `'begin_keywords`, `'end_keywords`. You can use these directives to specify a version of the LRM and apply that version's keyword set to the source code encapsulated by the directives. You can use the directives to resolve compilation errors caused by keywords introduced in newer versions of the LRM.

For example,

```
`begin_keywords "1800-2009"
class C;
    rand int implements;

    constraint c1 {
        implements inside {2,3,5};
    }
endclass

module m;
    C c=new();
    initial begin
        c.implements = 10;
        $display("implements = %0d\n", c.implements);
    end
endmodule
`end_keywords
```

The keyword `implements` was introduced in the *IEEE Standard for SystemVerilog* (IEEE Std P1800-2012). In the above example, the `'begin_keywords` and `'end_keywords` directives specify the LRM version IEEE Std P1800-2009. As a result, `implements` is handled as a regular identifier instead of a keyword. Without the directives, a compilation error is issued.

---

## Support for .triggered Property with Clocking Block Name

As defined in *IEEE SystemVerilog LRM*, clocking event of a clocking block is available directly by using the clocking block name, regardless of the actual clocking event used to declare the clocking block.

Besides using expression `@ as @(clocking_block_name)` to access clocking block event, VCS also allows the usage of clocking block name with `.triggered` property.

**Note:**

The feature allows the usage of clocking block name with the `.triggered` property only. No other event like properties, such as assignment, passing-through-port, and so on are allowed.

## Usage Examples

The following examples show how you can use clocking block name with `.triggered` property.

*Example 14-24 Example of Usage of Clocking Block with .triggered Property*

```
interface ifc(input clk);

    ...
    logic [7:0] data;

    clocking cb@(posedge clk);
    ...
    output data;
endclocking

modport producer(clocking cb);

endinterface

class driver;

    virtual ifc.producer vif;

    task drive;
        wait(vif.cb.triggered);
        vif.cb.data <= pkt.data;
        ...
    endtask
```

```
task check;
    if (vif.cb.triggered || pkt.size == len) begin
        ...
    end
endtask
endclass
```

---

## Support for Intra Assignment Delay With Non-Blocking Assignments in Program Block

VCS supports intra assignment delay with non-blocking assignment statements in program block with the `-ntb_opts re_nba_sched` compile time option, as specified in the section *9.4.5 of the IEEE Standard System Verilog Std. 1800-2012*.

### Limitations

This option is not supported in NLP flow.

---

## Extensions to SystemVerilog

This section contains descriptions of Synopsys enhancements to SystemVerilog. This section contains the following topics:

- “Unique/Priority Case/IF Final Semantic Enhancements (`-xlrn uniq_prior_final` Compile-Time Option)”
- “Single-Sized Packed Dimension Extension”
- “Covariant Virtual Function Return Types”
- “Self Instance of a Virtual Interface”

- “Support for Shuffle Method for MultiDimensional Arrays”
  - “Enhanced Clocking Block Behavior When Skew is negedge/posedge”
  - “Support for Slice of String Variable”
- 

## **Unique/Priority Case/IF Final Semantic Enhancements (-xlmr uniq\_prior\_final Compile-Time Option)**

The behavior of the compliance checking keywords `unique` and `priority` for `case` and for `if...else if...else` selection statements as defined in the Conditional if-else statement Section 12.4 “Conditional if-else statement” in some cases can cause spurious warnings when used inside a module's continuous assignment or `always` block. By default, VCS evaluate compliance with `unique` or `priority` on every update to the selection statement input.

To force `unique` and `priority` to evaluate compliance only on the stable and final value of the selection input at the end of a simulation timestep, VCS now provides a `-xlmr uniq_prior_final` compile-time option .

This can be useful, for example, when `always_comb` might trigger several times within a simulation time slot while its input values are getting stabilized. The `case` statements can get executed several times during the same time slot if it is valid for combinational blocks. While going through intermediate transitions, the `case` statement might get values that violate the `unique` or `priority` property and cause VCS to report multiple runtime warnings. When it is undesirable to receive intermediate warnings, the `-xlmr uniq_prior_final` compile time option can be used to evaluate compliance for only the final stable value of the input.

## Using Unique/Priority Case/If with Always Block or Continuous Assign

-x1rm uniq\_prior\_final behavior only applies to the use of unique and priority keywords when selection statements are used inside a module's continuous assignment statements or always blocks. The option is not applicable to selection statements in a program block or an initial block.

The following two examples illustrate this behavior:

*Example 14-25 unique case statement at the same timestep*

```
//test.sv:  
module top;  
reg cond;  
bit [7:0] a = 0,b, v1, v2;  
always_comb begin  
    if (cond) begin  
        unique case (a)  
            v1: begin b = 0; $display(" Executing Case  
                           with cond value 1 "); end  
            v2: begin b = 1; $display(" Executing Case  
                           with cond value 1 "); end  
        endcase  
    end  
    else begin  
        unique case (a)  
            v1: begin b = 0; $display(" Executing Case  
                           with cond value 0 "); end  
            v2: begin b = 1; $display(" Executing Case  
                           with cond value 0 "); end  
        endcase  
    end  
end  
  
initial begin  
#1 cond = 1;  
a=a+4; v1=4; v2=4;  
$display("\n TIME %0d ns : cond value %0b, a value %0d",
```

```

        $time, cond, a);
#0 cond = 0;
a=a+1; v1++; v2++;
$display("\n TIME %0d ns: cond value %0b, a value %0d",
        $time, cond, a);
end
endmodule

```

### **Simulation output without -xlrn uniq\_prior\_final:**

```
%> vcs -sverilog test.sv -R

Executing Case with condition value 0
RT Warning: More than one conditions match in 'unique case'
statement.
    "unique_case.sv", line 12, for top.
    Line    13 &    14 are overlapping at time      0.
Executing Case with cond value 0
RT Warning: More than one conditions match in 'unique case'
statement.
    "unique_case.sv", line 12, for top.
    Line    13 &    14 are overlapping at time      0.

TIME 1 ns : cond value 1, a value 4
Executing Case with cond value 1
RT Warning: More than one conditions match in 'unique case'
statement.
    "unique_case.sv", line 6, for top.
    Line    7 &    8 are overlapping at time      1.

TIME 1 ns: cond value 0, a value 5
Executing Case with cond value 0
RT Warning: More than one conditions match in 'unique case'
statement.
    "unique_case.sv", line 12, for top.
    Line    13 &    14 are overlapping at time      1.
```

### **Simulation output with -xlrn uniq\_prior\_final compile-time option:**

```
%> vcs -sverilog test.sv -xlrn uniq_prior_final -R
Executing Case with cond value 0:
```

```

RT Warning: More than one conditions match in 'unique case'
statement.
    "unique_case.sv", line 12, for top.
    Line    13 &    14 are overlapping at time      0.

TIME 1 ns : cond value 1, a value 4
Executing Case with cond value 1

TIME 1 ns: cond value 0, a value 5
Executing Case with cond value 0
RT Warning: More than one conditions match in 'unique case'
statement.
    "unique_case.sv", line 12, for top.
    Line    13 &    14 are overlapping at time      1.

```

**Example 14-26 unique if inside always\_comb**

```

//test.sv
module top;
reg cond;
bit [7:0] a = 0,b;
always_comb begin

unique if (a == 0 || a == 1) $display ("A is 0 or 1");
else if (a == 2) $display ("A is 2");

end

initial begin
#100;
a = 1;
#100 a = 2;
#100 a = 3;
#0 a++;
#0 a++;
#0 a++;
#10 $finish;

end

endmodule

```

## **Simulation output without -x1rm:**

```
%> vcs -sverilog test.sv -R

A is 0 or 1
A is 0 or 1
A is 0 or 1
A is 2
RT Warning: No condition matches in 'unique if' statement.
    "unique_if.sv", line 5, for top, at time 300.
RT Warning: No condition matches in 'unique if' statement.
    "unique_if.sv", line 5, for top, at time 300.
RT Warning: No condition matches in 'unique if' statement.
    "unique_if.sv", line 5, for top, at time 300.
RT Warning: No condition matches in 'unique if' statement.
    "unique_if.sv", line 5, for top, at time 300.
$finish called from file "unique_if.sv", line 17.
```

## **Simulation output with -x1rm uniq\_prior\_final:**

```
%> vcs -sverilog test.sv -x1rm uniq_prior_final -R

A is 0 or 1
A is 0 or 1
A is 0 or 1
A is 2
RT Warning: No condition matches in 'unique if' statement.
    "unique_if.sv", line 5, for top, at time 300.
$finish called from file "unique_if.sv", line 17.
```

## **Using Unique/Priority Inside a Function**

With this enhancement, if unique/priority case statement is used inside a function, VCS not only points to the current case statement but also provides a complete stack trace of where the function is called. The following example illustrate this behavior:

*Example 14-27 unique case used with nested loop inside function*

```
//test.sv
module top;
```

```

int i,j;
reg [1:0][2:0] a, b, c;
bit flag;

function foo;
    for (int i=0; i<2; i++)
        for (int j=0; j<3; j++)
            unique case (a[i][j])
                0: b[i][j] = 1'b0;
                1: b[i][j] = c[i][j];
            endcase
endfunction : foo

always_comb begin
    for(i=0; i<4; i++) begin
        if (i==2)
            foo();
    end
end

initial begin
    a = 6'b00x011;
end

endmodule : top

```

### **Simulation output without the `-x1rm` option:**

```
%> vcs -sverilog test.sv -R
```

RT Warning: No condition matches in 'unique case' statement.  
 "unique\_case\_inside\_func.sv", line 8, for top.foo, at time 0.

RT Warning: No condition matches in 'unique case' statement.  
 "unique\_case\_inside\_func.sv", line 8, for top.foo, at time 0.

### **Simulation output with `-x1rm uniq_prior_final`:**

```
%> vcs -sverilog test.sv -x1rm uniq_prior_final -R
```

RT Warning: No condition matches in 'unique case' statement.

```
"unique_case_inside_func.sv", line 8, for top.foo, at time 0.  
#0 in foo      at unique_case_inside_func.sv:8  
#1 in loop with j= 0      at unique_case_inside_func.sv:7  
#2 in loop with i= 1      at unique_case_inside_func.sv:6  
#3 in top      at unique_case_inside_func.sv:16  
#4 in loop with i= 2      at unique_case_inside_func.sv:14
```

**Note:**

The following limitations must be noted while using the `-x1rm` `uniq_prior_final` option for loop indices:

- It must be written in `for` statement. The `while` and `do...while` are not supported.
- The loop bounds must be the compile-time constants.
- `for(i= lsb; i<msb; i++)`
- Here, `lsb` and `msb` must be compile-time constant, or becomes constant when upper loops get unrolled.
- No other drivers of the loop variable must be in the loop body.

VCS also supports `unique/prior final` in a `for` loop that cannot be unrolled at compile time. For example, if you have a `for` loop whose range could not be determined at compile time and if there are errors during the last evaluation of such a `for` loop, VCS still reports the error. However, loop index information will not be provided. Even if multiple failures occur in different iterations, VCS reports only the last one.

**Important:**

Use `unique/priority case/if` statement only inside the `always` block, continuous assign, or inside a function. If you use it in other places, the final semantic is ignored.

## System Tasks to Control Warning Messages

Two system tasks `$uniq_prior_checkon` and `$uniq_prior_checkoff` enable you to switch on/off runtime warning messages for unique/priority if/case statements. The following example illustrates the use model of these tasks to ignore violations:

*Example 14-28 System tasks to control warning messages*

```
//test.sv
module m;
    bit sel, v1, v2;

//Disable this initial block to display all RT warning
messages
initial
begin
    $display($time, " Priority checker OFF\n");
    $uniq_prior_checkoff();
    #1;
    $display($time, " Priority checker ON\n");
    $uniq_prior_checkon();
end

initial
begin
//violation with this set of values (warning disabled)
    sel = 1'b1;
    v1 = 1'b1;
    v2 = 1'b1;
    #1;
//violation with this set of values (warning enabled)
    sel = 1'b0;
    v1 = 1'b0;
    v2 = 1'b0;
    #1;
end
always_comb begin
unique case(sel)
    v1: $display($time, " Hello");

```

```

v2: $display($time, " World");
endcase
end
endmodule

```

### **Simulation Output:**

```

%> vcs -sverilog test.sv -R

0 Priority checker OFF
0 Hello
0 Hello
1 Priority checker ON
1 Hello
RT Warning: More than one conditions match in 'unique case'
statement.
"system_task_control_warning.sv", line 28, for m.
Line 29 & 30 are overlapping at time 1.

```

## **Controlling Runtime Warning Messages Generated Using Unique/Priority If Constructs**

The `-xlrn uniq_prior_observed` compile-time option allows the runtime warning message to appear in the observed region of the current time step in compliance with the IEEE SystemVerilog LRM Std 1800™-2012 Section Section 12.4.2.1, which states the following:

*“A unique, unique0, or priority violation check is evaluated at the time the statement is executed, but violation reporting is deferred until the Observed region of the current time step. The violation reporting can be controlled by using assertion control system tasks.*

*Once a violation is detected, a pending violation report is scheduled in the Observed region of the current time step. It is scheduled on a violation report queue associated with the currently executing process.”*

The failure messages are reported in the following order:

1. All assert #0 and RT warning messages are interleaved among themselves.
2. All assert final messages are reported after step1 is done.

Consider the following messages:

```
`timescale 1ns/1ns
module top;
reg a,b,c;
always_comb begin
    Unique_case: unique case(1)
        a:$display("matched a at time ",,$time);
        b:$display("matched b at time ",,$time);
        default:;
    endcase
end
initial begin
    a = 0; b = 0;
    #5 a = 1; b = 1;
end
initial
$monitor("\n %t value of a : %b b : %b",$time,a,b);
Assobserved: assert #0 ($onehot({a,b}));
Assfinal: assert final ($onehot({a,b}));
p p1();
endmodule

program p();
initial begin
    #5 top.b = 0;
    #10 $finish;
end
endprogram
```

In this example, in the program block. #5 top.b=0 is assigned a value 0 at time 0 and a value 1 at time 5. Similarly, in the initial block, a and b are assigned a value 0 at time 0 and a value 1 at time 5.

You can compile this example using the following command:

```
% vcs -sverilog -assert svaext -xlrn  
uniq_prior_observed test.v  
  
% simv
```

VCS generates the following output:

```
Warning-[RT-MTOCMUCS] More than one condition match in  
statement test.v, 5  
More than one condition matches are found in 'unique case'  
statement inside top.Unique_case, at time 5ns.
```

Line number 6 and 7 are overlapping.

## Support for Unique0 in Conditional Statements

SystemVerilog provides support for unique0 conditional statements based on the *IEEE SystemVerilog LRM Std 1800™-2012 Section 12.4.2*. The unique0 conditional statements can identify improperly coded case statements that can lead to bugs at a later stage. The conditional statement is used to make a decision about whether a statement must be executed. The syntax for a conditional statement is as follows:

```
conditional_statement ::=  
[ unique_priority ] case ( cond_predicate ) statement_or_null  
unique_priority ::= unique | unique0 | priority  
cond_predicate ::=  
expression_or_cond_pattern { && expression_or_cond_pattern  
}  
expression_or_cond_pattern ::=  
expression | cond_pattern  
cond_pattern ::= expression matches pattern  
endcase
```

When `unique0` is used, it issues a warning if two or more of the case conditions are true at the same time. Alternatively, `unique0` does not issue any violation when none of the conditions matches. The usage of `unique0` enhances the performance and productivity of conditional statements in SystemVerilog assertions.

---

## Usage Example

The following examples illustrate usage of `unique0`:

```
module top;
    bit [2:0] a;
initial
begin
a=3;
end

always @(a)
begin
unique case(a) // values 3,5,6,7 cause a violation report
0,1: $display("0 or 1");
2: $display("2");
4: $display("4");
endcase

unique0 case(a) // values 3,5,6,7 do not cause a violation
report
0,1: $display("0 or 1");
2: $display("2");
4: $display("4");
endcase
end
endmodule
```

If the keyword `unique0` is used, there shall be no violation if no condition is matched.

On the other hand, unique case issues a violation message when no condition matches as follows:

```
Warning-[RT-NCMUCS] No condition matches in statement
uniq0_case.v, 11
No condition matches in 'unique case' statement. 'default'
specification is missing, inside top, at time 0s.
```

## Enhancements to the -xlrn uniq\_prior\_final Compile-Time Option

The -xlrn uniq\_prior\_final compile-time option and keyword argument tells you when:

- A case statement is modified by the unique, unique0, or priority keywords and there is a case item expression that is not unique
- A conditional if statement is modified by the unique, unique0, or priority keywords and both of the following:
  - the conditional expression is not met
  - there is no else statement for the if statement

When these conditions happens, VCS issues a violation report.

The violation reports, contains the following information:

- The hierarchical name of the module instance that contains the case or if statement as described above, or contains a call to a user-defined task or function that contains these statements.
- The value of the evaluation expression which must be true for a for loop statement to continue to iterate.

Also, a violation report is generated in the following scenarios:

- there is a call to a function that contains the `case` or `if` statements as described above
- the function called is a possible assignment expression for the conditional `? :` operator or in the RHS expression in a continuous assignment.

The following code examples illustrate these enhancements.

*Example 14-29 unique case Statement in Multiple Module Instances*

```

function foo (input a);
  unique case (a)
    1'b1: ; ← case item expressions
    1'b1: ; ← are not unique
  endcase
endfunction

module T (input wire a);
  always_comb
    foo(a); ← module T calls the function
  endmodule

module Top;
  reg a;
  T t1(a); ← multiple instances of module T
  T t2(a);
endmodule

initial
begin
  a = 1'b1;
end

endmodule

```

In [Example 14-29](#), the violation reports refers to the hierarchical names of the instances:

```
Warning-[RT-MTOCMUCS] More than one condition match in
statement
  More than one condition matches are found in 'unique case'
  statement inside
    $unit::foo, at time 0s.
```

Line number 3 and 4 are overlapping.

```
#0 in foo at test.v:2
#1 in Top.t1 at test.v:10
```

```
Warning-[RT-MTOCMUCS] More than one condition match in
statement
test.v, 2
  More than one condition matches are found in 'unique case'
  statement inside
    $unit::foo, at time 0s.
```

Line number 3 and 4 are overlapping.

```
#0 in foo at test.v:2
#1 in Top.t2 at test.v:10
```

VCS identifies both the instances and generates two violation reports.

*Example 14-30 for Loop Statement Iteratively Calling a Function with a unique case Statement*

```
function foo (input a);
    unique case (a)
        1'b1:;
        1'b1;; ← case item expressions
    endcase ← are not unique
endfunction

module T (input wire a);
    int P;
    always_comb
        for (int i=0;i<P;i++)
            foo(a); ← for loop that iteratively
    initial begin calls the function
        P = 2;
    end
endmodule

module Top;
    reg a;
    T t1(a);
    initial begin
        a = 1'b1;
    end
endmodule
```

In the `for` loop, the value of `i` must be less than `p`, then there are violation reports because the `unique case` statement does not have unique case item expressions.

In Example 14-30, the violation report includes the value of `i` in the iteration.

```
Warning-[RT-MTOCMUCS] More than one condition match in
statement
test1.v, 2
    More than one condition matches are found in 'unique case'
    statement inside
```

```
$unit::foo, at time 0s.
```

```
Line number 3 and 4 are overlapping.
```

```
#0 in foo at test1.v:2
#1 in Top.t1 at test1.v:12
#2 in loop with i= 0 at test1.v:11
```

```
Warning-[RT-MTOCMUCS] More than one condition match in
statement
test1.v, 2
More than one condition matches are found in 'unique case'
statement inside
$unit::foo, at time 0s.
```

```
Line number 3 and 4 are overlapping.
```

```
#0 in foo at test1.v:2
#1 in Top.t1 at test1.v:12
#2 in loop with i= 1 at test1.v:11
```

**Example 14-31 unique case Statement in a Function Call in a Possible Assignment Expression with the Conditional Operator**

```
function foo (input a);
    unique case (a)
        1'b1: ;
        1'b1: ;
    endcase
endfunction

module Top;
    reg a, b;
    reg cond;

initial
begin
    a = 1'b1;
    cond = 1'b0;
end

assign b = cond ? 1'b1 : foo(a);

endmodule
```

case item expressions  
are not unique

function call is an assignment expression  
of the conditional operator

In Example 14-31, the following is the violation report:

```
Warning- [RT-MTOCMUCS] More than one condition match in
statement
test2.v, 2
    More than one condition matches are found in 'unique case'
statement inside
    $unit::foo, at time 0s.
```

Line number 3 and 4 are overlapping.

```
#0 in foo at test2.v:2
#1 in Top at test2.v:18
```

## Limitations

This enhancement supports only unique/priority case

- unique0 if is not supported.

---

## Single-Sized Packed Dimension Extension

VCS has implemented an extension to a single-sized packed dimension SystemVerilog signals and multidimensional arrays (MDAs). This section provides examples of using this extension for a single-sized packed dimension and explains how VCS expands the single size.

You can use the extension for these basic data types: bit, reg, and wire (using other basic data types with this extension is an error condition) The following is an example:

```
bit [4] a;
```

VCS expands the packed dimension [4] into [0:3].

For packed MDAs, for example:

```
bit [4][4] a;
```

VCS expands the packed dimensions [4][4] into [0:3][0:3].

You can use this extension in several ways. The following is an example of using this extension in a user defined type:

```
typedef reg [8] DREG;
```

The following is an example of using this extension in a structure, union, and enumerated type:

```

struct packed {
    DREG [20][20] arr4;
} [2][2] st1;
union packed {
    DBIT [20][20] arr5;
} [2][2] unl;
enum logic [8] {IDLE, XX=8'bxxxxxxxxx, S1=8'bzzzzzzzz,
S2=8'hff} arr3;

```

The following is an example of a user-defined structure and union with a packed memory or MDA:

```

typedef bit [2][24] DBIT;

typedef reg [2][24] DREG;

typedef struct packed {
    DBIT [20][20] arr1;
} ST;

ST [2][2] st;

typedef union packed {
    DREG [20][20] arr2;
} UN;

UN [2][2] un;

```

You can also use this extension for specifying module ports. For example:

```

module mux2( input wire [3] a,
             input wire [3] b,
             output logic [3] y);

```

You can use this extension in the parameter list of a user-defined function or task. For example:

```
function automatic integer factorial (input [32] operand);
```

You can use this extension in the definition of a parameter. For example:

```
parameter reg [2][2][2] p2 = 8;
```

## Error Conditions

The following are error conditions for this extension:

- Using the dollar sign (\$) as the size. For example:

```
reg [8:$] a;  
reg [$] b;
```

- Using basic data types other than `bit`, `reg`, and `wire`. For example:

```
typedef shortint [8] DREG;
```

---

## Covariant Virtual Function Return Types

VCS supports, as an extension to SystemVerilog, covariant virtual function return types.

A covariant return type allows overriding a superclass method with a return type that is a derived type of the superclass method's return type. Covariant return types minimize the need for dynamic casts (upcasting or downcasting).

### *Example 14-32 Sample code for covariant function return types*

```
class Base;  
    virtual function Base clone();  
        Base b = new this;  
        return b;  
    endfunction  
endclass
```

```

class Derived extends Base;
    virtual function Derived clone();
        Derived d = new this;
        return d;
    endfunction
endclass

```

Without covariant types, the signature of the `Derived::clone()` above would have to be the same as in the `Base` class, like the following:

```

class Derived extends Base;
    virtual function Base clone();
        Derived d = new this;
        return d;
    endfunction
endclass

```

This leads to code like the following for users of the class:

```

Derived d = new;
Base b = d.clone();      // automatic down-cast to Base
Derived d2;
if(!$cast(d2, b)) begin
    b = null;
    $error(...); // some exception
end

```

Instead, with covariant return types, the code is simplified to:

```

Derived d = new;
Derived d2 = d.clone();

```

---

## Self Instance of a Virtual Interface

You can create a self instance of a virtual interface that points to itself when it is initialized. For example:

```
interface intf;
    int data1;
    int data2;
    virtual intf vi;
    initial
        vi = interface::self();
endinterface

module top;
    intf i0();
    initial #1 i0.vi.data1 = 100;
    always @(i0.data1)
        $display("trigger success");
endmodule
```

In this example, the virtual interface named `vi` is initialized with the expression:

```
vi = interface::self();
```

The `interface::self()` expression enables you provide a string variable that is effectively the `%m` format specification of the interface instance that VCS returns for assignment to the virtual interface variable. You use the `interface::self()` expression to initialize virtual interface variables in methodologies like UVM and VMM. It enables you to write components that are configurable with a string `is the %m` of the virtual interface that the component drives or monitors.

The expression `interface::self()` must be entered precisely. Otherwise it is a syntax error. Also notice the required delay (in this case `#1`) in the initialization of virtual interface `vi`. This delay is required to prevent a race condition.

This implementation is in accordance with the IEEE SystemVerilog LRM Std 1800™-2012 Section 9.7 “Fine-grain process control” that specifies:

“The `self()` function returns a handle to the current process, that is, a handle to the process making the call.”

SVA-bind is supported with self instances of virtual interfaces.

Note:

A self instance of a virtual interface is not supported in partition compile.

The following conditions are required for a self instance of a virtual interface:

- The self instance must be defined in the scope.
- The virtual interface type in the interface declaration must be the same as the interface that includes itself.
- Within an interface, you can only use the virtual `interface::self()` expression in a context that is valid for initializing a virtual interface. Any other use of the `interface::self()` expression results in a compilation error.
- Within an interface, you can use the virtual `interface::self()` expression in a context that is valid for initializing a virtual interface. Any other use of the `interface::self()` expression results in a compilation error.

## UVM Example

The following is an example of a self-instance of a virtual interface:

```
/* interface definition */
interface bus_if; //ports.
//signal declaration.

...
initial begin
    uvm_resource_db#(virtual bus_if)::set("*",
        $sformatf("%m"), interface::self());
end
endinterface

/* instantiated bus interface in design. */
//Add "bus()" to module called "top".
bind top bus_if bus();

/*Example config_db usage: */
if(!uvm_config_db#(virtual bus_if)::get(this, "",
    "top.bus", bus))
    `uvm_error("TESTERROR", "no bus interface available");
else
    'uvm_info("build", "got bus_if", UVM_LOW)
```

OR

```
/*Example resource_db usage: */
if(!uvm_resource_db#(virtual
bus_if)::read_by_type(get_full_name(), bus, this))
    `uvm_error("TESTERROR", "no bus interface available");
else
    'uvm_info("build", "got bus_if", UVM_LOW)
```

---

## Support for Shuffle Method for MultiDimensional Arrays

Array ordering methods reorder the elements of any unpacked array. These can be fixed or dynamically sized arrays except for associative arrays. Among the ordering methods, the shuffle method randomizes the order of the elements in the array.

When the shuffle method is called through a multidimensional array element, the method randomizes the order of the elements of the target array, if this array is not an associative array.

If the array calling shuffle method is another multidimensional array, then shuffle method only randomizes the element order of the leftmost dimension. VCS does not re-order the elements of other sub-dimensions.

---

## Use Model

VCS supports the usage of shuffle method for multidimensional arrays through the following compile-time option:

```
-xlrn sv_mda_shuffle
```

---

## Usage Example

The following example shows how the shuffle method works on fixed sized multidimensional integer array:

```
module test;
    int mda[3][4][5];
    initial begin
        mda = '{'{'{0, 1, 2, 3, 4},      '{5, 6, 7, 8, 9},
        '{10, 11, 12, 13, 14}, '{15, 16, 17, 18, 19}}};
```

```

        '{'{20, 21, 22, 23, 24}, '{25, 26, 27, 28, 29},
'{30, 31, 32, 33, 34}, '{35, 36, 37, 38, 39}},
        '{'{40, 41, 42, 43, 44}, '{45, 46, 47, 48, 49},
'{50, 51, 52, 53, 54}, '{55, 56, 57, 58, 59}}
    };
    mda[0][1].shuffle();
    mda[2].shuffle();
    mda.shuffle();
end
endmodule

```

**When you execute `mda[0][1].shuffle()`, the output is,**

```

{
'{'{0, 1, 2, 3, 4}, '{8, 6, 9, 5, 7}, '{10, 11, 12, 13, 14},
'{15, 16, 17, 18, 19}}
'{'{20, 21, 22, 23, 24}, '{25, 26, 27, 28, 29}, '{30, 31,
32, 33, 34}, '{35, 36, 37, 38, 39}}
'{'{40, 41, 42, 43, 44}, '{45, 46, 47, 48, 49}, '{50, 51,
52, 53, 54}, '{55, 56, 57, 58, 59}}
}

```

**In the result, the element in `mda[0][1]` is re-ordered.**

**When you execute `mda[2].shuffle()`, the output is,**

```

{
'{'{0, 1, 2, 3, 4}, '{5, 6, 7, 8, 9}, '{10, 11, 12, 13, 14},
'{15, 16, 17, 18, 19}}
'{'{20, 21, 22, 23, 24}, '{25, 26, 27, 28, 29}, '{30, 31,
32, 33, 34}, '{35, 36, 37, 38, 39}}
'{'{55, 56, 57, 58, 59}, '{45, 46, 47, 48, 49}, '{50, 51,
52, 53, 54}, '{40, 41, 42, 43, 44}}
}

```

**In the result, only the element of `mda[2]` is re-ordered, and the element order of sub-dimension arrays, such as `mda[2][0]` is unaltered.**

**When you execute `mda.shuffle()`, the output is,**

```
{  
'{{40, 41, 42, 43, 44}, '{45, 46, 47, 48, 49}, '{50, 51,  
52, 53, 54}, '{55, 56, 57, 58, 59}}  
'{{0, 1, 2, 3, 4}, '{5, 6, 7, 8, 9}, '{10, 11, 12, 13, 14},  
'{15, 16, 17, 18, 19}}  
'{{20, 21, 22, 23, 24}, '{25, 26, 27, 28, 29}, '{30, 31,  
32, 33, 34}, '{35, 36, 37, 38, 39}}  
}
```

**This example shows how the shuffle method works on dynamically sized multidimensional integer arrays:**

```
int mda[$][5] ={  
'{0, 1, 2, 3, 4}  
'{5, 6, 7, 8, 9}  
'{10, 11, 12, 13, 14}  
'{15, 16, 17, 18, 19}  
}
```

**When you execute `mda[2].shuffle()`, the output is:**

```
{  
'{0, 1, 2, 3, 4}  
'{5, 6, 7, 8, 9}  
'{13, 11, 14, 10, 12}  
'{15, 16, 17, 18, 19}  
}
```

**When you execute `mda.shuffle()`, the output is:**

```
{  
'{15, 16, 17, 18, 19}  
'{5, 6, 7, 8, 9}  
'{10, 11, 12, 13, 14}  
'{0, 1, 2, 3, 4}  
}
```

---

## Enhanced Clocking Block Behavior When Skew is negedge/posedge

By default, VCS overrides the clocking event with the skew when the skew is specified as posedge/negedge. However, you can use the `-ntb_opts no_cb_edge_override` option to avoid overriding the clocking event at input, output, and inout. The following is the behavior of this option at different clocking directions:

- **Input:** Value is sampled at the specified clocking skew delay before the clocking event and the update happens at the clocking event.
- **Output:** The output is updated at the specified clocking skew delay after the clocking event.

### Usage Example

The following example shows the usage of the `-ntb_opts no_cb_edge_override` option.

*Example 14-33 test.v*

```
module top;
    reg clk=0;
    reg [3:0] write, data, sdata;
    always #5 clk = ~clk;

    clocking cb@(posedge clk);
        default input negedge output negedge;
        input data;
        output write;
    endclocking

    always @ (cb.data) begin
        sdata = cb.data;
    end
```

```

always @ (write) begin
$display($time,, "Clocking Block Write");
end
always @ (cb.data) begin
$display($time,, "Clocking Block Read");
end

initial begin
@(posedge clk);
#3 data = 4'ha; // t = 8s
#3 data = 4'hb; // t = 11s
@(posedge clk);
#3 cb.write <= 4'hc; // t = 18s
#3 cb.write <= 4'hd; // t = 21s
end
endmodule

```

Run the example using the following commands:

```

% vcs -sverilog test.v -ntb_opts no_cb_edge_override
% simv

```

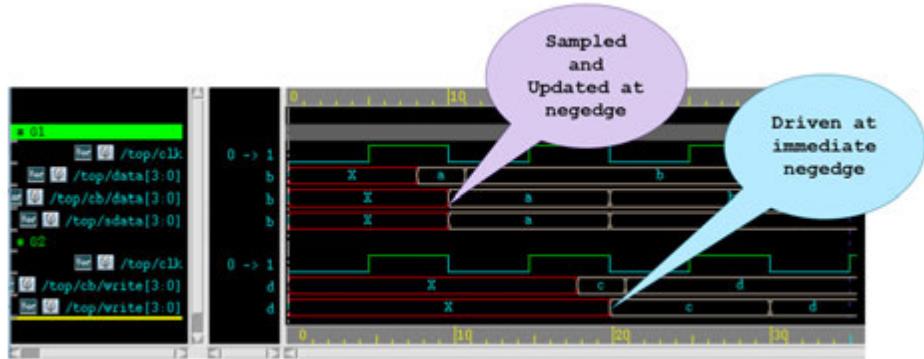
The following table shows the difference in the outputs with and without the `-ntb_opts no_cb_edge_override` option.

*Table 14-1 Outputs*

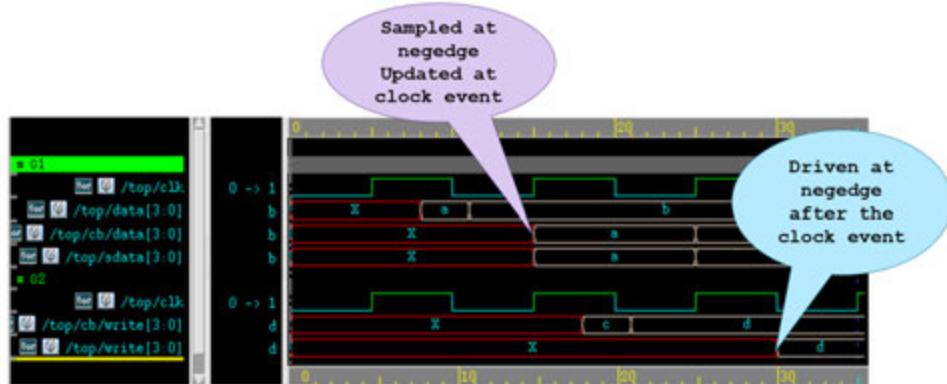
| <b>With the Option</b> | <b>Without the Option</b> |
|------------------------|---------------------------|
| 10 Read                | 10 Read                   |
| 15 Read                | 20 Read                   |
| 25 Read                | 20 Write                  |
| 30 Write               | 30 Write                  |

The following figures shows the difference in behavior in the Wave window.

*Figure 14-1 Default Behavior*



*Figure 14-2 Behavior with the -ntb\_opts no\_cb\_edge\_override option*




---

## Support for Slice of String Variable

VCS supports string functions while accessing slice of string using the `-x1rm sv_string_slice` option. The following example shows the usage of slice operator on string.

*Example 14-34 test.v*

```
module top;
```

```
string s0 = "Hello";
initial begin
$display("%s",s0[0:4]);
$display("%s",s0[0:2]);
end
endmodule
```

Run the example using the following commands:

- % vcs -sverilog -xlrn sv\_string\_slice test.v
- % simv

Following is the output:

```
Hello
Hel
```



# 15

## Aspect Oriented Extensions

---

Aspect-Oriented Programming (AOP) methodology complements the Object-Oriented Programming (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 SystemVerilog allow testbench engineers to design test cases more efficiently, using fewer lines of code.

AOP addresses issues or concerns that prove difficult to solve when using OOP to write constrained-random testbenches. These concerns include the following:

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 the SystemVerilog language is currently limited in the sense that an aspect extension affects the behavior of only a single class. It is useful to enable test engineers to design code that efficiently addresses concerns, such as context-sensitive behavior and unanticipated extensions.

AOP is used in conjunction with object-oriented programming. By compartmentalizing code containing aspects and 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. Aspect-oriented extensions of Native testbench should be used only for constrained-random test specifications with the aim of minimizing code.

Aspect-oriented extensions of Native testbench should not be used to do the following:

- 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 test benches, see the *Reference Verification Methodology User Guide*.

---

## Aspect-Oriented Extensions in SystemVerilog

In SystemVerilog, AOP is supported by a set of directives and constructs that need to be processed before compilation. Therefore, a SystemVerilog program with these aspect-oriented directives and constructs would need to be processed as per the definition of these directives and constructs in SystemVerilog to generate an equivalent SystemVerilog program that is devoid of aspect extensions, and consists of traditional SystemVerilog. Conceptually, AOP is implemented as pre-compilation expansion of code.

This chapter explains how aspect-oriented extensions in SystemVerilog are directives to SystemVerilog compiler as to how the pre-compilation expansion of code needs to be performed.

In SystemVerilog, 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 cannot be nested.

An aspect-oriented extension in SystemVerilog is defined using a new top-level *extends directive*. Terms “aspect” and “extends directive” have been used interchangeably throughout the

document. Normally, a class is extended through derivation, but an extends directive defines modifications to a pre-existing class by doing *in-place* extension of the class. *In-place* extension modifies the definition of a class by adding new member fields and member methods, and changing the behavior of earlier defined class methods, without creating any new subclass(es). That is, aspect-oriented extensions change the original class definition without creating subclasses. These changes affect all instances of the original class that is extended by aspect-oriented extensions.

An extends directive for a class defines a scope in the SystemVerilog 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 Aspect-Oriented Extensions as a Precompilation Expansion

As a precompilation expansion, the Aspect-Oriented Extension code is processed by VCS to modify class definitions that it extends as per the directives in aspect-oriented extensions.

A *symbol* is a valid identifier in a program. Classes and class methods are symbols that can be affected by Aspect-Oriented Extensions. The Aspect-Oriented Extension 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 , “The hide\\_list Details,”](#). The preprocessed input program, now devoid of aspect-oriented extension, 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 bold 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](#)”.

`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 `coverage_group` in a class.

`advice_code`

Specifies to a block of statements.

`statement`

Is a SystemVerilog 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 is created as part of the process of *weaving*, which is a part of precompilation expansion of code. The overall details of the process of “weaving” are explained in [Precompilation 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.

**The proceed statement**

The `proceed` keyword specifies a SystemVerilog statement that can be used within advice code. The `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 , “The `hide\_list` Details,”](#).

## Weaving Advice Into the Target Method

The target method is either the class method or some other new method that is created as part of the process of *weaving*. “Weaving” of all advices in the input program comprises several steps of *weaving of an advice into the target method*. Weaving of an advice into its target method involves the following:

A new method is created with the same method prototype as the target method and with the advice code block as the code block of the new method. This method is referred to as the *advice method*.

The following table shows the rest of the steps involved in weaving of the advice for each type of placement element (*before*, *after*, and *around*).

*Table 15-1 Placement Elements*

| Element | Description                                                                                                                                                                                                                                                                                                                                                                 |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| before  | Inserts a new method-call statement that calls an advice method. The statement is inserted as the first statement to be executed before any other statements.                                                                                                                                                                                                               |
| after   | Creates a new method A with the target method prototype, with its first statement being a call to the target method. Second statement with A is a new method call statement that calls the advice method. All the instances in the input program where the target method is called are replaced by newly-created method calls to A. A is replaced as the new target method. |
| around  | All the instances in the input program where the target method is called are replaced by newly-created method calls to the advice method.                                                                                                                                                                                                                                   |

Within an extends directive, you can specify only one advice that 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*.

### *Example 15-1    **before** Advice*

Target method:

```
class packet;
    task myTask();
        $display("Executing original code\n");
    endtask
```

```
endclass
```

#### Advice:

```
before task myTask ();
    $display("Before in aoel\n");
endtask
```

Weaving of the advice in the target method yields the following.

```
task myTask();
    mytask_before();
    $display("Executing original code\n");
endtask

task mytask_before();
    $display("Before in aoel\n");
endtask
```

Note that the SystemVerilog language does not impose any restrictions on the names of newly-created methods during precompilation expansions, such as *mytask\_before*. Compilers can adopt any naming conventions for those methods that are created as a result of the *weaving* process.

### *Example 15-2 after Advice*

Target method:

```
class packet;
    task myTask();
        $display("Executing original code\n");
    endtask
endclass
```

#### Advice:

```
after task myTask ();
    $display("Before in aoel\n");
endtask
```

Weaving of the advice in the target method yields the following.

```
task myTask_newTarget();
    myTask();
    myTask_after();
endtask

task myTask();
    $display("Executing original code\n");
endtask

task myTask_after ();
    $display("After in 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()`.

### *Example 15-3 around Advice*

Target method:

```
class packet;
    task myTask();
        $display("Executing original code\n");
    endtask
endclass
```

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 15-4    **around** Advice With **proceed***

Target method:

```

class packet;
    task myTask();
        $display("Executing original code\n");
    endtask
endclass

```

Advice:

```

around task myTask ();
    proceed;
    $display("Around in aoel\n");
endtask

```

Weaving of the advice in the target method yields the following:

```

task myTask_around();
    myTask();
    $display("Around in aoe1\n");
endtask

task myTask();
    $display("Executing original code\n");
endtask

```

As a result of weaving, all the method calls to `myTask()` in the input program code are replaced by method calls to `myTask_around()`. The `proceed` statement in the around code is replaced with a call to the target method `myTask()`. Also, `myTask_around()` replaces `myTask` as the target method for `myTask()`.

---

## Precompilation Expansion Details

Precompilation expansion of a program containing the aspect oriented-extensions 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 cannot 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 [Section , “The hide\\_list Details,”](#).

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 *dominate\_list* (see “[dominate\\_list](#)”). 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\_list* always overrides precedence established by the order in which extends directives are coded. Only when the precedence is not established after analyzing the dominate lists of directives, is the order of coding used to define the order of precedence.

Within an extends directive, there is an inherent precedence between advices. Advices that are defined later in the directive have higher precedence than those defined earlier.

Precedence does not change the order between adding of introductions and weaving of advices in the code. Precedence defines the order in which introductions to a class are added to the class, and the order in which advices to methods belonging to a class are woven into the class methods.

### *Example 15-5 Precedence Using **dominates***

```
// Beginning of file test.sv
class packet;
    // Other member fields/methods
    //...
    task send();
        $display("Sending data\n");
    endtask
```

```

endclass

program top ;

    initial begin
        packet p;
        p = new();
        p.send();
    end
endprogram

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 test.sv

```

In [Example 15-5](#), multiple aspect extensions for a class named *packet* are defined in a single SystemVerilog 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`. As `aspect_3` is coded later 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`. It 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. 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 the list, L. This is the order in which these advices are woven into the class method, thereby, affecting the runtime 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 the 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 that how the order of precedence of advices for a class method affects how advices are woven into their target method. Thus, the relative order of execution of advice code blocks. The *before* and *after* advices within an aspect to a target method are unrelated to each other. 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 code block of the *before* advice executes before the target method code block, and the *after* advice code block executes after the target method code block. When an *around* advice is used with a *before* or *after* advice in the same aspect, code weaving depends upon their precedence with respect to each other. Depending upon the precedence of the *around* advice with respect to other advices in the aspect for the same target method, the *around* advice either may be woven before all or some of the other advices, or may be woven after all of the other advices.

As an example, weaving of advices 1, 2, 3, 4 specified in aspect extensions in [Example 15-5](#) leads to the expansion of code in the manner shown in [Example 15-6](#). Advices are woven in the order of increasing precedence {2, 3, 4, 1} as explained earlier.

#### *Example 15-6 After Weaving Advice-2 of Class **packet***

```
// Beginning of file test.sv

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
endclass
```

```

        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 test.sv

```

**Example 15-6 shows how 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. The instances of method calls to the target method `packet::send` are modified, such that the code block from *advice 2* executes after the code block of the target method `packet::send`.**

**Example 15-7** shows how 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. The instances of method call to the target method `packet::send_Created_a` are modified. The code block from **advice 3** executes *around* the code block of the `packet::send_Created_a` method. Also, note that the `proceed` statement from the advice code block is replaced by a `call to send_Created_a`. At the end of this step, `send_around_Created_c` becomes the new target method for weaving of further advices to `packet::send`.

### *Example 15-7 After Weaving Advice-3 of Class packet*

```
// Beginning of file test.sv

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 test.sv

```

**Example 15-8** shows how 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 is different if `advice 4 (before advice)` is defined earlier than `advice 3 (around advice)` within `aspect_3`, as it affects the order of precedence of `advice 3` and `advice`. In this scenario, `advice 3 (around advice)` weaves around the code block from `advice 4 (before advice)`, unlike the current outcome.

### *Example 15-8 After Weaving Advice-4 of Class packet*

```

// Beginning of file test.sv

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 test.sv

```

**Example 15-9 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` are replaced by the method call to `packet::send_Created_f`.**

### ***Example 15-9 After Weaving All Advices {2,3,4,1}of Class `packet`***

```

// Beginning of file test.sv

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 test.sv

```

**When executed, the output of this program is as follows:**

```

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

```

**Example 15-10 shows the program in which `aoe1` dominates `aoe2`.**

### *Example 15-10 Around Advice With **dominates**-I*

```
// Begin file test.sv
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

program top;
    foo f;

    initial begin
        f = new();
        f.myTask();
    end
endprogram
// End file test.sv
```

When **aoe1** **dominates** **aoe2** and when the program is executed, its output is as follows:

```
Executing original code
around in aoe2
around in aoe1
```

[Example 15-11](#) shows the program in which `aoe2` dominates `aoe1`.

### *Example 15-11 Around Advice With **dominates**-II*

```
// Begin file test.sv
class foo;
    int i;
    task myTask();
        $display("Executing original code\n");
    endtask
endclass

extends aoe1 (foo);
    around task myTask();
        proceed;
        $display("around in aoe1\n");
    endtask
endextends

extends aoe2 (foo) dominates (aoe1);
    around task myTask();
        proceed;
        $display("around in aoe2\n");
    endtask
endextends

program top;
    foo f;
    initial begin
        f = new();
        f.myTask();
    end
endprogram
// End file test.sv
```

On the other hand, when `aoe2` dominates `aoe1` as in [Example 15-11](#), the output is as follows:

```
Executing original code
around in aoe1
around in aoe2
```

## Symbol Resolution Details

As introductions and advices defined within `extends` directives are preprocessed as a precompilation expansion of the input program, the preprocessing occurs earlier than the final symbol resolution stage within a compiler. Therefore, it is possible for aspect-oriented extensions code to reference symbols that are added to the original class definition using aspect-oriented extensions. Because advices are woven after introductions are added to the class definitions, advices can be specified for introduced member methods and can reference introduced symbols.

An advice to a class method can access and modify the member fields and methods of the class object to which the class method belongs. An advice to a class function can access and modify the variable that stores the return value of the function.

Furthermore, members of the original class definition can also reference symbols introduced by aspect extensions using the `extern` declarations (?). `Extern` declarations can also be used to reference symbols introduced by an aspect extension to a class in some other aspect extension code that extends the same class.

An introduction that has the same identifier as a symbol that is already defined in the target scope as a member property or member method is not permitted.

## Examples:

### Example 15-12 before Advice on Class Task

```
// Begin file test.sv
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

program top;
    packet p;

    initial begin
        p = new();
        p.foo(100);
    end
endprogram
// End file test.sv
```

The `extends` directive in [Example 15-12](#) 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 reference.

### **Example 15-13 after Advice on Class Function**

```
// Begin file test.sv
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

program top ;
    packet p;

    initial begin
        p = new();
        $display("Output is: %d\n", p.bar());
    end
endprogram

// End file test.sv
```

An advice to a function can access and modify the variable that stores the return value of the function as shown in [Example 15-13](#). 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 as follows:

```
Point 1: Value = 5
Point 2: Value = 5
Point 3: Value = 6
Output is: 6
```

## The `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 are previously visible in the target scope, and it is an error for an extension to introduce a symbol that hides a global or super-class symbol.

The `hide_list` option contains a comma-separated list of options such as:

- The *virtuals* option permits the hiding (that is, overriding) of virtual methods defined in a super class. Virtual methods are the only symbols that may be hidden; global, and file-local tasks and functions may not be hidden. Furthermore, all introduced methods must have the same virtual modifier as their overridden super-class and overriding sub-class methods.
- The *rules* option permits the extension to suspend access rules and to specify advice that changes protected and local virtual methods; by default, extensions cannot change protected and local virtual methods.
- An empty option list removes all permissions, that is, it resets permissions to default.

In [Example 15-14](#), the `print` method introduced by the `extends` directive hides the `print` method in the super class.

### **Example 15-14 Change Permission Using *hide virtuals***

```
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

program test;
    packet tr;
    pbase base;

    initial begin
        tr = new();
        tr.print();
        base = tr;
        base.print();
    end
endprogram
```

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 such permission is granted at an aspect item within an aspect, then the *virtuals-permission* is said to be *on* or the *status* of *virtuals-permission* is said to be *on* at that aspect item and at all the aspect items following that, until a hide list that forfeits the permission is encountered. If *virtuals-permission* is not *on* or the *status* of *virtuals-permission* is not *on* at an aspect item, then the *virtuals-permission* at that item is said to be *off* or the *status* of *virtuals-permission* at that item is said to be *off*.
- b. Permission to suspend access rules and to specify advice that changes protected and local virtual methods (option "rules") is referred to as *rules-permission*. If within an aspect, at an aspect item, such permission is granted, then the *rules-permission* is said to be *on* or the *status* of *rules-permission* is said to be *on* at that aspect item and at all the aspect items following that, until a hide list that forfeits the permission is encountered. If *rules-permission* is not *on* or the *status* of *rules-permission* is not *on* at an aspect item, then the *rules-permission* at that item is said to be *off* or the *status* of *rules-permission* at that item is said to be *off*.

Permission for one of the above types of hide permissions does not affect the other. Status of *rules-permission* and *hide-permission* varies with the position of an aspect item within the aspect. Multiple `hide_list(s)` may appear in the extension. In an aspect, whether an introduction or an advice that can be affected by hide permissions is permitted to be defined at a given position within the aspect extension is determined by the status of the relevant hide permission at the position. A `hide_list` at a given position in an aspect can change the status of *rules-permission* and/or *virtuals-permission* at that position and all following aspect items until any hide permission status is changed again in that aspect using `hide_list`.

[Example 15-15](#) illustrates how the two hide permissions can change at different aspect items within an aspect extension.

### *Example 15-15 Hide Permissions*

```
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();
        rules_test();
    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

program test;
    packet tr;

initial begin
    tr = new();
    tr.print1();
    tr.foo();
    tr.print2();
end
endprogram

```

## Examples

Introducing new members into a class:

[Example 15-16](#) shows how aspect-oriented extensions can be used to introduce new members into a class definition. `myaspect` adds a new property, constraint, coverage group, and method to the *packet* class.

### *Example 15-16 Introducing New Member*

```
class packet;
    rand bit[31:0] hdr_len;
endclass

extends myaspect(packet);
    integer sending_port;
    event cg_trigger;

constraint con2 {
    hdr_len == 4;
}

covergroup cov2 @(cg_trigger);
    coverpoint sending_port;
endgroup

task print_sender();
    $display("Sending port = %0d\n", sending_port);
endtask
endextends

program test;
    packet tr;

    initial begin
        tr = new();
        void'(tr.randomize());
        tr.sending_port = 1;
        tr.print_sender();
        -> tr.cg_trigger;
    end
endprogram
```

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, aspect-oriented extensions defined in the manner shown

in [Example 15-17](#) 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 15-17 Non-Permissible Introduction*

```
class foo;
    rand integer myfield;
    integer x;
endclass

extends myaspect(foo);
    integer x;

    constraint con1 {
        myfield == 4;
    }
endextends

program test;
    foo tr;

    initial begin
        tr = new();
        $display("Non-permissible introduction error....!");
        void'(tr.randomize());
    end
endprogram
```

## Examples of Advice Code

In Example 15-18, the `extends` directive adds advices to the `packet::send` method.

### Example 15-18 before-after Advices

```
// Begin file test.sv
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

program test;
    packet p;

    initial begin
        p = new();
        p.send();
    end
endprogram

// End file test.sv
```

When Example 15-18 is executed, the output is as follows:

```
Before sending packet
Sending data
After sending packet
```

In Example 15-19, extends directive myaspect adds advice to turn off constraint c1 before each call to the foo::pre\_randomize method.

### Example 15-19 Turn-off Constraint Using **before** Advice

```
class foo;
    rand integer myfield;

    constraint c1 {
        myfield == 4;
    }
endclass

extends myaspect(foo);

    before function void pre_randomize();
        c1.constraint_mode(0);
    endfunction
endextends

program test;
    foo tr;

    initial begin
        tr = new();
        void'(tr.randomize());
        $display("myfiled value = %d, constraint mode OFF (!= 4)!", tr.myfield);
    end
endprogram
```

In Example 15-20, extends directive myaspect adds advice to set a property named valid to 0 after each call to the foo::post\_randomize method.

### *Example 15-20 Change Property Value After **post-randomize()***

```
class foo;
    integer valid;
    rand integer myfield;

    constraint c1 {
        myfield inside { [0:6] };
    }
endclass

extends myaspect(foo);
    after function void post_randomize();
        if (myfield > 6)
            valid = 0;
        else
            valid = 1;
    endfunction
endextends

program test;
    foo tr;

    initial begin
        tr = new();
        void'(tr.randomize());
        $display("valid = %0d ", tr.valid);
    end
endprogram
```

[Example 15-21](#) shows an aspect extension that defines an around advice for the class method, `packet::send`. When the code in example is compiled and run, the around advice code is executed instead of original `packet::send` code.

### **Example 15-21 Changing Test Functionality Using *around* Advice**

```
// Begin file test.sv
class packet;
    integer len;
    task setLen( integer i);
        len = i;
    endtask

    task send();
        $display("Sending data\n.");
    endtask
endclass

program test;
    packet p;

    initial begin
        p = new();
        p.setLen(5000);
        p.send();
        p.setLen(10000);
        p.send();
    end
endprogram

extends myaspect(packet);
    around task send();
        if (len < 8000)
            proceed;
        else
            $display("Dropping packet\n");
    endtask
endextends

// End file test.sv
```

This [Example 15-21](#) 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 as follows:

Sending data  
Dropping packet

# 16

## Using Constraints

---

This chapter explains VCS support for the following constraints features:

- “Support for Array Slice in Unique Constraints”
- “Support for Object Handle Comparison in Constraint Guards”
- “Support for Pure Constraint Block”
- “Support for SystemVerilog Bit Vector Functions in Constraints”
- “Inconsistent Constraints”
- “Constraint Debug”
- “Constraint Guard Error Suppression”
- “Support for Array and Cross-Module References in std::randomize()”

- “Support for Cross-Module References in Constraints”
- “State Variable Index in Constraints”
- “Using DPI Function Calls in Constraints”
- “Using Foreach Loops Over Packed Dimensions in Constraints”
- “Randomized Objects in a Structure”
- “Support for Typecast in Constraints”
- “Strings in Constraints”
- “SystemVerilog LRM 1800™-2012 Update”
- “Enhancement to the Randomization of Multidimensional Array Functionality”
- “Supporting Random Array Index”
- “Supporting System Function Calls”
- “Supporting Foreach Loop Iteration over Array Select”
- “Support for Enumerated Type Methods in a Constraint Expression”

## Support for Array Slice in Unique Constraints

You can specify slices of unpacked array variables in unique constraints as defined in the IEEE SystemVerilog LRM Std 1800™-2012 Section 18.5.5 “Uniqueness constraints”.

### *Example 16-1 Array Slice in Unique Constraint*

```
module top;
```

```

class c;
    rand bit[2:0] a[7];
    rand bit[2:0] b;

    constraint cst1 {
        unique { a[0:2], a[6], a[3:5], b };
    }
endclass: c

c c1;

initial
begin
    c1 = new;
    c1.randomize();
end

endmodule: top

```

In [Example 16-1](#), the array slices, `a[0:2]` and `a[3:5]` are used in the specification of the unique constraint `cst1`.

## **Limitation**

Specifying loop variables of the `foreach` statement inside array slices is not supported with unique constraints.

### *Example 16-2 Loop Variables Inside an Array Slice*

```

class C;
    rand int x[5];

    constraint cst {
        foreach (x[i]) {
            if (i > 0)  unique { x[i-1+:2] };
        }
    }
endclass

```

```

program automatic test;
    C obj = new;
    initial obj.randomize;
endprogram

```

In [Example 16-2](#), the loop variable *i* is specified in the array slice *x[i-1+:2]*. The following error message is issued:

```

Error-[NYI-CSTR-AS] NYI constraint: array slice
test.sv, 6
$unit, "this.x[(i - 1)+:2]"
The expression 'this.x[(i - 1)+:2]' contains an array slice
that is not yet supported in unique constraint.
Please remove the array slice from the unique constraint,
or replace it with entire array or simple array select.

```

## Support for Object Handle Comparison in Constraint Guards

You can use the equal `==` and not equal `!=` operators to specify object handle comparisons in constraint guards as defined in the IEEE SystemVerilog LRM Std 1800™-2012 Section 18.5.13 “Constraint guards”. You can compare between two object handles or between an object handle and the value **null**. The object handles being compared must be instances of the same class. Using object handle comparison as constraint guards can prevent potential randomization errors caused by non-existent or incorrect object handles.

### *Example 16-3 Comparison Between Two Object Handles*

```

module top;

class A;
    rand int i;

```

```

constraint cst0 {
    i > 0;
    i < 10;
}
endclass: A

class C;
    rand bit[3:0] x;
    rand A a1,a2,a3;

    constraint cst1 {
        if (a1 == a3) { x == 3; }
        if (a1 != a3) { x == 5; }
    }

    function new;
        a1 = new;
        a2 = new;

        a3 = a2;
    endfunction

endclass: C

C c1;

initial
begin
    c1 = new;
    repeat(3) begin
        c1.randomize();
        $display("x = %p\ta1 = %p\ta2= %p", c1.x, c1.a1, c1.a2);
    end

    c1.a1 = c1.a2;

    repeat(3) begin
        c1.randomize();
        $display("x = %p\ta1 = %p\ta2= %p", c1.x, c1.a1, c1.a2);
    end
end

```

```
endmodule: top
```

In [Example 16-3](#), a different constraint is applied to the variable `x` depending on the object handles `a1` and `a3`. If `a1` equals to `a3`, `x` is constrained to 3. If `a1` is not equal to `a3`, `x` is constrained to 5. The following is the result of the example:

```
x = 5  a1 = '{i:2}      a2= '{i:5}
x = 5  a1 = '{i:8}      a2= '{i:5}
x = 5  a1 = '{i:9}      a2= '{i:1}
x = 3  a1 = '{i:1}      a2= '{i:1}
x = 3  a1 = '{i:3}      a2= '{i:3}
x = 3  a1 = '{i:7}      a2= '{i:7}
```

#### *Example 16-4 Comparison Between an Object Handle and Null*

```
module top;

class A;
    rand int i;

    constraint cst0 {
        i > 0;
        i < 10;
    }
endclass: A

class C;
    rand bit[3:0] x;
    rand A a1;
    A a2;

    constraint cst1 {
        if (a2 != null) { x == 3; }
    }

    function new;
        a1 = new;
    endfunction
```

```

endclass: C

C c1;

initial
begin
    c1 = new;
    repeat(3) begin
        c1.randomize();
        $display("x = %p\ta1 = %p\ta2 = %p", c1.x, c1.a1, c1.a2);
    end

    c1.a2 = new;
    repeat(3) begin
        c1.randomize();
        $display("x = %p\ta1 = %p\ta2 = %p", c1.x, c1.a1, c1.a2);
    end
end

endmodule: top

```

In [Example 16-4](#), the 4-bit variable `x` is constrained to value 3 when the object handle `a2` is not `null`. The following is the result of the example:

```

x = 6  a1 = '{i:5}      a2 = null
x = 1  a1 = '{i:5}      a2 = null
x = 9  a1 = '{i:1}      a2 = null
x = 3  a1 = '{i:1}      a2 = '{i:0}
x = 3  a1 = '{i:3}      a2 = '{i:0}
x = 3  a1 = '{i:7}      a2 = '{i:0}

```

## Limitations

The following are the limitations with this functionality:

- Object handles returned by function calls cannot be used for comparison in constraint guards.

- An array of objects cannot be used for comparison in constraint guards. Comparing singleton members is supported. The following constraint specification results in a “Not Yet Implemented” runtime error.

```
constraint cst1 {
    foreach (a_arry[j]) {
        if (a_arry[j] != null) { x inside { [2:5] }; }
    }
}
```

Error message:

```
Error-[NYI] Not Yet Implemented
orig_null.sv, 18
Feature is not yet supported: objects in object
(in)equality must currently refer to singleton members
(no array elements, function calls, null, ...)
```

## Support for Pure Constraint Block

You can specify pure constraint block as defined in the IEEE SystemVerilog LRM Std 1800™-2012 Section 18.5.2 “Constraint inheritance”. A pure constraint block, specified with the `pure` keyword defines a constraint prototype in a virtual class. The constraint implementation is provided when a non-virtual class is derived from the virtual class. The constraint that overrides a pure constraint block can be declared using a constraint block, a constraint prototype or an external constraint of the same name in the body of the derived class.

### *Example 16-5 Pure Constraint Implementation in a Derived Class*

```
virtual class B;
    pure constraint t1;
```

```

        pure constraint q1;
endclass

class C extends B;

    rand int z,z1;
    randc int x,y;
    rand bit a,b;

    constraint t1{
        x inside {2,3,5};
        y inside {3,4,7};
        z==x+y;
    }

    constraint q1{
        z1==(a+b);
    }
endclass

module m;
    C c=new();
    initial begin
        repeat (3) begin
            c.randomize();
            $display("x = %0d y = %0d z = %0d\n", c.x, c.y, c.z);
            $display("a = %0d b = %0d z1 = %0d\n", c.a, c.b, c.z1);
        end
    end
endmodule

```

In [Example 16-5](#), the pure constraint block is declared in the virtual base class `B`. The constraints of the pure constraint block are implemented in class `C` that is derived from class `B`. The following is the result of the example:

```

x = 5   y = 7   z = 12
a = 1   b = 0   z1 = 1
x = 2   y = 3   z = 5
a = 0   b = 1   z1 = 1

```

```
x = 3  y = 4  z = 7
a = 0  b = 0  z1 = 0
```

**Example 16-6 Pure Constraint Implementation in a Hierarchy of Derived Classes**

```
virtual class A;
    pure constraint q;

    virtual function obj ();
        endfunction
    endclass

    virtual class B extends A;
        pure constraint s;
        int i;

        function obj ();
            i=10;
        endfunction
    endclass

    class C extends B;
        rand bit [7:0]x,y,z;
        rand bit [3:0]a;

        constraint q {
            x < 8'h80; y < 8'h80; z < 8'h80;
            z == x+y;
        }

        constraint s{
            a inside { 0, 10 };
        }
    endclass

    module top;
        C c=new;
        initial begin
            repeat(2) begin
```

```

    c.randomize();
    c.obj();
    $display("x = %0d  y = %0d  z = %0d", c.x, c.y, c.z);
    $display("a = %0d\n", c.a);
end
end
endmodule

```

In [Example 16-6](#), the pure constraint blocks are declared in the virtual base class A and the virtual class B that is derived from class A. The constraints of the pure constraint blocks are implemented in class C that is derived from class B. The following is the result of the example:

```
x = 91  y = 20  z = 111
a = 10
```

```
x = 13  y = 45  z = 58
a = 10
```

### *Example 16-7 Pure Constraint Error With Missing Constraint Implementation*

```

virtual class A;
  rand int i;
  pure constraint t;
endclass

class B extends A ;
  rand bit [7:0]x,y,z;
endclass

module m;
  B b=new;

  initial begin
    b.randomize();
  end
endmodule

```

In [Example 16-7](#), the pure constraint block is declared in the virtual base class A. The derived class B does not include the constraint implementation for the pure constraint block. As a result, the following error message is issued:

```
Error-[CSTR-PCNI] 'pure' constraint not implemented
test.v, 6
$unit
'pure' constraint 't' has not been implemented in class 'B'
```

#### *Example 16-8 Pure Constraint Error With Missing Virtual Declaration*

```
class A;
    pure constraint c1;
endclass

class B;
    rand int x;
    randc int y;

    constraint c1 {
        x == 12;
        y inside {2,3,5};
    }
endclass

program test;
    B c ;
    initial begin
        c=new();
        c.randomize(x,y);
    end
endprogram
```

In [Example 16-8](#), the pure constraint block is declared in the base class A. Pure constraint blocks must be declared inside a virtual class. As a result, the following error message is issued:

```
Error-[CSTR-IUPC] Illegal use of pure constraint
test.v, 2
$unit
    Only 'virtual' (i.e. abstract) class may contain 'pure'
constraints.
```

### *Example 16-9 Pure Constraint Error With Mismatched Prototypes*

```
virtual class A;
    pure static constraint q1;
endclass;

virtual class B extends A;
    pure static constraint v1;
endclass;

class C extends B;
    rand bit [3:0]a[3];
    rand bit [3:0]b;
    rand int dyn_arr[];

constraint q1 {
    foreach ( dyn_arr [k] )
        { dyn_arr [k] > 10; }

    dyn_arr.sum() == 500;
}

constraint v1 { unique {a, b}; }
endclass

module m;
    C c=new();
    initial begin
        c.dyn_arr =new[10];
        c.randomize();
    end
endmodule
```

In [Example 16-9](#), the pure constraint blocks are declared in the virtual base class A and the virtual class B that is derived from class A. The constraints of the pure constraint blocks are implemented in class C that is derived from class B. However, the constraint declarations in class C do not include the static keyword that is specified in the pure constraint blocks. As a result, the following error messages are issued:

```
Error-[CSTR-PCPM] 'pure' constraint prototype mismatch  
negative_pureconstraint3.v, 14  
$unit  
    The declaration of constraint 'q1' in class 'C' must match  
    its corresponding  
        'pure' constraint declaration in its base class hierarchy.
```

```
Error-[CSTR-PCPM] 'pure' constraint prototype mismatch  
negative_pureconstraint3.v, 21  
$unit, "constraint v1 { unique {this.a, this.b}; }"  
    The declaration of constraint 'v1' in class 'C' must match  
    its corresponding  
        'pure' constraint declaration in its base class hierarchy.
```

---

## Support for SystemVerilog Bit Vector Functions in Constraints

You can specify the following SystemVerilog bit vector functions in constraints:

- ["\\$countones Function"](#)
- ["\\$onehot Function"](#)
- ["\\$onehot0 Function"](#)
- ["\\$countbits Function"](#)

- “\$bits Function”

The \$countones, \$onehot, \$onehot0, \$countbits and \$bits are defined as bit vector system functions in the IEEE SystemVerilog LRM Std 1800™-2012. These functions are supported in the constraint context as an operator to the expression in its argument and can be used to create an iterated constraint expression, similar to the use of array reduction methods in constraints.

The following example explains the differences between handling \$countones, \$onehot, \$onehot0, and \$countbits as functions and as expressions in the constraint context:

```
rand bit [3:0] vector;

constraint cst { $countones (vector) == 4; }
```

As defined in IEEE SystemVerilog LRM Std 1800™-2012 Section 18.5.12, the semantic restrictions related to a function call in constraints require the solver to solve the random variable as a function argument first and then use the return value of the function as a state variable. In this example, if \$countones is treated as a system function, the random variable, `vector`, is used as a function argument. The random variable is unconstrained and it can be assigned any value between 4'b0000 and 4'b1111. For example, `vector` is randomized to a value 4'b1010. The function \$countones returns a value of 2 based on the vector value of 4'b0101. A constraint solver failure is issued because the return value of the function 2 is not equal to 4. In this case, there is a 31/32 chance that a constraint solver failure is issued. This is unlikely the intent.

If \$countones is treated as an operator to the expression in its argument, it can be used to create iterative expression involving the bits of the expression. In this case, you can constrain how many bits of the expression is 1'b1. The constraint \$countones (vector) == 4 is considered as follows:

```
vector[0] + vector[1] + vector[2] + vector[3] == 4
```

The solver generates a solution for the random variable vector: 4'b1111. This is the most likely the intent when \$countones, \$onehot, \$onehot0, and \$countbits are used in the constraint context and is the behavior supported by VCS.

For more information on the bit vector functions, see the IEEE SystemVerilog LRM Std 1800™-2012 Section 20.9 “Bit vector system functions”.

---

## \$countones Function

The \$countones system function returns an integer equal to the number of bits in the expression having value 1. You can use \$countones as a constraint expression to specify the number of bits inside the \$countones expression to be randomized to value 1.

### *Example 16-10 Using \$countones in Constraint Specification*

```
module top;

class c;
    rand bit[7:0] b;

    constraint cst1 {
        $countones (b) == 3;
    }
endclass: c
```

```

c c1;

initial
begin
    c1 = new;
    repeat(5) begin
        c1.randomize();
        $display("b = %b", c1.b);
    end
end

endmodule: top

```

In [Example 16-1](#), the 8-bit variable `b` is constrained to generate a random value with exactly 3 bits having value 1. The following is the result of the example.

```

b = 10000011
b = 01010001
b = 10010010
b = 01100001
b = 10001100

```

## \$onehot Function

The `$onehot` function returns true if one and only one bit in the expression have value 1. You can use `$onehot` as a constraint expression to specify one and only one bit inside the `$onehot` expression to be randomized to value 1.

### *Example 16-11 Using \$onehot in Constraint Specification*

```

module top;

class c;
    rand bit[7:0] b;

```

```

constraint cst1 {
    $onehot (b);
}
endclass: c

c c1;

initial
begin
    c1 = new;
    repeat(5) begin
        c1.randomize();
        $display("b = %b", c1.b);
    end
end

endmodule: top

```

In [Example 16-4](#), the 8-bit variable `b` is constrained to generate a random value with one and only one bit having value 1. The following is the result of the example.

```

b = 00000001
b = 01000000
b = 00001000
b = 10000000
b = 01000000

```

## \$onehot0 Function

The `$onehot0` function returns true if zero or one bit in the expression have value 1. You can use `$onehot0` as a constraint expression to specify zero or one bit inside the `$onehot0` expression to be randomized to value 1.

### *Example 16-12 Using \$onehot0 in Constraint Specification*

```
module top;

class c;
    rand bit[2:0] b;

    constraint cst1 {
        $onehot0 (b);
    }
endclass: c

c c1;

initial
begin
    c1 = new;
    repeat(5) begin
        c1.randomize();
        $display("b = %b", c1.b);
    end
end

endmodule: top
```

In [Example 16-12](#), the 3-bit variable `b` is constrained to generate a random value with zero or one bit having value 1. The following is the result of the example.

```
b = 100
b = 000
b = 100
b = 001
b = 010
```

---

## \$countbits Function

The `$countbits` function counts the number of bits that have a specific set of values (0, 1, x, z) in a bit vector. The syntax is as follows:

```
$countbits (expression, control_bit {, control_bit} )
```

The control bit is a 1-bit logic that can have '0, '1, 'x, or 'z values. If a value with a width greater than 1 is passed, only the least significant bit (LSB) is used. If any individual value appears more than once in the control bits, it is treated exactly as if it had appeared once.

The `$countbits` function returns an integer that is equal to the number of bits in the expression whose values match one of the control bit entries.

For example,

- `$countbits (expression, '1)` returns the number of bits in the expression having value 1.
- `$countbits (expression, '1, '0)` returns the number of bits in the expression having values 1 or 0.

Note:

As SystemVerilog constraints support only two-state values. The use of 'x or 'z as control bits results in an error message when used in the constraint context.

The expression argument is of a `bit-stream` type. For more information on the bit vector functions, see IEEE SystemVerilog LRM Std 1800™-2012 Section 20.9 “Bit vector system functions”.

### **Example 16-13 Using \$countbits in Constraint Specification**

```
program automatic test;
    class cls;
        rand bit [7:0] driver_port1, driver_port2;

        constraint c1 {
            //randomize 'driver_port1' in such a way that
            // number of "ones" or
            // no of active driver ports should be two
            $countbits(driver_port1, 1) == 2;

            // randomize 'driver_port2' in such a way that
            // number of "zeros" should be four
            $countbits(driver_port2, 0) == 4;
        }
    endclass

    cls obj = new;

    initial begin
        obj.randomize;
        assert($countbits(obj.driver_port1, 1) == 2);
        assert($countbits(obj.driver_port2, 0) == 4);
    end
endprogram
```

---

## **\$bits Function**

The `$bits` system function returns the number of bits required to hold an expression as a bit stream. The return type is `integer`. The syntax is as follows:

```
$bits (expression | data_type)
```

The value returned by `$bits` is determined without actual evaluation of the expression that it encloses. An error message is issued if you enclose a function that returns a dynamically sized data type. The `$bits` return value is valid during elaboration only if the expression contains fixed-size data types.

The `$bits` system function returns 0 when called with a dynamic sized expression that is empty. An error message is issued,

- When you use the `$bits` system function directly with a dynamically sized data type identifier.
- When you use the `$bits` system function on an object of an interface class type.

For more information on the bit vector functions, see IEEE SystemVerilog LRM Std 1800™-2012 Section 20.6.2 “Expression size system function”.

#### *Example 16-14 Using \$bits in Constraint Specification*

```
program automatic test;

    class cls1 #(parameter width1 = 8, width2 = 32);

        rand bit [width1-1:0] r1;
        rand bit [width2-1:0] r2;
        rand bit [7:0] r3;

        constraint c1 {
            r2[$bits(r1)-1:0] == '1; // randomize r2, such
                                // that LSB bits
                                // ($bits(r1) no of bits)
                                // to all ones
            r3 == $bits(r1); // randomize r3, such that r3
                            // should be equal to $bits(r1) or
                            // width of r1
        }
    endclass
```

```

cls1 #() obj1 = new; // object obj1 holds default parameter
                  values 8(w1), 32(w2)
                  // so, $bits(r1) will be '8'

cls1 #(4, 16) obj2 = new; // object obj2 holds over-
                           ridden parameter values
                           4(w1), 16(w2)
                           // so, $bits(r1) will be '4'

initial begin
    obj1.randomize;
    assert(obj1.r3 == 8);
    assert(obj1.r2[7:0] == '1');

    obj2.randomize;
    assert(obj2.r3 == 4);
    assert(obj2.r2[3:0] == '1');

end
endprogram

```

## Inconsistent Constraints

VCS correctly identifies inconsistent constraints while trying to find the minimal set causing the inconsistency. VCS supports two options to find inconsistent constraints: binary search and linear search. You can use two new options to set larger timeout values. The default timeout values for each iteration of the constraint solver are 100 seconds for the binary search and 10 seconds for the linear search. You can set larger timeout values in seconds. For example:

```
% simv +ntb_binary_debug_solver_cpu_limit=200
% simv +ntb_linear_debug_solver_cpu_limit=20
```

**Note:**

If the constraint solver timeout value is too low, VCS may not be able to find the minimal set of conflicting constraints. If the solver timeout value is too high, performance may degrade while finding a conflict. Therefore, setting optimal timeout values is important.

Inconsistent constraints are non-fatal by default. VCS continues to run after a constraint failure. Use the

+ntb\_stop\_on\_constraint\_solver\_error=0|1 option, where 1 enables stop on first error and 0 disables stop on first error to control how VCS handles these inconsistencies. For example, to make VCS stop the simulation on the first constraint failure, use the following command line:

```
simv +ntb_stop_on_constraint_solver_error=1
```

When VCS detects inconsistent constraints, the default printing mode only displays the failure subset. For example:

The constraint solver failed when solving the following set of constraints:

```
rand integer y; // rand_mode = ON
rand integer z; // rand_mode = ON
rand integer x; // rand_mode = ON
constraint c // (from this) (constraint_mode = ON)
{
  ( x < 1 ) ;
  ( x in { 3 , 5 , 7 : 11 } ) ;
}
```

You can use the

`+ntb_enable_solver_trace_on_failure=0|1|2|3 runtime`  
option as follows:

- 0 Print a one-line failure message with no details.
- 1 Print only the failure subset (this is the default).
- 2 Print the entire constraint problem and the failure subset.
- 3 Print only the failure problem. This is useful when the solver fails to determine the minimum subset.

---

## Constraint Debug

Generally, there are two kinds of constraint debug scenarios. In the first scenario, VCS solves the random variables but the you wish to get a better understanding of how the random variables are solved. This is about debugging the solved values. In the second scenario, VCS either times out when solving or solves after a long time. This is about performance debug.

The following sections describe the VCS features that can help with these kind of constraint debugs.

- “Partition”
- “Randomize Serial Number”
- “Solver Trace”
- “Test Case Extraction”
- “Using multiple `+ntb_solver_debug` arguments ”
- “Summary for the `+ntb_solver_debug` Option”

- “Support for Save and Restore Stimulus”
- 

## Partition

Whether it is `std::randomize` or the randomization of a class object, it generally involves one or more state and random variables. Constraints are used to describe relationships between these variables. An important concept of constrained randomization is the notion of partitions. In other words, a randomize call is partitioned into one or more smaller constraint problems to solve. At runtime, VCS groups all the related random variables involved in each randomization into one or more partitions. If there are no constraints between two random variables, they are not solved in the same partition. Here is an example to illustrate this concept:

```
class myClass;
    rand int x;
    rand int y;
    rand int z;
    rand byte a;
    rand byte b;
    bit c;
constraint m {
    x > z;
    c -> a == b;
}
constraint n {
    y > 0;
}

myClass obj = new;
obj.randomize(); // 1st randomize() call
obj.randomize() with {x!=y;}; // 2nd randomize() call
```

For the first randomize call, the following constraints are used to solve the five random variables:  $x$ ,  $y$ ,  $z$ ,  $a$ , and  $b$  and VCS creates three partitions for these random variables.

```
x > z;           // from the constraint block m
c -> a == b;    // from the constraint block m
y > 0;          // from the constraint block n
```

The random variables  $x$  and  $z$  are grouped in one partition because of a constraint ( $x > z$ ) relating the two together.

The random variables  $a$  and  $b$  are grouped in another partition because of the constraint ( $c -> a == b$ ).

There are no constraints between  $y$  and any other random variable. So,  $y$  is on a third partition of its own.

Because the random variables from different partitions are not constrained together, they do not have to be solved in any particular order.

For the second `randomize()` call, a new constraint is added in the inline constraint (that is `randomize()` with). The following are the four constraints for the same 5 random variables.

```
x > z;           // from the constraint block m
c -> a == b;    // from the constraint block m
y > 0;          // from the constraint block n
x != y;         // from the inline constraint
                // - randomize() with ..
```

For this second randomize call, two partitions are created.

The first partition has the random variables:  $x$ ,  $y$ , and  $z$  because the following constraints relate all three together: ( $x > z$ ), ( $y > 0$ ), and ( $x != y$ ).

The second partition has the random variables `a` and `b` because of the `(c -> a == b)` constraint.

---

## Randomize Serial Number

Each randomization in a simulation is assigned a serial number starting with 1. For example, if there are ten randomize calls (`std::randomize` or randomization of class objects) in a simulation, they are numbered from 1 to 10.

By default, the randomize serial numbers are not printed at runtime. To display the randomize serial numbers during simulation, you need to run the simulation with the `+ntb_solver_debug=serial` option.

```
% simv +ntb_solver_debug=serial
```

After each randomization completes, VCS prints the randomize serial number along with some runtime and memory data for the `randomize()` call.

Using a randomize serial number provides a mechanism to focus the constraint debug on a specific `randomize()` call. If the randomize serial number is used together with the partition number, it is the specified partition within the specified randomize call that becomes the focus for the constraint debug.

To specify the  $n^{\text{th}}$  partition of the  $m^{\text{th}}$  randomize call, the notation `m.n` is used.

---

## Solver Trace

To get more insight to how VCS solves a randomize call, you can enable solver trace reporting by using the `+ntb_solver_debug=trace` runtime option. The following is an example of the solver trace:

```
// Part.sv
class C;
    rand byte x, y, z, m, n, p, q;

constraint imply {
    x > 3 -> y > p;                                // C1
    z < bigadd ( x, q );                            // C2
    n != 0;   // C3
}

function byte bigadd (byte a, b);
    return (a + b);
endfunction

endclass

program automatic test;
    C obj = new;
    initial begin
        repeat (5) begin
            obj.randomize() with { m == z; };      // C4
        end
    end
endprogram
```

For this example, let us determine the partitions that are created by the solver.

The SystemVerilog LRM mandates that function arguments must be solved first in order to compute the function that is used to constraint other random variables. In other words, separate partitions must be created for  $x$ ,  $q$  and then for  $z$ .

- The constraint expression  $C1$  relates the random variables  $x$ ,  $y$ ,  $p$  together. So they are solved together in one partition.
- The constraint expression  $C2$  using function call in constraint requires that  $z$  is solved in a different partition from  $x$  and  $q$ .
- Since the random variable  $q$  is not related to any other random variables,  $q$  is solved in a partition on its own.
- Similarly, the random variable  $n$  is not related to any other random variables,  $n$  is solved in another partition on its own.
- The constraint expression  $C4$  is an inline constraint relating the two random variables  $m$  and  $z$  together. Therefore,  $m$  and  $z$  be solved together in one partition.
- Given the above descriptions, you can see four partitions are created.
  - Partition 1 to solve  $x$ ,  $y$ ,  $p$  together
  - Partition 2 to solve  $n$  alone
  - Partition 3 to solve  $q$  alone
  - Partition 4 to solve  $z$  and  $m$  together

To compile and run this example and enable solver trace for the third randomize call, use the following command:

```
% vcs -sverilog part.sv  
% simv +ntb_solver_debug=trace +ntb_solver_debug_filter=3
```

Part of the solver trace shows the partition information. The following is a part of the solver trace from the above command.

```
=====  
SOLVING constraints  
At file part.sv, line 20, serial 3  
  
Rng state is:  
01x0z11xzxxxx11zx1x0zx100zxxzzz0zxxxxxxxxxxxxzzzzzzzzzz  
xxzxxx  
Virtual class C, Static class C  
  
...  
Solving Partition 1 (mode = 2)  
  
rand bit signed [7:0] y; // rand_mode = ON  
rand bit signed [7:0] p; // rand_mode = ON  
rand bit signed [7:0] x; // rand_mode = ON  
  
...  
  
Solving Partition 2 (mode = 2)  
  
rand bit signed [7:0] n; // rand_mode = ON  
  
...  
  
Solving Partition 3 (mode = 2)  
  
rand bit signed [7:0] q; // rand_mode = ON  
  
...  
  
Solving Partition 4 (mode = 2)  
  
bit signed [7:0] fv_3 /* this .C::bigadd( x , q ) */ = -127;  
rand bit signed [7:0] z; // rand_mode = ON  
rand bit signed [7:0] m; // rand mode = ON
```

It is required to specify the `randomize()` calls and/or partitions to report the solver trace details. For example, the following command reports the solver trace for the second `randomize()` call and all partitions within this `randomize()` call of the simulation.

```
% simv +ntb_solver_debug=trace +ntb_solver_debug_filter=2
```

The following command reports the solver trace for the third partition of the fifth `randomize()` call of the simulation.:

```
% simv +ntb_solver_debug=trace +ntb_solver_debug_filter=5.3
```

If the solver trace is to be enabled for multiple `randomize` calls, you can specify the list of random serial and partition numbers (optionally) in a comma-separated list for the `+ntb_solver_debug_filter` option. For example, consider the following `randomize()` calls and their partitions:

- Serial number 2, all partitions of this second `randomize()` call
- Serial number 5, just the third partition of this fifth `randomize()` call
- Serial number 10, all partitions of this tenth `randomize()` call
- Serial number 15, just the 30th partition of this 15<sup>th</sup> `randomize()` call.

The following command reports the solver traces for these `randomize()` calls and their partitions:

```
% simv +ntb_solver_debug=trace \
+ntb_solver_debug_filter=2,5.3,10,15.30
```

The following command reports the solver traces for the `randomize()` calls or partitions listed in a text file, such as `serial_trace.txt` is the file name.

```
% simv +ntb_solver_debug=trace \
+ntb_solver_debug_filter=file:serial_trace.txt
```

The following command reports the solver traces for all `randomize()` calls in the simulation. Be aware that this may produce a lot of data if there are many `randomize()` calls in the simulation.

```
% simv +ntb_solver_debug=trace +ntb_solver_debug_filter=all
```

or

```
% simv +ntb_solver_debug=trace_all
```

The `+ntb_solver_debug_filter` is not needed on the second `simv` command line.

Note:

Reporting solver traces for all `randomize()` calls can generate very large data files. Using the `+ntb_solver_debug=trace` and `+ntb_solver_debug_filter=serial_num|file` options and arguments limit the solver trace reports to the ones on which you want to focus the constraint debug.

---

## Constraint Profiler

To debug any performance related issues, profiling is required to identify the top consumers of time/memory. VCS provides a constraint profiler feature that can be enabled by using the `+ntb_solver_debug=profile` runtime option and keyword argument.

```
% simv +ntb_solver_debug=profile
```

This `simv` command line runs the simulation and collects runtime and memory data on each of the `randomize()` calls in the simulation. The randomize calls/partitions that take the most time and memory are listed out in a constraint profile report in the file `simv.cst/html/profile.xml`, where `simv` is the name of the simulation executable.

To view the constraint profile report in `simv.cst/html/profile.xml`, open the file with the Firefox or Chrome Web browser. You cannot view this file in Internet Explorer on Windows.

The random serial numbers for the randomize calls and/or partitions that take the most time are listed in the `simv.cst/serial2trace.txt` file.

### Note:

The unified profiler also does constraint profiling. The Unified profiler is an LCA feature, for more information see the *VCS LCA Features Guide*.

---

## Test Case Extraction

The solver trace shows the list of variables and constraints for each of the partitions. By wrapping this data inside a SystemVerilog class in a `program` block, you can create a standalone test case to compile and simulate to shorten the debug time. If you wish to try different things to better understand the solver behavior and also wish fix the constraint issue, you can do it on this extracted test case instead of the original design to save compile and runtime.

To enable test case extraction, you can enable the solver trace reporting by using the `+ntb_solver_debug=extract` runtime option and keyword argument. You must specify the specific `randomize()` calls to extract the test cases for using the `+ntb_solver_debug_filter` option.

For example, test case extraction is enabled for the second `randomize` call, that is, `randomize serial = 2`:

```
% simv +ntb_solver_debug=extract +ntb_solver_debug_filter=2
```

This extracts a test case for each of the partitions of the `randomize()` call. Extracted test cases are saved in the `simv.cst/testcases` directory, where `simv` is the name of the simulation executable. The extracted test cases follow this naming convention:

```
extracted_r_serial#_p_partition#.sv
```

Once extracted, you can follow the commands to compile and run the standalone test case. For example, to simulate the extracted test case for the third partition of the second `randomize()` call of the original design:

```
cd simv.cst/testcases  
% vcs -sverilog extracted_r_2_p_3.sv -R
```

Similar to reporting solver traces for a single partition or for multiple `randomize()` calls and their partitions, you can enable test case extraction for these too. For example:

```
% simv +ntb_solver_debug=extract \  
+ntb_solver_debug_filter=5.3  
  
% simv +ntb_solver_debug=extract \  
+ntb_solver_debug_filter=2,5.3,10,15.30  
  
% simv +ntb_solver_debug=extract \  
+ntb_solver_debug_filter=file:serial_trace.txt
```

**Note:**

You can only extract test cases from a partition. If VCS fails before any partition is created, test case extraction does not work.

When VCS encounters a `randomize()` call that has no solution or has constraint inconsistencies, VCS automatically extracts a test for it and saves the extracted test case using the following naming convention:

```
% simv.cst/testcases/  
extracted_r_serial#_p_partition#_inconsistency.sv
```

When VCS fails to solve a `randomize()` call due to solver time outs, test case extraction is also automatically enabled for it and VCS saves the extracted test case using the following naming convention:

```
% simv.cst/testcases/  
extracted_r_serial#_p_partition#_timeout.sv
```

---

## Using multiple +ntb\_solver\_debug arguments

To use multiple `+ntb_solver_debug` arguments, such as `serial`, `trace`, `extract`, and `profile`, you can use plus (+) to combine them. For example:

```
% simv +ntb_solver_debug=serial+trace+extract \
+ntb_solver_debug_filter=3,4
```

---

## Summary for the +ntb\_solver\_debug Option

The runtime option `+ntb_solver_debug` provides you with many constraint debug features to debug constraints in batch mode.

### **+ntb\_solver\_debug=serial**

The serial number assignment to the randomizations in a simulation provides a method to identify the `randomize()` calls to be debugged next. Once identified, you can use this runtime option with appropriate arguments to report the trace and extract test cases. The constraint profiler also uses the same identification method to provide feedback, such as which specific `randomize()` calls to optimize for best performance improvements.

### **+ntb\_solver\_debug=trace**

This enables solver trace reporting for the specified `randomize()` calls. This helps you to understand how VCS solves the random variables for given `randomize` calls. The `+ntb_solver_debug_filter` option is required to specify a list of `randomize()` calls for which to enable the solver trace.

### **+ntb\_solver\_debug=profile**

This enables constraint profiling for the simulation at runtime. The profile report provides important information to you, such as which randomize calls should be targeted for improving constraint performance to bring down the total simulation runtime or memory.

### **+ntb\_solver\_debug=extract**

This enables test case extraction for the specified randomize calls. This creates standalone test cases for you to compile and run outside the original design. This should help in quicker turnaround time to experiment possible fixes as it is faster to compile and run a smaller test case. The `+ntb_solver_debug_filter` option is required to specify a list of randomize calls for which to enable test case extraction.

### **+ntb\_solver\_debug=verbose**

The `+ntb_solver_debug=verbose` option provides you more detailed information in the solver output. This information includes the file name and the line number for all the individual constraint conditions inside the constraint block from the solver output.

The sample solver output is as follows:

```
Solver failed when solving following set of constraints
rand bit[7:0] r1; // rand_mode = ON
constraint cons1 // (from this) (constraint_mode = ON)
                  (test.v:4)
{
    (r1 == 8'h1); // (test.v:5)
    (r1 == 8'h2); // (test.v:6)
}
```

---

## Support for Save and Restore Stimulus

While migrating from a previous release to a new release, you may like VCS to generate same stimulus or output in the new release as it is generated for the previous release, especially during the early phases of migration. But, VCS can generate different stimulus and output across releases due to:

- procedural flows controlled by non-deterministic functions, such as date, time, and so on. This completely depends on the design.
- randomize function calls, such as `obj.randomize()` or `std::randomize()`. This is due to the change in the random behavior of constraint solvers from previous release to the new release.

Therefore, VCS supports saving the stimulus of the design in one release and applying or restoring the saved stimulus on the same design in the new release. This helps you to get same stimulus across all releases.

## Use Model

The following is the use model for saving and restoring the same stimulus for the same design across all releases:

```
% simv +ntb_solver_replay=[save|restore]  
[+ntb_solver_replay_path=<path>]
```

Where,

`+ntb_solver_replay=save`

Saves the stimulus of a simulation.

```
+ntb_solver_replay=restore
```

Restores the stimulus of a simulation for the same design.

```
+ntb_solver_replay_path=<path>
```

This is optional. It allows you to specify the file path to save and read the stimulus. Without this option, the

```
+ntb_solver_replay=save
```

and the  

```
+ntb_solver_replay=restore
```

 options save and restore the stimulus to or from `./simv.replay` respectively.

## Limitations

The feature has the following limitations:

- in save and restore phases, you should use the same set of compile-time and runtime options (including random seed).
- You should compile or run the design in the same environment or flow. For example, if you use the partition compile flow while saving the stimulus, you should use the same partition compile flow to restore the saved stimulus.
- For any changes in the setup from save to restore phase, the correct result is not guaranteed.

---

## Constraint Guard Error Suppression

If a guard expression is false, and if there are no other errors during randomization, VCS suppresses errors in the implied expressions of guard constraints. For example, the following is a sample error message that VCS now suppresses:

```
Error-[CNST-NPE] Constraint null pointer error
test_guard.sv, 27
    Accessing null pointer obj.x in constraints.
    Please make sure variable obj.x is allocated.
```

Guarded constraints are defined in the IEEE SystemVerilog LRM Std 1800™-2012 Section 13.4; especially sections 13.4.5, 13.4.6, and 13.4.12.

The VCS constraint solver does not distinguish between implication (see IEEE SystemVerilog LRM Std 1800™-2012 Section 13.4.5) and if-else constraints (IEEE SystemVerilog LRM Std 1800™-2012 Section 13.4.6). They are equivalent representations in the VCS constraint solver. These are called guarded constraints in this document.

Hence, the two formats shown in [Example 16-15](#) are equivalent inside the VCS constraint solver.

### *Example 16-15 Guarded Expressions*

```
if (a | b | c)
{
    obj.x == 10;
```

-or-

```
(a | b | c) -> (obj.x == 10);
```

In [Example 16-15](#), the expression inside the `if` condition (or the left side of the implication operator) is the guard expression. The remaining part of the expression (the right side of the implication operator) is the implied expression.

Note:

If there are other types of errors or conflicts, VCS does not guarantee suppression of those errors in the implied expression of the guard constraint.

The LRM states that the implication operator (or the `if-else` statement) should be at the top level of each constraint. Therefore, a constraint may have at most one guard (or one implication operator).

---

## Error Message Suppression Limitations

The constraint guard error message suppression feature has some limitations, as explained in the following sections:

- “[Flattening Nested Guard Expressions](#)”
- “[Pushing Guard Expressions into Foreach Loops](#)”

### Flattening Nested Guard Expressions

If there are multiple nested guards for a constraint, VCS combines them into one guard. For example, given the following code:

```
if (a)
{
    if (b)
    {
        if (c)
        {
            obj.x == 10;
        }
    }
}
```

VCS flattens the guard expression into the following equivalent code:

```
if (a && b && c)
{
    obj.x == 10;
}
```

In the above example, if `a` is false and `b` has an error (for example, a null address error), VCS still issues the error message.

## Pushing Guard Expressions into Foreach Loops

VCS pushes constraint guards into foreach loops. For example, if you have:

```
if (a | b | c)
{
    foreach (array[i])
    {
        array[i].obj.x == 10;
    }
}
```

VCS transforms it into the following equivalent code:

```
foreach (array[i])
{
    if (a | b | c)
    {
        array[i].obj.x == 10;
    }
}
```

In the above example, if `a | b | c` is false and `array` has an error (for example, a null address error), VCS still issues the error message.

---

## Support for Array and Cross-Module References in `std::randomize()`

VCS allows you to use Cross-Module References (XMRs) in class constraints and inline constraints, in all applicable contexts. Here, XMR means a variable with static storage (anything accessed as a global variable).

VCS `std::randomize()` support allow the use of arrays and XMRs as arguments.

VCS supports all types of arrays:

- fixed-size arrays
- associative arrays
- dynamic arrays
- multidimensional arrays
- smart queues

Note:

VCS does not support multidimensional, variable-sized arrays.

Array elements are also supported as arguments to `std::randomize()`.

VCS supports all types of XMRs:

- class XMRs
- package XMRs
- interface XMRs

- module XMRs
- static variable XMRs
- any combination of the above

You can use arrays, array elements, and XMRs as arguments to `std::randomize()`.

## Syntax

```
integer fa[3];
success= std::randomize(fa);
success= std::randomize(fa[2]);
success= std::randomize(pkg::xmr);
```

## Example

```
module test;
integer i, success;
integer fa[3];
initial
begin
    foreach(fa[i]) $display("%d %d\n", i, fa[i]);
    success = std::randomize(fa);
    foreach(fa[i]) $display("%d %d\n", i, fa[i]);
end
endmodule
```

When `std::randomize()` is called, VCS ignores any `rand` mode specified on class member arrays or array elements that are used as arguments. This is consistent with how `std::randomize()` is specified in the SystemVerilog LRM. This means that for purposes of `std::randomize()` calls, all arguments have `rand` mode ON, and none of them are `randc`.

## Error Conditions

If you specify an argument to a `std::randomize()` array element, which is outside the range of the array, VCS prints the following error message:

```
Error-[CNST-VOAE] Constraint variable outside array error
```

Random variables are not allowed as part of an array index.

If you specify an XMR argument in a `std::randomize()` call, and that XMR cannot be resolved, VCS issues an error message.

---

## Support for Cross-Module References in Constraints

You can use Cross-Module References (XMRs) in class constraints and inlined constraints. You can refer to XMR variables directly or by specifying the full hierarchical name, where appropriate. You can use XMRs for all data types, including scalars, enums, arrays, and class objects.

VCS supports all types of XMRs:

- class XMRs
- package XMRs
- interface XMRs
- module XMRs
- static variable XMRs
- any combination of the above

## Syntax

```
constraint general
{
    varxmr1 == 3;
    pkg::varxmr2 == 4;
}

c.randomize with { a.b == 5; }
```

## Examples

The following is an example of a module XMR:

```
// xmr from module
module mod1;
    int x = 10;
class cls1;
    rand int i1 [3:0];
    rand int i2;
constraint constr
{
    foreach(i1[a]) i1[a] == mod1.x;
}
endclass

cls1 c1 = new();
initial
begin
    c1.randomize() with {i2 == mod1.x + 5;};
end
endmodule
```

The following is an example of a package XMR:

```
package pkg;
    typedef enum {WEAK, STRONG} STRENGTH;
    class C;
        static rand STRENGTH stren;
    endclass
```

```

    pkg::C inst = new;
endpackage

module test;
    import pkg::*;
    initial
    begin
        inst.randomize() with {pkg::C::stren == STRONG;};
        $display("%d", pkg::C::stren);
    end
endmodule

```

## Functional Clarifications

XMR resolution in constraints (that is, choosing to which variable VCS binds an XMR variable) is consistent with XMR resolution in procedural SystemVerilog code. VCS first tries to resolve an XMR reference in the local scope. If the variable is not found in the local scope, VCS searches for it in the immediate upper enclosing scope, and so on, until it finds the variable.

If you specify an XMR variable that cannot be resolved in any parent scopes of the constraint/scope where it is used, VCS errors out and prints an error message.

## XMR Function Calls in Constraints

VCS supports XMR function calls in class constraints, inlined constraints, and `std::randomize`. You can refer to XMR functions with or without specifying the full hierarchical name. XMR functions can return and have as arguments all supported data types, including scalar data types, enums, arrays, and class objects.

---

## State Variable Index in Constraints

VCS supports the use of state variables as array indexes in constraints and inline constraints, in all applicable contexts. These state variables must evaluate to the same type required by the index type of the array to which they are addressed.

Note:

String-type state variables in array indexes are not supported.

VCS supports the set of expressions (operators and constructs) that also work with loop variables as array indices in constraints. The set of supported expressions is restricted in the sense that they must evaluate in the constraint framework.

---

## Runtime Check for State Versus Random Variables

VCS supports state variables for array indexes, but not random variables. Therefore, the tool performs runtime checks for the randomness of the variable. The randomness may be affected if the variable is aliased (due to object hierarchy, module hierarchy, or XMR). When this runtime check finds a random variable being used as an array index, the tool issues an error message.

To differentiate random variables versus state variables, VCS uses the following scheme:

- For `randomize` with a list of arguments (`std::randomize` or `obj.randomize`), variables or objects in the argument list are considered to be random. Variables or objects outside the list (and not aliased by the random objects) are considered to be state variables.

- For randomize without a list of arguments (`obj.randomize`) variables declared as non-random, or declared as random but with `rand mode OFF`, variables are considered to be state variables.

---

## Array Index

The variable (or supported expression) used for an array index must be an integral data type. If the value of the expression or the state variable evaluates out of bounds, comes to a negative index value and references a non-existent array member or contains `X` or `Z`, VCS issues a runtime error message.

---

## Using DPI Function Calls in Constraints

VCS supports calling DPI functions directly from constraints. These DPI function calls must be pure and cannot have any side effects, as per the SystemVerilog LRM. For more information on DPI function call contexts (pure and non-pure), see IEEE SystemVerilog LRM Std 1800™-2012 Section 35.

The following are some examples of valid import DPI function declarations that you can call from constraints:

```
import "DPI-C" pure function int func1();  
import "DPI-C" pure function int func2(int a, int b);
```

[Example 16-16](#) shows a pure DPI function in C.

### *Example 16-16 Pure DPI Function in C*

```
#include <svdpi.h>
```

```

int dpi_func (int a, int b) {
    return (a+b); // Result depends solely on its inputs.
}

```

[Example 16-17](#) shows how to call a pure DPI function from constraints.

### *Example 16-17 Invoking a Pure DPI Function from Constraints*

```

import "DPI-C" pure function int dpi_func(int a, int b);
class C;
    rand int ii;
    constraint cstr {
        ii == dpi_func(10, 20);
    }
endclass

program tb;
initial begin
    C cc;
    cc = new;
    cc.randomize();
end
endprogram

```

## Invoking Non-pure DPI Functions from Constraints

VCS issues an error message when it detects a call to any context DPI function or other import DPI function for which the context is not specified or the import property is not specified as `pure`. VCS issues this error even if the DPI function actually has no side effects. To prevent this kind of error, explicitly mark the DPI function import declaration with the `pure` keyword.

For example, running [Example 16-18](#) with the C code shown in [Example 16-16](#) results in an error because the import DPI function is not explicitly marked as `pure`.

**Example 16-18 Invoking a DPI Function Not Marked *pure* from Constraints.**

```
import "DPI-C" function int dpi_func(int a, int b);
// Error: Only functions explicitly marked as
// pure can be called from constraints

class C;
    rand int ii;
    constraint cstr {
        ii == dpi_func(10, 20);
    }
endclass

program tb;
initial begin
    C cc;
    cc = new;
    cc.randomize();
end
endprogram
```

Similarly, running [Example 16-19](#) with the C code shown in [Example 16-16](#) results in an error because context import DPI functions cannot be called from constraints.

**Example 16-19 Invoking a context DPI Function from Constraints**

```
import "DPI-C" context function int dpi_func(int a, int b);
// Error: Calling 'context' DPI function
// from constraint is illegal.

class C;
    rand int ii;
    constraint cstr {
        ii == dpi_func(10, 20);
    }
endclass

program tb;
initial begin
```

```

C cc;
cc = new;
cc.randomize();
end
endprogram

```

Calling an import DPI function that is explicitly marked `pure` (as shown in [Example 16-16](#)) has undefined behavior if the actual implementation of the function does things that are not pure, such as:

- Calling DPI exported functions/tasks.
- Accessing SystemVerilog data objects other than the function's actual arguments (for example, via VPI calls).

For example, [Example 16-20](#) has undefined behavior (and may even cause a crash).

### *Example 16-20 Non-pure DPI Function in C*

```

#include <stdio.h>
#include <stdlib.h>
#include "svdpi.h"

int readValueOfFile(char * file) {
    int result = 0;
    char * buf = NULL;
    FILE * fp = fopen(file, "r");

    // Read the content of the file in 'buf' here...
    ...

    if (buf) return strlen(buf);
    else return 0;
}

int dpi_func () {
    char * str = getenv("ENV_VAL_OF_A");

```

```

int a = str ? atoi(str) : -1;
int b = readValueOfFile("/some/file");
int c;

svScope scp = svGetScopeFromName("$unit");
if (scp == NULL) {
    fprintf(stderr, "FATAL: Cannot set scope to $unit\n");
    exit(-1);
}
svSetScope(scp);

c = export_dpi_func();
return (a+b+c);
}

```

[Example 16-21](#) shows a DPI function marked `pure` that is actually doing non-pure activities. This results in an error.

### *Example 16-21 DPI Function Marked `pure` but Non-pure Activities*

```

import "DPI-C" pure function int dpi_func();
export "DPI-C" function export_dpi_func;

function int export_dpi_func();
    return 10;
endfunction

class C;
    rand int ii;
    constraint cstr {
        ii == dpi_func();
    }
endclass

program tb;
initial begin
    C cc;
    cc = new;
    cc.randomize();
end
endprogram

```

So, make sure that DPI functions called from the constraints explicitly use the `pure` keyword. Also make sure that the DPI function corresponding foreign language implementation is indeed pure (that is, it has no side effects).

---

## Using Foreach Loops Over Packed Dimensions in Constraints

VCS supports foreach loops over the following kinds of packed dimensions in constraints:

- “[Memories with Packed Dimensions](#)”
- “[MDAs with Packed Dimensions](#)”

You do not need to set any special compilation or runtime switches to make this work. VCS supports foreach loop variables for the entire packed dimensions of an array. For more information, see the section “[The foreach Iterative Constraint for Packed Arrays](#)” .

---

### Memories with Packed Dimensions

You can use foreach loops over memories with single or multiple packed dimensions, as shown in the following examples.

#### Single Packed Dimension

```
class C;
    rand bit [5:2] arr [2];
constraint Cons {
    foreach(arr[i,j])
        arr[i][j] == 1;
}
```

```
}
```

```
endclass
```

## Multiple Packed Dimensions

```
class C;
    rand bit [3:1][5:2] arr [2];
constraint Cons {
    foreach(arr[i,j,k]) {
        arr[i][j][k] == 1;
    }
}
```

```
endclass
```

---

## MDAs with Packed Dimensions

You can use foreach loops over MDAs with single or multiple packed dimensions, as shown in the following examples.

### Single Packed Dimension

```
class C;
    rand bit [5:2] arr [2][3];
constraint Cons {
    foreach(arr[i,j,k]) {
        arr[i][j][k] == 1;
    }
}
```

```
endclass
```

### Multiple Packed Dimensions

```
class C;
    rand bit [-1:1][5:2] arr [2][3];
```

```

constraint Cons {
    foreach(arr[i,j,k,l]) {
        arr[i][j][k][l] == 1;
    }
}
endclass

```

## Just Packed Dimensions

```

class C;
rand integer arr1;
rand bit [-1:0][5:2] arr2;
constraint cons_1 {
    // foreach over the packed dimensions;
    foreach (arr1[i]) {
        if (i > 15) {
            arr1[i] dist {0:= 10, 1:= 90};
        } else {
            arr1[i] dist {0:= 90, 1:= 10};
        }
    }
}
constraint Cons2 {
    foreach(arr2[i,j]) {
        arr2[i][j] == 1;
    }
}
endclass

```

VCS does not create implicit constraints that guarantee the array indexed by the variable (or expression) is valid. You must properly constrain or set the variable value so that the array is correctly addressed.

VCS also supports associative array indices. The indexes of these arrays may be integral data types or strings if the associative array is string-indexed. However, you cannot use expressions for associative arrays.

---

## The foreach Iterative Constraint for Packed Arrays

VCS has implemented `foreach` loop variables for the entire packed dimensions of an array in the constraint context.

In previous releases, a `foreach` loop for the dimensions of a multidimensional array in the constraint context required that at least one of the dimensions be unpacked. That restriction is removed, a multi-dimensional packed array in the constraint context is now fully supported.

The following code example illustrates this implementation.

### *Example 16-22 The foreach Iterative Constraint for Packed Arrays*

```
program prog;
  class my_class;
    rand reg [2][2][2][2] arr;
    constraint constr {
      foreach (arr[i,j,k,l]) {
        (i==0) -> arr [i][j][k][l] == 1;
        (i==1) -> arr [i][j][k][l] == 0;
      }
    }
  endclass
endprogram
```



all dimensions packed

In previous releases at least one of the dimensions of MDA array needed to be unpacked.

This code example results in the following error message in previous releases:

```
Error-[NYI-UFAIFE] NYI constraint: packed dimensions
doc_ex.sv,9
prog, "this.arr"
    arr has only packed dimensions and no unpacked dimensions.
    Foreach over packed dimensions is supported if the object
has at least one
    unpacked dimension.

1 error
```

Now, entirely packed arrays in the constraint context are not an error condition and do not result in this error message.

---

## Randomized Objects in a Structure

VCS has implemented randomized objects in a structure. The following code example illustrates this implementation.

### *Example 16-23 Randomized Object in a Structure*

```
program test;

class packet;
    randc int addr = 1;
    int crc;
    rand byte data [] = {1,2,3,4};
endclass
class packet_test;
    typedef struct {
        rand packet p1;           randomized object
        } header;                  in a structure
header hd;
```

```

function new();
    this.hd.p1 = new;
endfunction

endclass

packet_test pt = new;

initial begin
    pt.randomize(hd);
    end
endprogram

```

In previous releases declaring this class in a structure with the `rand` type-modifier keyword resulted in the following error message:

```

Error-[SV-NYI-CRUDST] Rand class object under structure
code_ex_rand_struct.sv, 10
"p1"
    Rand class objects which defined under structure is not
yet supported.

1 error

```

This code example compiles and runs without any errors since `rand` class objects inside a structure are implemented.

---

## Support for Typecast in Constraints

You can use a cast ( ' ) operator on constraints as defined in the IEEE SystemVerilog LRM Std 1800™-2012 Section 6.24, “Casting”.

### Syntax

The following is the syntax for casting constraints.

```
constant_cast ::= casting_type ' ( constant_expression )

cast ::= casting_type ' ( expression )

casting_type ::= simple_type | constant_primary | signing

simple_type ::= integer_type | ps_type_identifier |
               ps_parameter_identifier

integer_type ::= bit | logic | reg | byte | shortint | int | longint |
               integer | time
```

### Description

The following are the details about typecasting constraints.

- VCS only supports variables and constants of integral types. As a result, typecasting constraints are applied only to integral or equivalent types (see IEEE SystemVerilog LRM Std 1800™-2012 Section 6.22.2).
- Built-in integer types can be cast to each other.
- Packed arrays, structs, unions and enum types can be cast to built-in integral types if their base types are integral types.
- For casting an enumerated type, if an enumerated value is outside the defined range of the enum type, then VCS issues a constraint inconsistency message (through the Solver).

- Casting from a non-integral types (real, string, etc.) to integral types is not supported. VCS issues a compile-time error message.
- Casting from user-defined types (class, unpacked struct, unpacked unions, etc.) to integral types is not supported. VCS issues a compile-time error message.

## Examples

```
// First example.
typedef struct packed signed {
    bit [7:0] a;
    byte b;
    shortint c;
} p1;
rand p1 p;
rand int q;
constraint c1 { int' (p) == q; };
```

For example, p is a 32-bit packed signed struct and is randomized to the value of 'h815F\_8D74. The members of this packed struct are p.a = 8'h81, p.b = 8'h5F, p.c = 8'h8D74 (or -29324 as it is signed). Because of the equality constraint, q takes on the value of 8'h815F\_8474 or -2124444300 as int is a signed data type.

```
// Second example.
class C;
    typedef enum {
        red=0,
        yellow=5,
        green
    } light_t;

    rand light_t x;
    rand int y;

    constraint c0 { x == light_t' (y); }
endclass
```

```
C obj = new;
obj.randomize();
```

Note, in the previous example (“Second example”), `y` is always 0, 5, or 6. The following would produce a constraint solver failure message:

```
obj.randomize() with { y == 10; };
```

Following is a third example.

```
// Third example.
class cfg;
    rand bit en[16];
    constraint only_enable_4_GOOD {
        // Intent: only 4 of the 16 of the 1-bit elements are 1'b1
        en.sum() with (int'(item)) == 4;
    }
endclass
program automatic test;
    cfg obj = new;
    initial obj.randomize();
endprogram
```

According to the LRM, `sum()` returns a single value of the same type as the array element type; or, if specified, the expression in the 'with' clause. In the previous case, `sum()` will return the type 'int' because of the `with` clause, and not a single 'bit' as the type of the individual array elements. Without the use of the `with` clause and casting the item from a `bit` to an `int` type, there would have been a constraint failure as `sum()` would have returned a value of type 'bit', and no 1-bit value would be equal to 4.

```

// Fourth example.
class A;
    rand int x;
    constraint c {
        x >= 0;
        x < ( (41'(2 ** 40)) - 1024 );
    }
endclass

program automatic test;
    A obj = new;
    initial obj.randomize();
endprogram

```

The cast to increase the size of the expression ( $2^{**}40$ ) from 30 to 41 is needed so that  $2^{**}40$  is treated as the value 41'h100000000000. Otherwise, the expressions in the second constraint will be evaluated as 32-bit values, and ( $2^{**}40$ ) would have been evaluated to 32'h0.

## Strings in Constraints

You can compare string type state variables and string literals with the logical equality == and inequality != operators.

All other uses of strings in constraints, such as string concatenation, string methods (for example str.substr()), string replication and casting to or from strings are unsupported.

The following is an example of the supported use of strings in constraints:

```

bit [31:0] p = "string_lit";
string p2 = "string_var";
rand int flg;

```

```
constraint constr0 {
    if (p == "string_lit" && p2 != "string_var") flg == 1;
    else flg == 10;
}
```

---

## SystemVerilog LRM 1800™-2012 Update

The SystemVerilog constructs in this update are as follows:

- “Using Soft Constraints in SystemVerilog”
  - “Unique Constraints”
- 

### Using Soft Constraints in SystemVerilog

Input stimulus randomization in SystemVerilog is controlled by user-specified constraints. If there is a conflict between two or more constraints, the randomization fails.

To solve this problem, you can use soft constraints. Soft constraints are constraints that VCS disables if they conflict with other constraints.

VCS use a deterministic, priority-based mechanism to disable soft constraints. When there is a constraint conflict, VCS disables any soft constraints in reverse order of priority (that is, the lowest priority soft constraint is disabled first) until the conflict is resolved. The following sections explain how to use soft constraints with VCS:

- “Using Soft Constraints”
- “Soft Constraint Prioritization”

- “Soft Constraints Defined in Classes Instantiated as rand Members in Another Class”
- “Soft Constraints Inheritance Between Classes”
- “Soft Constraints in AOP Extensions to a Class”
- “Soft Constraints in View Constraints Blocks”
- “Discarding Lower-Priority Soft Constraints”

## Using Soft Constraints

Use the `soft` keyword to identify the soft constraints. Constraints that are not defined as soft constraints are hard constraints. The following [Example 16-24](#) shows a soft constraint:

### *Example 16-24 Soft Constraint*

```
class A;
    rand int x;
constraint c1 {
    soft x > 2; // soft constraint
}
endclass
```

[Example 16-25](#) shows a hard constraint.

### *Example 16-25 Hard Constraint*

```
class A;
    rand int x;
constraint c1 {
    x > 2; // hard constraint
}
endclass
```

## Soft Constraint Prioritization

VCS determines the priorities of soft constraints according to the set of rules described in this section. In general, VCS assigns increasing priorities to soft constraints as they climb the following list:

- Class parents in the inheritance graph
- Class members
- Soft constraints in the class itself
- Soft constraints in any `extends` blocks applied to a class

In this schema, soft constraints in any `extends` blocks applied to a class are assigned the highest priority.

The following notation is used to describe the priority of a given soft constraint (SC):

`priority(SCx)`: If the following is true:

$\text{priority}(\text{SC2}) > \text{priority}(\text{SC1})$

Then VCS disables constraint SC1 before constraint SC2 when there is a conflict.

### Within a Single Class

VCS assigns soft constraints declared within a class, thus increasing the priority by order of declaration. The soft constraints that appear later in the class body have higher priority than the soft constraints that appear earlier in the class body.

For example, in [Example 16-26](#),  $\text{priority}(\text{SC2}) > \text{priority}(\text{SC1})$ .

*Example 16-26 SC2 Higher Priority than SC1*

```
class A;
    rand int x;
constraint c1 {
    soft x > 10; // SC1
    soft x > 5; // SC2
}
endclass
```

In [Example 16-27](#), priority(SC2) > priority(SC1).

*Example 16-27 SC2 Higher Priority than SC1*

```
class A;
    rand int x;
constraint c1 {
    soft x > 10; // SC1
}
constraint c2 {
    soft x > 5; // SC2
}
endclass
```

## Soft Constraints Defined in Classes Instantiated as rand Members in Another Class

VCS assigns soft constraints declared within rand members of classes increasing the priority by order of member declaration. In [Example 16-28](#), the soft constraints contributed by `C.objB` are of higher priority than the soft constraints contributed by `C.objA`, because `C.objB` is declared after `C.objA` within class `C`.

[Example 16-28](#) also shows why some soft constraints are dropped, instead of honored, because of the relative priorities assigned to soft constraints are as follows:

- // `objC.x = 4` because SC6 is honored.

- // objC.objA.x = 4 because priority(SC4) > priority(SC1).

Here, SC4 is honored and SC1 is dropped. If SC1 were not dropped, it might have caused a conflict because objA.x cannot be 4 (objC.x in SC4) and 2 (SC1) at the same time.

- // objC.objB.x = 5 because priority(SC5) > priority(SC3) > priority(SC2).

Here, SC5 is honored and SC3 is dropped (otherwise, SC3 might conflict with SC5). SC2 is honored because it does not conflict with SC5 by honoring SC2, objC.objB.x = 5.

### *Example 16-28 SC3 Higher Priority than SC2 and SC1*

```

class A;
    rand int x;
    constraint c1 { soft x == 2; } // SC1
endclass

class B;
    rand int x;
    constraint c2 { soft x == 5; } // SC2
    constraint c3 { soft x == 3; } // SC3
endclass

class C;
    rand int x;
    rand A objA;
    rand B objB;
    constraint c4 { soft x == objA.x; } // SC4
    constraint c5 { soft objA.x < objB.x; } // SC5
    constraint c6 { soft x == 4; } // SC6
function
    new(); objA = new; objB = new;
endfunction
endclass

program test;
    C objC;

```

```

initial begin
    objC = new;
    objC.randomize();
    $display(objC.x); // should print "4"
    $display(objC.objA.x); // should print "4"
    $display(objC.objB.x); // should print "5"
end
endprogram

```

For array members where objects are allocated prior to randomization, priorities are assigned in increasing order by position in the array, where the soft constraints in element N have lower priority than the soft constraints in element N+1.

For array members where the objects are allocated during randomization, all soft constraints in allocated objects and their base classes and member classes have the same priority.

## Soft Constraints Inheritance Between Classes

Soft constraints in an inherited class have a higher priority than soft constraints in its base class. For example, in [Example 16-29](#), priority(SC2) > priority(SC1).

### *Example 16-29 SC2 Higher Priority than SC1*

```

class A;
    rand int x;
    constraint c1 {
        soft x > 2; // SC1
    }
endclass

class B extends A;
    constraint c1 {
        soft x > 3; // SC2
    }
endclass

```

## Soft Constraints in AOP Extensions to a Class

As defined in "[Aspect Oriented Extensions](#)", constraint blocks in AOP extensions modify the class definition as an AOP Introduction directive. When there are multiple AOP extensions to the same class, the AOP precedence must be considered. The AOP precedence defines the order in which introductions to a class are added to the class. First, symbols introduced by an AOP extension with a higher precedence are appended to the class. Subsequently, same AOP extensions are appended to the class in the order they appear in the extension.

For example,

```
class A;
    rand int x;
    constraint c1 {
        soft x == 1; // SC1
    }
endclass

extends A_aop1(A);
    constraint c2 {
        soft x == 2; // SC2
    }
endextends
```

This means that the new constraint block `c2` is added as a new symbol in the original class definition as a new member. As a result, class `A` gets modified as follows:

```
class A;
    rand int x;
    constraint c1 {
        soft x == 1; // SC1
    }
    constraint c2 { // from A_aop1(A)
```

```

        soft x == 2; // SC2
    }
endclass

```

After the AOP introduction to the class definition, VCS assigns priorities to multiple soft constraints in the modified class. In this case, as the constraint block c2 is declared later in the modified class, the priority of the soft constraint SC2 is higher than that of the soft constraint SC1, or  $\text{priority}(\text{SC2}) > \text{priority}(\text{SC1})$ . As a result, x is randomized to 2 in the example.

Consider another example, where soft constraints are added on the same random variable in multiple AOP extensions to the same class.

For example:

```

class A;
    rand int x;
    constraint c1 {
        soft x == 1; // SC1
    }
endclass

extends A_aop2 (A);
    constraint c2 {
        soft x == 2; // SC2
    }
    constraint c3 {
        soft x == 3; // SC3
    }
endextends

extends A_aop4 (A);
    constraint c4 {
        soft x == 4; // SC4
    }
endextends

extends A_aop5 (A);

```

```

constraint c5 {
    soft x == 5; // SC5
}
endextends

```

The last AOP extension `A_aop5` of the class `A` has the highest precedence as it is declared last. As the introduction is added to the class in the order of the precedence, this becomes the content of the modified class as follows:

```

class A;
    rand int x;
    constraint c1 {
        soft x == 1; // SC1
    }

    // A_aop5(A) - highest AOP precedence gets added to the
    class first
    constraint c5 {
        soft x == 5; // SC5
    }

    // A_aop4(A) - next highest AOP precedence gets added
    to the class next
    constraint c4 {
        soft x == 4; // SC4
    }

    // A_aop2(A) - lowest AOP precedence gets added to the
    class last
    // but within the same AOP extension, the constraint
    blocks c2 and c3
    // are added in the order of declaration.
    constraint c2 {
        soft x == 2; // SC2
    }
    constraint c3 {
        soft x == 3; // SC3
    }
endclass

```

VCS then computes the priorities for the soft constraints based on this modified class: as follows:

```
priority(SC3) > priority(SC2) > priority(SC4) > priority(SC5) >  
priority(SC1)
```

In this example, `x` is randomized to 3. It is noted that constraint blocks from the highest AOP precedence are added to the class first, Soft constraints from the highest AOP precedence may have an overall lower soft constraint priority.

The following is the original class definition for the class `A`:

```
class A;  
    rand int x;  
    constraint c1 {  
        soft x == 1; // SC1  
    }  
endclass
```

Few modified examples with the `dominate_list` along with the resulting soft priority assignments and solver results for the random variable `x` are listed in the [Table 16-1](#).

## Soft Constraints in View Constraints Blocks

VCS assigns soft constraints within a view constraint block increasing priority by order of declaration. Soft constraints that appear later have higher priority than those that appear earlier. For example, in [Example 16-30](#),  $\text{priority}(\text{SC3}) > \text{priority}(\text{SC2}) > \text{priority}(\text{SC1})$ .

### *Example 16-30 SC3 Higher Priority than SC1*

```
class A;  
    rand int a;
```

Table 16-1 Examples with the Dominates\_list

| Dominates_list variations                                                                                                                                                                                                                                                                           | Resulting AOP precedence and soft constraint priority                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | Solver result for x |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| <pre> extends A_aop2(A) dominates (A_aop5); constraint c2 {     soft x == 2; // SC2 } constraint c3 {     soft x == 3; // SC3 } endextends  extends A_aop4(A); constraint c4 {     soft x == 4; // SC4 } endextends  extends A_aop5(A); constraint c5 {     soft x == 5; // SC5 } endextends </pre> | <ul style="list-style-type: none"> <li>Precedence set by dominates: <math>A\_aop2 &gt; A\_aop5</math></li> <li>No specific precedence set between <math>A\_aop2</math> and <math>A\_aop4</math>. Therefore, the order of declaration depends on the precedence order between the two as follows: <math>A\_aop4 &gt; A\_aop2</math></li> <li>Overall precedence order is set as: <math>A\_aop4 &gt; A\_aop2 &gt; A\_aop5</math></li> </ul> <p>This means the soft constraint from <math>A\_aop5</math> is appended last to the class A, making this soft constraint (SC5) the highest soft priority.</p> | 5                   |

Table 16-1 Examples with the Dominates\_list

| Dominates_list variations                                                                                                                                                                                                                                                                                                                           | Resulting AOP precedence and soft constraint priority                                                                                                                                                                                                                                                                                                                                                                                                                                                          | Solver result for x |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| <pre> extends A_aop2(A) dominates (A_aop4);     constraint c2 {         soft x == 2; // SC2     }     constraint c3 {         soft x == 3; // SC3     } endextends  extends A_aop4(A);     constraint c4 {         soft x == 4; // SC4     } endextends  extends A_aop5(A);     constraint c5 {         soft x == 5; // SC5     } endextends </pre> | <ul style="list-style-type: none"> <li>Precedence set by dominates: A_aop2 &gt; A_aop4</li> <li>No specific precedence set between A_aop2 and A_aop5. Therefore, the order of declaration depends on the precedence order between the two as follows: A_aop5 &gt; A_aop2</li> <li>Overall precedence order is set as A_aop5 &gt; A_aop2 &gt; A_aop4</li> </ul> <p>This means the soft constraint from A_aop4 is appended last to the class A, making this soft constraint (SC4) the highest soft priority.</p> | 4                   |

Table 16-1 Examples with the Dominates\_list

| Dominates_list variations                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | Resulting AOP precedence and soft constraint priority                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | Solver result for x |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
| <pre> extends A_aop2(A);     constraint c2 {         soft x == 2; // SC2     }     constraint c3 {         soft x == 3; // SC3     } endextends  extends A_aop4(A) dominates (A_aop5;     constraint c4 {         soft x == 4; // SC4     } endextends  extends A_aop5(A);     constraint c5 {         soft x == 5; // SC5     } endextends </pre><br><pre> rand int b; constraint c1 {     soft a == 2; // SC1 } endclass </pre><br><pre> A objA; objA.randomize () with { soft a &gt; 2; // SC2 soft b == 1; // SC3 } </pre> | <ul style="list-style-type: none"> <li>Precedence set by dominates: A_aop4 &gt; A_aop5.</li> <li>No specific precedence set between A_aop2 and A_aop5. Therefore, the order of declaration depends on the precedence order between the two as follows: A_aop5 &gt; A_aop2.</li> <li>Overall precedence order is set as: A_aop4 &gt; A_aop5 &gt; A_aop2.</li> </ul> <p>This means the soft constraints from A_aop2 is appended last to the class A. Within the same A_aop2 extension, the order of introduction of new constraint blocks (c2 and c3) is the same as the order of declaration. Thus, it makes the soft constraint from the constraint block c3(SC3) the highest soft priority.</p> | 3                   |

## Discarding Lower-Priority Soft Constraints

You can use a `disable soft` constraint to discard lower-priority soft constraints, even when they are not in conflict with other constraints (see [Example 16-31](#)).

### *Example 16-31 Discarding Lower-Priority Soft Constraints*

```
class A;
rand int x;
constraint A1 {soft x == 3;}
constraint A2 {disable soft x;} // discard soft constraints
constraint A3 {soft x inside {1, 2);}
endclass
initial begin
A a= new();
a.randomize();
end
```

In [Example 16-31](#), constraint `A2` tells the solver to discard all soft constraints of lower priority on random variable `x`. This results in constraint `A1` being discarded. Now, only the last constraint (`A3`) needs to be honored. This example results in random variable `x` taking the values 1 and 2.

A `disable soft` constraint causes lower-priority soft constraints to be discarded even when they are not in conflict with other constraints. This feature allows you to introduce fresh soft constraints that replace default values specified in preceding soft constraints (see [Example 16-32](#)).

### *Example 16-32 Specifying Fresh Soft Constraints*

```
class B;
rand int x;
constraint B1 {soft x == 5;}
constraint B2 {disable soft x; soft x dist {5, 8);}
endclass
```

```

initial begin
B b = new();
b.randomize();
end

```

In Example 16-32, the disable soft constraint preceding the soft dist in block B2 causes the lower-priority constraint on variable `x` in block B1 to be discarded. Now, the solver assigns the values 5 and 8 to `x` with equal distribution (the result from the fresh constraint: `soft x dist {5, 8}`).

Compare the behavior of Example 16-32 with Example 16-33, where the disable soft constraint is omitted.

### *Example 16-33 Specifying Additional Soft Constraints*

```

class B;
rand int x;
constraint B1 {soft x == 5;}
constraint B3 {soft x dist {5, 8};}
endclass
initial begin
B b = new();
b.randomize();
end

```

In Example 16-33, the soft dist constraint in block B3 can be satisfied with a value of 5, so the solver assigns `x` the value 5. If you want the distribution weights of a soft dist constraint to be satisfied regardless of the presence of lower-priority soft constraints, you should first use a disable soft constraint to discard those lower-priority soft constraints.

### *Example 16-34 Usage of Random Variable in Guard Expression*

```
program tb;
    class cls;
        rand int x;
        rand bit cond;

        constraint C1 {soft x == 3;}
        constraint C2 {((cond) -> disable soft x; }
        constraint C3 {x inside {1, 2};}
    endclass

    initial begin
        cls obj = new;
        obj.randomize;
    end
endprogram
```

In Example 16-34, `cond` is a random variable that is used in the guard expression of `disable soft` constraint. VCS displays the following error message:

```
Error-[CNST-NYI-DSCNS] Disable soft constraint not supported
test.v, 12
The variable 'cond' is not a state variable.
Only state variables can be used in guard expressions of
disable soft constraints.
```

---

## Unique Constraints

VCS has implemented unique constraints as specified in IEEE SystemVerilog LRM Std 1800™-2012 Section 18.5.5 “Uniqueness constraints”.

A unique constraint, specified with the `unique` keyword, specifies that a group of random variables after randomization have different (or unique) values.

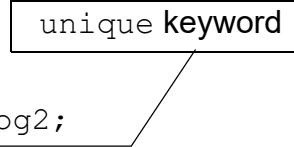
### *Example 16-35 Unique Constraint*

```
module mod;
    class myclass;
        rand logic [1:0] log0, log1, log2;
        constraint constr1 { unique {log0,log1,log2}; }
    endclass

    myclass mc1 = new;

    initial
    begin
        #25 if (mc1.randomize()==1)
            $display("\n\n at %0t log0=%0b log1=%0b log2=%0b \n",
                     $time,mc1.log0,mc1.log1,mc1.log2);
    end

endmodule
```



A callout box with a black border and white background contains the text "unique keyword". An arrow points from the bottom right corner of this box to the word "unique" in the SystemVerilog code, specifically within the constraint declaration.

The \$display system task displays the following:

```
at 25 log0=10 log1=11 log2=1
```

None of the random variables has the same value after randomization.

The SystemVerilog LRM has the following limitations on the random variables in the unique constraint:

- The random variables must be either:
  - a scalar equivalent types or
  - arrays whose leaf elements are equivalent types
- The random variables cannot be randc variables.

If the constraint solver cannot find unique values for the variables in the group, such as if the three variables in [Example 16-35](#) were scalar variables, so they could not have unique values, VCS displays the following error message:

```
Error-[CNST-CIF] Constraints inconsistency failure  
exp3.sv, 13  
    Constraints are inconsistent and cannot be solved.  
    Please check the inconsistent constraints being printed
```

---

## Enhancement to the Randomization of Multidimensional Array Functionality

VCS supports MDA with variable size dimensions. As defined in the IEEE SystemVerilog LRM Std 1800™-2012 Section 7.4.2 “Unpacked arrays”, unpacked or dynamic arrays can be made up of any data type. Arrays can have more than one dimension. These array of arrays are called multidimensional arrays. Therefore, you can extend the semantics of a single-dimensional array to MDA.

### Syntax

```
rand int m [][];  
m.size() constrains the first dimension (say, number of rows)  
m[0].size() constrains the second dimension for the first row  
m[1].size() constrains the second dimension for the second row
```

#### *Example 16-36 Example of MDA for Variable Size Dimensions*

```
rand int m [][];  
rand int m_length;  
constraint cst { m.size() == m_length; m_length inside  
{[1:3]}; }  
constraint cst2 {  
(m_length == 1) -> m[0].size == 5;  
(m_length == 2) -> { m[0].size == 3; m[1].size == 4; }
```

```

(m_length == 3) -> { m[0].size == 2; m[1].size == 5; m[2].size
== 1; }
}

```

- If `m_length` is set to 1, `m_data` structure may look like:

```
[0] * * * *
```

- If `m_length` is set to 2, `m_data` structure may look like

```
[0] * * *
```

```
[1] * * * *
```

- If `m_length` is set to 3, `m_data` structure may look like

```
[0] * *
```

```
[1] * * * *
```

```
[2] *
```

You can combine the dynamic array dimension with the fixed or associative array and queue dimensions. Any unpacked dimension in an array declaration can be a dynamic array dimension. For example:

```

rand int data1[] [4] [];
rand int data2[$] [2];
rand int data3[string] [];

```

### *Example 16-37 Example of MDA for Array Method Calls and Set Membership*

```

program tb;
    class cls;
        rand bit [7:0] mda1[][$];
        rand bit [7:0] a, b, c;

        constraint cons1 {
            // size constraints
            mda1.size == 2;
            foreach (mda1[i,])
                mda1[i].size inside {3,4,5,6};
        }
    endclass
endprogram

```

```

// array methods on rand MDA members
foreach (mda1[i,])
    mda1[i].sum with (int'(item)) <= 100;

// rand MDA members with the 'setmembership' operator
a inside {mda1[0], 1, 2, 3};
b inside {mda1[0][0], 4, 5, 6};

}
endclass

initial begin
    cls obj = new;
    obj.randomize;
end
endprogram

```

---

## Limitation

The feature has the following limitation:

- Randomization of MDA is not supported with `std::randomize` calls.

---

## Supporting Random Array Index

You can use random variable in array index expressions. You can use the random array index in fixed-size array, dynamic array, smart queue arrays.

The following examples shows the usage of random array index:

### Example 16-38 Usage of random array index

```

program test;
    class cls;
        rand bit [7:0] indx1, indx2;

```

```

rand bit [7:0] arr1[7:0], arr2[];

constraint cons1 {
    indx1 inside {0,2,4,6};

    // rand variable 'indx1' used as the array index
    arr1[indx1] == 11;
}

constraint cons2 {
    indx2 inside {0, 1};
    arr2.size inside {[2:10]};

    // rand variable 'indx2' used as the array index
    arr2[indx2] == 22;
}
endclass

initial begin
    cls obj = new;
    obj.randomize;
end
endprogram

```

---

## Limitation

The feature has the following limitation:

- Nested indices are not supported.

---

## Supporting System Function Calls

VCS supports usage of the following system function calls:

- [“\\$size\(\) System Function Call”](#)

- “\$clog2() System Function Call”

## \$size() System Function Call

VCS supports usage of \$size() system function call in the constraint code.

*Example 16-39 Example for the usage of \$size()*

```

program test;
    class cls;
        rand bit [7:0] arr1[1:0];
        rand bit [7:0] arr2[][][2:0][3:0];
        rand bit [7:0] size1, size2, size3;

        constraint cons_size {
            arr2.size == 2;

            // returns 'arr1' dimension size as '2'
            size1 == $size(arr1);

            // returns 'arr2' second dimension size as '3'
            size2 == $size(arr2[0]);

            // returns 'arr2' third dimension size as '4'
            size3 == $size(arr2[0], 2);
        }
    endclass

    initial begin
        cls obj = new;
        obj.randomize;
        $display("size1 = %0d, size2 = %0d, size3 = %0d",
                 obj.size1, obj.size2, obj.size3);
    end
endprogram

```

---

## \$clog2() System Function Call

VCS supports usage of `$clog2()` system function call or integer math function in the constraint scope as per IEEE SystemVerilog LRM Std 1800™-2012 Section 20.8.1 “Integer math functions”.

This significantly improves the verification productivity. You can use the `$clog2()` system function call directly in the constraint code to get the minimal number of bits required to store the value and to generate random stimulus based on the number of bits accordingly.

### Usage Example

Consider the following example for the usage of `$clog2()`:

```
program tb;
    class cls ;
        rand bit [15:0] a, b, c;

        constraint cons {
            c == a >> $clog2(b);
            b inside {3, 5, 14, 28, 46};
            a == 16'habcd;
        }
    endclass
    initial begin
        cls obj = new;
        repeat(3) begin
            obj.randomize;
            $display("a = 'h%4h, b = 'd%-4d, $clog2(b) = "
'd%0d, c = 'h%4h",
                obj.a, obj.b, $clog2(obj.b), obj.c);
        end
    end
endprogram
```

To run the example, use the following command:

```
% vcs -sverilog clog2.v
% ./simv
```

It generates the following output:

```
a = 'habcd, b = 'd14 , $clog2(b) = 'd4, c = 'h0abc
a = 'habcd, b = 'd3 , $clog2(b) = 'd2, c = 'h2af3
a = 'habcd, b = 'd28 , $clog2(b) = 'd5, c = 'h055e
```

---

## Supporting Foreach Loop Iteration over Array Select

VCS supports foreach loop iteration over array select, such as  
foreach (arr [0] [i] ).

### *Example 16-40 Example for foreach loop iteration over array select*

```
program tb;
    class cls;
        rand bit [7:0] arr[1:0][7:0];

        constraint cons_array {
            // foreach iteration over the array select 'arr[0]'
            foreach (arr [0] [i] ) {
                if (i <= 3) {
                    arr[0][i] inside {10, 20, 30, 40};
                } else {
                    arr[0][i] inside {50, 06, 70, 80};
                }
            }
        }
    endclass

    initial begin
        cls obj = new;
        obj.randomize;
    end
endprogram
```

---

## Support for Enumerated Type Methods in a Constraint Expression

VCS supports enumerated type methods in a constraint expression.  
The following enumerated type methods are supported:

- `first()`
- `last()`
- `next()`
- `prev()`
- `num()`

---

### Usage Example

Consider the following test case (`test.v`):

*Example 16-41 Usage of Enumerated Type Methods in a Constraint Expression*

```
program test;
    typedef enum {V, I, B, G, Y, O, R} colors_e;

    class C1;
        colors_e c0, c1 = I;
        rand colors_e rc;
        constraint c_in_range { rc == c1.next(); }
    endclass

    initial begin
        C1 obj1 = new();
        obj1.randomize();
    end
endprogram
```

Compile the test case as follows:

```
% vcs -sverilog test.v
```

---

## Limitation

The feature has the following limitation:

- Enumerated type methods can be used only on non-random `enum` variable inside constraints.

# 17

## Extensions for SystemVerilog Coverage

---

The extensions for SystemVerilog coverage include the following:

- “Support for Reference Arguments in `get_coverage()` and `get_inst_coverage()`”
- “Functional Coverage Methodology Using the SystemVerilog C/C++ Interface”

---

### Support for Reference Arguments in `get_coverage()` and `get_inst_coverage()`

The SystemVerilog LRM provides several predefined methods for every covergroup, coverpoint, or cross. For details, see Section 19.8, “Predefined coverage methods” in the *SystemVerilog LRM*

*IEEE Std. 1800-2012.* Two of these predefined methods, `get_coverage()` and `get_inst_coverage()`, support optional arguments.

You can use the `get_coverage()` and `get_inst_coverage()` predefined methods to query on coverage during the simulation run, so that you can respond to the coverage statistics dynamically.

The `get_coverage()` and `get_inst_coverage()` methods both accept, as optional arguments, a pair of integer values passed by reference.

---

## **get\_coverage() method**

The numerator and denominator assigned by the `get_coverage()` method depend on the scope.

In the covergroup scope, `get_coverage()` assigns the weighted sum of the coverage of merged coverpoints and crosses to its first argument.

In the coverpoint or cross scope, the first argument to `get_coverage()` is the number of covered bins in the merged coverpoint or cross, and the second argument is the total number of bins.

In all cases, weighted sums are rounded to the nearest integer and the second argument is set to the sum of weights.

---

## **get\_inst\_coverage() method**

When the optional arguments are entered with the method in the coverpoint scope or cross scope, the `get_inst_coverage()` method assigns the value of the covered bins to the first argument, and assigns the number of bins for the given coverage item to the second argument. These two values correspond to the numerator and the denominator used for calculating the coverage score (before scaling by 100).

In covergroup scope, the `get_inst_coverage()` method assigns the weighted sum, rounded to the nearest integer, of coverpoint and cross coverage to the first argument and assigns the sum of the weights of the coverpoint or cross items to the second argument.

---

## **Functional Coverage Methodology Using the SystemVerilog C/C++ Interface**

This section describes a SystemVerilog-based functional coverage flow. The flow supports functional coverage features—data collection, reporting, merging, grading, analysis, GUI, and so on.

The SystemVerilog functional coverage flow contains the following features:

- Performs RTL coverage using covergroups and cover properties.
- Performs C coverage using covergroups.
- Integrates easily with the existing testbench environment.
- Provides coverage analysis capabilities — reporting, grading merging, and GUI.

- Has no negative impact on RTL simulation performance.

Functional coverage is very important in verifying correct functionality of a design. SystemVerilog natively supports functional coverage in RTL code.

However, because C/C++ code is now commonly used in a design (with PLI, DPI, DirectC, and so on), there is no systematic approach to verify the functionality of C/C++.

The SystemVerilog C/C++ interface feature provides an application programming interface (API) so that C/C++ code can use the SystemVerilog functional coverage infrastructure to verify its coverage.

**Note:**

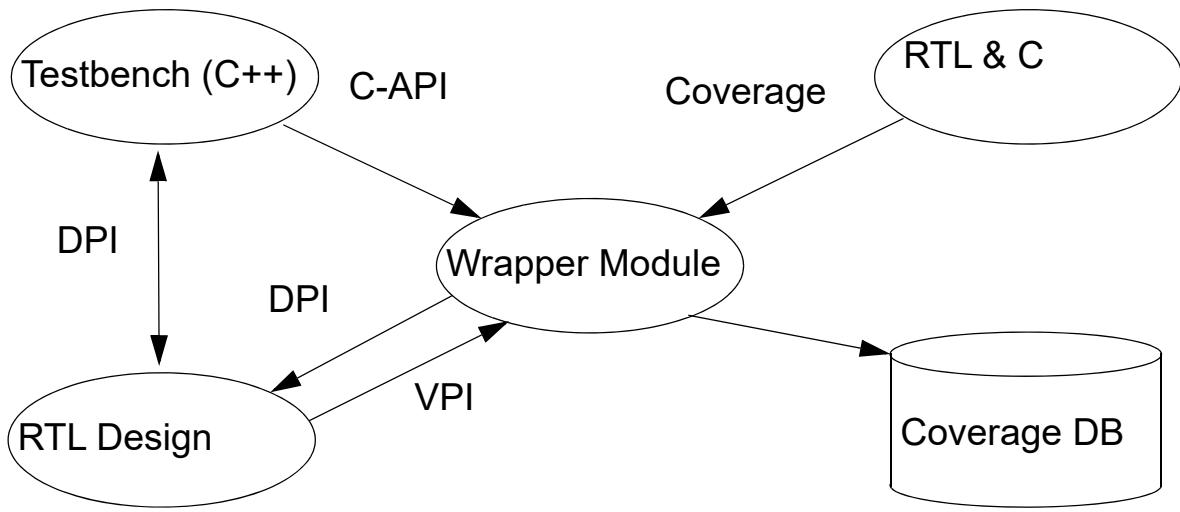
When you use the SystemVerilog C/C++ interface feature, you need to include the header file, `svCovgAPI.h`.

---

## SystemVerilog Functional Coverage Flow

Figure 17-1 illustrates the functional coverage flow:

Figure 17-1 SystemVerilog C/C++ Functional Coverage Flow



DPI is the SystemVerilog Direct Programming Interface. For details and examples of using DPI, see “SystemVerilog DPI” in the *SystemVerilog LRM IEEE Std. 1800 - 2012*.

VPI is the Verilog Procedural Interface. For information about using VPI with SystemVerilog, see, Chapter, “VPI object model diagrams” in the *SystemVerilog LRM IEEE Std. 1800 - 2012*.

Covergroups are defined in SystemVerilog, and then they are used to track the functional coverage of C/C++ code through the C-API (C Application Programming Interface). There are two major parts to C/C++ functional coverage interface:

- Covergroup(s)
- C/C++ testbench using those covergroups

---

## Covergroup Definition

The following section lists the covergroup limitations for C/C++ functional coverage.

- It cannot have a sampling clock.
- It must be declared in \$unit.
- It cannot be inside another scope (for example, modules, programs, and so on).
- It must not be instantiated anywhere in else SystemVerilog code.
- Arguments can only be in int, enum (base type int), and bit vector types. The SystemVerilog-to-C data-type mapping is compliant with DPI. [Table 17-1](#) shows the mapping of the supported types:

*Table 17-1 SystemVerilog-to-C Data-Type Mapping by DPI*

| SystemVerilog | C             |
|---------------|---------------|
| int           | int           |
| bit           | unsigned char |
| bit [m:n]     | svBitVec32    |
| enum int      | int           |

- Definitions must appear in files that are separate from the DUT because the definitions are compiled separately with the VCS command-line option, -c\_covg.

After you define the covergroups, compile them with -c\_covg (that is, -c\_covg <covergroup\_file>). If you have multiple covergroup files, you must precede each of them with the -c\_covg option (that is, -c\_covg <covergroup\_file1> -c\_covg <covergroup\_file2> ...).

The `-sverilog` and `+vpi` options are also needed when compiling with `-c_covg`.

After compiling the covergroups to be used with C/C++, the C-API allows for the allocation of covergroup handles, manual triggering of the covergroup sample, and the ability to de-instance and free the previously declared covergroup handle.

The following is a list of the C-API functions:

- `svCovgNew`/`svCovgNew2`
- `svCovgSample`/`svCovgSample2`
- `svCovgDelete`

Detailed specifications for these functions appear in “[C/C++ Functional Coverage API Specification](#)” .

The following examples demonstrate the use model.

## SystemVerilog (Covergroup for C/C++): covg.sv

```
cp: coverpoint count {
    bins b = {data};
    ...
}
endgroup
```

## C Testbench: test.c

```
int my_c_testbench ()
{
    svCovgHandle cgh;
    // C variables
    int data;
    int count;
```

## Approach #1: Passing Arguments by Reference

```
// Create a covergroup instance; pass data as a value
// parameter and count as a reference parameter;
// coverage handle remembers references
cgh = svCovgNew("cg", "cg_inst", SV_SAMPLE_REF, data,
&count);

// Sample stored references
svCovgSample(cgh); // sampling by the stored reference
...

// Delete covergroup instance
svCovgDelete(cgh);
```

## Approach #2: Passing Arguments by Value

```
// Create a covergroup instance; pass data and count as
// value parameters
cgh = svCovgNew("cg", "cg_inst", SV_SAMPLE_VAL, data,
count);

// Sample values passed for covergroup ref arguments
svCovgSample(cgh, count); // sampling the value of count
...

// Delete covergroup instance
svCovgDelete(cgh);
```

## Compile Flow

Compile the coverage model (`covg.sv`) using `-c_covg` together with the design and the C testbench

This step assumes that you invoke the C testbench from the `dut.sv` design through some C interface (for example, DPI, PLI, and so on). For example:

```
vcs -sverilog dut.sv test.c -c_covg +vpi covg.sv
```

## Runtime

At runtime (executing `simv`), the functional coverage data is collected and stored in the coverage database.

---

## C/C++ Functional Coverage API Specification

This section gives detailed specifications for the C/C++ functional coverage C-API.

**`svCovgHandle svCovgNew (char* cgName, char* ciName, int refType, args ...);`**

**`svCovgHandle svCovgNew2 (char* cgName, char* ciName, int refType, va_list v);`**

### Parameters

*cgName*

Covergroup name.

*ciName*

Covergroup instance name (should be unique).

*refType*

`SV_SAMPLE_REF` or `SV_SAMPLE_VAL`.

*args*

A variable number of arguments for creating a new covergroup instance.

*vl*

Represents a C predefined data structure (`va_list`) for maintaining a list of arguments.

## Description

Create a covergroup instance using the covergroup and instance names. If there is no error, return `svCovgHandle`, otherwise return NULL. The C variable sampling type (either reference or value) is specified using `refType`. The sampling type is stored in `svCovgHandle`. The `svCovgNew2` function is similar to `svCovgNew` except that you provide it with `va_list`, instead of a variable number of arguments (represented by "...") to `svCovgNew`.

For value sampling, pass values for non-reference and reference arguments in the order specified in the covergroup declaration, and set `refType` to `SV_SAMPLE_VAL`.

For reference sampling, pass values for non-reference arguments and addresses for reference arguments in the order specified in the covergroup declaration. References must remain valid during the life of the covergroup instance. Set `refType` to `SV_SAMPLE_REF`.

Type checking is not performed for arguments. You need to pass correct values and addresses.

**int svCovgSample(svCovgHandle *ch*, args ...);**

**int svCovgSample2(svCovgHandle *ch*, va\_list *vl*);**

## Parameters

*ch*

Handle to a covergroup instance created by `svCovgNew()`.

*args*

A variable number of arguments for sampling a covergroup by value, if `refType = SV_SAMPLE_VAL` in `svCovgNew()`.

*vl*

Represents a C predefined data structure (`va_list`) for maintaining a list of arguments.

## Description

Sample a covergroup instance using the sampling style stored in `svCovgHandle` and return 1 (TRUE) if no error, otherwise return 0 (FALSE). The `svCovgSample2` function is similar to `svCovgSample` except that you provide `va_list`, instead of a variable number of arguments (represented by "..."), to `svCovgSample`.

For value sampling, provide values for reference arguments in the order specified in the covergroup declaration. Type checking is not performed for value arguments. It is your responsibility to pass correct values.

For reference sampling, use stored addresses for reference arguments in `svCovgHandle`.

**int svCovgDelete(svCovgHandle *ch*);**

## Parameters

*ch*

Handle to a covergroup instance created by `svCovgNew()` (or `svCovgNew2`).

## **Description**

Delete a covergroup instance and return 1 (TRUE) if no error, otherwise return 0 (FALSE).

# 18

## OpenVera-SystemVerilog Testbench Interoperability

---

The primary purpose of OpenVera-SystemVerilog interoperability in VCS Native Testbench is to enable you to reuse OpenVera classes in new SystemVerilog code without rewriting OpenVera code into SystemVerilog.

This chapter describes the following topics:

- “[Scope of Interoperability](#)”
- “[Importing OpenVera Types Into SystemVerilog](#)”

Using the SystemVerilog package import syntax to import OpenVera data types and constructs into SystemVerilog.

- “Data Type Mapping”

The automatic mapping of data types between the two languages as well as the limitations of this mapping (some data types cannot be directly mapped).

- “Connecting to the Design”

Mapping of SystemVerilog modports to OpenVera where they can be used as OpenVera virtual ports.

- “Notes to Remember”

- “Usage Model”

- “Limitations”

---

## Scope of Interoperability

The scope of OpenVera-SystemVerilog interoperability in VCS Native Testbench is as follows:

- Classes defined in OpenVera can be used directly or extended in SystemVerilog testbenches.
- Program blocks must be coded in SystemVerilog. The SystemVerilog interface can include constructs, such as modports and clocking blocks, to communicate with a design.
- OpenVera code must not contain program blocks, bind statements, or predefined methods. It can contain classes, enums, ports, interfaces, tasks, and functions.

- OpenVera code can use virtual ports for sampling, driving, or waiting on design signals that are connected to the SystemVerilog testbench.

---

## Importing OpenVera Types Into SystemVerilog

OpenVera has two user-defined types: enums and classes. These types can be imported into SystemVerilog by using the SystemVerilog package `import` syntax:

```
import OpenVera::openvera_class_name;  
import OpenVera::openvera_enum_name;
```

It allows you to use `openvera_class_name` in the SystemVerilog code in the same way as a SystemVerilog class. This includes the ability to perform the following:

- Create objects of the `openvera_class_name` type.
- Access or use properties and types defined in `openvera_class_name` or its base classes.
- Invoke methods (virtual and non-virtual) defined in `openvera_class_name` or its base classes.
- Extend `openvera_class_name` to SV classes.

However, this does not import the names of base classes of `openvera_class_name` into SystemVerilog (that requires an explicit import). For example:

```

// OpenVera
    class Base {
        .
        .
        .
        task foo(arguments) {
            .
            .
            .
        }
        virtual task (arguments) {
            .
            .
            .
        }
    class Derived extends Base {
        virtual task vfoo(arguments) {
            .
            .
            .
        }
    }

// SystemVerilog
import OpenVera::Derived;
Derived d = new; // OK
initial begin
    d.foo(); // OK (Base::foo automatically
             // imported)
    d.vfoo(); // OK
end
Base b = new; // not OK (do not know that Base is a
              // class name)

```

The previous example would be valid if you add the following line before the first usage of the `Base` class name.

```
import OpenVera::Base;
```

Continuing with the previous example, SystemVerilog code can extend an OpenVera class as shown below:

```
// SystemVerilog
import OpenVera::Base;
class SVDerived extends Base;
    virtual task vmt()
        begin
            .
            .
            .
        end
    endtask
endclass
```

**Note:**

- If a derived class redefines a base class method, the arguments of the derived class method must exactly match the arguments of the base class method.
- Explicit import of each data type from OpenVera can be avoided by a single `import OpenVera::*`.

```
// OpenVera
    class Base {
        integer i;
        .
        .
        .
    }
    class wrappedBase {
        public Base myBase;
    }
// SystemVerilog
    import OpenVera::wrappedBase;
    class extendedWrappedBase extends wrappedBase;
        .
        .
        .
    endclass
```

In this example, `myBase.i` can be used to refer to this member of `Base` from the SV side. However, if SV also needs to use objects of the `Base` type, then you must include the following:

```
import OpenVera::Base;
```

---

## Data Type Mapping

This section describes how various data types in SystemVerilog are mapped to OpenVera and vice versa:

- *Direct mapping*: Many data types have a direct mapping in the other language and no conversion of data representation is required. In such cases, 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 the 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`. However, `reg` in OpenVera can be directly mapped to `logic` in SystemVerilog. 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 users. This is because in OpenVera, mailboxes and semaphores are represented by `integer` *ids* and VCS cannot reliably determine if an `integer` value represents a mailbox *id*.

---

## Mailboxes and Semaphores

Mailboxes and semaphores are referenced using object handles in SystemVerilog whereas in OpenVera, they are referenced using integral *ids*. VCS supports the mapping of mailboxes between these two languages.

For example, consider a mailbox created in SystemVerilog. To use it in OpenVera, you need to get *id* for the mailbox. The `get_id()` function, available as a VCS extension to SystemVerilog, returns this value:

```
function int mailbox::get_id();
```

The `get_id()` function is 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 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 occurs.

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 results in a simulation error.

To use an OpenVera mailbox in SystemVerilog, you need to get a handle to the mailbox object using a system function call. The system function, `$get_mailbox`, returns this handle:

```
function mailbox $get_mailbox(int id);
```

The function is 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 `string`. In SystemVerilog, the default value is the empty string (""). It is illegal to assign `null` to `string` in SystemVerilog. Currently, Native Testbench-OpenVera (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 is 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 the 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 are implicitly converted to the appropriate 4-state type (of the same bit length). For the conversion of bit vector types, see the discussion in Section 2.5.

OpenVera enum types can be imported to SystemVerilog using the following syntax:

```
import OpenVera::openvera_enum_name;
```

It is 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 OpenVera classes are illustrated in the following example:

```

class OVclass{
    enum Opcode {Add, Sub, Mul};
}

import OpenVera::OVclass;
OVclass::Opcode SVvar;
SVvar=OVclass::Add;

```

## Integers and Bit-Vectors

The mapping between SystemVerilog and OpenVera integral types are shown in the following table:

| <b>SystemVerilog</b> | <b>OpenVera</b>    | <b>2/4 or 4/2 value conversion?</b> | <b>Change in sign?</b> |
|----------------------|--------------------|-------------------------------------|------------------------|
| integer              | integer            | N<br>(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<br>(equivalent types)             | N (Both unsigned)      |
| bit [m:n]            | reg [abs(m-n)+1:0] | Y                                   | N (Both unsigned)      |
| time                 | reg [63:0]         | Y                                   | N (Both unsigned)      |

### Note:

If a value or sign conversion is needed between the actual and formal arguments of a task or function, then the argument cannot be passed by reference.

---

## Arrays

Arrays can be passed as arguments to tasks and functions from SystemVerilog to OpenVera and vice versa. The formal and actual array arguments must have equivalent element types, the same number of dimensions with corresponding dimensions of the same length. These rules follow the SystemVerilog LRM.

- A SystemVerilog fixed array dimension of the form  $[m:n]$  is directly mapped to  $[\text{abs}(m-n)+1]$  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 the `string` key type is equivalent to a SystemVerilog associative array with `string` key type provided the element types are equivalent.

Other types of SystemVerilog associative arrays have no equivalent in OpenVera and hence, they cannot be passed across the language boundary.

Some examples of compatibility are described in the following table:

| OpenVera        | SystemVerilog         | Compatibility |
|-----------------|-----------------------|---------------|
| integer a[10]   | integer b[11:2]       | Yes           |
| integer a[10]   | int b[11:2]           | No            |
| reg [11:0] a[5] | logic [3:0][2:0] b[5] | Yes           |

A 2-valued array type in SystemVerilog cannot be directly mapped to a 4-valued array in OpenVera. However, a cast may be performed as follows:

```
// OpenVera
    class Foo {
        .
        .
        .
        task vera_method(integer array[5]) {
            .
            .
            .
            .
        }
    }
// SystemVerilog
    int array[5];
    typedef integer array_t[5];
    import OpenVera::Foo;
    Foo f;
    .
    .
    .
    f.vera_method(array);           // Error: type mismatch
    f.vera_method((array_t')array)); // OK
    .
    .
    .
```

---

## Structs and Unions

Unpacked structs/unions cannot be passed as arguments to OpenVera methods. Packed structs/unions can be passed as arguments to OpenVera: they are 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

This section consists of the following subsections:

- “[Mapping Modports to Virtual Ports](#)”
- “[Semantic Issues With Samples, Drives, and Expects](#)”

---

### Mapping Modports to Virtual Ports

This section relies on the following extensions to SystemVerilog supported in VCS:

- “[Virtual Modports](#)”
- “[Importing Clocking Block Members Into a Modport](#)”

## Virtual Modports

VCS supports a *reference* to a modport in an interface to be declared using the following syntax:

```
virtual interface_name.modport_name virtual_modport_name;
```

For example:

```
interface IFC;
    wire a, b;
    modport mp (input a, output b);
endinterface

IFC i();
virtual IFC.mp vmp;
.
.
.
vmp = i.mp;
```

## Importing Clocking Block Members Into a Modport

VCS allows a reference to a clocking block member to be made by omitting the clocking block name.

For example, in SystemVerilog a clocking block is used in a modport as follows:

```
interface IFC(input clk);
    wire a, b;
    clocking cb @ (posedge clk);
        input a;
        input b;
    endclocking
    modport mp (clocking cb);
endinterface
```

```

program mpg(IFC ifc);
.
.
.
.
virtual IFC.mp vmp;
.
.
.
.
vmp = i.mp;
@(vmp.cb.a); // here you need to specify cb explicitly
.
endprogram
module top();
.
.
.
IFC ifc(clk); // use this to connect to DUT and TB
mpg mpg(ifc);
dut dut(...);
.
.
.
endmodule

```

VCS supports the following extensions that allow the clocking block name to be omitted from `vmp.cb.a`.

```

// Example-1
interface IFC(input clk);
    wire a, b;
    clocking cb @ (posedge clk);
        input a;
        input b;
    endclocking
    modport mp (import cb.a, import cb.b);
endinterface

program mpg(IFC ifc);
.
.
.
virtual IFC.mp vmp;

```

```

.
.
.

    vmp = i.mp;
    @ (vmp.a); // cb can be omitted; 'cb.a' is
                // imported into the modport

endprogram
module top();
.

.

    IFC ifc(clk); // use this to connect to DUT and TB
    mpg mpg(ifc);
    dut dut(...);

.

.

endmodule

// Example-2
interface IFC(input clk);
    wire a, b;
    bit clk;
    clocking cb @ (posedge clk);
        input a;
        input b;
    endclocking
    modport mp (import cb.*); // All members of cb
                                // are imported.
                                // Equivalent to the
                                // modport in
                                // Example-1.
endinterface

program mpg(IFC ifc);
.

.

    IFC i(clk);
.

.

    virtual IFC.mp vmp;
.
.
```

```

        .
        .
        vmp = i.mp;
        @ (vmp.a); // cb can be omitted;
                    //'cb.a' is imported into the modport
    endprogram

module top();
    .
    .
    IFC ifc(clk); // use this to connect to DUT and TB
    mpg mpg(ifc);
    dut dut(...);
    .
    .
endmodule

```

A SystemVerilog modport can be implicitly converted to an OpenVera virtual port provided the following conditions are satisfied:

- The modport and the virtual port have the same number of members.
- Each member of the modport converted to a virtual port must either be: (1) a clocking block, or (2) imported from a clocking block using the `import` syntax above.
- For different modports to be implicitly converted to the same virtual port, the corresponding members of the modports (in the order in which they appear in the modport declaration) be of bit lengths. If the members of a clocking block are imported into the modport using the `cb.*` syntax, where `cb` is a clocking block, then the order of those members in the modport is determined by their declaration order in `cb`.

## Example

```
// OpenVera
port P {
    clk;
    a;
    b;
}

class Foo {
    P p;
    task new(P p_) {
        p = p_;
    }

    task foo() {
        .
        .
        .
        @(p.$clk);
        .
        variable = p.$b;
        p.$a = variable;
        .
        .
        .
    }
}

// SystemVerilog
interface IFC(input clk);
    wire a;
    wire b;

    clocking clk_cb @(clk);
        input #0 clk;
    endclocking

    clocking cb @ (posedge clk);
        output a;
        input b;
    endclocking
}
```

```

modport mp (import clk_cb.* , import cb.*); // modport
    // can aggregate signals from multiple clocking blocks.

endinterface: IFC

program mpg(IFC ifc);
    import OpenVera::Foo;
    .
    .
    virtual IFC.mp vmp = ifc.mp;
    Foo f = new(vmp); // clocking event of ifc.cb mapped to
                      // $clk in port P
                      // ifc.cb.a mapped to $a in port P
                      // ifc.cb.b mapped to $b in port P
    .
    f.foo();
    .
    .
    .
endprogram

module top();
    .
    .
    IFC ifc(clk); // use this to connect to DUT and TB
    mpg mpg(ifc);
    dut dut(...);
    .
    .
endmodule

```

### Note:

In the above example, you can also directly pass the `vmp` modport from an interface instance:

```
Foo f = new(ifc.mp);
```

---

## Semantic Issues With Samples, Drives, and Expects

When OpenVera code wants to sample a DUT signal through a virtual port (or interface), if the current time is not at the relevant clock edge, the current thread is suspended until that clock edge occurs and then the value is sampled. Native Testbench-OpenVera (NTB-OV) implements this behavior by default. On the other hand, in SystemVerilog, sampling never blocks and the value that was sampled at the most recent edge of the clock is used. Analogous differences exist for drives and expects.

---

## Notes to Remember

### Blocking Functions in OpenVera

When a SystemVerilog function calls a virtual function that may resolve to a blocking OpenVera function at runtime, the compiler cannot determine with certainty whether the SystemVerilog function will block. VCS issues a warning at compile time and let the SystemVerilog function block at runtime.

Besides killing descendant processes in the same language domain, `terminate` invoked from OpenVera also kills descendant processes in SystemVerilog. Similarly, `disable fork` invoked from SystemVerilog also kills descendant processes in OpenVera. `wait_child` also waits for SystemVerilog descendant processes and `wait fork` also waits 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 is maintained across the language domain.

```
//OV
class OVclass{
    rand integer ri;
    constraint cnst{...}
}

//SV
OVclass obj=new();
SVclass Sobj=new();
Sobj.randomize();
obj.randomize() with
{obj.ri==Sobj.var;};
```

## Functional Coverage

There are some differences in functional coverage semantics between OpenVera and SystemVerilog. These differences are currently being eliminated by changing OpenVera semantics to conform to SystemVerilog. In the interoperability mode, `coverage_group` in OpenVera and `covergroup` in SystemVerilog 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 are unified and keywords such as `coverpoint`, `bins` are used from SystemVerilog instead of OpenVera keywords.

Here is an example of usage of coverage groups across the language boundary:

```
// OpenVera
class A
{
    B b;
    coverage_group cg {
        sample x(b.c);
        sample y(b.d);
        cross cc1(x, y);
        sample_event = @ (posedge CLOCK);
    }
    task new() {
        b = new;
    }
}
// SystemVerilog

import OpenVera::A;

initial begin
    A obj = new;
    obj.cg.option.at_least = 2;
    obj.cg.option.comment = "this should work";
    @(posedge CLOCK);
    $display("coverage=%f", obj.cg.get_coverage());
end
```

---

## Usage Model

Any `define from the OpenVera code is visible in SystemVerilog once they are explicitly included.

Note:

OpenVera #define must be rewritten as `define for ease of migration to SystemVerilog.

### For Two-Step Flow:

#### Compilation

```
% vcs [compile_options] -sverilog -ntb_opts interop  
[other_NTB_options] file4.sv file5.vr file2.v file1.v
```

#### Simulation

```
% simv [simv_options]
```

### For Three-Step Flow:

#### Analysis

```
% vlogan -sverilog -ntb_opts interop [other_NTB_options] \  
[vlogan_options] file4.sv file5.vr file2.v file1.v  
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

Note:

Specify the VHDL bottommost entity first, and then move up in order.

#### Elaboration

```
% vcs [elab_options] top_cfg/entity/module
```

## Simulation

```
% simv [simv_options]
```

Note:

- If RVM class libs are used in the OpenVera code, use `-ntb_opts rvm` with the `vlogan` command.
- Using `-ntb_opts interop -ntb_opts rvm` with `vcs` or `vlogan`, automatically translates `rvm_` macros in the OpenVera package to `vmm_` equivalents.

---

## Limitations

- Classes extended/defined in SystemVerilog cannot be instantiated by OpenVera. OpenVera verification IP needs to be compiled with the Native Testbench syntax and semantic restrictions. These restrictions are detailed in the *Native Testbench Coding Guide*, included in the VCS release.
- SystemVerilog contains several data types that are not supported in OpenVera including real, unpacked-structures, and unpacked-unions. OpenVera cannot access any variables or class data members of these types. A compiler error occurs 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.

- When using VMM RVM Interoperability, you should only register VMM or RVM scenarios with a generator in the same language. You can instantiate an OpenVera scenario in a SystemVerilog scenario, but only a SystemVerilog scenario can be registered with a SystemVerilog generator. You cannot register OpenVera multi-stream scenarios on a SystemVerilog Multi-Stream Scenario Generator (MSSG).



# 19

## Using SystemVerilog Assertions

---

Using SystemVerilog Assertions (SVA) you can specify how you expect a design to behave and have VCS display messages when the design does not behave as specified.

```
assert property (@(posedge clk) req |-> ##2 ack)
               else $display ("ACK failed to follow the request);
```

The above example displays "ACK failed to follow the request", if ACK is not high two clock cycles after req is high. This example is a very simple assertion. For more information on how to write assertions, refer to Chapter 17 of the *SystemVerilog Language Reference Manual*.

VCS allows you to:

- Control the SVAs
- Enable or Disable SVAs

- Control the simulation based on the assertion results

This chapter describes the following:

- “Using SVAs in the HDL Design”
- “Controlling SystemVerilog Assertions”
- “Viewing Results”
- “Enhanced Reporting for SystemVerilog Assertions in Functions”
- “Controlling Assertion Failure Messages”
- “Reporting Values of Variables in the Assertion Failure Messages”
- “Reporting Messages When \$uniq\_prior\_checkon/\$uniq\_prior\_checkoff System Tasks are Called”
- “Assertion and Unique/Priority Re-Trigger Feature”
- “Enabling Lint Messages for Assertions”
- “Fail-Only Assertion Evaluation Mode”
- “Treating x as true on an Assertion Precondition”
- “Using SystemVerilog Constructs Inside vunits”
- “Calling \$error Task When Else Block is Not Present”
- “Disabling Default Assertion Success Dumping in -debug\_access Option”
- “List of supported IEEE Std. 1800-2012 Compliant SVA Features”
- “SystemVerilog Assertions Limitations”
- “SystemVerilog Assertions Limitations”

---

## Using SVAs in the HDL Design

You can instantiate SVAs in your HDL design in the following ways:

- “[Using VCS Checker Library](#)”
  - “[Binding SVA to a Design](#)”
  - “[Inlining SVAs in the Verilog Design](#)”
  - “[Number of SystemVerilog Assertions Supported in a Module](#)”
  - “[Inlining SVA in the VHDL design](#)”
- 

## Using VCS Checker Library

VCS provides you with SVA checkers, which can be directly instantiated in your Verilog/VHDL source files. You can find these SVA checkers files in `$VCS_HOME/packages/sva_cg` directory.

This section describes the use model to analyze, elaborate/compile and simulate the design with SVA checkers. For more information on SVA checker libraries and list of available checkers, see the *SystemVerilog Assertions Checker Library Reference Manual*.

### Instantiating SVA Checkers in Verilog

You can instantiate SVA checkers in your Verilog source just like instantiating any other Verilog module. For example, to instantiate the checker `assert_always`, specify the following:

```
module my_verilog();  
  ....  
  assert_always always_inst (.clk(clk), .reset(rst),
```

```
.test_expr(test_expr);  
...  
endmodule
```

The use model to simulate the design with SVA checkers is as follows:

For two-step flow:

## Compilation

```
% vcs [vcs_options] -sverilog +define+ASSERT_ON \  
+incdir+$VCS_HOME/packages/sva -y $VCS_HOME/packages/sva  
+libext+.v \  
Verilog_source_files
```

## Simulation

```
% simv [simv_options]
```

For more information on SVA checker libraries and a list of available checkers, see the *SystemVerilog Assertions Checker Library Reference Manual*.

For three-step flow:

## Analysis

```
% vlogan -sverilog [vlogan_options] +define+ASSERT_ON \  
+incdir+$VCS_HOME/packages/sva -y $VCS_HOME/packages/sva  
+libext+.v \  
Verilog_source_files
```

Note:

It is necessary to use `+define+ASSERT_ON` to turn on the assertions in all checker instances.

## Elaboration

```
% vcs [vcs_options] top_cfg/entity/module
```

## Simulation

```
% simv [simv_options]
```

For more information on SVA checker libraries and a list of available checkers, see the *SystemVerilog Assertions Checker Library Reference Manual*.

## Instantiating SVA Checkers in VHDL

To instantiate SVA checkers in the VHDL source file, you need to do the following:

- Analyze the required SVA checker files using vlogan. For example, the command line to analyze the checker files in the default WORK library is as follows:

```
% vlogan $VCS_HOME/packages/sva/*.v \
+incdir+$VCS_HOME/packages/sva -y $VCS_HOME/
packages/sva +libext+.v \
+define+ASSERT_ON -sverilog
```

- Analyze the SVA component package file.

You can find SVA checkers in \$VCS\_HOME/packages/sva directory. In the same directory, you can also find the sva\_lib VHDL package, containing the component definitions for all the checkers in the library. The name of this file is component.sva\_v.vhd.

For example, suppose you analyze the package file in the default WORK library, then the vhdlan command line is as follows:

```
% vhdlan $VCS_HOME/packages/sva/component.sva_v.vhd
```

- To use the compiled checkers, you must include the `sva_lib` package in your VHDL file. For example, the following lines include the `sva_lib` package analyzed into the default `WORK` library:

```
library WORK;
use WORK.sva_lib.all;
```

For more information on SVA checker libraries and list of available checkers, see the *SystemVerilog Assertions Checker Library Reference Manual*.

You can now instantiate SVA checkers in your VHDL file, like any other VHDL entity. For example, to instantiate the checker `assert_always`, perform the following:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library WORK;
use WORK.sva_lib.all;

entity my_ent(
    ...
);
end my_ent;

architecture my_arch of my_ent is
...
begin
    ...
checker_inst : assert_always port map(.clk(clk),
   .reset(rst), a(1));
    ...
end my_arch;
```

The use model to simulate the design with SVA checkers is as follows:

## Analysis

Always analyze Verilog before VHDL.

```
% vlogan [vlogan_options] Verilog_source_files  
% vhdlan [vhdlan_options] VHDL_source_files
```

## Elaboration

```
% vcs [vcs_options] top_cfg/entity/module
```

## Simulation

```
% simv [simv_options]
```

---

## Binding SVA to a Design

Using bind statements to bind SVAs to your Verilog design. This is another way to use SVAs. The advantage is that the bind statements allow you to bind SVAs to the Verilog designs without modifying or editing your design files.

The syntax for bind statement is as follows:

```
bind inst_name/module SVA_module #[SVA_parameters]  
SVA_inst_name [SVA_ports]
```

The bind statement for Verilog targets can be used anywhere within your Verilog source file. For example:

```
//Verilog file  
module dev (...);  
...  
endmodule  
  
bind dev dev_checker dc1 (.clk(clk), .a(a), .b(b));
```

As shown in the above example, the `bind` statement is specified in the same Verilog file.

The use model to simulate the design is as follows:

### Compilation

```
% vcs -sverilog [compile_options] Verilog_files
```

### Simulation

```
% simv [run_options]
```

---

## Inlining SVAs in the Verilog Design

VCS allows you to write inlined SVAs for both VHDL and Verilog design. For Verilog designs, you can write SVAs as part of the code or within pragmas as shown in the following example:

### Example 1: Writing Assertions as a part of the code

```
module dut(...);

    ...

sequence s1;
@(posedge clk) sig1 ##[1:3] sig2;
endsequence

...

endmodule
```

### Example 2: Writing Assertions using SVA pragmas (`//sv_pragma`)

```
module dut(...);

    ...


```

```

//sv_pragma sequence s1;
//sv_pragma      @ (posedge clk) sig1 ##[1:3] sig2;
//sv_pragma endsequence

/*sv_pragma
sequence s2;
    @ (posedge clk) sig3 ##[1:3] sig4;
endsequence
*/
.....
endmodule

```

As shown in Example 2, you can use SVA pragmas as `//sv_pragma` at the beginning of all SVA lines, or you can use the following to mark a block of code as SVA code:

```

/* sv_pragma
sequence s2;
    @ (posedge clk) sig3 ##[1:3] sig4;
endsequence
*/

```

## Use Model

The use model to analyze, elaborate/compile and simulate the designs having inlined assertions is as follows:

For two-step flow:

### Compilation

```
% vcs -sv_pragma [compile_options] Verilog_files
```

### Simulation

```
% simv [run_options]
```

For three-step flow:

## Analysis

```
% vlogan -sv_pragma [vlogan_options] file1.v file2.v
```

### Note:

If you have your assertions inlined using //sv\_pragma, use the analysis option -sv\_pragma as shown above.

## Elaboration

```
% vcs -sv_pragma [elab_options] design_unit
```

## Simulation

```
% simv [run_options]
```

---

## Inlining SVA in the VHDL design

Inlining SVAs in VHDL design is possible only by using the SVA pragmas. The location of the SVA implicitly specifies to which entity-architecture the SVA code is bound to. You can embed the SVA code in the concurrent portion on your VHDL code using the --sva\_begin and --sva\_end pragmas. These pragmas should be written within an architecture - end architecture definition block as shown in the following example:

```
architecture RTL of cntrl is
begin
  ...
  --sva_begin
  -- property p1;
  --      @(posedge clk) a && b ##1 !c ;
  -- endproperty : p1

  -- a_p1: assert property (p1) else $display ($time, " :
  --           Assertion a_p1 failed");
  --sva_end
end architecture RTL;
```

As soon as VCS encounters `--sva_begin`, it implicitly understands that the following lines until `--sva_end` are SVA constructs.

Within the inlined SVA code, you can:

- use VHDL signals, generics, and constants.
- write Verilog comments, compiler directives, and SVA pragmas.

However, you cannot use a VHDL variable within the inlined SVA code.

## Use Model

### Analysis

Always analyze Verilog before VHDL.

```
% vlogan -sverilog [vlogan_options] file1.v file2.v file3.v  
% vhdlan -sva [vhdlan_options] file2.vhd file1.vhd
```

Note:

- Use the `-sva` option, if you have SVA code inlined in your VHDL.
- For analysis, analyze the VHDL bottom-most entity first, then move up the order.

### Elaboration

```
% vcs [vcs_options] top_cfg/entity/module
```

### Simulation

```
% simv [simv_options]
```

You can also use the option `-sv_opts "vlog_opts_to_SVAs"` with `vhdlan` to specify Verilog options, such as `+define+macro -timescale=timeunit/precision` to the inlined SVA code as shown in the following example:

```
% vhdlan -sva -sv_opts "-timescale=1ns/1ns" myDut.vhd
```

The following example shows the usage of `'ifdef` within the inlined SVA code:

```
architecture RTL of cntrl is
begin
  ...
  --sva_begin
  -- `ifdef P1
  --   property p1;
  --     @(posedge clk) a && b ##1 !c ;
  --   endproperty : p1
  -- `else
  --   property p1;
  --     @(posedge clk) a || b ##1 !c ;
  --   endproperty : p1
  -- `endif
  --   a_p: assert property (p1)
  --     else $display ($time, " : Assertion a_p failed");
  --sva_end
end architecture RTL;
```

In this example, to select the first property `P1`, you need to specify `+define+P1` as an argument to `-sv_opts` option as follows:

```
% vhdlan -sva -sv_opts "+define+P1" myDut.vhd
```

---

## **Number of SystemVerilog Assertions Supported in a Module**

VCS supports up to 1,048,000 SystemVerilog assertions per module.

Note:

Large number of assertions in a module can cause performance issues. If the performance degrades, it is recommended to subdivide the module into multiple modules and distribute assertions over those modules.

---

## **Controlling SystemVerilog Assertions**

SVAs can be controlled or monitored using:

- “[Compilation/Elaboration and Runtime Options](#)”
- “[Concatenating Assertion Options](#)” on page 17
- “[Assertion Monitoring System Tasks](#)”
- “[Using Assertion Categories](#)”

---

## **Compilation/Elaboration and Runtime Options**

VCS provides various compilation/elaboration options to perform the following tasks:

- The following is the list of assertion options that are enabled when the `-assert enable_diag` compilation/elaboration option is used:

- `-assert success`
- `-assert summary`
- `-assert maxcover=N`
- `-assert maxsuccess=N`

**The following is the list of assertion options that are enabled when `-assert enable_hier` compilation/elaboration option is used:**

- `-assert hier`
- `-assert maxfail=N`
- `-assert finish_maxfail=N`

**The following is the list of assertion options that do not require the `-assert enable_diag` or `-assert enable_hier` option:**

- `-assert dumpoff`
- `-assert nocovdb`
- `-assert nopostproc`
- `-assert quiet`
- `-assert quiet1`
- `-assert no_fatal_action`
- `-assert report`
- `-assert vacuous`
- `-assert global_finish_maxfail=N`

- To disable all SVAs in the design, use the `-assert disable` compile-time option. To disable only the SVAs specified in a file, use the `-assert hier=<file_name>` compile-time option.
- To disable assertion coverage, use the `-assert disable_cover` compile-time option. By default, when you use the `-cm assert` option, VCS enables you to monitor your assertions for coverage, and write an assertion coverage database during simulation.
- To disable property checks (that is, `assert` and `assume` directives) and retain assertion coverage (that is, `cover` directives), use the `-assert disable_assert` option at compile time.
- Disable dumping of SVA information in the VPD file

You can use the `-assert dumpoff` option to disable the dumping of SVA information to the VPD file during simulation (for additional information, see “[Options for SystemVerilog Assertions](#)” ).

VCS allows you to do the following tasks during runtime:

- Terminate simulation after certain number of assertion failures.  
You can use either the `-assert finish_maxfail=N` or `-assert global_finish_maxfail=N` runtime option to terminate the simulation if the number of failures for any assertion reaches  $N$  or if the total number of failures from all SVAs reaches  $N$ , respectively.
- Show both passing and failing assertions

By default, VCS reports only failures. However, you can use the `-assert success` option to enable reporting of successful matches, and successes on `cover` statements in addition to failures.

- Limit the maximum number of successes reported

You can use the `-assert maxsuccesses=N` option to limit the total number of reported successes to  $N$ .

- Disable the display of messages when assertions fail

You can use the `-assert quiet` option to disable the display of messages when assertions fail.

- Enable or disable assertions during runtime

You can use the `-assert hier=file_name` option to enable or disable the list of assertions in the specified file.

- Generate a report file

You can use the `-assert report=file_name` option to generate a report file with the specified name. For additional information, see “[Options for SystemVerilog Assertions](#)” .

- Enable assertion control using wildcard characters.

You can use the `-assert enable_wildcard` option to enable the support of assertion control through wildcard characters.

You can enter more than one keyword, using the plus + separator. For example:

```
% vcs -assert maxfail=10+maxsucess=20+success
```

---

## Concatenating Assertion Options

VCS allows you to concatenate the compile-time and runtime assertion options on the compile command-line using the + separator. VCS identifies and appropriately passes these options to compile and runtime, wherever applicable. Concatenating all options simplifies the use model.

For example, you can concatenate compile-time option `-assert enable_diag` with the runtime option `-assert success`, as follows:

```
% vcs -assert success+enable_diag -R
```

When an option has the same name at both compile time and runtime, and are used on the `vcs` compile command line with the `-R` option, then it the option is applied only at compile time.

---

## Assertion Monitoring System Tasks

To monitor SystemVerilog assertions, use 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 `$assert_monitor` system task is equivalent to the standard `$monitor` system task. It continually monitors the specified assertions and displays what is happening with them (you can have it displayed only on the next clock of the assertion). The syntax is as follows:

```
$assert_monitor([0|1,] assertion_identifier...);
```

Where:

0

Specifies reporting on the assertion if it is active (VCS checks for its properties). For the remaining assertions, it specifies reporting, whenever they start.

1

Specifies reporting on the assertions only once, the next time they start.

If you specify neither 0 or 1, the default is 0.

*assertion\_identifier*...

A comma separated list of assertions. If one of these assertions is not declared in the module definition containing this system task, specify it by its hierarchical name.

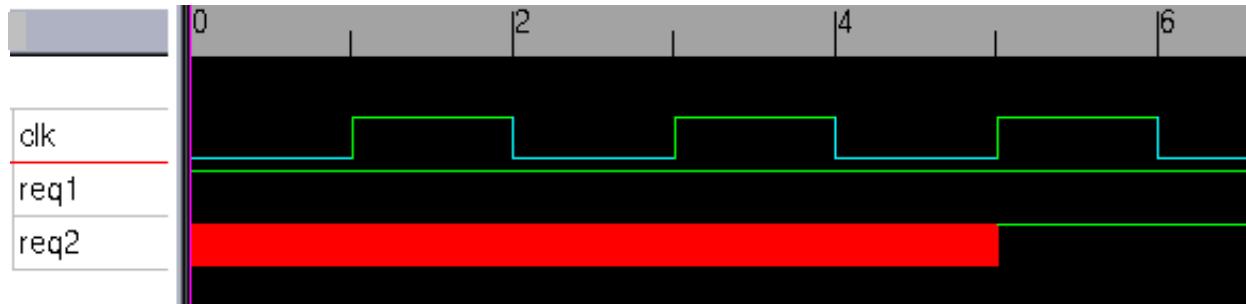
Consider the following assertion:

```
property p1;
  @ (posedge clk) (req1 ##[1:5] req2);
endproperty

a1: assert property(p1);
```

For property `p1` in assertion `a1`, a clock tick is a rising edge on signal `clk`. When there is a clock tick VCS checks to see if signal `req1` is true, and then to see if signal `req2` is true at any of the next five clock ticks.

In this example simulation, signal `clk` initializes to 0 and toggles every 1 ns, so the clock ticks at 1 ns, 3 ns, 5 ns and so on.



A typical display of this system task is as follows:

```
Assertion test.a1 ['design.v' 27] :  
5ns: tracing "test.a1" started at 5ns:  
      attempt starting found: req1 looking for: req2 or  
      any  
5ns: tracing "test.a1" started at 3ns:  
      trace: req1 ##1 any looking for: req2 or any  
      failed: req1 ##1 req2  
5ns: tracing "test.a1" started at 1ns:  
      trace: req1 ##1 any[* 2 ] looking for: req2 or any  
      failed: req1 ##1 any ##1 req2
```

Breaking this display into smaller chunks:

```
Assertion test.a1 ['design.v' 27] :
```

The display is about the assertion with the hierarchical name `test.a1`. It is in the source file named `design.v` and declared on line 27.

```
5ns: tracing "test.a1" started at 5ns:  
      attempt starting found: req1 looking for: req2 or  
      any
```

At simulation time, 5 ns VCS is tracing test.a1. An attempt at the assertion started at 5 ns. At this time, VCS found req1 to be true and is looking to see if req2 is true one to five clock ticks after 5 ns. Signal req2 doesn't have to be true on the next clock tick, so req2 not being true is okay on the next clock tick; that's what looking for "or any" means, anything else than req2 being true.

```
5ns: tracing "test.a1" started at 3ns:  
      trace: req1 ##1 any looking for: req2 or any  
      failed: req1 ##1 req2
```

The attempt at the assertion also started at 3 ns. At that time, VCS found req1 to be true at 3 ns and it is looking for req2 to be true some time later. The assertion "failed" in that req2 was not true one clock tick later. This is not a true failure of the assertion at 3 ns, it can still succeed in two more clock ticks, but it didn't succeed at 5 ns.

```
5ns: tracing "test.a1" started at 1ns:  
      trace: req1 ##1 any[* 2 ] looking for: req2 or any  
      failed: req1 ##1 any ##1 req2
```

The attempt at the assertion also started at 1 ns. [\* is the repeat operator. ##1 any[\* 2 ] means that after one clock tick, anything can happen, repeated twice. So the second line here says that req1 was true at 1 ns, anything happened after a clock tick after 1 ns (3 ns) and again after another clock tick (5 ns) and VCS is now looking for req2 to be true or anything else could happen. The third line here says the assertion "failed" two clock ticks (5 ns) after req1 was found to be true at 1 ns.

The `$assert_monitor_off` and `$assert_monitor_on` system tasks turn off and on the display from the `$assert_monitor` system task, just like the `$monitoroff` and `$monitoron` system turn off and on the display from the `$monitor` system task.

---

## Using Assertion Categories

You can categorize assertions and then enable and disable them by category. There are two ways to categorize assertions:

- [Using System Tasks](#)
  - [Using Assertion System Tasks](#)
- [Using Attributes](#)
- [Starting and Stopping Assertions Using Assertion System Tasks](#)
  - [Stopping Assertions by Category or Severity](#)

After you categorize assertions you can use these categories to stop and restart assertions.

## Using System Tasks

VCS has a number of system tasks and functions for assertions. These system tasks do the following:

- Set a category for an assertion
- Return the category of an assertion

### Using Assertion System Tasks

You can use the following assertion system tasks to set the category and severity attributes of assertions:

```
$assert_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.

```
$assert_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 tasks to retrieve the category and severity attributes of assertions:

```
$assert_get_severity("assertion_full_hier_name")
```

Returns the severity of action for an assertion failure.

```
$assert_get_category("assertion_full_hier_name")
```

Returns an unsigned integer for the category.

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, see [“Starting and Stopping Assertions Using Assertion System Tasks”](#).

## 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=l *) assert property (s1);
```

The attribute is retained in a `tokens.v` file when you use the `-Xman=4` compile-time option and the keyword argument.

## Starting and Stopping Assertions Using Assertion System Tasks

There are assertions system tasks for starting and stopping assertions. These system tasks are as follows:

### Stopping Assertions by Category or Severity

You can stop assertions by category using the following option:

```
$assert_category_stop(categoryValue,  
[maskValue[,globalDirective]]);
```

The option stops all assertions associated with the specified category.

You can stop assertions for the specified severity level using the following option:

```
$assert_severity_stop(severityValue,  
[maskValue[,globalDirective]]);
```

The option stops all assertions associated with the specified severity level.

where,

*categoryValue*

Since there is a *maskValue* argument, it is now the result of an anding operation between the assertion categories and the *maskValue* argument. If the result matches this value, these categories stop. Without the *maskValue* argument, this argument is the value you specify in `$assert_set_category` system tasks or `category` attributes.

*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*

The value can be either of the following values:

0

Enables an `$assert_category_start` system task that does not have a *globalDirective* argument, to restart the assertions stopped with this system task.

1

Prevents an `$assert_category_start` system task that does not have a *globalDirective* argument from restarting the assertions stopped with this system task.

## Starting Assertions by Category or Severity

You can start assertions by category using the following option:

```
$assert_category_start(categoryValue,
```

```
[maskValue[, globalDirective]]);
```

The option starts all assertions associated with the specified category.

You can start assertions for the specified severity level using the following option:

```
$assert_severity_start(severityValue,  
[maskValue[, globalDirective]]);
```

The option starts all assertions associated with the specified severity level. The severity level is an unsigned integer from 0 to 255.

where:

*categoryValue*

Since there is a *maskValue* argument, this argument is the result of an anding operation between the assertion categories and the *maskValue* argument. If the result matches this value, these categories start. Without the *maskValue* argument, this argument is the value you specify in `$assert_set_category` system tasks or `category` attributes.

*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 `$assert_category_stop` system task (that does not have a `globalDirective` argument) to stop the assertions started with this system task.

1

Prevents an `$assert_category_stop` system task that does not have a `globalDirective` argument from stopping the assertions started with this system task.

## Example Showing How to Use MaskValue

### Example 19-1 stops the odd numbered categories

*Example 19-1 MaskValue Numbering:*

```
$assert_set_category(top.d1.a1,1);
$assert_set_category(top.d1.a2,2);
$assert_set_category(top.d1.a3,3);
$assert_set_category(top.d1.a4,4);

.
.
.
.

$assert_category_stop(1,'h1);
```

The categories are masked with the `maskValue` argument and compared with the `categoryValue` argument as shown in the following table.

|            | bits | categoryValue |
|------------|------|---------------|
| category 1 | 001  |               |
| maskValue  | 1    |               |
| result     | 1    | match         |

|            |     |   |  |          |
|------------|-----|---|--|----------|
| category 2 | 010 |   |  |          |
| maskValue  | 1   |   |  |          |
| result     | 0   | 1 |  | no match |

|            |     |   |  |       |
|------------|-----|---|--|-------|
| category 3 | 011 |   |  |       |
| maskValue  | 1   |   |  |       |
| result     | 1   | 1 |  | match |

|            |     |   |  |          |
|------------|-----|---|--|----------|
| category 4 | 100 |   |  |          |
| maskValue  | 1   |   |  |          |
| result     | 0   | 1 |  | no match |

1. VCS logically *ands* the category value to the *maskValue* argument, which is 1.
2. The result of the *and* operation is true for categories 1 and 3 as per the calculation shown above. The result is false for categories 2 and 4.
3. VCS stops all the assertions which result in a true match with the *and* operation.

[Example 19-2](#) uses the *globalDirective* argument.

### *Example 19-2 Mask Value with Global Directive*

```
$assert_set_category(top.d1.a1,1);
$assert_set_category(top.d1.a2,2);
$assert_set_category(top.d1.a3,3);
$assert_set_category(top.d1.a4,4);
.
.
$assert_category_stop(1,'h1,1);
$assert_category_start(0,'h1);
```

The assertions that are stopped or started with `globalDirective` value 1, cannot be restarted or stopped with a call to `$assert_category_start`, without using the `globalDirective` argument. The above code cannot restart assertions.

The assertions can only be restarted with a call to `$assert_category_start` with `globalDirective`, as follows:

```
$assert_category_start(1, 'h1, 1);
```

or

```
$assert_category_start(1, 'h1, 0);
```

---

## Viewing Results

By default, VCS reports only assertion of the failures. However, you can use the `-assert success` runtime option to report both pass and failures.

Assertion results can be viewed:

- Using a Report File
- Using Verdi

For more information on viewing assertions using Verdi, refer to the “*Assertion Debug*” chapter in the Verdi and Siloti Command Reference Guide.

---

## Using a Report File

Using the `-assert report=file_name` option, you can create an assertion report file. VCS writes all the SVA messages to the specified file.

Assertion generates messages with the following format:

| File and line with the assertion                                     | Full hierarchical name of the assertion             | Start time | Status (succeeded at ..., failed at ..., not finished)         |
|----------------------------------------------------------------------|-----------------------------------------------------|------------|----------------------------------------------------------------|
| "design.v", 157: top.cnt_in.a2: started at 22100ns failed at 22700ns | Offending   ' (busData == mem[\$past(busAddr, 3)])' |            | Expression that failed (only with failure of check assertions) |

---

## Enhanced Reporting for SystemVerilog Assertions in Functions

This section describes an efficient reporting convention for functions containing assertions in the following topics:

- “[Introduction](#)”
- “[Use Model](#)”
- “[Name Conflict Resolution](#)”
- “[Checker and Generate Blocks](#)”

---

## Introduction

In earlier releases, when assertions were present inside functions, assertion path names were reported based on the position of the function call in the source file. For example, consider the following code:

```
module top;
bit b, a1, a2, a3, a4, a5;
function bit myfunc(input bit k);
    $display("FUNC name: %m");
    AF: assert #0(k && !k);
    return !k;
endfunction

always_comb a1=myfunc(b);
always_comb begin: A
begin: B
    a2=myfunc(b);
    begin a3=myfunc(!b); end
end
end
always_comb begin
    a4=myfunc(b);
    a5=myfunc(!b);
end
endmodule
```

If you run this code, it generates the following output:

```
"top.v", 5: top.\top.v_18_myfunc.AF : started .....
"top.v", 5: top.\top.v_17_myfunc.AF : started .....
"top.v", 5: top.\top.v_13_myfunc.AF : started .....
"top.v", 5: top.\top.v_12_myfunc.AF : started .....
"top.v", 5: top.\top.v_9_myfunc.AF : started .....
```

But the problem with this type of naming convention is that when code changes, the output of the simulation also changes. To overcome this limitation, a new naming convention is implemented under the `-assert funchier` compile-time option. This new naming convention is implemented as follows:

- Function names are generated based on the named blocks under which the functions are called. Each function name is appended with an index (index=0, 1, 2, 3...), where index 0 is assigned to the first function call, index 1 is assigned to the second function call, and so on.
- For unnamed blocks, the function name is based on the closest named block.
- If there is no named scope around the function call, then a module scope is used as a named block with an empty name.
- Each assertion status reporting the message contains the file name and the line number of the function caller.

---

## Use Model

Use the `-assert funchier` option to enable the new function naming convention, as shown in the following command:

```
% vcs -sverilog -assert funchier+svaext top.v  
% simv
```

If you run the above code using this command, it generates the following output:

```
"top.v", 5: top.myfunc_2.AF ("top.v", 18): started .....  
"top.v", 5: top.myfunc_1.AF ("top.v", 17): started .....  
"top.v", 5: top.\A.B.myfunc_1.AF ("top.v", 13): started ...
```

```
"top.v", 5: top.\A.B.myfunc_0.AF ("top.v", 12): started ....  
"top.v", 5: top.myfunc_0.AF ("top.v", 9): started .....
```

---

## Name Conflict Resolution

When a function name generated with the new naming convention conflicts with an existing block or identifier name in that scope, then the suffix index is incremented until the conflict is resolved.

---

## Checker and Generate Blocks

When a function is present inside a checker, the generated name of that function contains the checker name appended to all named blocks and identifiers in that checker.

Similarly, when a function is present inside a generate block, the generated name of that function contains the generated block name appended to all named blocks and identifiers in that generate block.

---

## Controlling Assertion Failure Messages

This section describes the mechanism to control failure messages for SystemVerilog Assertions (SVA), OpenVera Assertions (OVA), Property Specification Language (PSL) assertions, and OVA case checks.

This section contains the following topics:

- “Introduction”
- “Options for Controlling Default Assertion Failure Messages”

- “Options to Control Termination of Simulation”
  - “Option to Enable Compilation of OVA Case Pragmas”
- 

## Introduction

Earlier releases did not provide the flexibility to control the display of default messages for assertion (SVA, OVA, or PSL) failures based on the presence of an action block (for SVA) or a user message (for OVA and PSL). Also, there was no control over whether these assertion failures contributed to the failure counts for

`-assert [global_]finish_maxfail, or affected simulation if  
$ova_[severity|category]_action(<severity_or_category>,  
"finish") was specified.`

You can now use the options described in the following topics to enable additional controls on failure messages, and to terminate the simulation and compilation of OVA case pragmas.

---

## Options for Controlling Default Assertion Failure Messages

You can use the following runtime options to control the default assertion failure messages:

`-assert no_default_msg [=SVA | OVA | PSL]`

Disables the display of default failure messages for SVA assertions that contain a fail action block, and OVA and PSL assertions that contain user messages.

The default failure messages are displayed for:

- SVA assertions without fail action blocks
- PSL and OVA assertions that do not contain user messages

When used without arguments, this option affects SVA, OVA, and PSL assertions. You can use an optional argument with this option to specify the class of assertions that should be affected.

**Note:**

The `-assert quiet` and `-assert report` options override the `-assert no_default_msg` option. That is, if you use either of these options along with `-assert no_default_msg`, then the latter has no effect.

**The `-assert no_default_msg=SVA` option affects only SVA.**

**The `-assert no_default_msg=OVA` and `-assert no_default_msg=PSL` options affect both OVA and PSL assertions, but not SVA.**

In addition to the default message, an extra message is displayed by default, for PSL assertions that have a severity (info, warning, error, or fatal) associated with them. This message is considered as a user message, and no default message is displayed, if you use the `-assert no_default_msg[=PSL]` option.

## Example

Consider the following assertion:

```
As1: assert property (@(posedge clk) P1) else
$info("As1 fails");
```

By default, VCS displays the following information for each assertion failure:

```
"sva_test.v", 15: top.As1: started at 5s failed at 5s
Offending 'a'
Info: "sva_test.v", 15: top.As1: at time 5
As1 fails
```

If you use the `-assert no_default_msg` option at runtime, it disables the default message, and displays only the user message, as shown below:

```
Info: "sva_test.v", 15: top.As1: at time 5
As1 fails
```

---

## Options to Control Termination of Simulation

You can use the following runtime options to control the termination of simulation:

`-assert no_fatal_action`

Excludes failures on SVA assertions with fail action blocks for computation of failure count in the `-assert [global_]finish_maxfail=N` runtime option. This option also excludes failures of these assertions for termination of simulation, if you use the following command:

```
$ova_[severity|category]_action(<severity_or_category>,
"finish")
```

**Note:**

This option does not affect OVA case violations and OVA or PSL assertions with or without user messages.

Specifying `$fatal()` system task in the fail action block of an SVA assertion or in a fatal severity associated with a PSL assertion, results in termination of simulation irrespective of whether this option is used or not.

This option is useful when you want to exclude failures of assertions having fail action blocks from adding up to the global failure count, for the `-assert [global]_finish_maxfail=N` option.

## Example

Consider the following assertion:

```
As1: assert property (@(posedge clk) P1) else  
$info("As1 fails");
```

If you use the `-assert global_finish_maxfail=1` option at runtime, then the simulation terminates at the first `As1` assertion failure. Now, if you use `-assert global_finish_maxfail=1 -assert no_fatal_action` at runtime, then the failure of assertion `As1` does not cause the simulation to terminate.

`-ova_enable_case_maxfail`

Includes OVA case violations in computation of global failure count for the `-assert global_finish_maxfail=N` option.

Note:

The `-assert finish_maxfail=N` option does not include OVA case violations. This option maintains a per-assertion failure count for termination of simulation.

## Example

Consider an OVA case pragma, as shown in the following code, to check the case statements for full case violations:

```
reg [2:0] mda[31:0][31:0];
//ova full_case on;
initial begin
    for(i = 31; i >= 0; i = i - 1) begin
        for(j = 0; j <= 31; j = j + 1) begin
            case(mda[i][j])
                1: begin
                    testdetect[i][j] = 1'b1;
                end
            endcase
            #1;
        end
    end
end
```

The above code violates full case check. Therefore, case violations are displayed as follows:

```
Select expression value when violation happened for last
iteration : 3'b000
Ova [0]: "ova_case_full.v", 20: Full case violation at
time 9 in a
Failed in iteration: [ 31 ] [ 9 ]
```

By default, these violations are not considered in the failure count for the `-assert global_finish_maxfail=N` option. But if you use the `-ova_enable_case_maxfail` option at runtime, then the case violations are added in the failure count.

---

## Option to Enable Compilation of OVA Case Pragmas

You can use the following compile-time option to enable compilation of OVA case pragmas:

`-ova_enable_case`

Enables the compilation of OVA case pragmas only when used without the `-Xova` or `-ova_inline` option. All the inlined OVA assertion pragmas are ignored.

Note:

`-Xova` or `-ova_inline` is the superset of the `-ova_enable_case` option. They are used to compile both the case pragmas and assertions.

### Example

Consider the following code:

```
//ova parallel_case on;
//ova full_case on; /* case pragma*/
always @(negedge clock)
    case (opcode)
        //ova check_bool (alu_out>10, "ddd", negedge clock); /* assertion pragma */
        3'h0: alu_out = accum;
        3'h1: alu_out = accum;
        3'h2: alu_out = accum + data;
        3'h3: alu_out = accum & data;
        3'h4: alu_out = accum ^ data;
        3'h5: alu_out = data;
        3'h6: alu_out = accum;
    endcase
```

The above code contains both OVA case pragmas and assertions. This option ignores the OVA assertion pragmas and compiles only the case pragmas.

---

## Reporting Values of Variables in the Assertion Failure Messages

You can use the `-assert offending_values` compile-time option to enable reporting of the values of all variables used in the assertion failure messages, as shown:

```
"test.sv", 12: test.a1: started at 5s failed at 5s
Offending 'ack'
$rose(sh) = 0
$fell(rst) = 1
ack = 'b0
```

The values of the variables contained in the failing portion of a property are generated using the following formats:

**Table 19-1 Reporting Formats**

| Variable Type                        | Format                                                                                                                                                                                                                                                 |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Simple scalar bit or logic variables | <code>&lt;var_name&gt; = %b</code>                                                                                                                                                                                                                     |
| Bitvector variables                  | <code>&lt;var_name&gt; = %h</code>                                                                                                                                                                                                                     |
| Int and integer variables            | <code>&lt;var_name&gt; = %0d</code>                                                                                                                                                                                                                    |
| Real, realtime variables             | <code>&lt;var_name&gt; = %g</code>                                                                                                                                                                                                                     |
| Sampled value function calls         | <code>&lt;\$sampled_function_call(argument list)&gt; = %b</code><br><b>Note:</b> The values of the variables in the argument list are not reported. Also, if sampled value functions are nested, only the value of the top-level function is reported. |
| Unknown Expressions                  | <code>&lt;expr&gt; = %h</code>                                                                                                                                                                                                                         |
| User-defined function calls          | <code>&lt;function_name&gt;(list of var_name - value pairs)</code>                                                                                                                                                                                     |
| Part select and bit select items     | VCS reports with the sampled value of the selector, if it is a variable.                                                                                                                                                                               |

---

## Limitations

The reporting of failing assertion variables is not supported for the following:

- When an assertion is used in the action block of another assertion.
- For other directives, such as `assume`, `restrict`. This feature is supported only for `assert` directive.
- Sequence method
- PSL
- Randomize with

- When the type is one of the following:
  - Parameter/constant
  - Local variable
  - Clocking block variables
  - Dynamic types including class variables

## **Reporting Messages When \$uniq\_prior\_checkon/ \$uniq\_prior\_checkoff System Tasks are Called**

VCS reports messages when the `$uniq_prior_checkon`/  
`$uniq_prior_checkoff` system tasks are called in the source  
code or from UCLI. This feature allows you to control assertion  
failures.

Consider [Example 19-3](#),

### *Example 19-3 test.v*

```
module m;
  bit [1:0]a;
  bit b,c;
  initial begin
    repeat (8)
    begin
      #1;
      c=1'b1;
      a=2'b10;
      #1;
      a=2'b11;
      $monitor($time, "a %0d \n",a);
    end
  end
  always_comb
```

```

unique case(a)
  2'b10: b=1'b0;
  2'b10: b=1'b1;
  default: b=1'b0;
endcase

initial
begin
  #2 $uniq_prior_checkon();
  #8 $uniq_prior_checkoff();
end
endmodule

```

The following output is generated when the **\$uniq\_prior\_checkon/\$uniq\_prior\_checkoff system tasks are called in the source code:**

```

Starting Unique/Priority checks at time 2s : Level = 0 arg
= * (Source - test.v,23)
Stopping Unique/Priority checks at time 10s : Level = 0 arg
= * (Source - test.v,24)

```

#### *Example 19-4 test.ucli*

```

run 5
call {$uniq_prior_checkoff}
run 2
call {$uniq_prior_checkon}
run
exit

```

Consider [Example 19-3](#) and [Example 19-4](#). The following output is generated when the **\$uniq\_prior\_checkon/\$uniq\_prior\_checkoff system tasks are called from UCLI:**

```

Stopping Unique/Priority checks at time 5s : Level = 0 arg
= * (from inst m (UCLI))
Starting Unique/Priority checks at time 7s : Level = 0 arg
= * (from inst m (UCLI))

```

---

## Assertion and Unique/Priority Re-Trigger Feature

When `$uniq_prior_checkon` system task is called, unique/priority case reports the previous violations. Also, it flushes out all the pending violations recording in the `always_comb` block during the off period. The re-trigger feature reports previous violations only and does not re-execute any process.

Consider [Example 19-5](#),

*Example 19-5 example.v*

```
module Top;
    function foo (input a);
        U1: unique case (a)
            1'b1:;
        endcase
        assert #0 (a);
    endfunction
    reg a;
    always_comb begin:AC
        $display($time, "AC is called");
        foo(a);
    end
    always@(*) begin:AS
        $display($time, "AS is called");
        foo(a);
    end
    initial begin
        a = 1'b0;
    end
    initial begin
        $uniq_prior_checkoff(0, Top); $asserton(0,Top);
        #1 $uniq_prior_checkon(0, Top); $asserton(0,Top);
    end
endmodule
```

In this example, `foo` function is called in `always_comb` and `always @` blocks. With this new feature, when `$uniq_prior_check` system task is turned on at time 1 ns, then unique or priority messages are triggered from all process blocks.

Note:

Other statements inside the `always` block are not executed.

You can compile the example using the following command:

```
% vcs -sverilog -assert svaext example.v  
-xlrn uniq_prior_final
```

```
% simv
```

VCS generates the following output:

```
Stopping Unique/Priority checks at time 0s : Level = 0 arg  
= Top (Source - example.v,21)  
Starting assertion attempts at time 0s: level = 0 arg = Top  
(from inst Top (example.v:22))  
          OAS is called  
          OAC is called  
          OAC is called  
"example.v", 6: Top.foo.unnamed: started at 0s failed at 0s  
          Offending 'a'  
#0 in foo.unnamed           at Example.v:6  
#1 in Top                   at Example.v:15  
  
"example.v", 6: Top.foo.unnamed: started at 0s failed at 0s  
          Offending 'a'  
#0 in foo.unnamed           at example.v:6  
#1 in Top                   at example.v:11  
  
Starting Unique/Priority checks at time 1s : Level = 0 arg  
= Top (Source - example.v,23)  
Starting assertion attempts at time 1s: level = 0 arg = Top  
(from inst Top (example.v:24))
```

```
Warning-[RT-NCMUCS] No condition matches in statement
example.v, 3
No condition matches in 'unique case' statement. 'default'
specification is missing, inside Top.foo.U1, at time 1s.

#0 in foo.U1      at Example.v:3
#1 in Top         at Example.v:15
```

```
Warning-[RT-NCMUCS] No condition matches in statement
example.v, 3
No condition matches in 'unique case' statement. 'default'
specification is missing, inside Top.foo.U1, at time 1s.

#0 in foo.U1      at example.v:3
#1 in Top         at example.v:11
```

---

## Flushing Off the Assertion Re-Trigger Feature

To flush off all these assertion re-trigger feature, you can use the `-assert disable_flush` compile-time or runtime option.

For the example provided above, when you pass the `-assert disable_flush` option at runtime, VCS flushes off all the assertions that are re-triggered.

For example, you can compile and simulate the example using the following commands:

```
% vcs -sverilog example.v -xlrn uniq_prior_final
% simv -assert disable_flush
```

VCS generates the following output:

```
Stopping Unique/Priority checks at time 0s : Level = 0 arg
= Top (Source - example.v,20)
```

```

OAS is called
OAC is called
OAC is called
Starting Unique/Priority checks at time 1s : Level = 0 arg
= Top (Source - example.v,21)

```

---

## Enabling Lint Messages for Assertions

You can use the `+lint=sva` option at compile time to enable lint messages for SystemVerilog Assertions with the rules listed in [Table 19-2](#). You can also use the `+lint=<ID>` or `+lint=all` compile-time option to enable this feature.

Note:

For SVA-LDA, SVA-NCRT, and SVA-PWLNT lint IDs, you can use the `+lint=<ID>:<N>:<M>` option to control the display of the lint messages. Here, `N` denotes the number of times the lint message shall appear and `M` denotes the threshold limit to be set. For the remaining lint IDs specified in the [Table 19-2](#), you can use the `+lint=<ID>:<N>` option to control the display of messages.

*Table 19-2 List of New LINT IDs*

| Assertion Rule Description                                            | Lint Message                                               |
|-----------------------------------------------------------------------|------------------------------------------------------------|
| Assertion with large delays                                           | Lint-[SVA-LDA] Large delay assertion                       |
| Consequent contains throughout operator on non-consecutive repetition | Lint-[SVA-NCRT] Non-consecutive repetition in 'throughout' |
| Assertions using \$past with >5 clock cycles                          | Lint-[PWLNT] PAST with large number of ticks               |
| Non-singular edge found in assertion clock                            | Lint-[SVA-NSEF] Non-singular edge found                    |
| Complex clock expression is used with an assertion                    | Lint-[SVA-CE] Complex expression found                     |

| <b>Assertion Rule Description</b>                                     | <b>Lint Message</b>                                      |
|-----------------------------------------------------------------------|----------------------------------------------------------|
| Non-sampled variable used in the action block of an assertion         | Lint-[SVA-NSVU] Non-sampled variable used                |
| Assertions using local variables                                      | Lint-[SVA-LVU] Local variable used                       |
| Assertion declared outside a module                                   | Lint-[SVA-ADOM] Assertion declared outside module        |
| Disable iff expression used inside an assertion                       | Lint-[SVA-DIU] 'disable iff' used in assertion statement |
| Assertions used inside generate "for" loop                            | Lint-[SVA-AGFL] Assertions in generate for loop          |
| Unnamed assertion                                                     | Lint-[SVA-UA] Unnamed Assertion                          |
| Cover on a sequence                                                   | Lint-[SVA-SCU] Sequences inside 'Cover' statement        |
| Deferred assertion                                                    | Lint-[SVA-DAU] Deferred assertion used                   |
| Assertions using antecedent expression that results in an empty match | Lint-[SVA-AEM] Antecedent empty match                    |
| Consequent expression of an assert property is always true            | Lint-[SVA-CAT] Consequent always true                    |
| Implication in a cover property                                       | Lint-[SVA-ICP] Implication in cover property             |
| Using (clk iff gate_expr) inside an assert property                   | Lint-[SVA-LCE] Logical 'AND' in clock expression         |

| <b>Assertion Rule Description</b>                       | <b>Lint Message</b>                                   |
|---------------------------------------------------------|-------------------------------------------------------|
| Pass action block in assert property                    | Lint-[SVA-PAB] Pass action block                      |
| \$info/\$display statement in a cover property          | Lint-[SVA-IUC] Info messages used in cover            |
| Assertion with empty “begin -- end” action block        | Lint-[SVA-EFAB] Empty fail action block               |
| Assertion in a loop without using the loop index        | Lint-[SVA-FINUA] For-loop index not used in assertion |
| Assertion with a severity task in the pass action block | Lint-[SVA-STPAB] Severity task in pass action block   |

---

## Fail-Only Assertion Evaluation Mode

Fail-only is an assertion evaluation mode by which VCS provides an optional optimization controlled by the `-assert failonly` compile-time option. This option enables fail-only mode for concurrent assertions.

Immediate/deferred assertions and concurrent assertions without pass action blocks, local variables, match operators, or multiple clocks tend to benefit from this evaluation mode.

Note:

VCS ignores the fail-only assertion evaluation mode, if you use any of the following options:

`-assert enable_diag, -debug_access, -cm assert`

By default, VCS reports the start and end times of assertion evaluation attempts. Therefore, each attempt needs to be stored in memory until it matures. This can cause slowdown if there are multiple attempts pending till the simulation is finished.

If you use the `-assert failonly` option at compile time, VCS reports only assertion failures with their end times. This reduces the memory footprint and speeds up the simulation.

Consider [Example 19-6](#).

*Example 19-6 test.v*

```
`timescale 1ns/1ns
module top();
    reg a,b,c;
    reg clk = 0;
```

```

A1: assert property( @(posedge clk) a ##1 !c[*1:$] |-> b);

always
  #1 clk = ~clk;
initial begin
  a = 1;
  b = 0; c = 0;

  #100 b = 1;
  #2 $finish;
end
endmodule

```

The following output is generated with the `-assert failonly` option:

```
"test.v", 6: top.A1: failed at 101ns
```

In the above output, VCS reports only end times.

The following output is generated without `-assert failonly` (default mode) option:

```

"test.v", 6: top.A1: started at 99ns failed at 101ns
  Offending 'c'
"test.v", 6: top.A1: started at 97ns failed at 101ns
  Offending 'c'
.....
"test.v", 6: top.A1: started at 1ns failed at 101ns
  Offending 'c'

```

In the default mode, VCS reports start and end times for each failing attempt.

## Key Points to Note

- VCS reports only assertion failures with their end times.

- VCS reports only a single failure for multiple failures of a single assertion maturing at the same time.
- Behavior with coverage:
  - Reports unique matches only
  - The `-cm assert` option is not supported with the `-assert failonly` option.

Consider [Example 19-7](#).

### *Example 19-7 cov.v*

```
module top;
  reg a,b,c;
  reg clk=0;
  C2: cover property( @(posedge clk) b[=1:$] ##1 c);

  initial begin
    a=1;b=0;c=1;

    #25 b=1;
    #110 $finish;
  end
  always #5 clk=~clk;
endmodule
```

The following output is generated with the `-assert failonly` option:

```
"cov.v", 7: top.C2, 9 match
[4 succeeds at same time 45] hence counted only once
hence count is 9 vs 12 in default mode
```

The following output is generated without the `-assert failonly` (default mode) option:

```
"cov.v", 7: top.C2, 13 attempts, 12 match
```

---

## Limitations

The feature has the following limitations:

- Reporting of offending expressions upon failure is not supported.
- Success reporting (including vacuous success) is not supported.
- VPI callback on success (including vacuous success) is not supported.
- Attempt start time reporting is not supported.

---

## Treating x as true on an Assertion Precondition

VCS treats x values as FALSE while evaluating expressions. Therefore, it is not possible to see an assertion failure when the antecedent sequence expression is not satisfied. For example, consider the following assertion:

```
known_a : assert property (@(posedge clk) value_p  
| -> value_o);
```

In the above expression, `value_o` (consequent property expression) is evaluated only when `value_p` (antecedent sequence expression) is 1. VCS does not evaluate consequent property expression when `value_p` is x. This behavior is in compliance with *IEEE System Verilog LRM 1800-2012*.

However, VCS enables the feature to treat x as TRUE while evaluating the antecedent of assertions using the `-assert allow_antec_x` option. Therefore, for the following assertion:

```
known_a : assert property (@(posedge clk) value_p  
| -> value_o);
```

When `-assert allow_antec_x` option is used, VCS evaluates consequent property expression when `value_p` is x. Therefore, `value_p` (antecedent sequence expression) is considered TRUE and `value_o` (consequent property expression) is evaluated.

## Use Model

To enable this feature, use the `-assert allow_antec_x` option at compile time. Following is the use model:

```
% vcs <filename> -sverilog -assert allow_antec_x
```

## Usage Example

The following example illustrates a case where evaluation of antecedent of assertion is considered TRUE.

*Example 19-8 test.v*

```
module top;

bit clk;

logic a = 1'bx;
logic b = 1'bx;
logic c = 1'bx;

always clk = #2 ~clk;

always@(*)
  c = a & b;

initial begin
  a = 1'b0;    b = 1'b0;
#2 a = 1'b1; b = 1'b0; $display($time,", a = %h, b = %h, c = %h", a,b,c);
#2 a = 1'bx; b = 1'b1; $display($time,", a = %h, b = %h, c = %h", a,b,c);
#2 a = 1'b1; b = 1'b1; $display($time,", a = %h, b = %h, c = %h", a,b,c);
#2 a = 1'bx; b = 1'b0; $display($time,", a = %h, b = %h, c = %h", a,b,c);
#2 a = 1'bx; b = 1'b1; $display($time,", a = %h, b = %h, c = %h", a,b,c);
#2 a = 1'bx; b = 1'bx; $display($time,", a = %h, b = %h, c = %h", a,b,c);
#2 a = 1'b0; b = 1'b1; $display($time,", a = %h, b = %h, c = %h", a,b,c);
#20 $finish;
end

A1 : assert property (@(edge clk) (a & b) |-> c);

endmodule
```

The compile command is as follows:

```
% vcs test.v -sverilog -assert allow_antec_x
```

Following is the simulation command:

```
% simv
```

The output is as follows:

```
2, a = 1, b = 0, c = 0
4, a = x, b = 1, c = 0
6, a = 1, b = 1, c = x
"test.v", 26: top.A1: started at 6s failed at 6s
Offending 'c'
8, a = x, b = 0, c = 1
10, a = x, b = 1, c = 0
12, a = x, b = x, c = x
"test.v", 26: top.A1: started at 12s failed at 12s
Offending 'c'
14, a = 0, b = 1, c = x
"test.v", 26: top.A1: started at 14s failed at 14s
Offending 'c'
```

---

## Using SystemVerilog Constructs Inside vunits

VCS supports using SystemVerilog and SystemVerilog Assertions inside a Property Specification Language (PSL) verification unit (vunit). This feature enables the following:

- Allows SV or SVA code inside a vunit.
- Allows you to easily bind the checkers containing assertions and model a vunit code to a design.
- Allows module instantiation inside a vunit.

Use the `-assert svvunit` compile-time option, as shown in the following example to enable this feature. You can specify this option with the `vcs` or `vlogan` command as follows:

```
% vcs -assert svvunit <filename.v> <design_filename.v> \
<vunit_filename.psl>
```

or

```
% vlogan -assert svvunit <filename.v> <design_filename.v> \  
<vunit_filename.psl>
```

where,

*<vunit\_filename.psl>* is the PSL vunit that contains the SV or the SVA code. For example:

```
% vcs -assert svvunit test.v design.v vunit_checker.psl
```

---

## Limitations

The feature has the following limitations:

- Inheritance of vunits is not supported.
- vunit binding is supported only for modules.
- SV checker (checker/endchecker construct) instantiation in vunit is not supported.
- VHDL entity instantiation in vunit is not supported.
- In the above use model, you cannot specify PSL constructs in any vunit specified in the same vlogan command. You must separate the PSL vunits from SV vunits and use them in two separate compilations, as shown in the following examples:

```
% vlogan vunit_psl.psl design.v  
% vlogan -assert svvunit vunit_sv.psl test.v
```

where,

- `vunit_psl.psl` is a vunit that contains PSL code
- `vunit_sv.psl` is a vunit that contains SV or SVA code.

## Calling \$error Task When Else Block is Not Present

You can use the following runtime option to enable the calling of the `$error` task for the assert statements as per the IEEE 1800-2012 SystemVerilog LRM:

```
-assert error_default_action_block
```

If you use this option, VCS calls the `$error` task if an assert statement fails and else clause is not specified. This feature is supported for immediate, deferred, and concurrent assertions.

Consider the following test case (`test.v`):

```
module assertit;
    logic clk = 0; initial repeat (10) #1 clk++;
    logic [2:0] a = 0; always @(posedge clk) a <= a + 1;
    a1: assert property (@(posedge clk) !a[2]);
    a2: assert property (@(posedge clk) !a[2]) else
        $error("failed");
endmodule
```

The output for both asserts (a1 and a2) are identical for the preceding test case as follows:

```
"test.v", 6: assertit.a1: started at 9s failed at 9s
          Offending '(!a[2])'
Error: "test.v", 6: assertit.a1: at time 9
"test.v", 7: assertit.a2: started at 9s failed at 9s
          Offending '(!a[2])'
Error: "test.v", 7: assertit.a2: at time 9
failed
```

---

## **Disabling Default Assertion Success Dumping in -debug\_access Option**

By default, VCS does not dump the SystemVerilog assertion successes to the VPD file when the `-debug_access` option is enabled. This optimization can improve the VCS performance.

You can use the `-assert dumpsuccess` option at runtime to enable dumping of the SystemVerilog assertion successes to the VPD file.

---

## **List of supported IEEE Std. 1800-2012 Compliant SVA Features**

The following features are supported in compliance with the IEEE SystemVerilog LRM Std 1800<sup>TM</sup>-2012. For more details, see IEEE 1800<sup>TM</sup>-2012 Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language Manual.

- Overlapping operators in multiclock environment
- Immediate and Deferred Assertions
- `weak` and `strong` sequence operators

Note:

For limitations, see the “[Limitations](#)” section.

- Implication and equivalence operators (`->` and `<->`)
- `until` operator in four variants:

- until
- s\_until
- until\_with
- s\_until\_with
- nexttime operator in four variants:
  - nexttime property\_expr
  - nexttime [N] property\_expr
  - s\_nexttime property\_expr
  - s\_nexttime [N] property\_expr
- always operator in three variants:
  - always property\_expr
  - always [cycle\_delay\_const\_range\_expression] property\_expr
  - s\_always [constant\_range] property\_expr strong
- eventually operator in three variants
  - eventually [constant\_range] property\_expr
  - s\_eventually property\_expr
  - s\_eventually
- followed-by operator (#-#, #=#)
- accept\_on and reject\_on abort conditions
- Inferred value functions:

- `$inferred_clock`
- `$inferred_disable`
- `$inferred_enable`

**Note:**

`$inferred_enable` is a VCS extension to the Inferred functions and not a standard LRM feature.

- Local variable initialization and input
- Global clocking
- Global clocking past value functions:
  - `$past_gclk(expression)`
  - `$rose_gclk(expression)`
  - `$fell_gclk(expression)`
  - `$stable_gclk(expression)`
  - `$changed_gclk(expression)`
- Global clocking future value functions:
  - `$future_gclk(expression)`
  - `$rising_gclk(expression)`
  - `$falling_gclk(expression)`
  - `$steady_gclk(expression)`
  - `$changing_gclk(expression)`
- `let` construct
- Checker endchecker construct:

- Checker declarations
- Checker instantiation
- Procedural and Static checkers
- Default clocking in checkers
- Default disable in checkers
- Checker instantiation in a procedural for loop
- Random variable support in checker

**Note:**

To enable the `checker` `endchecker` construct, you must use the `-assert checker` option at compile time.

- Immediate and deferred assertions
- edge operator
- Bit/part/field select support in the let operator
- Elaboration system tasks
- VPI support

## Support for `$countbits` System Function

VCS supports the IEEE 1800-2012 SystemVerilog LRM system function `$countbits`. The following `$countbits` function counts the number of bits that have a specific set of values (for example, 0, 1, X, Z) in a bit vector:

```
$countbits (expression, control_bit { , control_bit })
```

This function returns an integer value equal to the number of bits in expression whose values match one of the `control_bit` entries. For example:

- `$countbits (expression, '1)` returns the number of bits in expression having value 1.
- `$countbits (expression, '1, '0)` returns the number of bits in expression having values 1 or 0.
- `$countbits (expression, 'x, 'z)` returns the number of bits in expression having values X or Z.

This support allows efficient implementation of basic non-temporal assertions in the presence of unknown values.

---

## Support for Real Data Type Variables

VCS supports the following as per the IEEE 1800-2012 SystemVerilog LRM:

- Real data type variables in the sub-expressions of a concurrent assertion.
- Sampling of real data type variables similar to the sampling of any integral variable.

---

## Support for \$assertcontrol Assertion Control System Task

VCS supports the IEEE 1800-2012 SystemVerilog LRM assertion control system task `$assertcontrol`. The `$assertcontrol` system task provides the capability to enable, disable, or kill the assertions based on the assertion type or directive type. Similarly,

this task also provides the capability to enable or disable action block execution of assertions and expect statements based on the assertion type or directive type.

### Syntax:

```
$assertcontrol ( control_type [ , [ assertion_type ] [ , [ directive_type ] [ , [ levels ] [ , list_of_scopes_or_assertions ] ] ] ] ) ;
```

The `$assertcontrol` system task provides finer assertion selection controls than `$asserton`, `$assertoff`, and `$assertkill` system tasks.

## Limitations

The feature has the following limitations:

- The `Unique0` assertion type is not supported.
- The assertion type 16 (Expect statement) is not supported.
- The control type values from 6 to 11 (`PassOn`, `PassOff`, `FailOn`, `FailOff`, `NonvacuousOn`, `VacuousOff`) are not supported.
- The control type `Kill` is not supported for Unique and Priority assertion types.

---

## Enabling IEEE Std. 1800-2012 Compliant Features

You must use the `-assert svaext` compile-time option to enable the IEEE Std. 1800-2012 compliant SVA features.

## **Limitations**

The feature has the following limitations:

- In VCS, strong and weak properties are not distinguished in terms of their reporting at the end of simulation. In all cases, if a property evaluation attempt did not complete evaluation, it is reported as unfinished evaluation attempt, and allows you to decide whether it is a failure or a success.
- Checker declaration are allowed in unit scope only.
- Bind construct with checkers is not supported.

The limitations on debug support are as follows:

- UCLI support for new assertions is not supported.

---

## **SystemVerilog Assertions Limitations**

This section describes the limitations that apply to SystemVerilog assertions.

### **Debug Support for New Constructs**

UCLI support for new assertions (LTL operators, checker block) supported under `-assert svaext` is not fully qualified.

## Note on Cross Features

Some of the new features in assertions (LTL operators, checker block) under `-assert svaext` have known limitations with cross-feature support, such as Debug and Coverage. Check with the Synopsys support for any unexpected results with cross-feature behavior for these new constructs.

Some known issues:

- `-cm property_path` is not available for the new constructs.
- New sequence operators when used as sampling event for covergroups may not function well.

# 20

## Using Property Specification Language

---

VCS supports the Simple Subset of the IEEE 1850 Property Specification Language (PSL) standard. Refer to Section 4.4.4 of the *IEEE 1850 PSL LRM* for the subset definition.

You can use PSL in Verilog, VHDL, or mixed designs along with SystemVerilog Assertions (SVA), SVA options, SVA system tasks, and OpenVera (OV) classes.

---

### Including PSL in the Design

You can include PSL in your design in any of the following ways:

- Inlining the PSL using the `//psl` or `/*psl */` pragmas in Verilog and SystemVerilog, and `--psl` pragma in VHDL.

- Specifying the PSL in an external file using a verification unit (vunit).
- 

## Examples

The following examples shows how to inline PSL in Verilog using the `//psl` and `/*psl */` pragmas, and in VHDL using the `--psl` pragma.

### In Verilog

```
module mod;
  ...
  // psl a1: assert always {r1; r2; r3} @ (posedge clk);

  /* psl
     A2: assert always {a;b} @ (posedge clk);
     ...
   */
endmodule
```

### In VHDL

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity vh_ent is
  ...
end vh_ent;

architecture arch_vh_ent of vh_ent is
  ...
  -- psl default clock is (clk'event and clk = '1');
  -- psl sequence seq1 is {in1; [*2]; test_sig};

  -- psl property p1 is
  -- (never seq1);
```

```
-- psl A1: assert p1 report " : Assertion failed P1";
end arch_vh_ent;
```

The following examples show how to use `vunit` to include PSL in the design.

### In Verilog

```
vunit vunit1 (verilog_mod)
{
    a1: assert always {r1; r2; r3} @ (posedge clk);
}
```

### In VHDL

```
vunit test(vh_entity)
{
    default clock is (clk'event and clk = '1');

    property foo is
        always ({a = '0'}) |=> {(b = prev(b) and c = prev(c))};
        assume foo;
}
```

---

## Use Model

If you inline the PSL code, you must compile/analyze it with the `-psl` option.

If you use `vunit`, you must compile/analyze the file that contains the `vunit` with the `-pslfile` option. You do not need to use this option if the file has the `.psl` extension.

### Compilation

```
% vcs -psl [vcs_options] Verilog_files
```

## **Simulation**

```
% simv
```

---

## **Examples**

To simulate the PSL code inlined in a Verilog file (`test.v`), execute the following commands:

```
% vcs -psl test.v  
% simv
```

To simulate the `vunit` specified in an external file with the `.psl` extension (`checker.psl`), execute the following commands:

```
% vcs dev.v checker.psl  
% simv
```

To simulate the `vunit` specified in an external file without the `.psl` extension (`checker.txt`), execute the following commands:

```
% vcs dev.v -pslfile checker.txt  
% simv
```

To simulate both the PSL code inlined in a Verilog file (`test.v`), and the `vunit` specified in an external file (`checker.psl` or `checker.txt`), execute the following commands:

```
% vcs -psl -test.v checker.psl  
% simv
```

or

```
% vcs -psl -test.v -pslfile checker.txt  
% simv
```

## **Analysis**

```
% vlogan -psl [vlogan_options] Verilog_files
% vhdlan -psl [vhdlan_options] VHDL_files
```

### **Note:**

Specify the VHDL bottommost entity first, then move up in order.

## **Elaboration**

```
% vcs -psl top_cfg/entity/config
```

### **Note:**

Ensure that you specify the `-psl` option while elaborating the design.

## **Simulation**

```
% simv
```

---

## **Examples**

To simulate the PSL code that is inlined in a mixed design (`test.v` and `dut.vhd`), execute the following commands:

```
% vlogan -psl test.v
% vhdlan -psl dut.vhd
% vcs -psl top
% simv
```

To simulate both the PSL code inlined in a VHDL file (`test.vhd`), and the `vunit` specified in an external file (`checker.psl` or `checker.txt`), execute the following commands:

```
% vhdlan -psl test.vhd checker.psl
% vcs -psl top
% simv
```

or

```
% vhdlan -psl test.vhd -pslfile checker.txt  
% vcs -psl top  
% simv
```

---

## PSL Assertions Inside VHDL Block Statements in Vunit

This section describes support for Property Specification Language (PSL) assertions inside VHDL block statements in a `vunit`.

This section contains the following topics:

- “[Introduction](#)” on page 6
- “[Use Model](#)” on page 7
- “[Limitations](#)” on page 8

---

### Introduction

VCS supports the usage of PSL assertions inside VHDL block statements in `vunit`. This feature extends the capability of VHDL block statements in a `vunit` by allowing PSL assertions inside VHDL block statements.

### Example

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;  
use ieee.numeric_std.all  
USE ieee.std_logic_unsigned.all
```

```

vunit ins2 (half_dsp_block(rtl)) {

    default clock is clock'event and clock = '0';

    signal sig1: integer;
    sequence S1 is {a0(3 downto 0) = "1000" [->1]}
    @ (clock'event and clock = '0');

    blk1: block is
    begin
        property P1 is always {b2 ; S1} @ (clock'event
and clock = '1');

        A1: assert always P1 @ (clock'event and clock = '1');
            sig1 <= conv_integer(a0);

    end block blk1;

    A2: assert always {sig1 = 8};
}

```

## Use Model

Use the `-assert psl_in_block` analysis option to enable the usage of PSL assertions inside VHDL block statements in vunit, as shown in the following command:

```
$> vhdlan test.vhd -psl -assert psl_in_block test.psl
```

The following PSL constructs are allowed in the VHDL block statements. You must specify these constructs in `block_statement_part`, and not in `block_declarative_part`.

- Property and sequence declarations
- Assert statement
- Assume statement

- Restrict statement
  - Cover statement
- 

## Limitations

The following are the limitations of using PSL assertions inside VHDL block statements in a vunit:

- This enhancement does not support the following non-PSL VHDL constructs, which are currently not allowed inside a block statement in a vunit.
  - Guard expressions
  - Port declarations and port maps
- The following PSL constructs are not supported in the VHDL block statements:
  - Generic declarations and generic maps
  - Default clocks
  - Vunit inheritance

---

## PSL Macro Support in VHDL

VCS supports the `%if` and `%for` PSL macros in VHDL as described in IEEE-1850-2010 PSL. You can use these macros to conditionally or iteratively generate PSL statements. This removes the need for to rewrite the entire PSL code (a time-consuming task). The following sections explain how to use these constructs:

- “Using the `%for` Construct” on page 9
  - “Using the `%if` Construct” on page 12
  - “Using Expressions with `%if` and `%for` Constructs” on page 13
  - “PSL Macro Support Limitations” on page 14
- 

### Using the `%for` Construct

The `%for` construct replicates a piece of code.

The syntax for `%for` iteration range is:

```
%for /var/ in /expr1/ ... /expr2/ do  
...  
%end
```

The syntax for `%for` iteration list is:

```
%for /var/ in { /item/ [, /item/] * } do  
...  
%end
```

Following are the arguments:

- `var` — Variable name

- `expr` — Expression on which macro substitution is performed. This argument should be a numeric decimal value.
- `item` — Value to be substituted for instances of the variable name on each iteration of the `%for` macro. This value can only contain alphanumeric characters ('a' to 'z', 'A' to 'Z', and '0' to '9') and underscores.

If an item contains only digits, it is treated as a number during expression evaluation.

Bareword macro substitution is not done on items in the `%for` iteration list. However, `%{ } style` macro substitutions are done on these items. This provides the flexibility to control the strings in the list. For example, consider the following code:

```
%for xx in { aa, bb, cc } do
%for yy in { xx, %{xx}, zz } do
...
```

The loop iterator `yy` takes the following values:

- `xx, aa, zz` in the first iteration of loop `xx`
- `xx, bb, zz` in the second iteration of loop `xx`
- `xx, cc, zz` in the third iteration of loop `xx`

When a macro substitution of a list item iterator occurs, it is only done on one level of substitution. That is, if the list item value itself is a name that matches the name of a macro iterator, then the value of that iterator is not substituted. The value substituted is the string defined in the item list. Consider the code in [Example 20-1](#).

### *Example 20-1 Macro Substitution*

```
%for xx in 1...2 do
  %for yy in { xx, zz } do
```

```
Lbl_{yy}_{xx} : assert ...
%end
%end
```

In [Example 20-1](#), when the `yy` iterator value is substituted, the resulting value is `xx`, and not the current value of the `xx` iterator (1 or 2):

```
Lbl_xx_1: assert ...
Lbl_zz_1: assert ...
Lbl_xx_2: assert ...
Lbl_zz_2: assert ...
```

The `%{ } macro substitution` within a quote ("") delimited string is supported. Bareword string substitution is not allowed within a quoted string. For example, the following code:

```
%for xx in 1 ... 2 do
    report "xx = %{xx}";
%end
```

Expands to:

```
Report "xx = 1";
Report "xx = 2";
```

You can use the `%` character as a string delimiter. No macro substitution is performed within `%` delimited strings.

For macro expansion, any occurrence of macro keywords that include the `%` character (`%for`, `%if`, `%then`, `%else`, `%end`, and the `%{ ... }` substitution macro) takes priority over string initiation. For example:

```
report %xx = %{xx}%;
```

The above example results in a syntax error at { and an unterminated string (starting delimiter is the last % on the line).

---

## Using the %if Construct

The `expr` argument of the `%if` macro must evaluate to an integer.

- If the expression resolves to an integer other than zero, then the expression is true and the `%then` clause is processed.
- If the expression resolves to zero, then the expression is false and the `%else` statement, if present, is processed.

The syntax for `%if` is as follows:

```
%if /expr/ %then  
...  
%end
```

or

```
%if /expr/ %then  
...  
%else  
  
%end
```

---

## Using Expressions with %if and %for Constructs

You can use the following in the expressions with `%if` and `%for` constructs:

- Decimal literals
- Alphanumeric strings
- Operators:  
`=, -, *, /, %, =, !=, <, <=, >, >=`
- Parentheses:  
`( ` ( ` and ` ) ' )`

All arithmetic operations are integral.

VCS generates an error message if:

- An operand of an arithmetic operation is non-numeric
- Either operand evaluates to a non-alphanumeric string

Comparison operations are integral if both operands are integral. If either operand is alphanumeric (after substitution), then lexical comparison is performed.

For example, consider the following expression:

```
%if (foo(1) == 0)
```

This expression is an error, because the left operand of the equality does not evaluate to an alphanumeric string.

---

## PSL Macro Support Limitations

- The `%for` and `%if` macros are not supported in inline PSL pragmas
- The `%{ }` macro substitution cannot span lines. However, `%for` and `%if` header constructs can span lines
- The `%for` and `%if` macros are not supported within encrypted blocks. Macro text can contain encrypted blocks
- VHDL-style extended identifiers are not supported as `%for` macro iterator names
- The `%{ }` style replacement macros within other `%{ }` style replacement macros are not allowed

**Example:** `%{ ii + %{jj + 1} }`

- Octal and hexadecimal literal numeric values are not supported
- A nested `%for` iterator name cannot use the same name as an outer `%for` macro's iterator name. For example:

```
%for xx in 1...2 do  
    %for xx in 3...4 do  
    ...
```

- C preprocessor directives (for example, `#define`, `#ifdef`, `#else`, `#include`, and `#undef`) are not supported.

---

## Using SVA Options, SVA System Tasks, and OV Classes

VCS enables you to use all assertion options with SVA, PSL, and OVA. For example, to enable PSL coverage and debug assertions while elaborating/compiling the PSL code, execute the following commands:

For two-step flow:

```
% vcs -psl -cm assert -debug_access -assert enable_diag  
test.v  
% simv -cm assert -assert success
```

For three-step flow:

```
% vhdlan -psl dut.vhd checkers.psl  
% vhdlan test.vhd  
% vcs top -psl -cm assert -debug_access -assert enable_diag  
% simv -cm assert -assert success
```

For information on all assertion options, see Appendix - Compilation/Elaboration Options.

You can control PSL assertions in any of the following ways:

- Using the \$asserton, \$assertoff, or \$assertkill SVA system tasks.
- Using NTB-OpenVera assert classes.

Note that VCS treats the assume PSL directive as the assert PSL directive.

---

## Limitations

The VCS implementation of PSL has the following limitations:

- VCS does not support binding `vunit` to an instance of a module or entity
- VCS does not support generic declarations and generic maps in VHDL block statements in a `vunit`
- VCS does not support the following data types in your PSL code -- `shortreal`, `real`, `realtime`, associative arrays, and dynamic arrays
- VCS does not support the `union` operator and union expressions in your PSL code
- Clock expressions have the following limitations:
  - You must not include the `rose()` and `fell()` built-in functions
  - You must not include endpoint instances
- Endpoint declarations must have a clocked SERE with either a clock expression or default clock declaration
- VCS does not support the `%for` and `%if` macros
- VCS supports only the `always` and `never` FL invariance operators in top-level properties. Ensure that you do not instantiate top-level properties in other properties
- VCS supports all LTL operators, except `sync_abort` and `async_abort`. You can apply the abort operator only to the top property
- VCS does not support the `assume_guarantee`, `restrict`, and `restrict_guarantee` PSL directives

# 21

## Using SystemC

---

The VCS SystemC Co-simulation Interface enables VCS and the SystemC modeling environment to work together when simulating a system coded in the Verilog, VHDL, and SystemC languages.

With this interface, you can use the most appropriate modeling language for each part of the system and verify the correctness of the design. For example, the VCS SystemC Co-simulation Interface allows you to:

- Use a SystemC module as a reference model for the VHDL or Verilog RTL design under test in your testbench
- Verify a Verilog or VHDL netlist after synthesis with the original SystemC testbench
- Write test benches in SystemC to check the correctness of Verilog and VHDL designs
- Import legacy VHDL or Verilog IP into a SystemC description

- Import third-party VHDL or Verilog IP into a SystemC description
- Export SystemC IP into a Verilog or VHDL environment when only a few of the design blocks are implemented in SystemC
- Use SystemC to provide stimulus to your design

The VCS/SystemC Co-simulation Interface creates the necessary infrastructure to co-simulate SystemC models with Verilog or VHDL 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, it is not required to update or write to these files.

During co-simulation, the VCS/SystemC Co-simulation Interface is responsible for:

- Synchronizing the SystemC kernel and VCS
- Exchanging data between the two environments

Note:

- The unified profiler can report CPU time profile information about the SystemC part or parts of a design. For more information, see chapter "*The Unified Simulation Profiler*" in the *VCS User Guide*.
- Examples of Verilog/VHDL instantiated in SystemC and SystemC instantiated in Verilog/VHDL are included in the `$VCS_HOME/doc/examples/systemc` directory.

For more information about SystemC co-simulation interface and how you can use SystemC with VCS for your design, see the **VCS SystemC User Guide** available in the SolvNet online support site.

# 22

## C Language Interface

---

Generally, C and C++ code is added with both Verilog and VHDL. There are many different mechanisms to add C and C++ code and how you add these code designs depends on your objective as well as the performance and restrictions of each mechanism. VCS supports the following ways to use C and C++ in your design:

- “Using PLI”
- “Using VPI Routines”
- “Using VHPI Routines”

VHPI enables you to use foreign architecture-based models written in C language in the VCS VHDL using DirectC.

- “Using DirectC”
- Using SystemC - See [Using SystemC](#)
- Using SystemVerilog DPI routines - See the SystemVerilog LRM.

For the description of PLI 1.0, PLI 2.0, and VHPI routines, see the *C Language Interface Reference Manual*.

Note:

PLI1.0 refers to TF and ACC routines, and PLI2.0 refers to VPI.

---

## Using PLI

PLI is the programming language interface (PLI) between C/C++ functions and VCS. It helps 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. Later during simulation VCS can call these functions.

VCS supports PLI 1.0 and PLI 2.0 routines for the PLI. Therefore, you can use VPI, ACC, or TF routines to write the PLI application. See [Appendix , "PLI Access Routines"](#).

This chapter covers the following topics:

- [“Writing a PLI Application”](#)
- [“Functions in a PLI Application”](#)
- [“Header Files for PLI Applications”](#)
- [“PLI Table File”](#)
- [“Enabling ACC Capabilities”](#)

---

## Writing a PLI Application

When writing a PLI application, you need to perform the following tasks:

1. Write the C/C++ functions of the application calling the VPI, ACC, or TF routines to access data inside VCS.
2. Associate user-defined system tasks and system functions with the C/C++ functions in your application. VCS calls these functions when it compiles or executes these system tasks or system functions in the Verilog source code. In VCS, associate the user-defined system tasks and system functions with the C/C++ functions in your application using a PLI table file (see “[PLI Table File](#)”). 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/Analyze, elaborate, 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, however, omit the source code for this function at compile time.

To use the debugging features, perform the following tasks:

1. Write a PLI table file, limiting the scope and operations of the ACC routines used by the debugging features.
2. Compile/Analyze, elaborate, and simulate your design, specifying the table file.

These procedures are not mutually exclusive. It is possible that you have a PLI application that you write and use during the debugging phase of your design. If so, you can write a PLI table file that both:

- Associates user-defined system tasks or system functions with the functions in your application and limits the scope and operations called by your functions for faster performance.
- Limits scope and operations of the functions called by the debugging features in VCS.

---

## Functions in a PLI Application

When you write a PLI application, you typically write a number of functions. The following are the 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, however, this call function must be present. It is not unusual for there to be more than one call function. You will need a separate user-defined system task for each call function. If the function returns a value, then you must write a user-defined system function for it instead of a user-defined system task.
- The function that VCS calls during compilation to check if the user-defined system task has the correct syntax. You can omit this check function.
- The function that VCS calls for miscellaneous reasons, such as the execution of \$stop, \$finish, or other reasons, such as a value change. When VCS calls this function, it passes a reason argument to it that explains why VCS is calling it. You can omit this miscellaneous function.

These are the functions you instruct VCS about in the PLI table file; apart from these PLI applications can have several more functions that are called by other functions.

**Note:**

You do not specify a function to determine the return value size of a user-defined system function; instead you specify the size directly in the PLI table file.

---

## **Header Files for PLI Applications**

For PLI applications, you need to include one or more of the following header files:

`vpi_user.h`

For PLI Applications whose functions call IEEE Standard VPI routines as documented in the *IEEE Verilog Language Reference Manual*.

`acc_user.h`

For PLI Applications whose functions call IEEE Standard ACC routines as documented in the *IEEE Verilog Language Reference Manual*.

`vcsuser.h`

For PLI applications whose functions call IEEE Standard TF routines as documented in the *IEEE Verilog Language Reference Manual*.

vcs\_acc\_user.h

For PLI applications whose functions call the special ACC routines implemented exclusively for VCS.

You can find these header files in the `$VCS_HOME/your_platform/lib` directory.

---

## PLI Table File

The PLI table file (also referred to as the `pli.tab` file) is used to:

- Associate user-defined system tasks and system functions with functions in a PLI application. This enables VCS to call these functions when it compiles or executes the system task or function.
- Limit the scope and operation of the PLI functions called by the debugging features. See “[Specifying Access Capabilities for PLI Functions](#)” and “[Specifying Access Capabilities for VCS Debugging Features](#)”.

## Syntax

The following is the syntax of the PLI table file:

`$name PLI_specifications [access_capabilities]`

Where,

`$name`

Specify the name of the user-defined system task or function.

## `PLI_specifications`

Specify one or more specifications, such as the name of the C function (mandatory), size of the return value (mandatory only for user-defined system functions), and so on. For a complete list of PLI specifications, see “[PLI Specifications](#)”.

## `access_capabilities`

Specify the access capabilities of the functions defined in the PLI application. Use this to control the PLI 1.0 or PLI 2.0 functions’ ability to access the design hierarchy. For more information, see “[Access Capabilities](#)”.

Synopsys recommends you to enable this feature while using PLIs to improve the runtime performance.

## **PLI Specifications**

The PLI specifications are as follows:

### `call=function`

Specifies the name of the function defined in the PLI application. This is mandatory.

### `check=function`

Specifies the name of the check function.

### `misc=function`

Specifies the name of the misc function.

### `data=integer`

Specifies the value passed as the first argument to the call, check, and miscellaneous functions. The default value is 0.

Use this argument if you want more than one user-defined system task or function to use the same call, check, or misc function. In such a case, specify a different integer for each user-defined system task or function that uses the same call, check, or misc function.

*size=number*

Specifies the size of the returned value in bits. While this is mandatory for user-defined system functions, you can ignore or specify 0 for user-defined system tasks. For user-defined system functions, specify a decimal value for the number of bits. For example, `size=64`. If the user-defined system function returns a real value, specify `r`. For example, `size=r`

*args=number*

Specifies the number of arguments passed to the user-defined system task or function.

*minargs=number*

Specifies the minimum number of arguments.

*maxargs=number*

Specifies the maximum number of arguments.

*nocelldefinepli*

This option is functionally equivalent to the `+nocelldefinepli+1` compile option. This option disables debugging and dumping of FSDB/VPD for objects in modules defined under the `'celldefine` compiler directive.

`persistent`

Marks the specified function as being called in the PLI application, even if the corresponding system task or function is not called in the design. If the function is not found or defined in the PLI application, VCS exits with an undefined reference error message.

Note that if you use the `-debug_access` option during elaboration, VCS marks all PLI functions as being persistent.

Setting the `PERSISTENT_FLAG` environment variable to 1 marks all PLI functions as being persistent.

### **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, such as interrupts.

### **Example 2**

```
$set_true size=16 call=set_true
```

In this line, there is an associated user-defined system function that returns a 15-bit return value. VCS calls the function named `set_true` when it executes this system function.

**Note:**

Do not enter blank space inside a PLI specification. The following example of PLI specifications does not work:

```
$set_true size = 16 call = set_true
```

## Access Capabilities

You can specify access capabilities in a PLI table file for the following reasons:

- PLI functions associated with your user-defined system task or system function. To do this, specify the access capabilities on a line in a PLI table file after the name of the user-defined system task or system function and its PLI specifications. For more details, see “[Specifying Access Capabilities for PLI Functions](#)” .
- For the debugging features VCS can use. To do this, specify access capabilities alone on a line in a PLI table file, without an associated user-defined system task or system function. For more details, see “[Specifying Access Capabilities for VCS Debugging Features](#)”.

In many ways, specifying access capabilities for your PLI functions, and specifying them for VCS debugging features is the same. However, the capabilities that you enable, and the parts of the design to which you can apply them are different.

## Specifying Access Capabilities for PLI Functions

The format for specifying access capabilities is as follows:

```
acc=|+=|-  
=| :=capabilities:module_names|interface_names|sig_name|inst_name  
e [+]|%CELL|%TASK|*
```

Where,

acc

Keyword that begins a line for specifying access capabilities.

= | += | -= | :=

Operators for adding, removing, or changing access capabilities.  
The operators in this syntax are as follows:

=

A shorthand for +=.

+=

Specifies adding the access capabilities to the parts of the design that follow, as specified by module name, interface name, %CELL, %TASK, or \* wildcard character.

--

Specifies removing the access capabilities from the parts of the design that follow, as specified by module name, interface name, %CELL, %TASK, or \* wildcard character.

:=

Specifies changing the access capabilities of the parts of the design that follow, as specified by module name, interface name, %CELL, %TASK, or \* wildcard character, to only those in the list of capabilities on this specification. A specification with this operator can change the capabilities specified in a previous specification.

capabilities

Comma-separated list of access capabilities. The capabilities that you can specify for the functions in your PLI specifications are as follows:

r or read

Reads the values of nets and registers in your design.

`rw` or `read_write`

Both reads from and writes to the values of registers or variables (but not nets) in your design.

`wn`

Enables writing values to nets.

`cbk` or `callback`

To be called when named objects (nets registers, ports) change value.

`cbka` or `callback_all`

To be called when named and unnamed objects (such as primitive terminals) change value.

`frc` or `force`

Forces values on nets and registers.

`prx` or `pulserx_backannotation`

Sets pulse error and pulse rejection percentages for module path delays.

`s` or `static_info`

Enables access to static information, such as instance or signal names and connectivity information. Signal values are not static information.

`tchk` or `timing_check_backannotation`

Back-annotates timing check delay values.

`gate or gate_backannotation`

Back-annotates delay values on gates.

`mp or module_path_backannotation`

Back-annotates module path delays.

`mip or module_input_port_backannotation`

Back-annotates delays on module input ports.

`mipb or module_input_port_bit_backannotation`

Back-annotates delays on individual bits of module input ports.

`module_names`

Comma-separated list of module identifiers.

Specifying modules enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability in all instances of the specified module.

`interface_names`

Comma-separated list of interface identifiers or names.

Specifying interfaces enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability in all instances of the specified interface.

`sig_name`

Signal name on which PLI capabilities are applied. `sig_name` is the full hierarchical path.

For example,

```
<mod_name>.<sig_name> or  
<mod_name>.<inst_name>.<sig_name>  
  
<interface_name>.<sig_name>
```

Note:

- Signal names defined inside the generate block are not supported.
- Structure type is not supported.
- Partition compile and PIP flows are not supported
- The `pli.tab` file does not support debug capabilities in the VHDL part of the design.

`inst_name`

Full hierarchical path of the instance name on which PLI capabilities are applied. You can use the “\*” wildcard character at the end of an instance path.

Note:

Partition compile and PIP flows are not supported.

+

Specifies adding, removing, or changing the access capabilities for not only the instances of the specified modules/interfaces, but also the instances hierarchically under the instances of the specified modules/interfaces.

Note:

The '[+]' feature is not supported for the debug capabilities specified at the signal level (`sig_name`) .

%CELL

Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability in all instances of module definitions compiled under the `'celldefine` compiler directive and all module definitions in Verilog library directories and library files (as specified with the `-y` and `-v` analysis options).

%TASK

Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability in all instances of module definitions that contain the user-defined system task or system function associated with the PLI function.

%TFARGS

Same as %TASK, but also enables debug capability on the arguments to the TF call.

\*

Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability throughout the entire design. Using wildcard characters impact the performance of VCS.

Note:

Do not use blank spaces when specifying access capabilities.

The following examples are the PLI specification examples from the previous section with access capabilities added to them. The examples wrap to more than one line, however, when you edit your PLI table file, ensure that there are no line breaks in these lines.

### **Example 1**

```
$val_proc call=val_proc check=check_proc misc=misc_proc  
acc+= rw,tchk:top,bot acc-=tchk:top
```

This example adds the access capabilities for reading and writing to nets and registers, and for back-annotating timing check delays, to these PLI functions, and enables them to do these things in all instances of modules `top` and `bot`. It then removes the access capability for back-annotating timing check delay values from these PLI functions in all instances of module `top`.

### **Example 2**

```
$value_passer size=0 args=2 call=value_passer persistent  
acc+=rw:%TASK acc-=rw:%CELL
```

This example adds the access capability to read from and write to the values of nets and registers to these PLI functions. It enables them to do these things in all instances of modules declared in module definitions that contain the `$value_passer` user-defined system task. The example then removes the access capability to read from and write to the values of nets and registers from these PLI functions, in module definitions compiled under the `'celldefine` compiler directive and all module definitions in the Verilog library directories and library files.

### **Example 3**

```
$set_true size=16 call=set_true acc+=rw:*
```

This example adds the access capability to read from and write to the values of nets and registers to the PLI functions. It enables them to do this throughout the entire design.

### **Example 4**

```
acc=rw:top.I1,top.I2
```

This example adds the access capabilities on instances top.I1, top.I2.

### **Example 5**

```
acc=rw:mod1.s1,mod1.s2
```

This example adds the access capabilities on signals s1, s2 of mod1.

### **Example 7**

```
acc=rw:top.I1.s1,top.I1.s2,top.I2.s1,top.I2.s2
```

This example adds the access capabilities on signals s1, s2 of instances top.I1, top.I2.

### **Example 8**

```
acc=rw:top.I1+
```

This example adds the access capabilities on instance top.I1 and its sub-instances.

### **Example 9**

```
acc=rw:top.I1.*
```

This example adds read and write capabilities on instance top.I1 and all its sub-instances.

## Usage Example

Consider the following files, test.v, driver.c, and pli.tab:

### *Example 22-1 Testcase test.v*

```
module top;
    integer myInt;
    dut d1();
    initial
    begin
        force myInt = 20;
        $display(" After Force d1.g myInt at %0t is %0d %0d",
        $time, d1.g,myInt);
        #5;
        $myFrc(myInt);
        $display(" After PLI call d1.g myInt at %0t is %0d %0d",
        $time, d1.g,myInt);
        #5
        release myInt;
        $myDrv(d1.g);
        $display(" After PLI call d1.g myInt at %0t is %0d %0d",
        $time, d1.g,myInt);
    end
endmodule

module dut;
    integer g;
endmodule
```

### *Example 22-2 driver.c*

```
#include "acc_user.h"

myDrv()
{
    handle reg = acc_handle_object("top.d1.g");
    static s_setval_delay delay_s = { { 0, 1, 0, 0.0 },
    accNoDelay };
    static s_setval_value value_s = {accIntVal};
    value_s.value.integer=0;
    acc_set_value(reg, &value_s, &delay_s);
```

```

}

myFrc()
{
    handle reg = acc_handle_object("top.myInt");
    static s_setval_delay delay_s = { { 0, 0, 0, 0, 0.0 },
accForceFlag };
    static s_setval_value value_s = {accIntVal};
    value_s.value.integer =2;
    acc_set_value(reg, &value_s, &delay_s);

}

```

**Example 22-3 pli.tab**

```

$myDrv call=myDrv acc=rw:top.d1
$myFrc call=myFrc acc=frc:top.myInt

```

**Compile the test.v code as follows:**

```
% vcs driver.c -P pli.tab test.v -R -sverilog
```

**Perform simulation:**

```
% simv
```

**VCS displays the following output:**

```

After Force d1.g myInt at 0 is x 20
After PLI call d1.g myInt at 5 is x 2
After PLI call d1.g myInt at 10 is 0 2

```

## **Specifying Access Capabilities for VCS Debugging Features**

The format for specifying these capabilities for VCS debugging features is as follows:

acc= | += | -

= | :=*capabilities*:*module\_names*|*interface\_names*[+] | %CELL| \*

Here:

acc

Keyword that begins a line for specifying access capabilities.

= | += | -= | :=

Operators for adding, removing, or changing access capabilities.

Note:

The removal of capability is done after all additions/changes are performed.

*capabilities*

Comma separated list of access capabilities.

*module\_names*

Comma separated list of module identifiers. The specified access capabilities are added, removed, or changed for all instances of these modules.

*interface\_names*

Comma separated list of interface identifiers. The specified access capabilities are added, removed, or changed for all instances of these interfaces.

+

Specifies adding, removing, or changing the access capabilities for not only the instances of the specified modules/interfaces, but also the instances hierarchically under the instances of the specified modules/interfaces.

%CELL

Specifies all modules compiled under the `'celldefine` compiler directive and all modules in the Verilog library directories and library files (as specified with the `-y` and `-v` options.)

\*

Specifies that debug capability is applied to all modules in the design.

The access capabilities and the interactive commands are as follows:

| ACC Capability                      | What it enables your PLI functions to do                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>r</code> or <code>read</code> | For specifying “reads” in your design, it enables commands for performing the following: <ul style="list-style-type: none"><li>• Creating an alias for another UCLI command (<code>alias</code>)</li><li>• Displaying UCLI help</li><li>• Specifying the radix of displayed simulation values (<code>ofORMAT</code>)</li><li>• Displaying simulation values</li><li>• Descending and ascending the module hierarchy</li><li>• Displaying the set breakpoints on signals</li><li>• Displaying the port names of the current location, and the current module instance or scope, in the module hierarchy</li><li>• Displaying the names of instances in the current module instance or scope</li><li>• Displaying the nets and registers in the current scope</li></ul> |

| <b>ACC Capability</b>                      | <b>What it enables your PLI functions to do</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>rw</code> or <code>read_write</code> | <ul style="list-style-type: none"> <li>• Moving up the module hierarchy</li> <li>• Deleting an alias for another UCLI command</li> <li>• Ending the simulation</li> </ul> <p>Both reads from and writes to the values of registers or variables (but not nets) in your design.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>cbk</code> or <code>callback</code>  | <p>Commands perform the following tasks:</p> <ul style="list-style-type: none"> <li>• Setting a repeating breakpoint. In other words always halting simulation, when a specified signal changes value</li> <li>• Setting a one shot breakpoint. In other words, halting simulation the next time the signal changes value, however, not the subsequent time it changes value</li> <li>• Removing a breakpoint from a signal</li> <li>• Showing the line number or number in the source code of the statement or statements that causes the current value of a net</li> <li>• A longer way to specify this capability is with the <code>callback</code> keyword.</li> </ul> <p>Commands perform the following tasks:</p> <ul style="list-style-type: none"> <li>• Forcing a net or a register to a specified value so that this value cannot be changed by subsequent simulation events in the design</li> <li>• Releasing a net or register from its forced value</li> </ul> <p>A longer way to specify this capability is with the <code>force</code> keyword.</p> |
| <code>frc</code> or <code>force</code>     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

## 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
```

Note that there are no blank spaces in this specification. Including blank spaces causes syntax error.

## Example 2

The following specifications enable most interactive commands for most of the modules in a design. They then change the ACC capabilities preventing breakpoint and force commands in instances of modules in Verilog libraries and modules designated as cells with the `'celldefine` compiler directive.

```
acc+=rw, cbk, frc:top+ acc:=rw:%CELL
```

In this example, the first specification enables the interactive commands that are enabled by the `rw`, `cbk`, and `frc` capabilities for module `top`, which, in this example, is the top-level module of the design, and all module instances under it. The second specification limits the interactive commands for the specified modules to only those enabled by the `rw` (same as `r`) capability.

## Using the PLI Table File

To specify the PLI table file, specify 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 in the `vcs` command line, you can also enter C source files, compiled `.o` object files, or `.a` libraries in 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
```

When you include `.o` object files and `.a` libraries, you do not have to recompile the PLI application every time you compile your design.

---

## Enabling ACC Capabilities

VCS allows you to enable ACC capabilities throughout your design and also specify ACC capabilities in specific parts of your design (as described in “[PLI Table File](#)”). It also enables you to specify selected write capabilities using a configuration file. As enabling ACC capabilities has an impact on performance, VCS allows you to enable only the ACC capabilities you need.

### Enabling ACC Capabilities Globally

You can enter the `+acc+level_number` compile time option to globally enable ACC capabilities throughout your design.

Note:

Using the `+acc+level_number` option significantly impedes the simulation performance of your design. Synopsys recommends that you use a PLI table file to enable ACC capabilities for only the parts of your design where you need them. For more information about enabling ACC capabilities for specific parts of a design, see “[PLI Table File](#)”.

You can specify additional ACC capabilities using `level_number` as follows:

```
+acc+3
```

Enables read, write, and callback capabilities. This option also enables module path delay annotation.

+acc+4

Enables read, write, and callback capabilities. This option also enables module path and gate delay annotation.

## Using the Configuration File

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 PLIACC 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};
```

OR

```
signal {list_of_signal_hierarchical_names}
{accWrite};
```

Where,

set

Keyword preceding a property that applies to the entire design.

writeOnMem

Enables ACC write to memories (any single or multi-dimensional array of the reg data type) throughout the entire design.

noAccWrite

Disables ACC write capabilities throughout the entire design.

accWrite

Enables ACC write capabilities.

module

Keyword specifying that the `accWrite` attribute in this statement applies to all instances of the modules in the list, specified by module identifier.

`list_of_module_identifiers`

Comma separated list of module identifiers (also called module names).

`instance`

Keyword specifying that the `accWrite` attribute in this statement applies to all instances in the list.

`list_of_module_instance_hierarchical_names`

Comma separated list of module instance hierarchical names.

**Note:**

Follow the Verilog syntax for signal names and hierarchical names of module instances.

`tree`

Keyword specifying that the `accWrite` attribute in this statement applies to all instances of the modules in the list, specified by module identifier, and also applies to all module instances hierarchically under these module instances.

`depth`

An integer that specifies how far down the module hierarchy from the specified modules you want to apply the `accWrite` attribute. You can specify a negative value. A negative value specifies descending to the leaf level and counting up levels of the hierarchy to apply these attributes. This specification is optional. Enclose this specification in parentheses: ( )

signal

Keyword specifying that the `accWrite` attribute in this statement applies to all signals in the list.

`list_of_signal_hierarchical_names`

Comma separated list of signal hierarchical names.

## Selected ACC Capabilities

This section describes the compile time and runtime options that enable VCS and PLI applications to use only the ACC capabilities you need. The procedure to use these options is as follows:

Use the following runtime options to instruct VCS to keep track of, or learn, the ACC capabilities:

*Table 22-1 Runtime Options to Enable PLI Learn*

| Option                                                        | Description                                |
|---------------------------------------------------------------|--------------------------------------------|
| <code>+vcs+learn+pli</code>                                   | Enables module level PLI learn             |
| <code>+vcs+learn+pli+instpli</code>                           | Enables instance level PLI learn           |
| <code>+vcs+learn+pli+sigpli</code>                            | Enables signal PLI learn at module level   |
| <code>+vcs+learn+pli+sigpli<br/>+vcs+learn+pli+instpli</code> | Enables signal PLI learn at instance level |

Note:

PLI learn at signal or instance level is enabled by default, if the PLI table file has any hierarchical path specified.

VCS uses this information to create a secondary PLI table file named `pli_learn.tab`. You can use this table file to recompile your design so that subsequent simulations use only the ACC capabilities that are needed.

1. Instruct 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).
2. Simulate again with a `simv` executable in which only the ACC capabilities you need are enabled.

## Learning What Access Capabilities are Used

Include the `+vcs+learn+pli` runtime option to instruct VCS to learn the access capabilities that are used by the modules in your design and write them into a secondary PLI table file named, `pli_learn.tab`.

This file is considered as 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, modifies the access capabilities that are specified in a first PLI table file, or other means of specifying access capabilities, so that you enable only the capabilities you need in subsequent simulations.

See the contents of the `pli_learn.tab` file that VCS writes to understand the access capabilities that are used during simulation. The following is an example of this file:

```
////////////// SYNOPSYS INC /////////////
//          PLI LEARN FILE
// AUTOMATICALLY GENERATED BY VCS(TM) LEARN MODE
///////////////
acc=r:testfixture
    //SIGNAL STIM_SRLS:r
acc=rw:SDFFR
    //SIGNAL S1:rw
```

The following line in this file specifies that during simulation, the read capability was needed for signals in the module named testfixture.

```
acc=r:testfixture  
    //SIGNAL STIM_SRLS:r
```

The comment lets you know that the only signal for which this capability is needed is the signal named, STIM\_SRLS. This line is in the form of a comment because the syntax of the PLI table file does not permit specifying access capabilities on a signal-by-signal basis.

The following line in this file specifies that during simulation, the read and write capabilities are needed for signals in the module named, SDFFR, specifically for the signal named S1.

```
acc=rw:SDFFR  
    //SIGNAL S1:rw
```

The following are the examples of the pli\_learn.tab file for +vcs+learn+pli+instpli and +vcs+learn+pli+sigpli options:

Consider [Example 22-1](#), [Example 22-2](#), and [Example 22-3](#).

Compile the test.v code as follows:

```
% vcs driver.c -P pli.tab test.v -R -sverilog
```

Perform simulation using the +vcs+learn+pli+instpli option:

```
% simv +vcs+learn+pli+instpli
```

VCS displays the following output:

```
////////// SYNOPSYS INC //////////  
// PLI LEARN FILE
```

```

// AUTOMATICALLY GENERATED BY VCS(TM) LEARN MODE
///////////////////////////////
acc=frc:top
    //SIGNAL myInt:frc
acc=rw:top.d1
    //SIGNAL g:rw

```

Perform the simulation using the `+vcs+learn+pli+sigpli` option:

```
% simv +vcs+learn+pli+sigpli
```

VCS displays the following output:

```

//////////////////// SYNOPSYS INC /////////////////////
//                      PLI LEARN FILE
// AUTOMATICALLY GENERATED BY VCS(TM) LEARN MODE
///////////////////////
acc=frc:top.myInt

acc=rw:dut.g

```

### **Signs of a Potentially Significant Performance Gain**

You might see one of following comments in the `pli_learn.tab` file:

- `// !VCS_LEARNED: NO_ACCESS_PERFORMED`

This indicates that none of the enabled access capabilities were used during the simulation.

- `// !VCS_LEARNED: NO_DYNAMIC_ACCESS_PERFORMED`

This indicates that only static information was accessed through access capabilities and there was no value change information during simulation.

These comments indicate that there is a potentially significant performance gain when you apply the access capabilities in the `pli_learn.tab` file.

## Compiling to Enable Only the Access Capabilities You Need

After you have run the simulation to learn what access capabilities are actually used by your design, you can then recompile the design with the information you have learned, so the resulting `simv` executable uses only the access capabilities you require.

When you recompile your design, include the `+applylearn` compile time option.

If you have 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, have specified a PLI table file with the `-P` compile-time option, specify this PLI table file again, using the `-P` option along with the `+applylearn` option.

### Note:

If you change your design after VCS writes the `pli_learn.tab` file, and you want to make sure that you are using only the access capabilities you need, you will need to have VCS write another one by including the `+vcs+learn+pli` runtime option and then compiling your design again with the `+applylearn` option.

## **Limitations**

VCS does not maintain a history of all access capabilities. However, the capabilities that are maintained and specified in the `pli_learned.tab` file are as follows:

- `r` - read
- `rw` - read and write
- `cbk` - callbacks
- `cbka` - callback all including unnamed objects
- `frc` - forcing values on signals

The `+applylearn` compile-time option does not work if you also use either the `+multisource_int_delays` or `+transport_int_delays` compile time option, because interconnect delays need global access capabilities.

If you enter the `+applylearn` compile-time option more than once on the `vcs` command line, VCS ignores all occurrences except the first.

### **Important:**

The `+applylearn` option is for performance and if you enter this option with options for debugging, such as `-debug_access`, VCS ignores the debugging options.

---

## **Using VPI Routines**

To enable VPI capabilities in VCS, use the `+vpi` compilation/elaboration option, as shown in the following example:

```
% vcs +vpi top -P test.tab test.c
```

The header file for the VPI routines is \$VCS\_HOME/include/vpi\_user.h.

The +vpi option is automatically enabled as part of – debug\_access.

The +vpi+1 option prevents VPI capability on tasks and functions. This takes precedence over +vpi.

The +vpi+1+assertions option is same as +vpi+1, but allows VPI capability on assertions. This takes precedence over +vpi+1.

You can register your user-defined system tasks/function-related callbacks using the vpi\_register\_systf VPI routine. For more information, see “[Support for the vpi\\_register\\_systf Routine](#)”.

You can also use the PLI .tab file to associate your user-defined system tasks with your VPI routines. For more information, see “[PLI Table File for VPI Routines](#)”.

---

## Support for VPI Callbacks

The vpi\_register\_cb() callback mechanism can be registered for callbacks to occur for simulation events, such as value changes on an expression or terminal, or the execution of a behavioral statement. When the cb\_data\_p-> reason field is set to one of the following, the callback occurs as follows:

- cbForce/cbRelease — After a force or release is occurred
- cbAssign/cbDeassign — After a procedural assign or deassign statement is executed

VPI callbacks cbForce and cbRelease are supported with the following limitations:

- The force and release commands generates a callback only if cb\_data\_p > obj is a valid handle. If it is set to NULL, it does not generate a callback.
- For cbForce, cbRelease, cbAssign, and cbDeassign callbacks, the handle that you supplied while registering the callback is returned and not the corresponding statement handle [NULL handles are not allowed].

For more information about the VPI callbacks, see section *Simulation-event-related callbacks in the Verilog IEEE LRM 1364-2001*.

---

## Support for the vpi\_register\_systf Routine

VCS supports the vpi\_register\_systf VPI access routine. To use this routine, you need to make an entry in the vpi\_user.c file. You can copy this file from \$VCS\_HOME/etc/vpi.

Consider the following example:

```
/*=====
Copyright (c) 2003 Synopsys Inc
=====*/
/* Fill your start up routines in this array, Last entry
should be
zero, */
extern void register_me();
void (*vlog_startup_routines[])(() = {
register_me,
    0 /* Last Entry */ };
```

entry here

In this example:

- The routine named `register_me` is externally declared.
- It is also included in the array named `vlog_startup_routines`.
- The last entry in the array is zero.

For example,

```
% vcs top.v vpi_user.c +vpi
```

You can also write a PLI table file for VPI routines. See “[PLI Table File for VPI Routines](#)”.

---

## Integrating a VPI Application With VCS

If you create one or more shared libraries for a VPI application, the application must not contain the `vlog_startup_routines` array.

Instead, enter the `-load` compile-time option to specify the registration routine. The syntax is as follows:

```
-load shared_library:registration_routine
```

It is not required to specify the path name of the shared library, if that path is part of your `LD_LIBRARY_PATH` environment variable.

The following are some examples of using this option:

- `-load lib1.so: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.so: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.so:my_register -load lib2.so: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` options to specify multiple shared libraries and their registration routines.**
- `-load lib1.so:my_register`  
**The registration routine `my_register()` is in `lib1.so`. The location of `lib1.so` is in the `LD_LIBRARY_PATH` environment variable.**
- `-load /usr/lib/mylib.so:my_register`  
**The registration routine `my_register()` is in `lib1.so`, which is in `/usr/lib/mylib.so`, and not in the `LD_LIBRARY_PATH` environment variable.**

## PLI Table File for VPI Routines

The PLI table file for VPI routines works similar to and with the same syntax as a PLI table file for user-defined system tasks that execute C functions. The following is an example of such a PLI table file:

```
$set_mipd_delays call=PLIbook_SetMipd_calltf
```

```
check=PLIbook_SetMipd_completf  
acc=mip,mp,gate,tchk,rw:test+
```

Note that this entry includes `acc=` even though the C functions in the PLI specification call VPI routines instead of PLI 1.0 routines. The syntax has not changed; you use the same syntax for enabling PLI 1.0 and PLI 2.0 routines.

This PLI table file is used for an example file named `set_mipd_delays_vpi.c`, which is available with *The Verilog PLI Handbook* by Stuart Sutherland, Kluwer Academic Publishers, Boston, Dordrecht, and London.

---

## Virtual Interface Debug Support

A Virtual Interface is a reference object that can either be initially assigned at its declaration or not assigned. You can debug the Virtual Interface object when it is initially assigned or not assigned within a module or a class.

To debug the Virtual Interface objects, the VPI properties defined in the SystemVerilog LRM, such as `vpiVirtual`, `vpiActual`, and `vpiInterfaceDecl`, are supported. For more information about these properties, see the *IEEE SystemVerilog LRM*.

## Example

The following example shows the VPI routines usage for Virtual Interface Debug:

virtual\_interface.sv

```
interface ifc (input logic clk);  
    event reset;
```

```

        int      ifci;
        modport tracker (input clk);
endinterface: ifc

package p;

class C;

    virtual ifc.tracker busmpIF;          VI declared in Class  
scope

    virtual ifc busIF;
    int i;

    function new (virtual ifc inf);
        busIF = inf;
    endfunction // new

    function test(virtual ifc inf);
        busIF = inf;
        $display("hello");
    endfunction: test
endclass: C
endpackage: p

module mod( input logic clk);
    import p::*;

    ifc trkIF(.clk(clk));

    virtual ifc modbusIF = trkIF;          VI declared in Module  
scope
    virtual ifc.tracker modportIF2;

    C c;

    initial begin
`ifdef DUMP
        $vcdpluson;
`endif
        c = new(trkIF);
        c.test(modbusIF);
    end
endmodule

```

```

        modbusIF.ifci <= 10;
#1
$getVar;
$display("end the first round\n");
#1
modbusIF.ifci <= 11;
$getVar;
$display("end the second round.");
end
endmodule: mod

```

## pli.c

```

#include <stdio.h>
#include <stdlib.h>
#include "vcs_vpi_user.h"
#include "sv_vpi_user.h"

void traverse(){
    vpiHandle Han, iterHan, scanHan, cls, obj, intfHan,
Href, Hactual;

    vpi_configure(vpiDisplayWarnings,"true");

    intfHan = vpi_handle_by_name("mod.vbusIF",NULL);
    vpi_printf("\tVAR `%s'\n", vpi_get_str(vpiName,intfHan));
    vpi_printf("\t--- DefName `%s'\n\t--- FullName:%s\n\t-
-- vpiType:%s\n",
                vpi_get_str(vpiDefName,intfHan),
    vpi_get_str(vpiFullName,intfHan),
                vpi_get_str(vpiType,intfHan));
    if(vpi_get(vpiVirtual, intfHan)){
        vpi_printf("\t%s is Virtual
Interface\n",vpi_get_str(vpiName,intfHan));
    }
    Hactual = vpi_handle(vpiActual, intfHan);
    if ( Hactual )
    {
        vpi_printf("\n\tActual `%s'\n",
vpi_get_str(vpiName,Hactual));
        vpi_printf("\t--- DefName `%s'\n\t--- FullName:%s\n\t-
-- vpiType:%s\n",

```

```

        vpi_get_str(vpiDefName,Hactual),
vpi_get_str(vpiFullName,Hactual),
        vpi_get_str(vpiType,Hactual));
    if(vpi_get(vpiVirtual, Hactual)){
        vpi_printf("\tActual Handle is Virtual Interface\n");
    }
}
}

```

### **pli.tab**

```
$getVar call=traverse acc+=r:* acc+=cbk:*
```

To compile this example code, use the following commands:

```
% vcs -P pli.tab pli.c virtual_interface.sv -
debug_access+all -sverilog

% simv -verdi
```

## **Limitations**

The following are the limitations with this functionality:

- Virtual Interface passed as a method port is not displayed in Verdi.
- Virtual Interface as an array is not supported.
- Virtual Interface debugging is not supported in UCLI.
- \$vcdplusmsglog do not dump Virtual Interface.

## **Unimplemented VPI Routines**

VCS does not support all the functionalities specified for VPI routines in the *IEEE Standard Verilog® Hardware Description Language, 1364-2001*, as some routines would be rarely used and some of the data access operations of other routines would be rarely used. The unimplemented routines are as follows:

- vpi\_get\_data
- vpi\_put\_data
- vpi\_sim\_control

Object data model diagrams in the *IEEE Verilog Language Reference Manual* specify that some VPI routines should be able to access data that is rarely needed. These routines and the data they cannot access are as follows:

vpi\_get\_value

- Cannot retrieve the value of `var` select objects (see diagram 26.6.8 Variables in *IEEE Standard Verilog® Hardware Description Language, 1364-2001*) and `func` call objects (diagram 26.6.18 Task, function declaration in *IEEE Standard Verilog® Hardware Description Language, 1364-2001*).
- 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 (see diagram 26.6.8 Variables in *IEEE Standard Verilog® Hardware Description Language, 1364-2001*) and primitive objects (see diagram 26.6.13 Primitive, `prim` term in *IEEE Standard Verilog® Hardware Description Language, 1364-2001*).

vpi\_get\_delays

Cannot retrieve the values of continuous assign objects (see diagram 26.6.24 Continuous assignment in *IEEE Standard Verilog® Hardware Description Language, 1364-2001*) or procedurally assigned objects.

vpi\_put\_delays

Cannot put values on continuous assign objects (see diagram 26.6.24 Continuous assignment in *IEEE Standard Verilog® Hardware Description Language, 1364-2001*) or procedurally assigned objects.

vpi\_register\_cb

Cannot register the following types of callbacks that are defined for this routine:

cbError

cbPliError

---

## Modified VPI Features

VCS complies with some of the constants that are standardized to comply with the IEEE LRM 1800-2009.

[Table 22-2](#) provides the modified constants for the handle type returned by `vpi_get(vpiType, handle)`.

*Table 22-2 Modified Constants for the Handle Type Returned by  
vpi\_get(vpiType, handle)*

| In G-2012.09 and earlier releases   | From H-2013.06 release          |
|-------------------------------------|---------------------------------|
| <code>vpiImmediateAssertType</code> | <code>vpiImmediateAssert</code> |
| <code>vpiImmediateAssumeType</code> | <code>vpiImmediateAssume</code> |

|                              |                    |
|------------------------------|--------------------|
| vpiImmediateCoverType        | vpiImmediateCover  |
| vpiAssertType                | vpiAssert          |
| vpiAssumeType                | vpiAssume          |
| vpiCoverType                 | vpiCover           |
| *vpiImmediateFinalAssertType | vpiImmediateAssert |
| *vpiImmediateFinalAssumeType | vpiImmediateAssume |
| *vpiImmediateFinalCoverType  | vpiImmediateCover  |
| vpiEndedOp                   | vpiTriggeredOp     |
| vpiModPortPort               | vpiModportPort     |

\* Use `vpi_get(vpiFinal, <assert_handle>)` to determine if they are final type.

[Table 22-3](#) provides the modified constants used in iterators.

*Table 22-3 Modified Constants Used in Iterators*  
`vpi_iterate(constant)`

| In G-2012.09 and earlier releases | From H-2013.06 release                                      |
|-----------------------------------|-------------------------------------------------------------|
| vpiSequence                       | vpiExpr                                                     |
| vpiSequenceExpr                   | vpiExpr                                                     |
| vpiModPort                        | vpiModport                                                  |
| vpiIdentifier                     | vpiSeqFormalDecl <b>when ref handle is</b> vpiSequenceDecl  |
| vpiIdentifier                     | vpiPropFormalDecl <b>when ref handle is</b> vpiPropertyDecl |

## Example

`pli.c`

```
modport_iter = vpi_iterate(vpiModPort, refHandle);
```

```
// here refHandle points to object of type interface
```

## Error Message

```
< ...: error: 'vpiModPort' undeclared (first use in this  
function)
```

## Solution

You must change the code to comply with the LRM 1800-2009.

[Table 22-4](#) provides the constants that are updated for the new value.

*Table 22-4 Constants That Are Updated for the New Value*

| Modified Constants   |
|----------------------|
| vpiPortType          |
| vpiInterfacePort     |
| vpiMember            |
| vpiStructUnionMember |
| vpiAssertion         |
| vpiClockingEvent     |
| vpiDisableCondition  |
| vpiIfOp              |
| vpiElseOp            |
| vpiCompAndOp         |
| vpiCompOrOp          |
| vpiAssignmentOp      |
| vpiAcceptOnOp        |
| vpiRejectOnOp        |
| vpiSyncAcceptOnOp    |
| vpiSyncRejectOnOp    |

|                           |
|---------------------------|
| vpiOverlapFollowedByOp    |
| vpiNonOverlapFollowedByOp |
| vpiNexttimeOp             |
| vpiAlwaysOp               |
| vpiEventuallyOp           |
| vpiUntilOp                |
| vpiUntilWithOp            |
| vpiImpliesOp              |
| vpiInsideOp               |

## Example

plic.c

```
assertIter = vpi_iterate(700, 0x0);Solution
```

## Warning Message

Warning-[VCS-VPI-DEPRECATED] VPI value deprecated

In 'vpi\_iterate' call, the VPI value for  
`vpiAssertion`(700) is deprecated and will not be  
supported in the next release.  
Please check, fix and recompile your PLI program.

## Solution

When using the hardcoded values, change the code to use the constants.

## Using VHPI Routines

VHPI enables you to use foreign architecture-based models written in C language in VCS VHDL.

---

## Diagnostics for VPI/VHPI PLI Applications

As defined in the LRM, VPI/VHPI remains silent when an error occurs. The application checks for error status to report an error. If error detection mechanisms are not in place, the C code of the application must be modified and recompiled. In addition, you might have to recompile the HDL code, if required.

However, you can use the following runtime diagnostic options to make the PLI application to report errors without code modification:

- `-diag vpi`
- `-diag vhpi`

For more information, see “[Diagnostics for VPI/VHPI PLI Applications](#)”.

---

## Using DirectC

DirectC is an extended interface between Verilog and C/C++. It is an alternative to the PLI. Unlike the PLI, DirectC enables you to do the following:

- More efficiently pass values between Verilog module instances and C/C++ functions by calling the functions directly along with actual parameters in your Verilog code.
- Pass more types of data between Verilog and C/C++. With the PLI, you can only pass Verilog information to and from a C/C++ application. With DirectC you do not have this limitation.

With DirectC, for example, you can model a simulation environment for your design in C/C++ in which you can pass pointers from the environment to your design and store them in Verilog signals, and at a later simulation time, pass these pointers to the simulation environment.

Similarly, you can use DirectC to develop applications to run with VCS to which you can pass pointers to the location of simulation values for your design.

DirectC is an alternative to PLI, however, DirectC is not a replacement for PLI. Certain functionalities can only be enabled using PLI. 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. However, this functionality cannot be enabled with DirectC.

You can use Direct C/C++ function calls for existing and proven C code as well as C/C++ code that you write in the future. You can also use them without much rewriting of, or additions to, your Verilog code. You call them the same way you call (or enable) a Verilog function or Verilog task.

This section describes the DirectC interface in the following sections:

- “[Using Direct C/C++ Function Calls](#)”
- “[Using Direct Access](#)”
- “[Using Abstract Access](#)”
- “[Enabling C/C++ Functions](#)”
- “[Extended BNF for External Function Declarations](#)”

---

## Using Direct C/C++ Function Calls

To enable a direct call of a C/C++ function during simulation, perform the following tasks:

1. Declare the function in your Verilog code.
2. Call the function in your Verilog code.
3. Compile/Elaborate your design and C/C++ code using elaboration/compile-time options for DirectC.

However, there are complications to this otherwise straightforward procedure.

DirectC allows the invocation of C++ functions that are declared in C++ using the `extern "C"` linkage directive. The `extern "C"` directive is necessary to protect the name of the C++ function from being mangled by the C++ compiler. Plain C functions do not undergo mangling, and therefore, do not need any special directive.

The declaration of these functions involves specifying a direction for the parameters of the C function, because, in the Verilog environment, they become analogous to Verilog tasks as well as functions. Verilog tasks are similar to void C functions in that they do not return a value. However, Verilog tasks do have input, output, and inout arguments, whereas C function parameters do not have explicitly declared directions. For more information, see “[Declaring the C/C++ Function](#)”.

You can use the following access modes for C/C++ function calls. These modes do not make much difference in your Verilog code; they only pertain to the development of the C/C++ function. They are as follows:

- The slightly more efficient direct access mode - this mode has rules for how values of different types and sizes are passed to and from Verilog and C/C++. For more information about this mode, see “[Using Direct Access](#)”.
- The slightly less efficient, but with better error handling abstract access mode. 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 includes a library of accessory functions for using these handles. For more information, see “[Using Abstract Access](#)”.

The abstract access library of accessory functions contains operations for reading and writing values and for querying about argument types, sizes, and so on. 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, 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, decide how you want 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 a small change in your Verilog code.

Using abstract access is “safer” because the library of accessory functions for abstract access issues error messages that help you to debug the interface between C/C++ and Verilog. With direct access, errors result in segmentation faults, memory corruption, and so on.

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.

## Functioning of C/C++ Code in a Verilog Environment

Similar to 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, similar to the arguments of Verilog tasks. Do not specify them as such in your C code, but you can 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`

Therefore, 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

```

Where,

`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 22-5](#) for a description of what these types specify.

*small\_bit\_vector*

Specifies a bit-width of a returned vector `bit`. A C/C++ function cannot return a four-state vector `reg`, but it can return a vector `bit` if its bit-width is 32 bits or less.

*function\_id*

The name of the C/C++ function.

*direction*

One of the following keywords: `input`, `output`, `inout`. In a C/C++ function, these keywords specify the same thing that they specify in a Verilog task; see [Table 22-6](#).

*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 (that is,, `input`, `output`, `inout`) applies to all arguments that follow it until the next direction occurs; the default direction is `input`.

*Table 22-5 C/C++ Function Return Types*

| <b>Return Type</b>   | <b>Specifies</b>                                                                                                         |
|----------------------|--------------------------------------------------------------------------------------------------------------------------|
| <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 reg 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 reg.                                                            |
| <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 22-6 C/C++ Function Argument Directions*

| <b>keyword</b>      | <b>Specifies</b>                                                                                                                                            |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <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 <code>input</code> keyword. |
| <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 22-7 C/C++ Function Argument Types**

| keyword | Specifies                                                                                         |
|---------|---------------------------------------------------------------------------------------------------|
| real    | The C/C++ function reads or writes the address of a Verilog real data type.                       |
| reg     | The C/C++ function reads or writes the value or address of a Verilog reg.                         |
| bit     | The C/C++ function reads or writes the value or address of a Verilog reg in two state simulation. |
| int     | The C/C++ function reads or writes the address of a C/C++ int data type.                          |
| pointer | The C/C++ function reads or writes the address that a pointer is pointing to.                     |
| string  | The C/C++ function reads from or writes to the address of a string.                               |

## Example 1

```
extern "A" reg return_reg (input reg r1);
```

This example declares a C/C++ function named `return_reg`. This function returns the value of a scalar reg. When you 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 you call this function, the value of an 8-bit vector bit (a reg in two state simulation) named `r3` is passed to the function. This function uses direct access.

The keyword `input` is omitted. This keyword can be omitted if the first argument specified is an input argument.

### **Example 3**

```
extern string return_string();
```

This example declares a C/C++ function named `return_string`. This function returns a character string and takes no arguments.

### **Example 4**

```
extern void receive_string( input string r5);
```

This example declares a C/C++ function named `receive_string`. It is a void function. At some time earlier in the simulation, another C/C++ function passed the address of a character string to reg `r5`. When you 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 you 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 you 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 you 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 you 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`. Bit width for vector bit `r7` is not specified. This allows you 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 you 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](#)”. When it finishes executing, a new value is passed by reference to bit `r9`.

## Calling the C/C++ Function

After declaring the C/C++ function, you can call it in your Verilog code. You call a void C/C++ function in the same manner as you call a Verilog task-enabling statement, that is, by entering the function name and its arguments, either on a separate line in an `always` or `initial` block, or in the procedural statements in a Verilog task or function declaration. Unlike Verilog tasks, you can call a C/C++ function in a Verilog function.

You call a non-void (returns a value) C/C++ function in the same manner as you call a Verilog function call, that is, by entering its name and arguments, either in an expression on the RHS of a procedural assignment statement in an `always` or `initial` block, or in a Verilog task or function declaration.

## Examples

```
r2=return_reg(r1);
```

The value of scalar reg `r1` is passed to C/C++ function `return_reg`. It returns a value to reg `r2`.

```
r4=return_vector_bit(r3);
```

The value of vector bit `r3` is passed to C/C++ function `return_vector_bit`. It returns a value to vector bit `r4`.

```
r5=return_string();
```

The address of a character string is passed to reg `r5`.

```
receive_string(r5);
```

The address of a character string in reg `r5` is passed to C/C++ function `receive_string`.

```
r6=return_pointer();
```

The address pointed to in a pointer in C/C++ function `return_pointer` is passed to reg `r6`.

```
get_pointer(r6);
```

The address in reg `r6` is passed to C/C++ function `get_pointer`.

```
memory_reorg(mem1,mem2);
```

In this example, all the values in memory `mem2` are passed to C/C++ function `memory_reorg`, and when it finishes executing, it passes new values to memory `mem1`.

```
incr(r7);
```

In this example, the value of bit `r7` is passed to C/C++ function `incr`, and when it finishes executing, it passes a new value to bit `r7`.

## Storing Vector Values in Machine Memory

If you are using direct access you must know how vector values are stored in memory. This information is also helpful if you are using abstract access.

Verilog four-state simulation values (1, 0, x, and z) are represented in machine memory with data and control bits. The control bit differentiates between the 1 and x and the 0 and z values, as shown in the following table:

| Simulation Value | Data Bit | Control Bit |
|------------------|----------|-------------|
| 1                | 1        | 0           |
| x                | 1        | 1           |
| 0                | 0        | 0           |
| z                | 0        | 1           |

When a routine returns Verilog data to a C/C++ function, how that data is stored depends on whether it is from a two-state or four-state value, and whether it is from a scalar, a vector, or from an element in a Verilog memory.

For a four-state vector (denoted by the keyword reg), the Verilog data is stored in type `vec32`, which for abstract access is defined as follows:

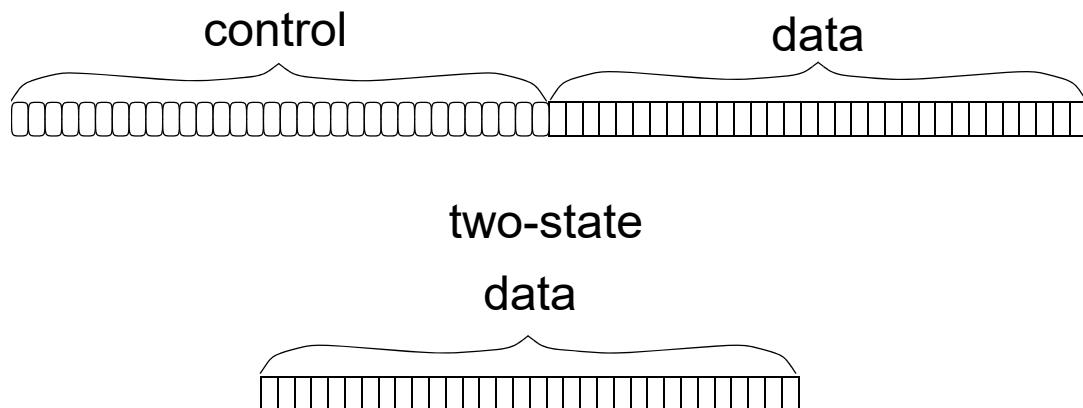
```
typedef unsigned int U;
typedef struct { U c; U d; } vec32;
```

So, type `vec32*` has two members of type `U`; member `c` is for control bits and member `d` is for data bits.

For a two-state vector bit, the Verilog data is stored in type `U*`.

Vector values are stored in arrays of chunks of 32 bits. For four-state vectors there are chunks of 32 bits for data values and 32 bits for control values. For two-state vectors, there are chunks of 32 bits for data values.

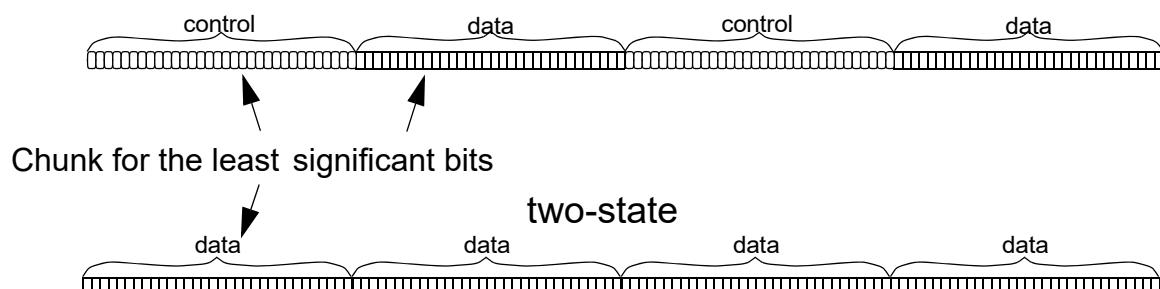
*Figure 22-1 Storing Vector Values  
four-state*



Long vectors, more than 32 bits, have their value stored in more than one group of 32 bits and can be accessed by chunk. Short vectors, 32 bits or less, are stored in a single chunk.

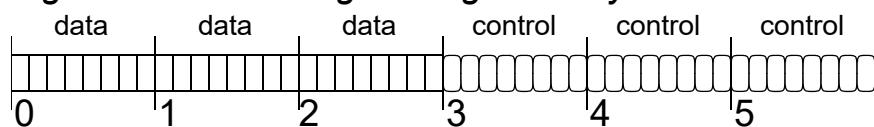
For long vectors, the chunk for the least significant bits come first, followed by the chunks for the more significant bits.

*Figure 22-2 Storing Vector Values of More than 32 Bits  
four-state*



In an element in a Verilog memory, for each eight bits in the element, there is a data byte and a control byte with an additional set of bytes for remainder bit. Therefore, if a memory has 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 22-3 Storing Verilog Memory Elements in Machine Memory*



## Converting Strings

There are no \*true\* strings in Verilog, and a string literal, like "some\_text," is only a notation for vectors of bits, based on the same principle as binary, octal, decimal, and hexadecimal numbers. So there is a need for a conversion between the two representations of "strings": the C-style representation (which actually is a pointer to the sequence of bytes terminated with null byte) and the Verilog vector encoding a string.

DirectC includes the `vc_ConvertToString()` routine that you can use to convert a Verilog string to a C string. Its syntax is as follows:

```
void vc_ConvertToString(vec32 *, int, char *)
```

There are scenarios in which a string is created on the Verilog side and is passed to C code, and therefore, has to be converted from Verilog representation to C representation.

Consider the following example:

```
extern void WriteReport(string result_code, .... /* other  
stuff */);
```

Example of a valid call:

```
WriteReport("Passes", ....);
```

Example of incorrect code:

```
reg [100*8:1] message;  
. . .  
message = "Failed";  
. . .  
WriteReport(message, ....);
```

This call causes a core dump because the function expects a pointer and gets some random bits instead.

It might 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, ...)
```

Following are the 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 Pass by Reference

You can use pass by reference with DirectC. The following source files: `main.v` and `pythag.c`, illustrate using pass by reference.

### **main.v**

```
extern void pythag(inout real);
module main;
real p;
initial begin
    p = 7.89;
    pythag(p);
    $finish;
end
endmodule
```

### **pythag.c**

```
#include <stdio.h>
void pythag(double *p)
{
    printf ("Passed real value from verilog p=%f \n", *p);
}
```

You can try out this example with the following command-line:

```
% vcs +vc main.v pythag.c -R -l somv.log
```

At runtime, VCS displays the following:

```
Passed real value from verilog p=7.890000
```

---

## Using Direct Access

Direct access for C/C++ routines whose formal parameters are of the following types:

|       |        |         |         |        |
|-------|--------|---------|---------|--------|
| int   | int*   | double* | void*   | void** |
| char* | char** | scalar  | scalar* |        |
| U*    | vec32  | UB*     |         |        |

Some of these type identifiers are standard C/C++ types; those that are not, are 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 22-8 For Input Arguments*

| argument type                                         | C/C++ formal parameter data type | Passed by |
|-------------------------------------------------------|----------------------------------|-----------|
| int                                                   | int                              | value     |
| real                                                  | double*                          | reference |
| pointer                                               | void*                            | value     |
| string                                                | char*                            | value     |
| bit                                                   | scalar                           | value     |
| reg                                                   | scalar                           | value     |
| bit [] - 1-32 bit wide vector                         | U                                | value     |
| bit [] - open vector, any vector wider than 32 bits   | U*                               | reference |
| reg [] - 1-32 bit wide vector                         | vec32*                           | reference |
| array [] - open vector, any vector wider than 32 bits | UB*                              | reference |

**Table 22-9 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);
```

In this example, the function named `return_reg` returns the value of a scalar reg. The value of a scalar reg is passed to it. The header of the C/C++ function is as follows:

```
extern "C" scalar return_reg(scalar reti);
scalar return_reg(scalar reti);
```

If `return_reg()` is a C++ function, it must be protected from name mangling, as follows:

```
extern "C" scalar return_reg(scalar reti);
```

Note:

The `extern "C"` directive has been omitted in subsequent examples for brevity.

A scalar reg is passed by value to the function so the parameter is not a pointer. The parameter's type is scalar.

## Example 2

Consider the following C/C++ function declared in the Verilog source code:

```
extern "C" bit [7:0] return_vector_bit (bit [7:0] r3);
```

In this example, the function named `return_vector_bit` returns the value of a vector bit. The "`C`" entry specifies direct access.

Typically, a declaration includes this when some other functions use abstract access. The value of an 8-bit vector bit is passed to it. The header of the C/C++ function is as follows:

```
U return_vector_bit(U returner);
```

A vector bit is passed by value to the function because the vector bit is less than 33 bits so the parameter is not a pointer. The parameter's type is U.

## Example 3

Consider the following C/C++ function declared in the Verilog source code:

```
extern void receive_pointer ( input pointer r6 );
```

In this example, the function named `receive_pointer` does not return a value. The argument passed to it is declared as a pointer. The header of the C/C++ function is as follows:

```
void receive_pointer(*pointer_receiver);
```

A pointer is passed by value to the function so the parameter is a pointer of type `void`, a generic pointer. In this example, it is not required to know the type of data that it points to.

#### **Example 4**

Consider the following C/C++ function declared in the Verilog source code:

```
extern void memory_rewriter (input bit [1:0] array [1:0]
                            mem2, output bit [1:0] array [1:0] mem1);
```

In this example, the function named `memory_rewriter` has two arguments, one declared as an input, the other as an output. Both arguments are bit memories. The header of the C/C++ function is as follows:

```
void memory_rewriter(UB *out[2], *in[2]);
```

Memories are always passed by reference to a C/C++ function so the parameter named `in` is a pointer of type `UB` with the size that matched the memory range. The parameter named `out` is also a pointer, because its corresponding argument is declared to be output. Its type is also `UB` because it outputs to a Verilog memory.

## **Example 5**

Consider the following C/C++ function declared in the Verilog source code:

```
extern void incr (inout bit [] r7);
```

In this example, the function named `incr`, that does not return a value, has an argument declared as `inout`. No bit-width is specified, but the `[]` entry for the argument specifies that it is not a scalar bit. The header of the C/C++ function is as follows:

```
void incr (U *p);
```

Open bit-width parameters are always passed to by reference. A parameter whose corresponding argument is declared to be `inout` is passed to and from by reference. So there are two reasons for parameter `p` to be a pointer. It is a pointer to type `U` because its corresponding argument is a vector bit.

## **Example 6**

Consider the following C/C++ function declared in the Verilog source code:

```
extern void passbig1 (input bit [63:0] r8,
                      output bit [63:0] r9);
```

In this example, the function named `passbig1`, that does not return a value, has input and output arguments declared as `bit` and larger than 32 bits. The header of the C/C++ function is as follows:

```
void passbig (U *in, U *out)
```

In this example, the parameters `in` and `out` are pointers to type `U`. They are pointers because their corresponding arguments are larger than 32 bits and type `U` because their corresponding arguments are type `bit`.

### Example 7

Consider the following C/C++ function declared in the Verilog source code:

```
extern void passbig2 (input reg [63:0] r10,  
                      output reg [63:0] r11);
```

In this example, the function named `passbig2`, that does not return a value, has input and output arguments declared as non-scalar `reg`. The header of the C/C++ function is as follows:

```
void passbig2(vec32 *in, vec32 *out)
```

In this example, the parameters `in` and `out` are pointers to type `vec32`. They are pointers because their corresponding arguments are non-scalar type `reg`.

### Example 8

Consider the following C/C++ function declared in the Verilog source code:

```
extern void reality (input real real1, output real real2);
```

In this example, the function named `reality`, that does not return a value, has `input` and `output` arguments of declared type `real`. The header of the C/C++ function is as follows:

```
void reality (double *in, double *out)
```

In this example, the parameters `in` and `out` are pointers to type `double` because their corresponding arguments are type `real`.

## Using the `vc_hdrs.h` File

When you compile/elaborate your design for DirectC (by including the `+vc` elaboration/compile-time option), VCS writes a file in the current directory named `vc_hdrs.h`. In this file, there are `extern` declarations for all the C/C++ functions that you declared in your Verilog code. For example, if you compile/elaborate 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 copy from these declarations, you will always use the right type of parameter in your function header and you do not have to learn the rules for direct access. However, it is recommended to copy the `extern` declaration by VCS.

## 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-dimensional, two-dimensional, 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 values:

- 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.

Abstract access is useful when do not have to worry about the type you use for parameters, because you always use a special pointer type called `vc_handle`.

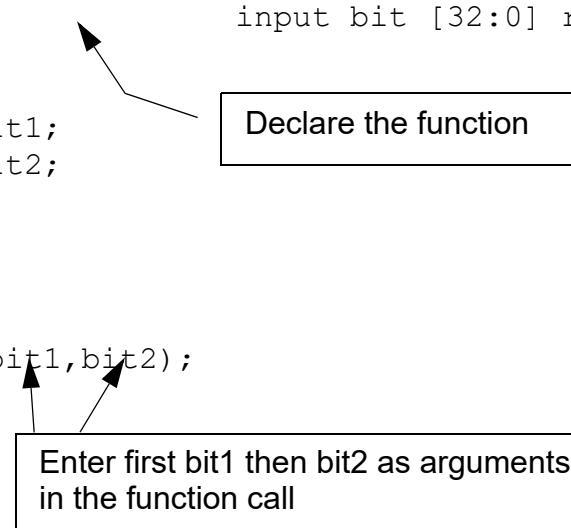
In abstract access, VCS creates a descriptor for every argument that you enter in the function call in your Verilog code. The `vc_handle` is a pointer to the descriptor for the argument. It is defined as follows:

```
typedef struct Veric_Descriptor *vc_handle;
```

## Using vc\_handle

In the function header, the `vc_handle` for a Verilog reg, bit, or memory is based on the order that you declare the `vc_handle` and the order that you entered its corresponding reg, bit, or memory in the function call in your Verilog code. For example, you could have declared the function and called it in your Verilog code as follows:

```
extern "A" void my_function( input bit [31:0] r1,  
                           input bit [32:0] r2);  
  
module dev1;  
reg [31:0] bit1;  
reg [32:0] bit2;  
initial  
begin  
.  
. .  
. .  
my_function(bit1,bit2);  
. .  
. .  
end  
endmodule
```



Declare the function

Enter first bit1 then bit2 as arguments in the function call

This is using abstract access, so VCS created descriptors for `bit1` and `bit2`. These descriptors contain information about their value, but also other information such as whether they are scalar or vector, and whether they are simulating in two-state 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 provides 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 naming convention used for access routines indicate 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, however, you must make sure that they can be overwritten in Verilog. Therefore, 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. For the summary of access routines, see “[Summary of Access Routines](#)”.

### **int vc\_isScalar(vc\_handle)**

Returns a 1 value if the `vc_handle` is for a one-bit reg or bit; returns a 0 value for a vector reg or bit or any memory including memories with scalar elements. For example:

```
extern "A" void scalarfinder(input reg r1,
                               input reg [1:0] r2,
                               input reg [1:0] array [1:0] r3,
                               input reg array [1:0] r4);

module top;
reg r1;
reg [1:0] r2;
reg [1:0] r3 [1:0];
reg r4 [1:0];
initial
scalarfinder(r1,r2,r3,r4);
endmodule
```

In this example, a routine named `scalarfinder` and input a scalar reg, a vector reg and two memories (one with scalar elements) are declared.

The declaration contains the "A" specification for abstract access. You typically include it in the declaration when other functions 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("\nil=%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 displays 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("\n i1=%d i2=%d i3=%d i4=%d\n\n",i1,i2,i3,i4);
}

```

The function displays 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("\n i1=%d i2=%d i3=%d i4=%d\n\n",i1,i2,i3,i4);
}

```

The function displays 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 displays 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 displays 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 displays the following:

```
The vc_handle to a 2state Vector is:  
h6=1
```

```
The vc_handles to 2state scalars or  
      memories and 4state are:  
h1=0 h2=0 h3=0 h4=0 h5=0 h7=0 h8=0
```

### **int vc\_width(vc\_handle)**

Returns the width of a vc\_handle. For example:

```
void memcheck_int(vc_handle h)
{
    int i;

    int mem_size = vc_arraySize(h);

    /* determine minimal needed width, assuming signed int */
    for (i=0; (1 << i) < (mem_size-1); i++) ;

    if (vc_width(h) < (i+1)) {
        printf("Register too narrow to be assigned %d\n",
               (mem_size-1));
        return;
    }

    for(i=0;i<8;i++) {
        vc_putMemoryInteger(h,i,i*4);
        printf("memput : %d\n",i*4);
    }
    for(i=0;i<8;i++) {
        printf("memget:: %d \n",vc_getMemoryInteger(h,i));
    }

}
```

### **int vc\_arraySize(vc\_handle)**

Returns the number of elements in a memory or multi-dimensional array. The previous example also shows the usage of vc\_arraySize().

### **scalar vc\_getScalar(vc\_handle)**

Returns the value of a scalar reg or bit. For example:

```
void rotate_scalars(vc_handle h1, vc_handle h2, vc_handle
```

```

h3)
{
    scalar a;

    a = vc_getScalar(h1);
vc_putScalar(h1, vc_getScalar(h2));
    vc_putScalar(h2, vc_getScalar(h3));
    vc_putScalar(h3, a);
    return;
}

```

### **void vc\_putScalar(vc\_handle, scalar)**

Passes the value of a scalar reg or bit to a `vc_handle` by reference. The previous example also shows the usage of `vc_putScalar()`.

### **char vc\_toChar(vc\_handle)**

Returns the 0, 1, x, or z character. For example:

```

void print_scalar(vc_handle h) {
    printf("%c", vc_toChar(h));
    return;
}

```

### **int vc\_toInteger(vc\_handle)**

Returns an int value for a `vc_handle` to a scalar bit or a vector bit of 32 bits or less. For a vector reg or a vector bit with more than 32 bits this routine returns a 0 value and displays the following warning message:

DirectC interface warning: 0 returned for 4-state value  
(`vc_toInteger`)

The following is an example of Verilog code that calls a C/C++ function that uses this routine:

```

extern void rout1 (input bit onebit, input bit [7:0] mobits);

module top;
reg /*2value*/ onebit;
reg /*2value*/ [7:0] mobits;
initial
begin
rout1(onebit,mobits);
onebit=1;
mobits=128;
rout1(onebit,mobits);
end
endmodule

```

Note 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 displays 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 you modify the C/C++ function in the previous example, it is as follows:

```
void vector_printer (vc_handle h)
{
vec32 b,*c;
c=vc_4stVectorRef(h);
b=*c;
printf("\n b is %s \n\n",vc_toStringF(h,'b'));
printf("\n b is %s \n\n",vc_toStringF(h,'o'));
printf("\n b is %s \n\n",vc_toStringF(h,'d'));
printf("\n b is %s \n\n",vc_toStringF(h,'x'));
}
```

This example now displays:

```
b is zx01zx01
```

b is XZX

b is X

b is XX

### **void vc\_putReal(vc\_handle, double)**

Passes by reference a real (double) value to a vc\_handle. For example:

```
void get_PI(vc_handle h)
{
    vc_putReal(h, 3.14159265);
}
```

### **double vc\_getReal(vc\_handle)**

Returns a real (double) value from a vc\_handle. For example:

```
void print_real(vc_handle h)
{
    printf("[print_real] %f\n", vc_getReal(h));
}
```

### **void vc\_putValue(vc\_handle, char \*)**

This function passes, by reference, through the vc\_handle, a value represented as a string containing the 0, 1, x, and z characters. For example:

```
extern void check_vc_putvalue(output reg [] r1);

module tester;
reg [31:0] r1;
```

```

initial
begin
check_vc_putvalue(r1);
$display("r1=%0b",r1);
$finish;
end
endmodule

```

In this example, the C/C++ function is declared in the Verilog code specifying that the function passes a value to a four-state reg (and, therefore, can hold X and Z values).

```

#include <stdio.h>
#include "DirectC.h"

void check_vc_putvalue(vc_handle h)
{
    vc_putValue(h,"10xz");
}

```

The `vc_putValue` routine passes the string "10xz" to the reg `r1` through the `vc_handle`. The Verilog code displays:

`r1=10xz`

### **void vc\_putValueF(vc\_handle, char \*, char )**

This function passes by reference, through the `vc_handle`, a value for which you specify a radix with the third parameter. The valid radices are 'b', 'o', 'd', and 'x'. For example the following Verilog code declares a function named `assigner` that uses this routine:

```

extern void assigner (output reg [31:0] r1,
                      output reg [31:0] r2,
                      output reg [31:0] r3,
                      output reg [31:0] r4);

module test;

```

```

reg [31:0] r1,r2,r3,r4;
initial
begin
assigner(r1,r2,r3,r4);
$display("r1=%0b in binary r1=%0d in decimal\n",r1,r1);
$display("r2=%0o in octal r2 =%0d in decimal\n",r2,r2);
$display("r3=%0d in decimal r3=%0b in binary\n",r3,r3);
$display("r4=%0h in hex r4= %0d in decimal\n\n",r4,r4);
$finish;
end
endmodule

```

**The following is the C/C++ function:**

```

#include <stdio.h>
#include "DirectC.h"

void assigner (vc_handle h1, vc_handle h2, vc_handle h3,
vc_handle h4)
{
vc_putValueF(h1,"10",'b');
vc_putValueF(h2,"11",'o');
vc_putValueF(h3,"10",'d');
vc_putValueF(h4,"aff",'x');
}

```

**The Verilog code displays the following:**

```

r1=10 in binary r1=2 in decimal

r2=11 in octal r2 =9 in decimal

r3=10 in decimal r3=1010 in binary

r4=aff in hex r4= 2815 in decimal

```

```
void vc_putPointer(vc_handle, void*)
void *vc_getPointer(vc_handle)
```

These functions pass a generic type of pointer or string to a `vc_handle` by reference. Do not use these functions for passing Verilog data (the values of Verilog signals). Use them for passing C/C++ data instead. `vc_putPointer` passes this data by reference to Verilog and `vc_getPointer` receives this data in a pass by reference from Verilog. You can also use these functions for passing Verilog strings.

For example:

```
extern void passback(output string, input string);
extern void printer(input pointer);

module top;
reg [31:0] r2;
initial
begin
passback(r2,"abc");
printer(r2);
end
endmodule
```

This Verilog code passes the string `abc` to the `passback` C/C++ function by reference, and that function passes it by reference to reg `r2`. The Verilog code then passes it by reference to the C/C++ function `printer` from reg `r2`.

```
passback(vc_handle h1, vc_handle h2)
{
vc_putPointer(h1, vc_getPointer(h2));
}

printer(vc_handle h)
{
printf("Procedure printer prints the string value %s\n\n",

```

```
    vc_getPointer (h)) ;  
}
```

The function named `printer` prints the following:

```
Procedure printer prints the string value abc
```

### **void vc\_StringToVector(char \*, vc\_handle)**

Converts a C string (a pointer to a sequence of ASCII characters terminated with a null character) into a Verilog string (a vector with 8-bit groups representing characters). For example:

```
extern "C" stringFullPath(string filename);  
// find full path to the file  
// C string obtained from C domain  
  
extern "A" void s2v(string, output reg[]);  
// string-to-vector  
// wrapper for vc_StringToVector().  
  
`define FILE_NAME_SIZE 512  
  
module Test;  
  reg [`FILE_NAME_SIZE*8:1] file_name;  
  // this file_name will be passed to the Verilog code that  
  // expects  
  // a Verilog-like string  
  .  
  .  
  .  
  initial begin  
    s2vFullPath("myStimulusFile"), file_name); // C-string to  
    // Verilog-string  
    // bits of 'file_name' represent now 'Verilog string'  
  end  
  .  
  .  
  .  
endmodule
```

The C code is as follows:

```
void s2v(vc_handle hs, vc_handle hv) {
    vc_StringToVector((char *)vc_getPointer(hs), hv);
}
```

### **void vc\_VectorToString(vc\_handle, char \*)**

Converts a vector value to a string value.

### **int vc\_getInteger(vc\_handle)**

Same as vc\_toInteger.

### **void vc\_putInteger(vc\_handle, int)**

Passes an int value by reference through a vc\_handle to a scalar reg or bit or a vector bit that is 32 bits or less. For example:

```
void putter (vc_handle h1, vc_handle h2, vc_handle h3,
vc_handle h4)
{
int a,b,c,d;
a=1;
b=2;
c=3;
d=9999999;

vc_putInteger(h1,a);
vc_putInteger(h2,b);
vc_putInteger(h3,c);
vc_putInteger(h4,d);
}
```

### **vec32 \*vc\_4stVectorRef(vc\_handle)**

Returns a vec32 pointer to a four-state vector. Returns NULL if the specified vc\_handle is not to a four-state vector reg. For example:

```

typedef struct vector_descriptor {
    int width; /* number of bits */
    int is4stte; /* TRUE/FALSE */
} VD;

void WriteVector(vc_handle file_handle, vc_handle a_vector)
{
    FILE *fp;
    int n, size;
    vec32 *v;
    VD vd;
    fp = vc_getPointer(file_handle);

    /* write vector's size and type */
    vd.is4state = vc_is4stVector(a_vector);
    vd.width = vc_width(a_vector);
    size = (vd.width + 31) >> 5; /* number of 32-bit chunks */
    /* printf("writing: %d bits, is 4 state: %d, #chunks:
       %d\n", vd.width, vd.is4state, size); */
    n = fwrite(&vd, sizeof(vd), 1, fp);
    if (n != 1) {
        printf("Error: write failed.\n");
    }

    /* write the vector into a file; vc_*stVectorRef
       is a pointer to the actual Verilog vector */
    if (vc_is4stVector(a_vector)) {
        n = fwrite(vc_4stVectorRef(a_vector), sizeof(vec32),
                   size, fp);
    } else {
        n = fwrite(vc_2stVectorRef(a_vector), sizeof(U),
                   size, fp);
    }
    if (n != size) {
        printf("Error: write failed for vector.\n");
    }
}

```

**U \*vc\_2stVectorRef(vc\_handle)**

Returns a U pointer to a bit vector that is larger than 32 bits. If you specify a short bit vector (32 bits or fewer) this routine returns a NULL value. For example:

In this example, the Verilog code declares a 32-bit vector bit, `r1`, and a 33-bit vector bit, `r2`. The values of both are passed to the C/C++ function `big_2state`.

When you 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 you pass bit vector `r2` because it has more than 32 bits. Note 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 does not have a null value, whatever it points to is passed to `u1`. If it does have a null value, the function prints a message about the short bit vector. In this example, you can expect it to display this message.

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. Because Verilog values are stored in 32-bit chunks in C/C++, the function finds out how many chunks are needed to store the long bit vector. It then loads one chunk at a time into `u2` and prints the chunk starting with the most significant bits. This function displays the following:

```
Short 2 state vector passed to up1  
width of h2 is 33  
the number of chunks needed for h2 is 2  
loading into u2 1 2
```

```
void vc_get4stVector(vc_handle, vec32 *)  
void vc_put4stVector(vc_handle, vec32 *)
```

Passes a four-state vector by reference to a `vc_handle` to and from an array in C/C++ function. `vc_get4stVector` receives the vector from Verilog and passes it to the array and `vc_put4stVector` passes the array to Verilog.

These routines work only if there are sufficient elements in the array for all the bits in the vector. The array must have an element for every 32 bit in the vector plus an additional element for any remaining bits. For example:

```
extern void copier (input reg [67:0] r1,  
                    output reg [67:0] r2);  
  
module top;  
  
reg [67:0] r1,r2;  
  
initial
```

In this example, there are two 68-bit regs. Values are assigned to all the bits of one reg and both of these regs are parameters to the C/C++ function named `copier`.

```
copier(vc_handle h1, vc_handle h2)
{
vec32 holder[3];
vc_get4stVector(h1,holder);
vc_put4stVector(h2,holder);
}
```

This function declares a `vec32` array of three elements named `holder`. It uses three elements because its parameters are 68-bit regs so you need an element for every 32 bits and one more for the remaining four bits.

The Verilog code displays the following:

```
void vc_get2stVector(vc_handle, U *)
void vc_put2stVector(vc_handle, U *)
```

Passes a two-state vector by reference to a `vc_handle` to and from an array in C/C++ function. `vc_get2stVector` receives the vector from Verilog and passes it to the array and `vc_put4stVector` passes the array to Verilog.

These routines, such as the `vc_get4stVector` and `vc_put4stVector` routines, work only if there are sufficient elements in the array for all the bits in the vector. The array must have an element for every 32 bit in the vector plus an additional element for any remaining bits.

The differences between these routines and the `vc_get4stVector` and `vc_put4stVector` routines are the type of data they pass, two-state or four-state simulation values, and the type you declare for the array in the C/C++ function.

### **UB \*vc\_MemoryRef(vc\_handle)**

Returns a pointer of type UB that points to a memory in Verilog. For example:

```
extern void mem_doer ( input reg [1:0] array [3:0]
                      memory1, output reg [1:0] array
                      [31:0] memory2);

module top;
reg [1:0] memory1 [3:0];
reg [1:0] memory2 [31:0];
initial
begin
memory1 [3] = 2'b11;
memory1 [2] = 2'b10;
memory1 [1] = 2'b01;
memory1 [0] = 2'b00;
mem_doer(memory1,memory2);
```

```

$display("memory2[31]=%0d",memory2[31]);
end
endmodule

```

In this example, two memories, one with 4 addresses, `memory1`, the other with 32 addresses, `memory2` are declared. You assign values to the addresses of `memory1`, and then pass both memories to the C/C++ function `mem_doer`.

```

#include <stdio.h>
#include "DirectC.h"

void mem_doer(vc_handle h1, vc_handle h2)
{
    UB *p1, *p2;
    int i;

    p1 = vc_MemoryRef(h1);
    p2 = vc_MemoryRef(h2);

    for ( i = 0; i < 8; i++) {
        memcpy(p2,p1,8);
        p2 += 8;
    }
}

```

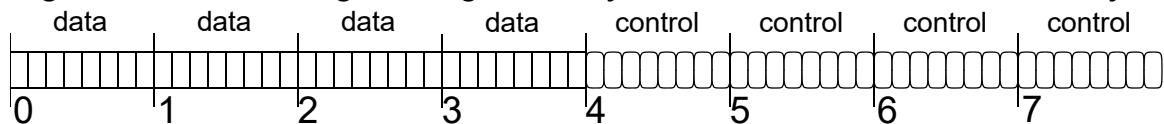
The purpose of the C/C++ function `mem_doer` is to copy the four elements in Verilog memory `memory1` into the 32 elements of `memory2`.

The `vc_MemoryRef` routines return pointers to the Verilog memories and the machine memory locations they point to are also pointed to by pointers `p1` and `p2`. Pointer `p1` points to the location of Verilog memory `memory1`, and `p2` points to the location of Verilog memory `memory2`.

The function uses a `for` loop to copy the data from Verilog memory `memory1` to Verilog memory `memory2`. It uses the standard `memcpy` function to copy a total of 64 bytes by copying eight bytes eight times.

This example copies a total of 64 bytes because each element of `memory2` is only two bits wide, however, for every eight bits in an element in machine memory there are two bytes, one for data and another for control. The bits in the control byte specify whether the data bit with a value of 0 is actually 0 or Z, or whether the data bit with a value of 1 is actually 1 or X.

*Figure 22-4 Storing Verilog Memory Elements in Machine Memory*



In an element in a Verilog memory, for each eight bits in the element there is a data byte and a control byte with an additional set of bytes for a remainder bit. Therefore, if a memory has 9 bits it would need two data bytes and two control bytes. If it has 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_doir( inout reg [25:1] array [3:0]
memory1);

module top;
reg [25:1] memory1 [3:0];
initial
begin
memory1 [0] = 25'b00000000xxxxxxxx11111111;
$display("memory1 [0] = %0b\n", memory1[0]);
mem_add_doir(memory1);
$display("\nmemory1 [3] = %0b", memory1[3]);
end
endmodule
```

In this example, there is a Verilog memory with four addresses, each element has 25 bits. This means that the Verilog memory needs eight bytes of machine memory because there is a data byte and a control byte for every eight bits in an element, with an additional data and control byte for any remainder bits.

In this example, in element 0 the 25 bits are assigned from right to left, eight 1 bits, eight unknown x bits, eight 0 bits, and one high impedance z bit.

```
#include <stdio.h>
#include "DirectC.h"

void mem_elem_doir(vc_handle h)
{
    U indx;
    UB *p1, *p2, t [8];
```

```

    indx = 0;
    p1 = vc_MemoryElemRef(h, indx);
    indx = 3;
    p2 = vc_MemoryElemRef(h, indx);
    memcpy(p2,p1,8);

    memcpy(t,p2,8);
    printf(" %d from t[0], %d from t[1]\n",
           (int)t[0], (int) t[1]);
    printf(" %d from t[2], %d from t[3]\n",
           (int)t[2], (int) t[3]);
    printf(" %d from t[4], %d from t[5]\n",
           (int)t[4], (int)t[5]);
    printf(" %d from t[6], %d from t[7]\n",
           (int)t[6], (int)t[7]);

}

```

C/C++ function `mem_elem_doir` uses the `vc_MemoryElemRef` routine to return pointers to addresses 0 and 3 in Verilog memory1 and pass them to UB pointers `p1` and `p2`. The standard `memcpy` routine then copies the eight bytes for address 0 to address 3.

The remainder of the function is additional code to show you data and control bytes. The eight bytes pointed to by `p2` are copied to array `t` and then the elements of the array are printed.

The combined Verilog and C/C++ code displays the following:

```

memory1 [0] = 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_door` passes the contents of the Verilog memory `memory1` element 0 to element 3.

In array `t`, the elements contain the following:

- [0] The data bits for the eight 1 values assigned to the element.
- [1] The data bits for the eight X values assigned to the element
- [2] The data bits for the eight 0 values assigned to the element
- [3] The data bit for the Z value assigned to the element
- [4] The control bits for the eight 1 values assigned to the element
- [5] The control bits for the eight X values assigned to the element
- [6] The control bits for the eight 0 values assigned to the element
- [7] The control bit for the Z value assigned to the element

### **scalar vc\_getMemoryScalar(vc\_handle, U indx)**

Returns the value of a one-bit memory element. For example:

```
extern void bitflipper (inout reg array [127:0] mem1);  
  
module test;  
reg mem1 [127:0];  
initial  
begin  
mem1 [0] = 1;  
$display("mem1[0]=%0d",mem1[0]);  
bitflipper(mem1);  
$display("mem1[0]=%0d",mem1[0]);  
$finish;  
end  
endmodule
```

In this example of Verilog code, a memory with 128 one-bit elements, assign a value to element 0 is declared, and display its value before and after you call a C/C++ function named `bitflipper`.

```
#include <stdio.h>
```

```

#include "DirectC.h"

void bitflipper(vc_handle h)
{
    scalar holder=vc_getMemoryScalar(h, 0);
    holder = ! holder;
    vc_putMemoryScalar(h, 0, holder);
}

```

In this example, a variable of type scalar, named `holder`, to hold the value of the one-bit Verilog memory element is declared. The routine `vc_getMemoryScalar` returns the value of the element to the variable. The value of `holder` is inverted and then the variable is included as a parameter in the `vc_putMemoryScalar` routine to pass the value to that element in the Verilog memory.

The Verilog code displays the following:

```

mem[0]=1
mem[0]=0

```

### **void vc\_putMemoryScalar(vc\_handle, U indx, scalar)**

Passes a value of type scalar to a Verilog memory element. You specify the memory by `vc_handle` and the element by the `indx` parameter. This routine is used in the previous example.

### **int vc\_getMemoryInteger(vc\_handle, U indx)**

Returns the integer equivalent of the data bits in a memory element whose bit-width is 32 bits or less. For example:

```

extern void mem_elem_halver (inout reg [] array [] memX);

module test;
reg [31:0] mem1 [127:0];
reg [7:0] mem2 [1:0];
initial

```

```

begin
mem1 [0] = 999;
mem2 [0] = 8'b1111xxxx;
$display("mem1[0]=%0d",mem1[0]);
$display("mem2[0]=%0d",mem2[0]);
mem_elem_halver(mem1);
mem_elem_halver(mem2);
$display("mem1[0]=%0d",mem1[0]);
$display("mem2[0]=%0d",mem2[0]);
$finish;
end
endmodule

```

In this example, when the C/C++ function is declared on your 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 you 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 you assign a value that included X values to the 0 element in memory `mem2`.

```

#include <stdio.h>
#include "DirectC.h"

void mem_elem_halver(vc_handle h)
{
int i = vc_getMemoryInteger(h, 0);
i = i/2;
vc_putMemoryInteger(h, 0, i);
}

```

This C/C++ function inputs the value of an element and then outputs half that value. The `vc_getMemoryInteger` routine returns the integer equivalent of the element you specify by `vc_handle` and

index number to an int variable `i`. The function halves the value in `i`. Then the `vc_putMemoryInteger` routine passes the new value by value to the specified memory element.

The Verilog code displays the following before the C/C++ function is called twice with the different memories as the arguments:

```
mem1[0]=999  
mem2[0]=X
```

Element `mem2[0]` has an X value because half of its binary value is x and the value is displayed with the `%d` format specification and, in this example, a partially unknown value is an unknown value. After the second call of the function, the Verilog code displays:

```
mem1[1]=499  
mem2[0]=127
```

This occurs because before calling the function, `mem1[0]` has 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]` has a value of 8'b1111xxxx, however, the data bits for the element would all be 1s (11111111). It is the control bits that specify 1 from x and this routine only deals with the data bits. Therefore, the `vc_getMemoryInteger` routine returns an integer value of 255 (the integer equivalent of the binary 11111111) to the C/C++ function, which is why the function outputs the integer value 127 to `mem2[0]`.

### **void vc\_putMemoryInteger(vc\_handle, U indx, int)**

Passes an integer value to a memory element that is 32 bits or fewer. You specify the memory by `vc_handle` and the element by the `indx` argument. This routine is used in the previous example.

### **void vc\_get4stMemoryVector(vc\_handle, U indx, vec32 \*)**

Copies the value in an Verilog memory element to an element in an array. This routine copies both the data and control bytes. It copies them into an array of type `vec32` which is defined as follows:

```
typedef struct { U c; U d; } vec32;
```

Therefore, type `vec32` has two members, `c` and `d`, for control and data information. This routine always copies to the 0 element of the array. For example:

```
extern void mem_elem_copier (inout reg [] array [] memX);

module test;
reg [127:0] mem1 [127:0];
reg [7:0] mem2 [64:0];
initial
begin
mem1 [0] = 999;
mem2 [0] = 8'b0000000z;
$display("mem1[0]=%0d",mem1[0]);
$display("mem2[0]=%0d",mem2[0]);
mem_elem_copier(mem1);
mem_elem_copier(mem2);
$display("mem1[32]=%0d",mem1[32]);
$display("mem2[32]=%0d",mem2[32]);
$finish;
end
endmodule
```

In the Verilog code, a C/C++ function is declared that is called twice. Note 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`. You must declare an array for this type, but as shown in this example, it is specified that it has only one element. The

`vc_get4stMemoryVector` routine copies the data from the Verilog memory element (in this example, specified as the 0 element) to the 0 element of the `vec32` array. It always copies to the 0 element. The `vc_put4stMemoryVector` routine copies the data from the `vec32` array to the Verilog memory element (in this case, element 32).

The call to `printf` is to describe 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 remaining 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 you declare the array to be of type `U` instead and you do not copy the control bytes, because there are none in two-state simulation.

**void vc\_put2stMemoryVector(vc\_handle, U indx, U \*)**

Copies Verilog data from a `U` array to a Verilog memory element. This routine is used in the previous example.

### **void vc\_putMemoryValue(vc\_handle, U idx, char \*)**

This routine works similar to 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 argument to specify the element (index) to which you want the routine to pass the value. For example:

```
#include <stdio.h>
#include "DirectC.h"

void check_vc_putvalue(vc_handle h)
{
    vc_putMemoryValue(h, 0, "10xz");
}
```

### **void vc\_putMemoryValueF(vc\_handle, U idx, char, char \*)**

This routine works similar to 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 similar to the `vc_toString` routine except that it used is for passing values to/from memory elements instead of to a reg or bit. You enter an argument to specify the element (index) whose value you want the routine to pass. For example:

```
extern void    memcheck_vec(inout reg[] array[]);

module top;
reg [0:7] mem[0:7];
integer i;

initial
begin
    for(i=0;i<8;i=i+1) begin
        mem[i] = 8'b00000111;
        $display("Verilog code says \"mem [%0d] = %0b\"", i,mem[i]);
    end

    memcheck_vec(mem);
end

endmodule
```

The C/C++ function that calls `vc_MemoryString` is as follows:

```
#include <stdio.h>
#include "DirectC.h"

void memcheck_vec(vc_handle h)
{
    int i;

    for(i= 0; i<8;i++) {
        printf("C/C++ code says \"mem [%d] is %s
\"\\n",i,vc_MemoryString(h,i));
    }
}
```

```
    }
}
```

The Verilog and C/C++ code display the following:

```
Verilog code says "mem [0] = 111"
Verilog code says "mem [1] = 111"
Verilog code says "mem [2] = 111"
Verilog code says "mem [3] = 111"
Verilog code says "mem [4] = 111"
Verilog code says "mem [5] = 111"
Verilog code says "mem [6] = 111"
Verilog code says "mem [7] = 111"
C/C++ code says "mem [0] is 00000111 "
C/C++ code says "mem [1] is 00000111 "
C/C++ code says "mem [2] is 00000111 "
C/C++ code says "mem [3] is 00000111 "
C/C++ code says "mem [4] is 00000111 "
C/C++ code says "mem [5] is 00000111 "
C/C++ code says "mem [6] is 00000111 "
C/C++ code says "mem [7] is 00000111 "
```

### **char \*vc\_MemoryStringF(vc\_handle, U indx, char)**

This routine works similar to the `vc_MemoryString` function except that you specify a radix with the third parameter. The valid radices are '`b`', '`o`', '`d`', and '`x`'. For example:

```
extern void    memcheck_vec(inout reg[] array[]);

module top;
reg [0:7] mem[0:7];

initial begin
mem[0] = 8'b00000111;
$display("Verilog code says \"mem[0]=%0b radix b\"",mem[0]);
$display("Verilog code says \"mem[0]=%0o radix o\"",mem[0]);
$display("Verilog code says \"mem[0]=%0d radix d\"",mem[0]);
$display("Verilog code says \"mem[0]=%0h radix h\"",mem[0]);
memcheck_vec(mem);
```

```
end  
  
endmodule
```

The C/C++ function that calls `vc_MemoryStringF` is as follows:

```
#include <stdio.h>  
#include "DirectC.h"  
  
void memcheck_vec(vc_handle h)  
{  
  
    printf("C/C++ code says \"mem [0] is %s radix b\"\n",
           vc_MemoryStringF(h, 0, 'b'));  
    printf("C/C++ code says \"mem [0] is %s radix o\"\n",
           vc_MemoryStringF(h, 0, 'o'));  
    printf("C/C++ code says \"mem [0] is %s radix d\"\n",
           vc_MemoryStringF(h, 0, 'd'));  
    printf("C/C++ code says \"mem [0] is %s radix x\"\n",
           vc_MemoryStringF(h, 0, 'x'));  
}
```

The Verilog and C/C++ code display the following:

```
Verilog code says "mem [0]=111 radix b"  
Verilog code says "mem [0]=7 radix o"  
Verilog code says "mem [0]=7 radix d"  
Verilog code says "mem [0]=7 radix h"  
C/C++ code says "mem [0] is 00000111 radix b"  
C/C++ code says "mem [0] is 007 radix o"  
C/C++ code says "mem [0] is 7 radix d"  
C/C++ code says "mem [0] is 07 radix x"
```

### **void vc\_FillWithScalar(vc\_handle, scalar)**

This routine fills all the bits or a reg, bit, or memory with all 1, 0, x, or z values (you can choose only one of these four values).

You specify the value with the scalar argument, which can be a variable of the scalar type. The scalar type is defined in the DirectC.h file as:

```
typedef unsigned char scalar;
```

You can also specify the value with integer arguments as follows:

|   |                    |
|---|--------------------|
| 0 | Specifies 0 values |
| 1 | Specifies 1 values |
| 2 | Specifies z values |
| 3 | Specifies x values |

If you declare a scalar type variable, enter it as the argument, and assign only the 0, 1, 2, or 3 integer values to it, they specify filling the Verilog reg, bit, or memory with the 0, 1, z, or x values.

You can use the following definitions from the DirectC.h file to specify these values:

```
#define scalar_0 0
#define scalar_1 1
#define scalar_z 2
#define scalar_x 3
```

The following Verilog and C/C++ code shows you how to use this routine to fill a reg and a memory using the following values:

```
extern void filler (inout reg [7:0] r1,
                     inout reg [7:0] array [1:0] r2,
                     inout reg [7:0] array [1:0] r3);
module top;
reg [7:0] r1;
reg [7:0] r2 [1:0];
reg [7:0] r3 [1:0];
initial
```

```

begin
$display("r1 is %0b",r1);
$display("r2[0] is %0b",r2[0]);
$display("r2[1] is %0b",r2[1]);
$display("r3[0] is %0b",r3[0]);
$display("r3[1] is %0b",r3[1]);
filler(r1,r2,r3);
$display("r1 is %0b",r1);
$display("r2[0] is %0b",r2[0]);
$display("r2[1] is %0b",r2[1]);
$display("r3[0] is %0b",r3[0]);
$display("r3[1] is %0b",r3[1]);
end
endmodule

```

**The C/C++ code for the function is as follows:**

```

#include <stdio.h>
#include "DirectC.h"

filler(vc_handle h1, vc_handle h2, vc_handle h3)
{
    scalar s = 1;
    vc_FillWithScalar(h1,s);
    vc_FillWithScalar(h2,0);
    vc_FillWithScalar(h3,scalar_z);
}

```

**The Verilog code displays the following:**

```

r1 is xxxxxxxx
r2[0] is xxxxxxxx
r2[1] is xxxxxxxx
r3[0] is xxxxxxxx
r3[1] is xxxxxxxx
r1 is 11111111
r2[0] is 0
r2[1] is 0
r3[0] is zzzzzzzz
r3[1] is zzzzzzzz

```

## **char \*vc\_argInfo(vc\_handle)**

Returns a string containing the information about the argument in the function call in your Verilog source code. For example, if you have the following Verilog source code:

```
extern void show(reg [] array []);
module tester;
reg [31:0] mem [7:0];
reg [31:0] mem2 [16:1];
reg [64:1] mem3 [32:1];
initial begin
    show(mem);
    show(mem2);
    show(mem3);
end
endmodule
```

Verilog memories `mem`, `mem2`, and `mem3` are all arguments to the function named `show`. If that function is defined as follows:

```
#include <stdio.h>
#include "DirectC.h"

void show(vc_handle h)
{
    printf("%s\n", vc_argInfo(h)); /* notice \n after the
string */
}
```

This routine displays the following:

```
input reg[0:31] array[0:7]
input reg[0:31] array[0:15]
input reg[0:63] array[0:31]
```

### **int vc\_Index(vc\_handle, U, ...)**

Internally, a multi-dimensional array is always stored as a one-dimensional array and this makes a difference in how it can be accessed. In order to avoid duplicating many of the previous access routines for multi-dimensional arrays, the access process is split into two steps. The first step, which this routine performs, is to translate the multiple indices into a single index of a linearized array. The second step is for another access routine to perform an access operation on the linearized array.

This routine returns the index of a linearized array or returns -1 if the U-type parameter is not an index of a multi-dimensional array or the vc\_handle parameter is not a handle to a multi-dimensional array of the reg data type.

```
/* get the sum of all elements from a 2-dimensional slice
   of a 4-dimensional array */
int getSlice(vc_handle vh_array, vc_handle vh_idx1,
vc_handle vh_idx2) {

    int sum = 0;
    int i1, i2, i3, i4, indx;

    i1 = vc_getInteger(vh_idx1);
    i2 = vc_getInteger(vh_idx2);
    /* loop over all possible indices for that slice */
    for (i3 = 0; i3 < vc_mdaSize(vh_array, 3); i3++) {

        for (i4 = 0; i4 < vc_mdaSize(vh_array, 4); i4++) {

            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-dimensional 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 22-10](#) summarizes all the access routines described in the previous section.

**Table 22-10 Summary of Access Routines**

| <b>Access Routine</b>                             | <b>Description</b>                                                                                                                                                                             |
|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int vc_isScalar(vc_handle)</code>           | Returns a 1 value if the <code>vc_handle</code> is for a one-bit reg or bit. It returns a 0 value for a vector reg or bit or any memory including memories with scalar elements.               |
| <code>int vc_isVector(vc_handle)</code>           | This routine returns a 1 value if the <code>vc_handle</code> is to a vector reg or bit. It returns a 0 value for a vector bit or reg or any memory.                                            |
| <code>int vc_isMemory(vc_handle)</code>           | This routine returns a 1 value if the <code>vc_handle</code> is to a memory. It returns a 0 value for a bit or reg that is not a memory.                                                       |
| <code>int vc_is4state(vc_handle)</code>           | This routine returns a 1 value if the <code>vc_handle</code> is to a reg or memory that simulates with four states. It returns a 0 value for a bit or a memory that simulates with two states. |
| <code>int vc_is2state(vc_handle)</code>           | This routine does the opposite of the <code>vc_is4state</code> routine.                                                                                                                        |
| <code>int vc_is4stVector(vc_handle)</code>        | This routine returns a 1 value if the <code>vc_handle</code> is to a vector reg. It returns a 0 value if the <code>vc_handle</code> is to a scalar reg, scalar or vector bit, or to a memory.  |
| <code>int vc_is2stVector(vc_handle)</code>        | This routine returns a 1 value if the <code>vc_handle</code> is to a vector bit. It returns a 0 value if the <code>vc_handle</code> is to a scalar bit, scalar or vector reg, or to a memory.  |
| <code>int vc_width(vc_handle)</code>              | Returns the width of a <code>vc_handle</code> .                                                                                                                                                |
| <code>int vc_arraySize(vc_handle)</code>          | Returns the number of elements in a memory.                                                                                                                                                    |
| <code>scalar vc_getScalar(vc_handle)</code>       | Returns the value of a scalar reg or bit.                                                                                                                                                      |
| <code>void vc_putScalar(vc_handle, scalar)</code> | Passes the value of a scalar reg or bit to a <code>vc_handle</code> by reference.                                                                                                              |
| <code>char vc_toChar(vc_handle)</code>            | Returns the 0, 1, x, or z character.                                                                                                                                                           |
| <code>int vc_toInteger(vc_handle)</code>          | Returns an int value for a <code>vc_handle</code> to a scalar bit or a vector bit of 32 bits or less.                                                                                          |
| <code>char *vc_toString(vc_handle)</code>         | Returns a string that contains the 1, 0, x, and z characters.                                                                                                                                  |

| Access Routine                                                                  | Description                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| char<br>*vc_toStringF(vc_handle,<br>char)                                       | Returns a string that contains the 1, 0, x, and z characters and allows you to specify the format or radix for the display. The char parameter can be 'b', 'o', 'd', or 'x'.                                                                                                                                                                                                                               |
| void<br>vc_putReal(vc_handle,<br>double)                                        | Passes by reference a real (double) value to a vc_handle.                                                                                                                                                                                                                                                                                                                                                  |
| double<br>vc_getReal(vc_handle)                                                 | Returns a real (double) value from a vc_handle.                                                                                                                                                                                                                                                                                                                                                            |
| void<br>vc_putValue(vc_handle,<br>char *)                                       | This function passes, by reference through the vc_handle, a value represented as a string containing the 0, 1, x, and z characters.                                                                                                                                                                                                                                                                        |
| void<br>vc_putValueF(vc_handle,<br>char, char *)                                | This function passes by reference through the vc_handle a value for which you specify a radix with the third parameter. The valid radices are 'b', 'o', 'd', and 'x'.                                                                                                                                                                                                                                      |
| void<br>vc_putPointer(vc_handle,<br>void*)<br>void<br>*vc_getPointer(vc_handle) | These functions pass, by reference to a vc_handle, a generic type of pointer or string. Do not use these functions for passing Verilog data (the values of Verilog signals). Use it for passing C/C++ data. vc_putPointer passes this data by reference to Verilog and vc_getPointer receives this data in a pass by reference from Verilog. You can also use these functions for passing Verilog strings. |
| void<br>vc_StringToVector(char *,<br>vc_handle)                                 | Converts a C string (a pointer to a sequence of ASCII characters terminated with a null character) into a Verilog string (a vector with 8-bit groups representing characters).                                                                                                                                                                                                                             |
| void<br>vc_VectorToString(vc_handl<br>e, char *)                                | Converts a vector value to a string value.                                                                                                                                                                                                                                                                                                                                                                 |
| int<br>vc_getInteger(vc_handle)                                                 | Same as vc_toInteger.                                                                                                                                                                                                                                                                                                                                                                                      |
| void<br>vc_putInteger(vc_handle,<br>int)                                        | Passes an int value by reference through a vc_handle to a scalar reg or bit or a vector bit that is 32 bits or less.                                                                                                                                                                                                                                                                                       |

| Access Routine                                                                                   | Description                                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| vec32<br>*vc_4stVectorRef(vc_handle)<br>)                                                        | Returns a vec32 pointer to a four state vector.<br>Returns NULL if the specified vc_handle is not to a four-state vector reg.                                                                                                  |
| U<br>*vc_2stVectorRef(vc_handle)<br>)                                                            | This routine returns a U pointer to a bit vector that is larger than 32 bits. If you specify a short bit vector (32 bits or fewer), this routine returns a NULL value.                                                         |
| void<br>vc_get4stVector(vc_handle,<br>vec32 *)<br>void<br>vc_put4stVector(vc_handle,<br>vec32 *) | Passes a four-state vector by reference to a vc_handle to and from an array in C/C++ function.<br>vc_get4stVector receives the vector from Verilog and passes it to the array.<br>vc_put4stVector passes the array to Verilog. |
| void<br>vc_get2stVector(vc_handle,<br>U *)<br>void<br>vc_put2stVector(vc_handle,<br>U *)         | Passes a two state vector by reference to a vc_handle to and from an array in C/C++ function.<br>vc_get2stVector receives the vector from Verilog and passes it to the array.<br>vc_put2stVector passes the array to Verilog.  |
| UB<br>*vc_MemoryRef(vc_handle)                                                                   | Returns a pointer of type UB that points to a memory in Verilog.                                                                                                                                                               |
| UB<br>*vc_MemoryElemRef(vc_handle, U indx)                                                       | Returns a pointer to an element (word, address or index) of a Verilog memory. You specify the vc_handle of the memory and the element.                                                                                         |
| scalar<br>vc_getMemoryScalar(vc_handle, U indx)                                                  | Returns the value of a one-bit memory element.                                                                                                                                                                                 |
| void<br>vc_putMemoryScalar(vc_handle, U indx, scalar)                                            | Passes a value, of type scalar, to a Verilog memory element. You specify the memory by vc_handle and the element by the indx parameter.                                                                                        |
| int<br>vc_getMemoryInteger(vc_handle, U indx)                                                    | Returns the integer equivalent of the data bits in a memory element whose bit-width is 32 bits or less.                                                                                                                        |
| void<br>vc_putMemoryInteger(vc_handle, U indx, int)                                              | Passes an integer value to a memory element that is 32 bits or fewer. You specify the memory by vc_handle and the element by the indx parameter.                                                                               |

| Access Routine                                              | Description                                                                                                                                                                                                                              |
|-------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void<br>vc_get4stMemoryVector(vc_handle, U indx, vec32 *)   | Copies the value in an Verilog memory element to an element in an array. This routine copies both the data and control bytes. It copies them into an array of type vec32.                                                                |
| void<br>vc_put4stMemoryVector(vc_handle, U indx, vec32 *)   | Copies Verilog data from a vec32 array to a Verilog memory element.                                                                                                                                                                      |
| void<br>vc_get2stMemoryVector(vc_handle, U indx, U *)       | Copies the data bytes, but not the control bytes, from a Verilog memory element to an array in your C/C++ function.                                                                                                                      |
| void<br>vc_put2stMemoryVector(vc_handle, U indx, U *)       | Copies Verilog data from a U array to a Verilog memory element.                                                                                                                                                                          |
| void<br>vc_putMemoryValue(vc_handle, U indx, char *)        | This routine works like the vc_putValue routine except that it is for passing values to a memory element instead of to a reg or bit. You enter an parameter to specify the element (index) you want the routine to pass the value to.    |
| void<br>vc_putMemoryValueF(vc_handle, U indx, char, char *) | This routine works like the vc_putValueF routine except that it is for passing values to a memory element instead of to a reg or bit. You enter an parameter to specify the element (index) you want the routine to pass the value to.   |
| char<br>*vc_MemoryString(vc_handle, U indx)                 | This routine works like the vc_toString routine except that it is for passing values to from memory element instead of to a reg or bit. You enter an parameter to specify the element (index) you want the routine to pass the value of. |
| char<br>*vc_MemoryStringF(vc_handle, U indx, char)          | This routine works like the vc_MemoryString function except that you specify a radix with the third parameter. The valid radices are 'b', 'o', 'd', and 'x'.                                                                             |
| void<br>vc_FillWithScalar(vc_handle, scalar)                | This routine fills all the bits or a reg, bit, or memory with all 1, 0, x, or z values (you can choose only one of these four values).                                                                                                   |

| Access Routine                       | Description                                                                                                                                                                                                                                                                                                                                                                                  |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| char<br>*vc_argInfo(vc_handle)       | Returns a string containing the information about the parameter in the function call in your Verilog source code.                                                                                                                                                                                                                                                                            |
| int vc_Index(vc_handle, U,<br>...)   | 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.                                                                                                                                                                   |
| int vc_Index2(vc_handle, U,<br>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,<br>U, U) | Specialized version of vc_Index() where the two U parameters are the indexes in a three-dimensional array.                                                                                                                                                                                                                                                                                   |
| U vc_mdaSize(vc_handle, U)           | 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` elaboration/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_handle` 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:

|                               | <b>no +abstract suffix</b> | <b>include the +abstract suffix</b> |
|-------------------------------|----------------------------|-------------------------------------|
| extern<br>(no mode specified) | direct access              | abstract access                     |
| extern "A"                    | abstract access            | abstract access                     |
| extern "C"                    | direct access              | direct access                       |

## Specifying the DirectC.h File

The C/C++ functions need the `DirectC.h` file to use abstract access. This file is located in `$VCS_HOME/include` (and there is a symbolic link to it at `$VCS_HOME/platform/lib/DirectC.h`). You need to instruct VCS where to look for it. You can accomplish this in the following three ways:

- Copy the `$VCS_HOME/include/DirectC.h` file to your current directory. VCS always looks for this file in your current directory.
- Establish a link in the current directory to the `$VCS_HOME/include/DirectC.h` file.
- Include the `-CC` option as follows:

```
-CC "-I$VCS_HOME/include"
```

---

## Extended BNF for External Function Declarations

A partial EBNF specification for external function declaration is as follows:

```
source_text ::= description +
description ::= module | user_defined_primitive |
extern_function_declaration
extern_function_declaration ::= extern access_mode
extern_func_type extern_function_name (
list_of_extern_func_args ? ) ;
access_mode ::= ( "A" | "C" ) ?
```

**Note:**

If access mode is not specified, then the command line option +abstract rules; default mode is "C".]

```
extern_func_type ::= void | reg | bit |
DirectC_primitive_type | bit_vector_type
bit_vector_type ::= bit [ constant_expression :
constant_expression ]
list_of_extern_func_args ::= extern_func_arg
( , extern_func_arg ) *
extern_func_arg ::= arg_direction ? arg_type
optional_arg_name ?
```

**Note:**

Argument direction (that is, input, output, inout) applies to all arguments that follow it until the next direction occurs; the default direction is input.

```
arg_direction ::= input | output | inout
arg_type ::= bit_or_reg_type | array_type |
DirectC_primitive_type
bit_or_reg_type ::= ( bit | reg ) optional_vector_range ?
optional_vector_range ::= [ ( constant_expression :
constant_expression ) ? ]
array_type ::= bit_or_reg_type array [ ( constant_expression :
constant_expression ) ? ]
```

*DirectC\_primitive\_type ::= int | real | pointer | string*  
**In this specification, `extern_function_name` and  
`optional_arg_name` are user-defined identifiers.**



# 23

## Support for VHDL 2002 and 2008

---

VCS supports the following VHDL standards:

- “[VHDL 2002 Protected Type](#)” - VHDL 2002 is supported with the `-vhdl102` switch or `VHDL_MODE=vhdl102` setup variable (which should be specified in the `synospys_sim.setup` file).
- “[VHDL 2008 Constructs](#)” - VHDL 2008 is supported with the `-vhdl108` switch or `VHDL_MODE=vhdl108` setup variable (which should be specified in the `synospys_sim.setup` file).

---

## VHDL 2002 Protected Type

Protected type is supported by VCS. For more information on protected type, refer to clause 3.5 in the VHDL 2002 LRM.

---

### Use Model

Protected type is supported with the following options:

`-vhdl02` or `-vhdl08 (vhdlan)` switch.

OR

`VHDL_MODE = vhdl08` or `VHDL_MODE = vhdl02` setup variable.

---

### Limitations of VHDL 2002 Protected Type

VHDL 2002 protected type works with the following limitations:

- VHPI interface to protected types is not supported.
- Protected type in subprograms is not supported.
- UCLI support is not yet implemented.

---

## VHDL 2008 Constructs

VCS supports the following VHDL 2008 constructs that are described in this section. To enable the VHDL 2008 constructs, specify the `-vhdl08` option in the `vhdlan` command line or set the `VHDL_MODE=vhdl08` variable in the `synopsys_sim.setup` file.

- “Array Types and Operators”
- “Adding Comments”
- “Use Clause and Aliases”
- “Support for Bit String Literals”
- “Support for TO\_STRING Conversion”
- “Support for External Names”
- “Specifying The all Keyword in the Process Sensitivity List”
- “Support for Logical Unary Reduction Operator”
- “Support for Matching Relational Operators for Bit and std\_ulogic”
- “Including Non-Static Expressions in Port Map”
- “Standard Environment Package”
- “Package Declaration and Instantiations”
- “Referencing Interface Lists”
- “Overriding the Value Assigned to a Signal”
- “Matching Case Statements”
- “Conditional Elaboration”
- “Condition Operator in an Expression”
- “Reading Output Port”
- “2008 IEEE Packages”
- “Resolved Elements”
- “Conditional and Selected Assignments”

- “Context Declaration”
  - “Improved I/O”
  - “Support for Implicitly Constrained Array Elements”
  - “Support for Unconstrained Element Types”
  - “Support for Enhanced Generics in Entity Interfaces”
  - “Support for Slices in Array Aggregates”
  - “Support for Type Conversion in VHDL 2008”
  - “Support for Case Expression Subtype”
  - “Support for Subtypes of Ports and Parameters”
  - “Support for Static Composite Expressions”
- 

## Array Types and Operators

As defined in the *VHDL 2008 LRM*, section 5.3.2.3 and 5.3.2.4, VCS supports predefined array types, STRING, BOOLEAN\_VECTOR, BIT\_VECTOR, INTEGER\_VECTOR, REAL\_VECTOR, and TIME\_VECTOR. These data types are defined in STANDARD package. For more information about these data types and its definition, see the *IEEE Std. VHDL 2008 LRM*.

Example,

```
entity test1 is
port(w1 : out integer_vector(2 downto 0);
      x1 : out boolean_vector(2 downto 0);
      y1 : out real_vector(2 downto 0);
      t1 : out time_vector(2 downto 0);
      );
end test1;
```

```

architecture ar_test1 of test1 is
begin
    w1<=(10,20,30);
    x1<=(TRUE,FALSE,TRUE);
    y1<=(123.234,9876.12345,12.1);
    t1 <= (10 ns, 20 ns, 60 ns);
end ar_test1;

```

---

## Adding Comments

As defined in the *IEEE Std. VHDL 2008 LRM*, a comment is either a single-line comment or a delimited comment. A single-line comment starts with two adjacent hyphens and extends up to the end of the line. A delimited comment starts with a solidus (slash) character immediately followed by an asterisk character and extends up to the first subsequent occurrence of an asterisk character immediately followed by a solidus character.

### Example1 - Single-line Comment

```

entity full_adder is
    port (A, B, CIN: in std_logic;sum, cout: out std_logic);
end full_adder;

architecture full_adder of full_adder is
begin
    SUM <= A xor B xor CIN; -- My comment for SUM
    COUT <= (A and B) or (B and CIN) or (CIN and A);
                                --My comment for CARRY
end full_adder;

```

### Example 2 - Delimited Comment

```

entity full_adder is
    port (A, B, CIN: in std_logic;sum, cout: out std_logic);
end full_adder;

```

```

architecture full_adder of full_adder is
begin
    SUM <= A xor B xor CIN; /* My comment for SUM */
    COUT <= (A and B) or (B and CIN) or (CIN and A);
                                /* My comment for CARRY */
end full_adder;

```

---

## Use Clause and Aliases

VCS supports use clauses as defined in the *IEEE Std. VHDL 1076-2008 LRM*, section 12.4. A use clause achieves direct visibility of declarations that are visible by selection. For information about the syntax and functionality, see the *IEEE Standard VHDL 1076-2008 LRM*.

Example,

```

package P is
    type color is (BLUE, GREEN, RED, YELLOW);
end package;
use work.P.color;
entity e is
end;
architecture arch of e is
    signal A1 : color ;
begin
    A1 <= GREEN ; // Now enum literal will be visible with
                    the use of type identifier with use
end architecture;

```

VCS also supports alias declarations as defined in VHDL 1076-2008 LRM, section 6.6. An alias declaration declares an alternate name for an existing named entity. For information about the syntax and functionality, see the *IEEE Standard VHDL 1076-2008 LRM*.

Example,

```

package P1 is
    type alu_type1 is (add,sub,mul,div);
end package;
use work.P1.alu_type1;
package P2 is
    alias alu_type is alu_type1;
end package;
use work.P2.alu_type;
entity e is
end;
architecture arch of e is
    signal a1, a2 : alu_type := add;
begin
    process
begin
    a1 <= a2;
    wait;
end process;
end;

```

## Support for Bit String Literals

As defined in the *IEEE Standard VHDL 1076-2008 LRM*, section 15.8, a bit string literal is formed by a sequence of characters (possibly none) enclosed between two quotation marks used as bit string brackets, preceded by a base specifier. The bit string literal can also be preceded by an integer specifying the length of the value represented by the bit string literal.

`bit_string_literal ::= [ integer ] base_specifier "[ bit_value ]"`

Where,

`bit_value ::= graphic_character { [ underline ] graphic_character }`  
`base_specifier ::= B | O | X | UB | UO | UX | SB | SO | SX | D`

For more information about the rules and definition of bit string literals, see the *IEEE Standard VHDL 1076-2008 LRM*.

VCS supports VHDL bit string literals to specify a value for vector of 0 and 1 elements in binary, octal, or hexadecimal form. You can use the bit string literals by adding the `-vhdl08` option in `vhdlan` command line. With this enhancement, you can:

- Specify elements other than 0 and 1:
  - In a binary literal, any non-bit character represents itself
  - In an octal literal, any non-octal-digit character is expanded to three occurrences of the character in the vector value
  - In a hexadecimal literal, any non-hexadecimal digit character is expanded to four occurrences of the character in the vector value
  - Embed underscore (\_) for readability as it will not be expanded.
- Specify the exact length of the vector represented by a literal
- Specify if the literal represents an unsigned or signed number
- Specify a vector value in the decimal form

VHDL 2008 allows you to specify the length of a bit string literal. The length of a bit string literal is the length of its string literal value. If a bit string literal includes the integer immediately preceding the base specifier, the length of the bit string literal is the value of the integer. Else, the length is the number of characters in the expanded bit value. The string literal value is obtained by adjusting the expanded bit value to the length of the bit string literal.

Examples,

```
6x"a" is equivalent to b"001010"  
32b"10" is equivalent to  
    b"00000000_00000000_00000000_00000010"  
7o"123" is equivalent to b"1010011"
```

VHDL 2008 adds signed bit string literals, which permits you to represent negative values in fixed-width representations. In a signed bit string literal, the leftmost element of the expanded bit value is duplicated, if the specified length is longer. If the specified length is shorter, elements might be truncated, if they are the same as the new leftmost element. For unsigned literals, any truncated elements must be 0.

Examples,

```
16so"6" is equivalent to b"11111111_11111110"  
6sx"fe" is equivalent to b"111110"  
8sb"100111000" is an error
```

VHDL 2008 allows you to specify bit string literals in decimal form with the expanded bit value having sufficient bits to contain the largest power of two (not greater than the value). Decimal bit string literals are always unsigned.

Examples,

```
d"47" is equivalent to b"101111"  
16d"25" is equivalent to b"00000000_00011001"  
7d"255" is an error
```

Note:

As the length of the expanded bit value is chosen so that the leftmost element is always 1, any specified length must not be less than that.

VHDL 2008 permits characters other than underscores and extended digits. This permits you to specify metalogical values for bit string literals to use them as aggregates for arrays of `std_ulogic`. If the radix is octal or hexadecimal, the character is duplicated as necessary.

Examples,

```
8x"Z5" is equivalent to "ZZZZ0101"  
7so"X3" is equivalent to "XXXX011"  
6b"ABC" is equivalent to "000ABC"  
    (not valid as an array of std_logic but still  
     permitted by the language)
```

---

## Support for TO\_STRING Conversion

VCS supports the `TO_STRING` (`x`) predefined function, as defined in the *VHDL 2008 LRM*, sections 5.2.6 and 5.7. The `TO_STRING` (`x`) function works with any scalar type and any one-dimensional array type where the element type is an enumerated type all of whose literals are character literals.

VCS supports the following VHDL 2008 overloaded versions of the `TO_STRING` predefined function:

- `TO_STRING (value: time; unit: time)` — use `unit` instead of the resolution limit
- `TO_STRING (value: real; digits: natural)` — if `digits` are greater than 0, then it controls the number of digits after decimal, and suppresses the exponent
- `TO_STRING (value: real; format: string)` — use a restricted subset of `printf(3)` conversions (no length modifiers, asterisks, or text other than a conversion)

Additionally, for all bit-based array types, you can define the octal and the hexadecimal string conversion functions as follows:

- TO\_OSTRING [*BitArrayType* **return** string]
- TO\_HSTRING [*BitArrayType* **return** string]

You can use the TO\_STRING (x) functions by adding the -vhdl108 option in vhdlan command line.

The `to_string/to_ostring/to_hstring` literals are built-in functions and following are some of the conversion examples:

| TO_STRING                                                     | Expected Output |
|---------------------------------------------------------------|-----------------|
| <code>to_string(bit('1'))</code>                              | “1”             |
| <code>to_string(1.5 ms, 2ns)</code><br>(time resolution 1 ns) | “750000 ns”     |
| <code>to_string(52.5, “%5.2f”)</code>                         | “52.50”         |
| <code>to_ostring(bit_vector("111111"))</code>                 | “77”            |

## Example

The following example illustrates the usage of `to_string` with `bit_vector` and `std_logic_vector`:

```
entity E is
end entity E;
architecture demo of e is
    signal s1 : string(1 to 5) := to_string(bit_vector'("10111"));
    signal s2 : string(1 to 5) := to_string(std_logic_vector'("1ZU0X"));
begin
process
begin
    report to_string(s1);
    report to_string(s2);
    wait;
end process;
end architecture demo;
```

---

## Support for External Names

VCS supports VHDL External Names, which is introduced in the 1076-2008 - IEEE Standard as specified in the section 8.7 of VHDL Language Reference Manual.

VHDL had never provided an easy way to access a signal or a shared variable inside a design hierarchy from any other levels of design/verification environment. With the introduction of External Names, this limitation is no longer valid and you can easily access a constant, a signal or a shared variable residing in any other VHDL design hierarchy.

In tune with the enhancements in the *IEEE Standard VHDL 1076-2008 LRM*, VCS also supports the *External Names* feature. For more information about *External Names* feature, see the *IEEE Standard VHDL 1076-2008 LRM*.

### Example

Leaf module:

```
entity e is
end entity;
architecture a of e is
signal s : std_logic :='1';
begin
end a;
```

Top module instantiating the Leaf module:

```
entity tb is
end entity;
architecture arch of tb is
signal s1 : std_logic ;
begin
```

```

inst : entity work.e; // instance
process is
begin
    s1 <= << signal .tb.inst.s:std_logic>>;//external name
    wait for 1 ns;
    wait;
end process;
end arch;

```

To run the example, use the following command:

```
% vhdlan -vhdl08 test.vhd
% vcs tb -R
```

## **Specifying The all Keyword in the Process Sensitivity List**

VCS supports the usage of the VHDL 2008 `all` keyword in the process sensitivity list as defined in the section 11.3 of *IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-2008)*. Use the `-vhdl08` option to enable this feature.

As per the VHDL 2008 LRM, you can use the `all` keyword in the process sensitivity list, as shown in the following command:

```
process_sensitivity_list ::= all | sensitivity_list
```

For example, consider the following process that has signals `s1` and `si` in the sensitivity list:

```

process (s1, si) is
begin
    s4(si) <= s1;
end process;
```

When you use process (all), the preceding code change as follows:

```
process (all) is
begin
    s4(si) <= s1;
end process;
```

---

## Support for Logical Unary Reduction Operator

VCS supports the use of unary logical operators and, or, nand, nor, xor, and xnor and are referred to as logical reduction operators, which is introduced in section 9.2.2 of *IEEE Standard VHDL 2008 Language Reference Manual*. Each such logical reduction operator can be applied for any one-dimensional array type whose element type is bit or boolean and this produces a bit or boolean result, respectively.

Example,

```
entity test is
end;

architecture arch of test is

signal bv1 : bit_vector(2 downto 0);
signal bs1,bs2,bs3,bs4,bs5,bs6,bs7,bs8 : bit;

begin
process
begin
    wait for 1 ns;
    bv1 <= "010";
    wait for 1 ns;
    bs2 <= and bv1;
    bs3 <= or bv1;
    bs4 <= nand bv1;
```

```

    bs5 <= nor bvl;
    bs6 <= xor bvl;
    bs7 <= xnor bvl;
    wait;
end process;
end;

```

---

## Support for Matching Relational Operators for Bit and std\_ulogic

VCS supports the usage of matching relational operators `?=`, `?/=`, `?<`, `?<=`, `?>`, and `?>=` that returns bit or `std_ulogic` results, which is introduced in section 9.2.3 of *IEEE Standard VHDL 2008 Language Reference Manual*.

The result type of each matching relational operator is same as the type of the operands (for scalar operands) or the element type of the operands (for array operands). For more information about matching relational operators, see the *IEEE Std. VHDL 2008 LRM*.

### Example

```

entity test is
end;

architecture arch of test is

signal b1,b2,b3,b4,b5,b6,b7,b8 :std_ulogic;

begin
process
begin
    wait for 1 ns;
    b1 <= '1';
    b2 <= '1';
    wait for 1 ns;
    b3 <= b1 ?= b2;

```

```

b4 <= b1 ?/= b2;
b5 <= b1 ?< b2;
b6 <= b1 ?<= b2;
b7 <= b1 ?> b2;
b8 <= b1 ?>= b2;
    wait;
end process;
end;

```

Output of this example is as follows:

```

value of b3 is '1';
value of b4 is '0';
value of b5 is '0';
value of b6 is '1';
value of b7 is '0';
value of b8 is '1';

```

## Including Non-Static Expressions in Port Map

VCS supports writing an expression in a port map to include non-static expressions involving the values of signals, as defined in section 6.5.7.3 of *IEEE Std. VHDL-2008 LRM*.

This enables you to include functional logic in a port map, and avoid the need to express the logic with a separate assignment statement and an intermediate signal. If the expression is not static, the port association is defined to be equivalent to association with an anonymous signal that is the target of a signal assignment with the expression on the right-hand side.

For example,

Consider the entity `test` with port “`a`”:

```

library ieee;
use ieee.std_logic_1164.all;

```

```

entity test is port(a : in std_logic);
end entity;

architecture arch of test is
begin
end arch;

```

And the entity top, where entity test is instantiated with port “a” having an expression in port mapping:

```

library ieee;
use ieee.std_logic_1164.all;

entity top is port(    a1 : in std_logic;
                      b1 : in std_logic);
end entity;

```

Because of this enhancement, you can include the non-static signal expression in a port map, in which the architecture is simpler as shown in t he following code:

```

architecture arch of top is
component test
    port (a : in std_logic);
end component;
begin
    vh_inst : test port map ( a => a1 and b1 );
end arch;

```

**Command Line:**

```

% vhdlan -vhdl108 basic_sig_exp.vhd
% vcs top
% ./simv

```

---

## Standard Environment Package

As defined in the section 16.5 of *IEEE Std. VHDL 2008 LRM*, package ENV contains declarations that provide a VHDL interface to the host environment. You can use the `std.env.all` package to provide a VHDL interface.

Example,

```
use IEEE.std_logic_1164.all;
use std.env.all; // Use the env package
entity e is
    port (clk : in bit );
end entity;
architecture arch of e is
begin
check : process(clk)
begin
    if (clk'event and clk = '1') then
        stop; // Call the stop
    end if;
end process;
end architecture;
```

Note:

Execution of STOP procedure causes the simulator to stop.

---

## Package Declaration and Instantiations

Generics provide a channel for communicating information to a block, a package, or a subprogram from its environment. VCS supports generic classes (constants, types, packages, and subprograms) in only *packages* as defined in the following sections.

As defined in the VHDL 2008 LRM, section 13.1 and 13.2, you can declare packages as design units. They are separately analyzed into a design library, and can be referenced by any other design unit that names the library. Thus, they are applicable only in the global scope. Using VHDL-2008, you can declare the packages locally within the declarative region of an entity, architecture, block, process, subprogram, protected type body, or enclosing package. However, VCS supports the following generics only in a *global scope*.

- **Generic classes:** the following generic classes are supported in a package:
  - Generic constants
  - Generic types
  - Generic packages
  - Generic subprograms

## Use Model

To use generics in your design source code, specify the `-vhdl08` option in the `vhdlan` command line, or enable VHDL 2008 using the `VHDL_MODE=vhdl08` variable in the `synopsys_sim.setup` file.

## Example

The following example includes packages with all the supported generic constants and generic types:

```
library ieee;
use ieee.std_logic_1164.all;

package uninst_pack is
  generic (g1 : integer);
  function foo return integer;
end package uninst_pack;

package body uninst_pack is
```

```

        function foo return integer is
begin
    return g1 + 11;
end;
end package body uninst_pack;

package inst_pack is new work.uninst_pack generic map (g1 => 31);
-----
library ieee;
use ieee.std_logic_1164.all;

package global is
    generic (type DT;
             left , right : DT;
             function inc(x : DT) return DT;
             procedure chk(x : DT; y : DT);
             package inst_pack is new work.uninst_pack generic map (<>));
use inst_pack.all;
procedure self_check;
end package global;

package body global is

procedure self_check is
    variable v : DT := left;
begin
    v := inc(v);
    chk(v, right);
end procedure;

end package body global;
-----
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

package normal_package is

function incr (x : integer) return integer;
function incr (x : std_logic_vector) return std_logic_vector;
procedure util_assert(i1 : integer; i2 : integer);
procedure util_assert(sv1 : std_logic_vector; sv2 : std_logic_vector);

end package normal_package;

package body normal_package is

function incr(x : integer) return integer is
begin
    report "--From Function incr1--";
    return x + 1;
end;

function incr(x : std_logic_vector) return std_logic_vector is
begin

```

```

        report "--From Function incr2--";
        return unsigned(x) + 1;
end;

procedure util_assert(i1 : integer; i2 : integer) is
begin
    assert i1 = i2 report to_string(i1) & " /= " & to_string(i2);
end procedure;

procedure util_assert(sv1 : std_logic_vector; sv2 : std_logic_vector)
is
begin
    assert sv1 = sv2 report to_string(sv1) & " /= " & to_string(sv2);
end procedure;
end package body normal_package;
-----
use work.normal_package.all;

package global_inst is new work.global
generic map(
    DT => integer,
    left => 8, right => 9,
    inc => incr, chk => util_assert,
    inst_pack => work.inst_pack
);

use work.global_inst.all;

entity top is
end entity top;

architecture behav of top is
begin

process is
begin
    self_check;
    wait;
end process;

end architecture behav;

```

To use an uninstantiated package, it is required to instantiate it with the actual generics, as shown in the following example:

```

package global_inst is new work.global
generic map(
    DT => integer,
    left => 8, right => 9,
    inc => incr, chk => util_assert,
    inst_pack => work.inst_pack
);

use work.global_inst.all;

```

## Command Line

To run the example, use the following commands:

```
% vhdlan -vhdl108 generics_package.vhd  
% vcs top -R
```

Note:

If the actual generic type is unconstrained and if a formal generic type is used in the context of the constrained type, VCS issues an error message.

## Limitation

Uninstantiated package declarations and package instantiations are only allowed at the design unit level (that is, at a global scope). Local package declaration and instantiations are not supported.

---

## Referencing Interface Lists

A generic interface list consists entirely of interface constant declarations, interface type declarations, interface subprogram declarations, and interface package declarations. VCS supports declaring the following interface constants, as defined in the VHDL 2008 Std. LRM sections, 6.5.6.1 and 6.5.6.2:

- Reference *generic constants* in a generic list
- Reference *generic type* in a declaration of a subsequent generic

You can use one formal generic declared within a given generic list to declare the subsequent generics in this list. With this enhancement, you can use the value of generic to constrain the size of a subsequent generic of an array type.

## Use Model

To use generics in your design source code, specify the `-vhdl08` option in the `vhdlan` command line, or enable VHDL 2008 using the `VHDL_MODE=vhdl08` variable in the `synopsys_sim.setup` file.

## Example

### *Example 23-1*

```
library ieee;
use ieee.std_logic_1164.all;

entity test is
    generic (left : integer:=3;
             right: integer:=1;
             arr: bit_vector(left downto right):="110");
    port      (A : in bit_vector(left downto right));
end test;

architecture arch of test is
begin

    L1: process
    begin
        report to_string(left) & " " & to_string(right);
        report to_string(arr);
        report to_string(A);
    end process L1;

end arch ;
```

## Command Line

To run the example, use the following commands:

```
% vhdlan -vhdl08 top.vhd
% vcs test -R
```

## Limitations

The support for referencing generics in the generic list works with the following limitations:

- Overriding generics using the elaboration and runtime option
  - gv | -gvalue generic\_name=value and elaboration option
  - gfile genericfile.txt

The values overridden using the elaboration/runtime options are not propagated to the subsequent generics in the list. In the following example, the value of the generic `size` when overridden using the `-gv size=16` option at elaboration time or runtime is not propagated to the subsequent generic, `left`. Thus, the generic, `left`, holds the previous value 7.

Example,

```
library ieee;
use ieee.std_logic_1164.all;

entity test is
    generic (
        size : integer:= 8;
        left : integer:= size-1;
        right : natural := 0
    );
    port (
        A : in bit_vector(left downto right)
    );
end test;
architecture arch of test is
begin
    L1: process
    begin
        report to_string(size);
```

```

        report to_string(left) & " " & to_string(right);
        report to_string(A);
    end process L1;
end arch;
```

## Command Lines

To override at elaboration time, execute the following commands:

```
% vhdlan test.vhd -vhdl08
% vcs test -gv size=16
% ./simv
```

To override at elaboration time, execute the following commands:

```
% vhdlan test.vhd -vhdl08
% vcs test -debug_access+all -lca -gfile generic_file.txt
% ./simv
```

To override at runtime, execute the following commands:

```
% vhdlan test.vhd -vhdl08
% vcs test -debug_access+all -lca
% ./simv -gv size=16
```

- Overriding generics in a VHDL design unit instantiated under Verilog top (VL-VH scenario).

The values of VHDL generics when overridden from a Verilog-top design is not propagated to the subsequent generics in the list. In the following example, the value of generic, `size` when overridden to 16 is not propagated to subsequent generic, `left`. Thus, the generic, `left`, holds the previous value 7.

Example,

```
library ieee;
use ieee.std_logic_1164.all;
```

```

entity test is
    generic (
        size : integer:= 8;
        left : integer:= size-1;
        right : natural := 0
    );
    port (
        A : in bit_vector(left downto right)
    );
end test;
architecture arch of test is
begin
    L1: process
    begin
        report to_string(size);
        report to_string(left) & " " & to_string(right);
        report to_string(A);
    end process L1;
end arch;

module top;
    parameter NUM = 15;
    wire [7:0] arr;
    test #(size(NUM)) test_inst(.A(arr));
endmodule

```

## Command Lines

To run the example, use the following commands:

```

% vhdlan -vhdl08 -nc example3.vhd
% vlogan -sverilog -nc example3.v
% vcs top -nc -q
% ./simv

```

---

## Overriding the Value Assigned to a Signal

Generally, it is required to override the value assigned to a signal during the verification cycle and force a different value onto the signal to verify various signal values. Overriding a signal enables you to:

- Set up a test scenario by forcing value to a state, which is normally achieved through a complex initialization sequence. Forcing the values allows you to bypass the sequence and thus reduces the verification time significantly.
- Inject erroneous values into the design to ensure that it detects errors.

VCS supports the VHDL 2008 standard for forcing and releasing the values of signals as defined in the VHDL 2008 LRM, section 10.5. With this capability, you can write VHDL testbench code to force and release signal values.

To enable the execution of force and release assignments in your testbench, specify the `-vhdl08` option in the `vhdlan` command line, or enable VHDL 2008 using the `VHDL_MODE=vhdl08` variable in the `synopsys_sim.setup` file.

This section discusses the following topics:

- “[Forcing and Releasing Values of Signals](#)”
- “[Forcing and Releasing Ports of a Design](#)”
- “[Assigning Composite Value to a Collection of Signals](#)”
- “[Forcing and Releasing Assignment Written in a Subprogram](#)”
- “[Forcing and Releasing Multiple Concurrent Assignments](#)”

- “Debugging the Force and Release Assignments”

## Forcing and Releasing Values of Signals

Forcing a value is a sequential assignment written within a process that forms a part of the testbench. To force a signal with a force assignment, use the following syntax:

```
signal_name <= force expression;
```

This causes a delta cycle and forces the named signal to take on the value of the expression in that delta cycle, regardless of the value assigned to the signal by any normal signal assignment. The signal is considered to be active during the delta cycle. If the forcing value is different from the previous value, an event occurs on the signal. The processes sensitive to changes on the signal value, then respond to the value change in the normal way.

The rules relating the type of expression to the type of the target signal are applicable for force assignments. The target signal name can be a normal signal name, or it can be an external signal name, or an alias. Generally, using external names is a common use case, as you might often need to force an internal signal of a design from a testbench.

After the signal is forced, you can update the signal with another force assignment to change the overriding value, which causes the signal to become active again and possibly to have another event. Note that, you can force the updated signal as often as needed.

To stop forcing a signal, execute a release assignment with the following syntax:

```
signal_name <= release;
```

This causes a further delta cycle, with the signal being active. However, with the `release` assignment, the signal is no longer forced and the current values of its sources are used to determine the signal value in the normal way. Thus, it enables you to take back the control of the signal.

## Forcing and Releasing Ports of a Design

The ports are a form of signal. As defined in the VHDL 2008 LRM, you can force and release ports of a design. The values of forcing and releasing ports are governed by the driving value and the effective value.

The driving value is the value presented externally by an entity and is determined by the internal sources within the entity. The effective value is the value seen internally by an entity and is determined by whatever is externally connected to the port, whether it is an explicitly declared signal or a port of an enclosing entity. Depending on the port mode and the external connections, the driving and effective values may be different.

For example, an `inout` mode port of the type `std_logic` might drive a `0` value, but the externally connected signal might have another source driving a `1` value. In this case, the resolved value of the signal is `x`, and that value is seen as the effective value of the `inout` mode port.

VHDL 2008 allows you to force driving values and effective values of a port independently by including a force mode in an assignment. For explicitly declared signals, where the driving and effective values are the same, the distinction makes no difference.

For ports and signal parameters, to force the driving value, include the keyword `out` in the force assignment, as shown in the following syntax:

```
signal_name <= force out expression;
```

Alternatively, to force the effective value, include the keyword `in` in the force assignment, as shown in the following syntax:

```
signal_name <= force in expression;
```

To stop forcing the port's or signal parameter's driving value, use the `release` assignment with the keyword `out`, as shown in the following syntax:

```
signal_name <= release out;
```

Similarly, to stop the forced effective value, use the `release` assignment with the keyword `in`, as shown in the following syntax:

```
signal_name <= release in;
```

You can force and release the driving values of ports of the mode `out`, `inout`, and `buffer`. Ports and signal parameters of mode `in` can be forced or released only on the effective value. They do not have a driving value, therefore, force or release with mode `out` is not allowed.

The ports and signal parameters of all modes except the linkage have effective values, and therefore you can force and release the effective value of a port or a signal parameter of any mode except linkage.

If you omit the force mode (`out` or `in`) in a force or release assignment, a default force mode applies. For assignments to ports and signal parameters of mode `in` and to explicitly declared signals, the default force mode is `in`, forcing the effective value. For assignments to ports of mode `out`, `inout`, or `buffer`, and to signal parameters of mode `out` or `inout`, the default force mode is `out`, forcing the driving value.

## Assigning Composite Value to a Collection of Signals

A testbench models a broken data driver connection by forcing a `Z` value on the output part of the bidirectional port, while allowing the input part of the port to operate normally.

As defined in the VHDL 2008 LRM, VCS allows you to assign such composite value to a collection of signals. You can write the collection of signals in the form of an aggregate on the left-hand side of the assignment, as shown in the following syntax:

```
(carry_out, sum) <= ('0' & a) + ('0' & b);
```

Note that this form of aggregate assignment is legal in VHDL 2008. However, you cannot write an aggregate of signal names as the target of a force or release assignment to force or release each of the signal values. Instead, you must write a separate force or release assignment for each of the signals.

In scenarios where you define a resolved signal of a composite type, such as an array type (that is, a signal with multiple sources, each of which is a composite value), the resolution function for the signal takes an array of composite values and determines a composite value as the resolved value of the signal. You cannot write a force or a release assignment with an element of such a signal as the target. However, you can only force or release the signal as a whole.

## **Forcing and Releasing Assignment Written in a Subprogram**

As defined in the VHDL 2008 LRM, VCS enables you to force and release assignment in a subprogram. A signal assignment written in a procedure that is not contained within a process can only assign to a signal parameter of the procedure.

VHDL 2008 allows force and release assignments in procedures outside of the processes to signals other than the signal parameters.

## **Forcing and Releasing Multiple Concurrent Assignments**

Multiple forces and releases might occur for a given signal during a single simulation cycle, as they are sequential assignments written in processes. As defined in the VHDL 2008 LRM, if a force and release both occur, the effect is as though the release is immediately overridden by the force, and so the signal remains forced. However, it is forced with the new force value.

As the effect of multiple forces is not defined in the LRM, it is recommended to avoid writing multiple forces in a testbench. The effect of multiple releases, however, is same as a single release, and a release assignment on a signal that is not forced has no effect.

## **Debugging the Force and Release Assignments**

UCLI supports debugging of the VHDL 2008 force and release assignment statements. For EVCD dumping, language force is considered as a test bench driver, and same is applicable to VHDL 2008 force also.

The following UCLI commands support the VHDL 2008 force and release assignment statements:

- The `drivers` command lists the force assignment statements just as non-force assignment statements are listed.
- The `show` and `get` commands shows the value of signals with force assignment statements.
- The `stop -event` command stops the simulation when a force/release assignment statement triggers a value change.
- The `force -freeze` command overrides any active force assignment statement.
- The `release` command releases any active force assignment statement.

---

## Matching Case Statements

VCS enables you to define the sequential matching `case` statement, as defined in the VHDL 2008 LRM, section 10.9. With this support, VCS allows you to write the matching `case` statement with do not care –.

You can specify the expression followed by sequential statements using `case-end case`, as shown in the following syntax:

```
case_statement ::= [ case_label : ]
case '?' expression is
  case_statement_alternative
  { case_statement_alternative }
end case '?' [ case_label ] ';'
```

To enable the VHDL 2008 constructs, specify the `-vhdl08` option in the `vhdlan` command line, or enable VHDL 2008 using the `VHDL_MODE=vhdl08` variable in the `synopsys_sim.setup` file.

### **Example 1:**

[Example 23-2](#) is an example with the matching case statement.

#### *Example 23-2 Usage of matching case Statement*

```
File: example.vhdl
library ieee;
use ieee.std_logic_1164.all;

entity ent is
    port (
        sel : in std_ulogic_vector(2 downto 0);
        in1 : in std_ulogic_vector(3 downto 0);
        in2 : in std_ulogic_vector(3 downto 0);
        out1 : out std_ulogic_vector(3 downto 0)
    );
end entity ent;

architecture arch of ent is
begin
    process(sel,in1,in2)
    begin
        case ? sel is
            when "10-" => report "AND operation"; out1 <= in1
                and in2;
            when "01-" => report "OR operation"; out1 <= in1
                or in2;
            when "00-" => report "NAND operation"; out1 <= in1
                nand in2;
            when others => null;
        end case ?;
    end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;
```

```

entity top is
end entity top;

architecture arch of top is
    signal in1_tb,in2_tb,out1_tb : std_ulogic_vector(3 downto
0);
    signal sel_tb : std_ulogic_vector(2 downto 0);
begin
    ent_inst : entity work.ent(arch) port
map(sel_tb,in1_tb,in2_tb,out1_tb);

    process
    begin
        in1_tb <= "1111"; in2_tb <= "0010";
        sel_tb <= "101";
        wait for 1 ns;
        report to_string(out1_tb);
        wait;
    end process;
end architecture arch;

```

You can run this example using the following command line:

```
% vhdlan -vhdl08 example.vhdl
% vcs top
% simv
```

VCS generates the following output:

```

Report NOTE at 0 NS in design unit ENT(ARCH) from process /
TOP/ENT_INST/_P0:
    "AND operation"
1 NS
Report NOTE at 1 NS in design unit TOP(ARCH) from process /
TOP/_P0:
    "0010"
(simv): Simulation complete, time is 1 NS.
```

---

## Conditional Elaboration

VCS enables you to define the sequential if-else-generate and case-generate statements, as defined in the VHDL 2008 LRM, section 11.8. With this support, VCS allows you to configure the alternatives.

You can specify the alternate sets of sequential or structural statements using if-else-generate, as shown in the following syntax:

```
generate_label : if [ alternative_label : ] condition generate
[block_declarative
begin ]

----- first alternative
[else [ alternative_label : ] generate
[ block declarative
begin ]

-----last alternative
[end [alternate_label];]
end generate [ generate_label ];
```

You can also include further conditions to test using the if-elsif-generate, as follows:

```
generate_label : if [ alternative_label : ] condition generate
[block_declarative
begin ]

----- first alternative
[end [alternate_label];]
{ elsif [ alternative_label : ] condition generate
[block declarative
begin ]
----- next alternative
[end [alternate_label];]
}
[else [ alternative_label : ] generate
[ block declarative
begin ]
```

```

-----last alternative
[end [alternate_label];]
end generate [ generate_label ];

```

Similarly, VCS also supports the `case-generate` statement that enables you to specify alternative sets of sequential or structural statements and the choice values for each alternative, as shown in the following syntax:

```

generate_label : case expression generate
when [ alternative_label : ] choice1 =>
[ block declarative
begin ]

----- first alternate
[end [alternate_label];]
when [ alternative_label : ] choice2 =>
[ block declarative
begin ]

----- next alternate
[end [alternate_label];]
when [ alternative_label : ] others =>
[ block declarative
begin ]

----- last alternate
[end [alternate_label];]
end generate [generate_label];

```

To enable the VHDL 2008 constructs, specify the `-vhdl08` option in the `vhdlan` command line, or enable VHDL 2008 using the `VHDL_MODE=vhdl08` variable in the `synopsys_sim.setup` file.

### **Example 1:**

[Example 23-3](#) is an example with the `if-else-generate` statement.

#### *Example 23-3 Usage of if-else-generate Statement*

```

File: example1.vhd
library ieee;

```

```

use ieee.std_logic_1164.all;

entity test is
generic(j:integer :=2);
end test;

architecture arch of test is

begin

IF1: if A1 : j>5 generate
    signal k : integer := 10;
    begin
        process
        begin
            report to_string(j);
            report to_string(k);
            wait;
        end process;
    elsif A2 : j <= 2  generate
        process
        begin
            report to_string(j+1);
            wait;
        end process;
    else  A3 : generate
        process
        begin
            report to_string(-j);
            wait;
        end process;
    end generate IF1;

end arch;

```

You can run this example using the following command line:

```

% vhdlan -vhdl08 example1.vhd
% vcs test -R
% simv

```

VCS generates the following output:

```
Report NOTE at 0 NS in design unit TEST(ARCH) from process
/TEST/IF1/_P0:
  "3"
(simv): Simulation complete, time is 0.
```

### Example 2:

[Example 23-4](#) is an example with the `case-generate` statement.

#### *Example 23-4 Usage of case-generate Statement*

```
File: example2.vhd
library ieee;
use ieee.std_logic_1164.all;

entity test is
generic(s:std_ulogic:='X');
end test;

architecture arch of test is

begin

CASE1: case s generate
when A1 : 'U' | 'X' | '0' | '1' | 'Z' =>
process
begin
report to_string(s);
wait;
end process;
when A2 : 'W' | 'L' | 'H' =>
process
begin
report to_string('P');
wait;
end process;
when A3 : others =>
process
begin
```

```

        report "others";
        wait;
        end process;
    end generate CASE1;
end arch;
```

You can run this example using the following command line:

```
% vhdlan -vhdl08 example2.vhd
% vcs test -R
% simv
```

VCS generates the following output:

```
Report NOTE at 0 NS in design unit TEST(ARCH) from process
/TEST/CASE1/_P0:
    "X"
(simv): Simulation complete, time is 0.
```

For using cross-module references across the language boundaries or for tool specific configuration, you must use only generate label names without alternate label as shown in [Example 23-5](#):

#### *Example 23-5 Usage of XMR across the if-else and case generate statements*

```
File: example1.vhdl
library ieee;
use ieee.std_logic_1164.all;

entity ent1 is
    port(
        in1 : in std_logic;
        in2 : in std_logic;
        sel : in std_logic;
        out1 : out std_logic
    );
end entity ent1;
```

```

architecture arch of ent1 is
begin
    process(in1,in2,sel)
    begin
        if(sel='1') then
            out1 <= in1;
        else
            out1 <= in2;
        end if;
    end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;

entity ent2 is
    generic (
        g0 : natural := 1;
        g1 : natural := 2;
        g2 : natural := 3
    );
    port(
        in1 : in std_logic;
        in2 : in std_logic;
        sel : in std_logic;
        out1 : out std_logic
    );
end entity ent2;

architecture arch of ent2 is
begin
    gen_lbl : if g0 = 2 generate
    elsif alt_lbl : g1 = 3 generate
    else last_lbl : generate
    begin
        gen_lbl2 : case g2 generate
        when first_alt : 4 =>
        when alt_lbl : others =>
            ent1_inst : entity ent1(arch) port
map(in1,in2,sel,out1);
        end generate gen_lbl2;
    end last_lbl;

```

```

        end generate gen_lbl;
    end architecture arch;

library ieee;
use ieee.std_logic_1164.all;

entity ent3 is
end entity ent3;

architecture arch of ent3 is
    signal in1_tb,in2_tb,sel_tb,out1_tb : std_logic;
begin

    ent2_inst : entity work.ent2(arch) port
map(in1_tb,in2_tb,sel_tb,out1_tb);

    process
    begin
        sel_tb <= '1'; in1_tb <= '1'; in2_tb <= '0';
        wait for 1 ns;
        sel_tb <= 'X'; in1_tb <= '0'; in2_tb <= '1';
        wait for 1 ns;
        sel_tb <= 'X'; in1_tb <= '1'; in2_tb <= '1';
        wait for 1 ns;
        sel_tb <= 'X'; in1_tb <= '0'; in2_tb <= '0';
        wait;
    end process;
end architecture arch;

File: example2.v
`timescale 1ns/1ns
module top;
ent3 ent3_inst ();
    initial $monitor("@time = %0t
%b",$time,top.ent3_inst.ent2_inst.gen_lbl.gen_lbl2.ent1_in
st.out1); // xmr to output port
endmodule
File: xprop.cfg
instance
{top.ent3_inst.ent2_inst.gen_lbl.gen_lbl2.ent1_inst}
{xpropOn}; // enabling xprop on ent1_inst using xprop config
file
```

You can run this example using the following command line:

```
% vhdlan -vhdl108 example1.vhdl
% vlogan -sverilog example2.v
% vcs top -xprop=xprop.cfg
% simv
```

VCS generates the following output:

```
@time = 0 1
@time = 1 x
@time = 2 1
@time = 3 0
```

---

## Condition Operator in an Expression

VCS enables you to use condition operator ?? in an expression, as defined in the VHDL 2008 LRM, section 9.2.9. Prior to VHDL 2008 a condition was required to have a Boolean value which was not convenient to model the designs that used bit or std\_ulogic as control signals as shown:

```
if ctrl_sig = '1' then ...
```

Now, you can convert the bit or the std\_ulogic value to a boolean value. For bit value, ?? converts 1 to true and 0 to false. For std\_ulogic value, ?? converts both 1 and H to true and all other values to false.

You can rewrite the if-statement condition shown above using the following implicit and explicit ways:

```
if ?? ctrl_sig then ... -- explicit way
```

```
if ctrl_sig then ... -- implicit way
```

To enable the VHDL 2008 constructs, specify the `-vhdl08` option in the `vhdlan` command line, or enable VHDL 2008 using the `VHDL_MODE=vhdl08` variable in the `synopsys_sim.setup` file.

### Example 1:

[Example 23-6](#) is an example with explicit condition operator.

#### *Example 23-6 Usage of explicit condition operator*

```
File: example1.vhd
library ieee;
use ieee.std_logic_1164.all;

entity ent is
    port ( in1 : in std_logic;
           in2 : in std_logic;
           sel : in std_logic;
           out1 : out std_logic
    );
end entity ent;

architecture arch of ent is
begin
    process(sel,in1,in2)
    begin
        if ?? sel then -- Using Conditional Operator Explicitly
            out1 <= in1;
        else
            out1 <= in2;
        end if;
    end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;

entity top is
```

```

end entity top;

architecture arch of top is
    signal in1_tb,in2_tb,out1_tb,sel_tb : std_logic;
begin
    ent_inst : entity work.ent(arch) port
map(in1_tb,in2_tb,sel_tb,out1_tb);

    process
    begin
        in1_tb <= '1'; in2_tb <= '0'; sel_tb <= '1';
        wait for 1 ns;
        report "out1 = " & to_string(out1_tb);
        wait;
    end process;
end architecture arch;

```

## **Example 2:**

[Example 23-7](#) is an example with implicit condition operator.

### *Example 23-7 Usage of Implicit Condition Operator*

```

File: example2.vhd
library ieee;
use ieee.std_logic_1164.all;

entity ent is
    port ( in1 : in std_logic;
           in2 : in std_logic;
           sel : in std_logic;
           out1 : out std_logic
      );
end entity ent;

architecture arch of ent is
begin
    process(sel,in1,in2)
    begin
        if sel then -- Using Conditional Operator Implicitly
            out1 <= in1;
    end if;
end process;
end architecture arch;

```

```

        else
            out1 <= in2;
        end if;
    end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;

entity top is
end entity top;

architecture arch of top is
    signal in1_tb,in2_tb,out1_tb,sel_tb : std_logic;
begin
    ent_inst : entity work.ent(arch) port
map(in1_tb,in2_tb,sel_tb,out1_tb);

    process
    begin
        in1_tb <= '1'; in2_tb <= '0'; sel_tb <= '1';
        wait for 1 ns;
        report "out1 = " & to_string(out1_tb);
        wait;
    end process;
end architecture arch;

```

You can run these examples using the following command line:

```
% vhdlan -vhdl08 example1.vhd
```

Or

```
% vhdlan -vhdl08 example2.vhd
% vcs top
% simv
```

VCS generates the following output:

```
1 NS
```

```
Report NOTE at 1 NS in design unit TOP(ARCH) from process /  
TOP/_P0:  
    "out1 = 1"  
(simv): Simulation complete, time is 1 NS.
```

---

## Reading Output Port

VCS enables you to read out-mode ports, as defined in the VHDL 2008 LRM, section 9.2.9. Prior to VHDL 2008, internal reading of out-mode ports driving value was not allowed. In order to achieve this, the common practice followed was to restore the out-mode port value in an internal signal and drive on to out-mode ports. Now, it is no longer required to restore to an internal signal.

To enable the VHDL 2008 constructs, specify the `-vhdl08` option in the `vhdlan` command line, or enable VHDL 2008 using the `VHDL_MODE=vhdl08` variable in the `synopsys_sim.setup` file.

### Example 1:

[Example 23-8](#) is an example for reading output parameter.

#### *Example 23-8 Usage for Reading Output Parameter*

```
File: example.vhdl  
library ieee;  
use ieee.std_logic_1164.all;  
  
entity outModeRead is  
    port(  
        in1 : in std_logic_vector(0 to 3);  
        in2 : in std_logic_vector(0 to 3);  
        out_port : out std_logic_vector(0 to 3)  
    );  
end entity outModeRead;  
  
architecture arch of outModeRead is  
    procedure proc(in1,in2 : in std_logic_vector(0 to 3);out1
```

```

        : out std_logic_vector(0 to 3)) is
begin
    out1 := out1 or in1 or in2; -- reading the output
                                parameter of procedure
                                on LHS
end procedure proc;
begin
    process(in1, in2)
        variable sig : std_logic_vector(0 to 3);
begin
    proc(in1,in2,sig);
        out_port <= out_port or sig; -- reading the output
                                    parameter of
                                    entity on LHS
    end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;

entity top is
end entity top;

architecture arch of top is
    signal in1_tb,in2_tb,out_port_tb : std_logic_vector(0
to 3);
begin
    inst : entity work.outModeRead(arch) port
map(in1_tb,in2_tb,out_port_tb);
    process
begin
        in1_tb <= "1101"; in2_tb <= "1100";
        wait for 1 ns;
        report to_string(out_port_tb);
        wait;
    end process;
end architecture arch;

```

**You can run this example using the following command line:**

```
% vhdlan -vhdl108 example.vhdl
% vcs top
```

```
% simv
```

VCS generates the following output:

```
1 NS
Report NOTE at 1 NS in design unit TOP(ARCH) from process /
TOP/_P0:
"11U1"
(simv): Simulation complete, time is 1 NS.
```

---

## 2008 IEEE Packages

The 2008 IEEE package includes:

- All the standard packages including pre-defined types in the VHDL Standard LRM.
- Several new packages including the fixed-point and the floating-point numbers represented as vectors of `std_ulogic` elements and a package providing access to the simulation environment.
- Operations to the standard packages that provide a consistent feature set across the suite. These operations provide a consistent set of conversion functions and I/O operations, such as `to_string`, `read`, `write`, and so on.

## Overview of Additional IEEE Packages

This section provides an overview of enhancement to the IEEE packages, supported data types, and the corresponding operations that are included in VHDL 2008.

*Table 23-1 VHDL 2008 Supported IEEE Packages*

| IEEE Packages  | Supported Data Types                                                                                                     | Supported Operations                                                                                                                                                        | Supported Functions and Procedures                                                                                                                                                                                                                                                                                                                                                               |
|----------------|--------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| std_logic_1164 | 1. STD_LOGIC<br>2. STD_ULOGIC<br>3. STD_LOGIC_VECTOR<br>4. STD_ULOGIC_VECTOR<br>5. X01<br>6. X01Z<br>7. UX01<br>8. UX01Z | <ul style="list-style-type: none"> <li>• logical (and, nand, or, nor, xor, xnor, not)</li> <li>• shift and rotate (sll,srl,rol,ror)</li> <li>• conditional (???)</li> </ul> | <ul style="list-style-type: none"> <li>• Conversion (To_bit,To_bitvector,To_StdULogic,To_StdLogicVector,To_StdULogicVector)</li> <li>• Strength strippers (TO_01,To_X01,To_X01Z,To_UX01)</li> <li>• Edge detection (rising_edge,falling_edge)</li> <li>• Unknown detection(Is_X)</li> <li>• String conversion (to_string, to_ostring,to_hstring,READ,WRITE,OREAD,OWRITE,HREAD,HWRITE)</li> </ul> |

| IEEE Packages | Supported Data Types     | Supported Operations                                                                                                                                                                                                                                                                                                                                                                            | Supported Functions and Procedures                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|---------------|--------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| numeric_bit   | 1. signed<br>2. unsigned | <ul style="list-style-type: none"> <li>• arithmetic (+,-,*,/,rem, mod, abs)</li> <li>• logical (and, nand, or, nor, xor, xnor, not)</li> <li>• comparison(&gt;,&lt;,=,/=,&gt;=,&lt;=)</li> <li>• matching relational (?&gt;,&lt;?,?=,?/,=?&gt;=,?&lt;=)</li> <li>• shift and rotate (sll, srl, rol, ror, sla, sra)</li> <li>• matching relational (?&gt;,&lt;?,?=,?/,=?&gt;=,?&lt;=)</li> </ul> | <ul style="list-style-type: none"> <li>• arithmetic (find_leftmost,find_rightmost)</li> <li>• comparison (MAXIMUM,MINIMUM)</li> <li>• shift and rotate (SHIFT_LEFT,SHIFT_RIGHT,ROTATE_LEFT,ROTATE_RIGHT)</li> <li>• Resize (RESIZE)</li> <li>• Conversion (TO_INTEGER,TO_UN SIGNED,TO_SIGNED)</li> <li>• Edge detection (rising_edge, falling_edge)</li> <li>• String conversion (to_string, to_ostring, to_hstring, READ,WRITE,OREAD, OWRITE,HREAD,HWR ITE)</li> </ul> |

| IEEE Packages        | Supported Data Types                                                       | Supported Operations                                                                                                                                                                                                                                                                                              | Supported Functions and Procedures                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|----------------------|----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| numeric_std          | 1. SIGNED<br>2. UNSIGNED<br>3. UNRESOLVED_SIGNED<br>4. UNRESOLVED_UNSIGNED | <ul style="list-style-type: none"> <li>• arithmetic (+,-,*,/rem, mod, abs)</li> <li>• logical (and, nand, or, nor, xor, xnor, not)</li> <li>• comparison(&gt;,&lt;,=,/=,&gt;=,&lt;=)</li> <li>• matching relational(?, ?, ?, ?, ?, ?, ?, ?)</li> <li>• shift and rotate (sll, srl, rol, ror, sla, sra)</li> </ul> | <ul style="list-style-type: none"> <li>• arithmetic (find_leftmost,find_rightmost)</li> <li>• comparison (MAXIMUM,MINIMUM)</li> <li>• shift and rotate (SHIFT_LEFT,SHIFT_RIGHT,ROTATE_LEFT,ROTATE_RIGHT)</li> <li>• Resize (RESIZE)</li> <li>• Conversion (TO_INTEGER,TO_UN SIGNED,TO_SIGNED)</li> <li>• Edge detection (rising_edge, falling_edge)</li> <li>• Match functions (STD_MATCH)</li> <li>• Translation (TO_01, TO_X01,TO_X01Z, TO_UX01,IS_X)</li> <li>• String conversion (to_string, to_ostring, to_hstring, READ,WRITE,OREAD, OWRITE,HREAD,HWR ITE).</li> </ul> |
| numeric_bit_unsigned | • BIT_VECTOR                                                               | <ul style="list-style-type: none"> <li>• arithmetic (+,-,*,/rem,mod)</li> <li>• comparison (&gt;,&lt;,=,/=,&gt;=,&lt;=)</li> <li>• matching relational (?, ?, ?, ?, ?, ?, ?, ?)</li> <li>• shift and rotate (sll,srl,rol,ror,sla,sra)</li> </ul>                                                                  | <ul style="list-style-type: none"> <li>• arithmetic (find_leftmost,find_rightmost)</li> <li>• comparison (MAXIMUM,MINIMUM)</li> <li>• shift and rotate (SHIFT_LEFT,SHIFT_RIGHT,ROTATE_LEFT,ROTATE_RIGHT)</li> <li>• Resize (RESIZE)</li> <li>• Conversion (TO_INTEGER, TO_bitvector)</li> </ul>                                                                                                                                                                                                                                                                              |

| IEEE Packages        | Supported Data Types                                                         | Supported Operations                                                                                                                                                                                                                      | Supported Functions and Procedures                                                                                                                                                                                                                                                                                       |
|----------------------|------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| numeric_std_unsigned | 1. STD_LOGIC<br>2. STD_ULOGIC<br>3. STD_LOGIC_VECTOR<br>4. STD_ULOGIC_VECTOR | <ul style="list-style-type: none"> <li>• arithmetic (+,-,*,/,rem,mod)</li> <li>• comparison (&gt;,&lt;,=,/=,&gt;=,&lt;=)</li> <li>• matching relational (?&gt;,&lt;?,?=,?/=?&gt;=?&lt;=)</li> <li>• shift and rotate (sla,sra)</li> </ul> | <ul style="list-style-type: none"> <li>• arithmetic (find_leftmost,find_rightmost)</li> <li>• comparison (MAXIMUM,MINIMUM)</li> <li>• shift and rotate (SHIFT_LEFT,SHIFT_RIGHT,ROTATE_LEFT,ROTATE_RIGHT)</li> <li>• Resize (RESIZE)</li> <li>• Conversion (TO_INTEGER, TO_stdlogicvector, to_stdulogicvector)</li> </ul> |

| IEEE Packages | Supported Data Types                                                   | Supported Operations                                                                                                                                                                                                                                                                                               | Supported Functions and Procedures                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------------|------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| fixed_pkg     | 1. UFIXED<br>2. UNRESOLVED_UFIXED<br>3. SFIXED<br>4. UNRESOLVED_SFIXED | <ul style="list-style-type: none"> <li>• arithmetic(+,-,*,/ ,rem,mod,abs)</li> <li>• logical ( and,nand,or,nor,xo r,xnor,not)</li> <li>• comparison(&gt;,&lt;,=,/ =,&gt;=,&lt;=)</li> <li>• matching relational(?&gt;,&lt;?,?= ,?=/,?&gt;=,&lt;=)</li> <li>• shift and rotate (sll,srl,rol,ror,sla,rsa)</li> </ul> | <ul style="list-style-type: none"> <li>• arithmetic (divide, reciprocal, remainder, modulo, add_carry, scalb, Is_Negative)</li> <li>• comparison (std_match, MAXIMUM, MINIMUM, find_leftmost, find_rightmost)</li> <li>• shift and rotate (SHIFT_LEFT,SHIFT_RIGHT,ROTATE_LEFT, ROTATE_RIGHT)</li> <li>• Resize (RESIZE)</li> <li>• Conversion (to_ufixed,to_unsigned, to_real,to_integer,to_sfixed,to_signed,to_slv,to_suv,to_UFix,to_SFix)</li> <li>• Range generation (ufixed_high,ufixed_low ,sfixed_high,sfixed_low ,UFix_high,UFix_low,S Fix_high,SFix_low,saturate)</li> <li>• Translation (TO_01, TO_X01 ,TO_X01Z, TO_UX01,IS_X)</li> <li>• String conversion( to_string, to_ostring,to_hstring,READ,WRITE,OREAD,O WRITE,HREAD,HWRITE,from_string,from_ostring,from_hstring)</li> </ul> |

| IEEE Packages | Supported Data Types                                                                                                                           | Supported Operations                                                                                                                                                                                                                                                                                            | Supported Functions and Procedures                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| float_pkg     | 1. float,<br>UNRESOLVED_float<br>2. float32,<br>UNRESOLVED_float32<br>3. float64,<br>UNRESOLVED_float64<br>4. float128,<br>UNRESOLVED_float128 | <ul style="list-style-type: none"> <li>• arithmetic (+,-,*,/rem, mod, abs)</li> <li>• logical (and, nand, or, nor, xor, xnor, not)</li> <li>• comparison (&gt;,&lt;,=,/=,&gt;=,&lt;=)</li> <li>• matching relational (?&gt;,&lt;?,?=,?/=,?&gt;=,&lt;=)</li> <li>• shift and rotate (sll,srl,rol,ror)</li> </ul> | <ul style="list-style-type: none"> <li>• validity check (Classfp)</li> <li>• arithmetic (add, subtract, multiply, divide, reciprocal, remainder, modulo, dividebyp2, mac, sqrt, Copysign, Scalb, Logb, Nextafter, Unordered, Finite, Isnan, zerofp, nanfp, qnanfp, pos_inffp, neg_inffp, neg_zerofp)</li> <li>• comparison (eq,ne,lt,gt,le,ge,std_match,find_rightmost,find_leftmost,maximum,minum)</li> <li>• shift and rotate (SHIFT_LEFT,SHIFT_RIGHT,ROTATE_LEFT,ROTATE_RIGHT)</li> <li>• Resize (RESIZE)</li> <li>• Conversion (to_float32,to_float64,to_float128,to_slv,to_suv,to_float,to_unsigned,to_signed,to_ufixed,to_sfixed,to_real,to_integer,realtobits,bitstoreal,break_number,normalize)</li> <li>• Result range generation (ufixed_high,ufixed_low,sfixed_high,sfixed_low,UFix_high,UFix_low,SFix_high,SFix_low,saturate)</li> </ul> |

| IEEE Packages | Supported Data Types | Supported Operations | Supported Functions and Procedures                                                                                                                                                                                                                           |
|---------------|----------------------|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|               |                      |                      | <ul style="list-style-type: none"> <li>• Translation (TO_01, TO_X01, TO_X01Z, TO_UX01, IS_X)</li> <li>• String conversion (to_string, to_ostring, to_hstring, READ, WRITE, OREAD, OWRITE, HREAD, HWRITE, from_string, from_ostring, from_hstring)</li> </ul> |

---

## Resolved Elements

VCS supports resolved elements, as defined in *VHDL 2008 LRM Section 6.3*.

The support for resolution is enhanced to include array elements and record elements.

Using this enhancement, `std_logic_vector` is automatically converted to `std_ulogic_vector` (or vice versa). This indicates that `std_logic_vector` is now a subtype of `std_ulogic_vector`.

The following is the general syntax for a subtype declaration:

```

subtype_declaration ::= 
    subtype identifier is subtype_indication ;
subtype_indication ::= 
    [ resolution_indication ] type_mark [ constraint ]
resolution_indication ::= 
    resolution_function_name | ( element_resolution )

element_resolution ::= array_element_resolution |
record_resolution
  
```

```

array_element_resolution ::= resolution_indication

record_resolution ::= record_element_resolution { ,
record_element_resolution }

record_element_resolution ::= record_element_simple_name
resolution_indication

type_mark ::=

    type_name
    | subtype_name

constraint ::=

    range_constraint
    | array_constraint
    | record_constraint

element_constraint ::=

    array_constraint
    | record_constraint

```

## Usage Example

The following is an example for array elements resolution:

```

entity DUT is
    port(in1,in2,in3 : in bit_vector(0 to 7);
         out1 : out bit_vector(0 to 7)
        );
end entity DUT ;

architecture arch of DUT is
    type bit_vector_vector is array (natural range <>) of
bit_vector(0 to 7);
    function func1 (arg : bit_vector) return bit is
        variable result : bit := '0';
    begin
        for i in arg'range loop
            result := result or arg(i);
        end loop;
        return result;
    end function func1;
-- subtype declaration with array elements
subtype res_bit_nest is ((func1)) bit_vector_vector(0 to 0);

```

```

        signal sig : res_bit_nest;
begin
sig(0) <= in1;
sig(0) <= in2;
sig(0) <= in3;
out1 <= sig(0);
end architecture arch;

entity top is
end entity top;

architecture arch of top is
    signal in1,in2,in3,out1 : bit_vector(0 to 7);
begin
    inst : entity work.DUT(arch) port map(in1,in2,in3,out1);

    process
    begin
        in1 <= "11000011"; in2 <= "10101010"; in3 <= "10011001";
        wait for 1 ns;
        report to_string(out1);
        wait;
    end process;
end architecture arch;

```

**To run the example, use the following commands:**

```
% vhdlan -vhdl08 test.vhdl
% vcs top
% simv
```

**The following is the output:**

```
1 NS
Report NOTE at 1 NS in design unit TOP(ARCH) from process /
TOP/_P0:
    "11111011"
(simv): Simulation complete, time is 1 NS.
```

**The following is an example for record elements resolution:**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```

package pack is
    type bit_vector_vector is array (natural range <>) of
    bit_vector(0 to 3);
    type signed_vector is array (natural range <>) of signed(0
    to 3);
        function wor (arg : bit_vector) return bit;
        function wxor (arg : signed) return std_ulogic;
    type REC is record
        E1 : bit_vector_vector(0 to 0);
        E2 : signed_vector(0 to 0);
    end record;
    -- subtype declaration with record elements
    subtype resolved_rec is (E1((wor)),E2((wxor))) REC;
end package pack;

package body pack is

    function wor (arg : bit_vector) return bit is
        variable result : bit := '0';
    begin
        for i in arg'range loop
            result := result or arg(i);
        end loop;
        return result;
    end function wor;

    function wxor (arg : signed) return std_ulogic is
        variable result : std_ulogic := '0';
    begin
        for i in arg'range loop
            result := result nor arg(i);
        end loop;
        return result;
    end function wxor;

end package body pack;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.pack.all;

```

```

entity DUT is
    port(in1,in2 : in bit_vector(0 to 3);
         in3,in4 : in signed(0 to 3);
         out1 : out bit_vector(0 to 3);
         out2 : out signed(0 to 3)
    );
end entity DUT;

architecture arch of DUT is
    signal sig : resolved_rec;
begin

    sig.E1(0) <= in1;
    sig.E1(0) <= in2;
    sig.E2(0) <= in3;
    sig.E2(0) <= in4;
    out1 <= sig.E1(0);
    out2 <= sig.E2(0);

end architecture arch;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top is
end entity top ;

architecture arch of top is
    signal in1,in2,out1 : bit_vector(0 to 3);
    signal in3,in4,out2 : signed(0 to 3);
begin

    inst : entity work.DUT(arch) port map
    (in1,in2,in3,in4,out1,out2);

    process
    begin
        in1 <= "1100"; in2 <= "1010"; in3 <= "1001"; in4 <= "1101";
        wait for 1 ns;
        report "out1 = " & to_string(out1) & " out2 = " &
        to_string(out2);

```

```
    wait;
end process;
end architecture arch;
```

To run the example, use the following commands:

```
Command lines to run
% vhdlan -vhdl08 test.vhdl
% vcs top
% simv
```

The output is as follows:

```
1 NS
Report NOTE at 1 NS in design unit TOP(ARCH) from process /
TOP/_P0:
  "out1 = 1110 out2 = 0000"
```

---

## Conditional and Selected Assignments

VCS supports the conditional and selected assignments, as defined in *VHDL 2008 LRM Sections 10.5 and 10.6*.

According to VHDL 2008 LRM, you can use conditional and selected signal and variable assignments inside processes and subprograms, which allows the consistency between sequential and concurrent assignments.

The conditional signal assignment represents an equivalent if statement that assigns values to signals or that forces or releases signals. The general syntax is as follows:

```
conditional_signal_assignment ::= 
  conditional_waveform_assignment
  | conditional_force_assignment
conditional_waveform_assignment ::=
```

```

target <= [ delay_mechanism ] conditional_waveforms ;
conditional_waveforms ::= 
    waveform when condition
    { else waveform when condition }
    [ else waveform ]
conditional_force_assignment ::= 
    target <= force [ force_mode ] conditional_expressions ;
conditional_expressions ::= 
    expression when condition
    { else expression when condition }
    [ else expression ]

```

The selected signal assignment represents an equivalent case statement that assigns values to signals or that forces or releases signals. The syntax is as follows:

```

selected_signal_assignment ::= 
    selected_waveform_assignment
    | selected_force_assignment
selected_waveform_assignment ::= 
    with expression select [ ? ]
        target <= [ delay_mechanism ] selected_waveforms ;
selected_waveforms ::= 
    { waveform when choices , }
    waveform when choices
selected_force_assignment ::= 
    with expression select [ ? ]
        target <= force [ force_mode ] selected_expressions ;
selected_expressions ::= 
    { expression when choices , }
    expression when choices

```

For more information about the syntax and the functionality, see the *IEEE Standard VHDL 1076-2008 LRM*.

## Use Model

To use Conditional and selected assignments in your design source code, specify the `-vhdl108` option in the `vhdlan` command line, or enable VHDL 2008 using the `VHDL_MODE=vhdl108` variable in the `synopsys_sim.setup` file.

## Usage Example

The following example illustrates the usage of conditional signal assignment inside process:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity counter is
    port(
        clk      : in std_logic;
        rst      : in std_logic;
        count   : out std_logic_vector(3 downto 0)
    );
end entity counter;

architecture arch of counter is
begin
    process(clk)
    begin
        if (rising_edge(clk)) then
            --- conditional signal assignment
            count <= (others=>'0') when rst else count + 1;
        end if;
    end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use std.textio.all;
```

```

entity tb is
end entity tb;

architecture arch of tb is
    signal clk,rst : std_logic;
    signal count : std_logic_vector(3 downto 0);
begin
    inst : entity work.counter(arch) port map(clk,rst,count);
    process
    begin
        rst <= '0';
        wait for 5 ns;
        rst <= '1';
        wait for 5 ns;
        rst <= '0';
        wait for 100 ns;
        wait;
    end process;

    process
    begin
        if( now /= 100 ns) then
            clk <= '0'; wait for 5 ns;
            clk <= '1'; wait for 5 ns;
        else
            wait;
        end if;
    end process;

    process(count)
        variable l : line;
    begin
        write(l,justify("@time = " & to_string(now) & " Count
= " & to_string(count)));
        writeline(output,l);
    end process;
end architecture arch;

```

To run the example, use the following commands:

```
% vhdlan -vhdl108 test.vhdl
% vcs tb
% simv
```

The output is as follows:

```
@time = 0 ns Count = UUUU
@time = 5 ns Count = 0000
@time = 15 ns Count = 0001
@time = 25 ns Count = 0010
@time = 35 ns Count = 0011
@time = 45 ns Count = 0100
@time = 55 ns Count = 0101
@time = 65 ns Count = 0110
@time = 75 ns Count = 0111
@time = 85 ns Count = 1000
@time = 95 ns Count = 1001
```

The following example illustrates the usage of selected signal assignment inside process:

```
library ieee;
use ieee.std_logic_1164.all;

entity dut is
    port (
        in1 , in2 : in std_logic;
        sel   : in std_logic;
        out1 : out std_logic
    );
end entity dut;

architecture arch of dut is
begin
    process(all)
    begin
        with sel select          -- selected signal assignment
            out1 <= in1 and in2 when '0',
                    in1 or in2 when '1',
                    'Z' when others;
```

```

        end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use std.textio.all;

entity tb is
end entity tb;

architecture arch of tb is
    signal in1,in2,sel,out1 : std_logic;
begin
    inst : entity work.dut(arch) port map(in1,in2,sel,out1);
process
    variable l : line;
begin
    in1 <= '0'; in2 <= '1'; sel <= '0';
    wait for 1 ns;
    write(l,"out1 = " & to_string(out1));
    writeline(output,l);
    in1 <= '0'; in2 <= '1'; sel <= '1';
    wait for 1 ns;
    write(l,"out1 = " & to_string(out1));
    writeline(output,l);
    wait;
end process;
end architecture arch;

```

To run the example, use the following commands:

```
% vhdlan -vhdl08 test.vhdl
% vcs tb
% simv
```

The output is as follows:

```
out1 = 0
out1 = 1
```

### Note:

Conditional and selected assignments are also supported for variables.

### Limitation

The following is the limitation for conditional and selected assignments feature:

- VHDL 2008 condition operator ?? is not yet supported for condition coverage.

---

## Context Declaration

VCS supports the context declaration and context clauses, as defined in *VHDL 2008 LRM Sections 13.3 and 13.4*.

Based on the VHDL 2008 LRM, you can define context declaration and its context clauses which can be referenced by design units. This allows you to gather a collection of libraries and use clauses in one declaration instead of repeating the collection for every design unit.

The benefits of context declaration are:

- You can avoid writing multiple long lists of library and use clauses in large design.
- You can organize contexts of design units that share common library and use clauses in a more efficient and consistent manner.
- You can implement built-in contexts for common purposes.

The following is the general syntax for a context declaration:

```

context_declaration ::= 
    context identifier is
        context_clause
    end [ context ] [ context_simple_name ] ;

context_clause ::= { context_item }

context_item ::= 
    library_clause
    | use_clause
    | context_reference

context_reference ::= 
    context selected_name { , selected_name } ;

```

## Usage Example

The following example shows the syntax of a context declaration:

**File1.vhd**

```

context context_clause1 is
    library ieee;
    use ieee.std_logic_1164.all;
end context context_clause1;

```

**File2.vhd**

```

context context_clause2 is
    library lib1;
    context lib1.context_clause1;
    library ieee;
    use ieee.std_logic_arith.all;
    use std.textio.all;
    use ieee.std_logic_textio.all;
end context context_clause2;

```

```

context work.context_clause2;

```

```

entity dut is
    port(
        in1 : in unsigned(0 to 3);
        in2 : in unsigned(0 to 3);
        out1 : out unsigned(0 to 3)
    );

```

```

        );
end entity dut;

architecture arch of dut is
begin
    process(in1, in2)
    begin
        case in1(0) is
            when '1' => out1 <= in1 + in2;
            when '0' => out1 <= in2 - in1;
            when others => out1 <= in1;
        end case;
    end process;
end architecture arch;

context work.context_clause2;

entity top is
end entity top;

architecture arch of top is
    signal in1,in2,out1 : unsigned(0 to 3);
begin
    inst : entity work.dut(arch) port map(in1,in2,out1);

    process
        variable l : line;
    begin
        in1 <= "1101" ; in2 <= "1001";
        wait for 1 ns;
        write(l,"out1 = " &
to_string(conv_std_logic_vector(out1,4)));
        writeline(output,l);
        wait;
    end process;
end architecture arch;

synopsys_sim.setup file

WORK > DEFAULT
DEFAULT : ./work
lib1 : ./lib1

```

To run the example, use the following commands:

```

% mkdir work lib1
% vhdlan -vhdl08 file1.vhd -work lib1
% vhdlan -vhdl08 file2.vhd
% vcs top
% simv

```

The following is the output:

```
out1 = 0110  
(simv): Simulation complete, time is 1 NS.
```

---

## Improved I/O

VHDL provides a set of enhancements to the I/O functions and string conversion features. You can easily debug the signal value change and also use read and write procedures by implementing these new functions and features.

The support for textual I/O is enhanced by including operations for all the standard types in the respective IEEE 2008 packages as defined in *VHDL 2008 LRM Section 16.4*. This support is also extended for octal and hexadecimal I/O to read and write data by enhancing the string I/O capability.

The following is the list of functions:

- `TO_STRING (x)` function - This function provides the ability to work with any scalar type and any one-dimensional array type where the element type is an enumerated type all of whose literals are character literals.
- Justify function - This function provides the ability to justify a string representation in a field of given width. It is useful for tabular formatting of output.
- Octal Read/write operations - This function provides the octal read and write capability. If the number of bits to be read is not divisible by three, then the number read is resized accordingly. In the case of write operation, padding bits are included.

- Hexadecimal Read/write operations - This function provides the hexadecimal read and write capability. If the number of bits is not divisible by four, then the number read is resized accordingly. In the case of write operation, padding bits are included.
- Tee procedure - It provides the ability to write a line both to the file output and to a separate named file and allows to avoid replicated write operations.
- Flush procedure - It provides the ability to effect all previous calls to the write procedure for a file be completed.

## Usage Example

The following is an example for some of the I/O functions:

```

library ieee;
use ieee.std_logic_1164.all,ieee.std_logic_unsigned.all;

entity counter is
    port(
        clk,rst,en : in std_logic;
        count : out std_logic_vector(3 downto 0)
    );
end entity counter;

architecture arch of counter is
begin
    process(clk,rst)
    begin
        if(rst) then
            count <= (others => '0');
        elsif(rising_edge(clk)) then
            if(en) then
                count <= count + 1;
            end if;
        end if;
    end process;
end architecture arch;

```

```

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

entity top is
end entity top;

architecture arch of top is
    signal clk,rst,en : std_logic;
    signal count : std_logic_vector(3 downto 0);
begin
    counter_inst : entity work.counter(arch) port
map(clk,rst,en,count);
    clk_proc : process
    begin
        if now = 100 ns then
            wait;
        else
            clk <= '0'; wait for 5 ns;
            clk <= '1'; wait for 5 ns;
        end if;
    end process clk_proc;

process
begin
    rst <= '1'; en <= '0';wait for 3 ns;
    rst <= '0'; en <= '1';wait for 100 ns;
    wait;
end process;

process(count)
    variable wl,rl : line;
    variable var : std_logic_vector(3 downto 0);
    file fpw : text open WRITE_MODE is "file.out";
    file fpr : text open READ_MODE is "file.out";
begin
    write(wl,justify(to_string(now,ns),right,10));
    write(wl,justify(to_string(count),right,15));
    tee(fpw,wl);
end process;

```

```
end architecture arch;
```

To run the example, use the following commands:

```
% vhdlan -vhdl08 test.vhdl  
% vcs top  
% simv
```

The output is as follows:

|      |      |
|------|------|
| 0ns  | UUUU |
| 0ns  | 0000 |
| 5ns  | 0001 |
| 15ns | 0010 |
| 25ns | 0011 |
| 35ns | 0100 |
| 45ns | 0101 |
| 55ns | 0110 |
| 65ns | 0111 |
| 75ns | 1000 |
| 85ns | 1001 |
| 95ns | 1010 |

---

## Support for Implicitly Constrained Array Elements

VHDL provides two kinds of composite types, namely, arrays and records. All the elements are of the same type in the case of array elements. The elements are of different types in the case of record elements.

VCS supports the use of implicitly constrained array element types as defined in VHDL 2008 LRM, section 5.3.

The following element types are supported from this release:

- Partially constrained subtypes
- Implicitly constrained array subtypes on:
  - Ports
  - Constants
  - Generics
  - Parameters

## Usage Example

```
package global is
    type BitVecArrTyp is array(natural range <>) of
        bit_vector;
end package global;

entity ent1 is
    port(in1, in2 :in work.global.BitVecArrTyp;
         out1 :out work.global.BitVecArrTyp);
end entity ent1;

architecture arch of ent1 is
begin
    process(in1, in2)
    begin
        for idx in in1'range loop
            out1(idx) <= in1(idx) and in2(idx);
        end loop;
    end process;
end architecture arch;

entity top is
end entity top;

architecture arch of top is
    signal in1, in2, out1 : work.global.BitVecArrTyp(0 to 1)(3
        downto 0);
begin
    ent1_inst : entity work.ent1(arch)port map (in1, in2,
        out1);

    in1 <= ("1010", "0101");
    in2 <= ("1110", "0111");

    process (out1)
    begin -- process
        report "out1 : ("" & to_string(out1(0)) &
                """", """ & to_string(out1(1)) & """")";
    end process;
end architecture arch;
```

The following are the commands to run the test case:

```
vhdlan -vhdl08 DocExample.vhd  
vcs top  
.simv
```

## Output

```
Report NOTE at 0 NS in design unit TOP(ARCH) from process /  
TOP/_P2:  
    "out1 : ("0000", "0000")"  
Report NOTE at 0 NS in design unit TOP(ARCH) from process /  
TOP/_P2:  
    "out1 : ("1010", "0101")"  
(simv): Simulation complete, time is 0.
```

---

## Support for Unconstrained Element Types

VCS supports the use of unconstrained element types as defined in the *VHDL 2008 LRM, Section 5.3*. This feature is enabled with the `-vhdl08` switch at the `vhdlan` stage.

The following element types are supported:

- Partially constrained array subtypes
- Implicitly constrained array subtypes on,
  - Ports
  - Constants
  - Generics
  - Parameters

- New built-in array attributes
- Unconstrained or partial array element constrained aliases
- Unconstrained array element access type
- Unconstrained array element type external names
- Explicitly constrained record element subtypes on,
  - Ports
  - Constants
  - Generics
  - Parameters

The following is the general syntax for definition of an array type and a record type:

### **Array Type:**

```

array_type_definition ::= 
    unbounded_array_definition | 
    constrained_array_definition

unbounded_array_definition ::= 
    array ( index_subtype_definition { , 
        index_subtype_definition } ) 
        of element_subtype_indication

constrained_array_definition ::= 
    array index_constraint of element_subtype_indication

index_subtype_definition ::= type_mark range <>

array_constraint ::= 
    index_constraint [ array_element_constraint ] 
    | ( open ) [ array_element_constraint ]

```

```

array_element_constraint ::= element_constraint

index_constraint ::= ( discrete_range { , discrete_range } )

discrete_range ::= discrete_subtype_indication | range

```

## Record Type:

```

record_type_definition ::=

  record
    element_declaration
    { element_declaration }
  end record [ record_type_simple_name ]

element_declaration ::=

  identifier_list : element_subtype_definition ;

identifier_list ::= identifier { , identifier }

element_subtype_definition ::= subtype_indication

record_constraint ::=

  ( record_element_constraint { ,
    record_element_constraint } )

record_element_constraint ::= record_element_simple_name
element_constraint

```

## Usage Example

The following is an example for unconstrained array element type:

### *Example 23-9 Unconstrained element type*

```

library ieee;
use ieee.std_logic_1164.all;

package global is
  type StdVecArrTyp is array(natural range <>) of

```

```

    std_logic_vector;
end package global;

library ieee;
use ieee.std_logic_1164.all;

entity ent1 is
  port(
    in1 : in work.global.StdVecArrTyp(open) (open);
    in2 : in work.global.StdVecArrTyp(open) (open);
    out1 : out work.global.StdVecArrTyp(open) (open)
  );
end entity ent1;

architecture arch of ent1 is
begin
  process(in1,in2)
  begin
    for idx in in1'range loop
      out1(idx) <= in1(idx) and in2(idx);
    end loop;
  end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;

entity top is
end entity top;

architecture arch of top is
  signal in1,in2,out1 : work.global.StdVecArrTyp(0 to 1)(3
  downto 0);
begin
  ent1_inst : entity work.ent1(arch)
    port map(in1 => in1, in2 => in2, out1 => out1);
process
  use ieee.std_logic_textio.all;
  use std.textio.all;
  variable l : line;
begin
  in1(0) <= "1101"; in1(1) <= "1111";

```

```

in2(0) <= "1101"; in2(1) <= "1001";
wait for 1 ns;
for i in in1'range loop
    write(l,string'(" in1(") & to_string(i) & string'(")
    : ") & to_string(in1(i)));
    write(l,string'(" in2(") & to_string(i) & string'(")
    : ") & to_string(in2(i)));
    write(l,string'(" out1(")& to_string(i) & string'(")
    : ") & to_string(out1(i)));
    writeline(output,l);
end loop;
wait;
end process;
end architecture arch;

```

**The following are the commands to execute:**

```
% vhdlan -vhdl08 test.vhd
% vcs top
% simv
```

## Output

```

in1(0) : 1101 in2(0) : 1101 out1(0) : 1101
in1(1) : 1111 in2(1) : 1001 out1(1) : 1001
(simv): Simulation complete, time is 1 NS.
```

The following is an example of unconstrained record element type:

### *Example 23-10 Unconstrained record element type*

```

library ieee;
use ieee.std_logic_1164.all;

package pkg is
    type slv1d is array(natural range <>) of std_logic_vector;
    type rec_typ is record
        r0 : slv1d;
        r1 : std_logic_vector;
    end record rec_typ;
end package pkg;
```

```

library ieee;
use ieee.std_logic_1164.all;
use work.pkg.all;

entity ent is
  port (
    in1 : in rec_typ(r0(0 to 1)(3 downto 0),r1(3 downto 0));
    out1 : out rec_typ(r0(0 to 1)(3 downto 0),r1(3downto 0))
  );
end entity ent;

architecture arch of ent is
begin
  process(in1)
  begin
    for i in in1.r0'range loop
      for j in in1.r0'element'ranging loop
        out1.r0(i)(j) <= not in1.r0(i)(j);
      end loop;
    end loop;
    out1.r1 <= not in1.r1;
  end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
use std.textio.all;
use work.pkg.all;

entity top is
end entity top;

architecture arch of top is
  signal in1,out1 : rec_typ(r0(0 to 1)(3 downto 0),r1(3
  downto 0));

begin
  inst : entity work.ent(arch) port map(in1,out1);
  process
    variable l : line;

```

```

begin
    in1.r0(0) <= "1101"; in1.r0(1) <= "1111"; in1.r1 <=
    "0101";

    wait for 1 ns;
    for i in out1.r0'range loop
        write(l,string'(" out1.r0(") & to_string(i) &
        string') : ") & to_string(out1.r0(i)));
    end loop;
    write(l,string'(" out1.r1 : ") & to_string(out1.r1));
    writeline(output,l);
    wait;
end process;
end architecture arch;

```

The following are the commands to execute:

```

% vhdlan -vhdl08 test.vhd
% vcs top
% simv

```

## Output

```
out1.r0(0) : 0010 out1.r0(1) : 0000 out1.r1 : 1010
```

## Limitations

The following are the limitations for unconstrained element types:

1. Unbounded records constrained by expression or association are not supported for the following objects:
  - Ports, parameters, generic constants, alias, access, external, and type conversion

### Note:

This includes parameters to operator overload and therefore comparison of aggregates of implicitly constrained unbounded record type is not supported.

2. Record aggregates assigned to record objects and those associated with record interface elements that have been explicitly constrained are not supported.
3. Formal sub-element association of record elements where the record element is constrained at the point of the interface element declaration (Port/Parameter/Generic).
4. Type conversion for unconstrained element type(arrays or records) is not supported.
5. Unbounded record and array element types associated with type generics are not supported.

---

## Support for Enhanced Generics in Entity Interfaces

VCS supports the use of enhanced generics in entity interfaces as defined in Section 6.5 of *VHDL 2008 LRM*.

With the support of enhanced generics in VHDL 2008, you can declare generic types, sub-programs, packages, and generic constants in an entity interface. This helps in declaring abstract and reusable entities with different types and design specialization.

### Usage Example

These examples show the use of generic types and functions.

#### **Example 1:**

#### **Using a Generic Type in an Entity Interface:**

### *Example 23-11 Usage of generic type in entity interface*

```
entity gen_mux is
    generic( type T );
    port(
        sel   : in bit;
        I0,I1 : in T ;
        Y     : out T
    );
end entity gen_mux;

architecture arch of gen_mux is
begin
    Y <= I0 when sel else I1;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity top is
end entity top;

architecture arch of top is
    signal sel : bit;
    signal a,b,Y : std_logic_vector(2 downto 0);
begin
    inst : entity work.gen_mux(arch)
        generic map (
            T => std_logic_vector(2 downto 0) )
        port map(
            sel => sel,
            I0 => a,
            I1 => b,
            Y => Y
        );
    Process
        Variable l : line;
    Begin
        a <= "110"; b <= "011"; sel <= '1';
        wait for 1 ns;
```

```

        write(l,"sel = " & to_string(sel) & " a = " &
to_string(a) & " b = " & to_string(b) & " y =
" & to_string(y));
        writeline(output,l);
        a <= "110"; b <= "011"; sel <= '0';
        wait for 1 ns;
        write(l,"sel = " & to_string(sel) & " a = " &
to_string(a) & " b = " & to_string(b) & " y =
" & to_string(y));
        writeline(output,l);
        wait;
    end process;
end architecture arch;

```

**Following are the commands to run the example:**

```

vhdlan -vhdl08 test.vhd
vcs top
./simv

```

## Output

```

sel = 1 a = 110 b = 011 y = 110
sel = 0 a = 110 b = 011 y = 011

```

## Example 2:

### Using a Sub-program in an Entity Interface:

*Example 23-12 Usage of sub-program in entity interface*

```

library ieee;
use ieee.std_logic_1164.all;

entity gen_counter is
    generic( type T; constant C : T ; function
count_increment(in1 : T) return T );
    port(
        clk,rst : in std_logic;
        count : out T
    );
end entity gen_counter;

```

```

architecture arch of gen_counter is
begin
    process(clk,rst)
        variable rst_value : T := C;
    begin
        if(rst) then
            count <= C;
        elsif(rising_edge(clk)) then
            count <=count_increment(count);
        end if;
    end process;
end architecture arch;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity top is
end entity top;

architecture arch of top is
    signal clk,rst: std_logic;
    signal count : std_logic_vector(1 downto 0);
    function increment(inl : in std_logic_vector) return
std_logic_vector is
    begin
        return (inl + 1);
    end function increment;
begin
    inst : entity work.gen_counter(arch)
        generic map (
            T => std_logic_vector(1 downto 0),
            C => (others => '0'),
            count_increment => increment
        )
        port map(
            clk => clk,
            rst => rst,
            count => count

```

```

        );
process
begin
    if(now = 50 ns) then
        wait;
    else
        clk <= '0'; wait for 5 ns;
        clk <= '1'; wait for 5 ns;
    end if;
end process;
Process
begin
    rst <= '1';
    wait for 3 ns;
    rst <= '0';
    wait for 101 ns;
    wait;
end process;
process(count)
Variable l : line;
begin
    write(l, "rst = " & to_string(rst) &
" clk = " & to_string(clk) & " count = " & to_string(count) &" time = " & to_string(now) );
    writeline(output,l);
end process;
end architecture arch;

```

**Following are the commands to run the example:**

```

vhdlan -vhdl08 -q test.vhd
vcs top
./simv

```

## Output

```

rst = U clk = U count = UU time = 0 ns
rst = 1 clk = 0 count = 00 time = 0 ns
rst = 0 clk = 1 count = 01 time = 5 ns
rst = 0 clk = 1 count = 10 time = 15 ns
rst = 0 clk = 1 count = 11 time = 25 ns

```

```
rst = 0 clk = 1 count = 00 time = 35 ns
rst = 0 clk = 1 count = 01 time = 45 ns
```

### Example 3

#### Using the Generic Package Instantiation in an Entity Interface:

*Example 23-13 Usage of generic package instantiation in an entity interface*

```
package pkg is
    generic(type T);
        function func(sel : in bit; in1,in2 : in T) return T ;
end package pkg;

package body pkg is
    function func(sel : in bit; in1,in2 : in T) return T is
    begin
        if(sel) then
            return in1;
        else
            return in2;
        end if;
    end function func;
end package body pkg;

library ieee;
use ieee.std_logic_1164.all;

entity gen_mux is
    generic(type Te;package pkg_inst is new work.pkg
generic map(T=>Te));
    port(
        sel: in bit;
        in1,in2 : in Te;
        out1 : out Te
    );
end entity gen_mux;

architecture arch of gen_mux is
    use pkg_inst.all;
begin
    out1 <= func(sel,in1,in2);
```

```

end architecture arch;

library ieee;
use ieee.std_logic_1164.all;
package pkg_inst is new work.pkg generic map(
    T => std_logic_vector(1 downto 0)
);

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use ieee.std_logic_textio.all;

entity top is
end entity top;

architecture arch of top is
    signal sel : bit;
    signal in1,in2,out1,temp: std_logic_vector(1 downto 0);
begin
    inst : entity work.gen_mux(arch)
        generic map (Te => std_logic_vector(1 downto 0),pkg_inst => work.pkg_inst)
            port map(
                sel => sel,
                in1 => in1,
                in2 => in2,
                out1 => out1
            );
Process
begin
    sel <= '1'; in1 <= "00";in2<= "01";
    wait for 3 ns;
    sel <= '0'; in1 <= "10";in2<= "11";
    wait for 3 ns;
    wait;
end process;

process(out1)
    variable l : line;
begin
    write(l,"sel = "&to_string(sel)&" in1 =

```

```

"&to_string(in1)&" in2 = "&to_string(in2)&" out1 =
"&to_string(out1)&" time = "&to_string(now));
      writeline(output,l);
end process;
end architecture arch;

```

Following are the commands to run the example:

```

% vhdlan -vhdl08 test.vhd
% vcs top
% simv

```

## Output

```

sel = 0 in1 = UU in2 = UU out1 = UU time = 0 ns
sel = 1 in1 = 00 in2 = 01 out1 = 00 time = 0 ns
sel = 0 in1 = 10 in2 = 11 out1 = 11 time = 3 ns

```

## Limitation

Following is a limitation for this feature:

- Enhanced generics are not supported in component declarations.

## Support for Slices in Array Aggregates

VHDL 2008 supports the use of slices in array aggregates as defined in Section 9.3.3.3 of VHDL 2008 LRM.

An array aggregate allows you to create an array from a collection of various elements. In VHDL 2008, you can form array aggregates from a combination of individual elements and slices of the array.

VHDL 2008 also allows you to write an aggregate as the target of signal assignment statement. The names in the aggregate can be a mixture of element-typed signals and array-typed signals.

An example of bit\_vector array aggregate can be written as:

('1', "1100", '0')

This forms a six-element vector from the single element '1', the vector value "1100", and the single element '0'. The vector value forms a slice of the final aggregate value.

## Usage Example

### Example 23-14 SliceAggrEx1.vhd

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity ent is
End entity ent;

Architecture arch of ent is
signal Sig1:std_logic_vector(7 downto 0) :=
('0',"011011","1");
Begin
    process begin
        wait for 1 ns;
        report "sig1:" & to_string(Sig1);
        wait;
    end process;
End architecture arch;
```

The following are the commands to run the test case:

```
% vhdlan -vhdl108 -q sliceAggrEx1.vhd
% vcs ent -q
% simv -q
```

## Output

```
Report NOTE at 1 NS in design unit ENT(ARCH) from process /
ENT/_P0:
"sig1:00110111"
```

## Example 2:

### Example 23-15 SliceAggrEx2.vhd

```
Library ieee;
Use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
Entity ent is
Port (
    a, b : in unsigned(7 downto 0);
    sum : out unsigned(7 downto 0);
    c_out :out std_ulogic
);
End entity ent;

Architecture arch of ent is
Begin
    (c_out,sum) <= ('0' & a ) + ( '0' & b);
End architecture arch;

Library ieee;
Use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity tb is
end entity;

architecture a of tb is
    signal a, b , sum : unsigned(7 downto 0);
    signal c_out : std_ulogic;
begin

process begin
    a <= "00101110";
    b <= "10110110";
    wait for 5 ns;
    a <= "00101110";
    b <= "10110110";
    wait for 5 ns;
    wait;
end process;

inst : entity work.ent port map (a => a , b=> b , sum
```

```

=> sum , c_out => c_out);

      process(c_out,sum) begin
        report "c_out=" & to_string(c_out) & " sum=" &
to_string(sum);
      end process;
end architecture;

```

The following are the commands to run the test case:

```

% vhdlan -vhdl108 -q sliceAggrEx2.vhd
% vcs tb -q
% simv -q

```

## Output

```

Report NOTE at 0 NS in design unit TB(A) from process /TB/_P1:
"c_out=U sum=UUUUUUUU"
Report NOTE at 0 NS in design unit TB(A) from process /TB/_P1:
"c_out=X sum=XXXXXXXX"
Report NOTE at 0 NS in design unit TB(A) from process /TB/_P1:
"c_out=0 sum=11100100"

```

## Example 3:

This example shows the usage of inner most sub-aggregate of a multidimensional array aggregate.

### *Example 23-16 Usage of inner most sub-aggregate*

```

Library ieee;
Use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity tb is
end entity;

architecture a of tb is
  signal clk : bit := '0';

  type typeArr2D is array (natural range 0 to 1 ) of
std_logic_vector(3 downto 0);

```

```

type typeArr3D is array (natural range <>)of typeArr2D;

constant sigArr2D_1 : typeArr2D := (0 => "0001" , 1 =>
"0010" );
constant sigArr3D_1 : typeArr3D(1 downto 0) := (sigArr2D_1
,sigArr2D_1);
signal sigArr3D_2 : typeArr3D(5 downto 0);
signal sigArr3D_3 : typeArr3D(2 downto 0) := ( sigArr2D_1
, sigArr3D_1);

begin

process(clk) begin
if (clk = '1')then
    sigArr3D_2 <= (sigArr2D_1,sigArr3D_1,sigArr3D_3);
end if;
end process;

process begin
wait for 5 ns;
clk <= '1';
wait for 5 ns;
report "sigArr3D_2(0) [0=> (" &
to_string(sigArr3D_2(0)(0)) & ") , 1=( " &
to_string(sigArr3D_2(0)(1)) & ") ]";
report "sigArr3D_2(1) [0=> (" &
to_string(sigArr3D_2(1)(0)) & ") , 1=( " &
to_string(sigArr3D_2(1)(1)) & ") ]";
report "sigArr3D_2(2) [0=> (" &
to_string(sigArr3D_2(2)(0)) & ") , 1=( " &
to_string(sigArr3D_2(2)(1)) & ") ]";
report "sigArr3D_2(3) [0=> (" &
to_string(sigArr3D_2(3)(0)) & ") , 1=( " &
to_string(sigArr3D_2(3)(1)) & ") ]";
report "sigArr3D_2(4) [0=> (" &
to_string(sigArr3D_2(4)(0)) & ") , 1=( " &
to_string(sigArr3D_2(4)(1)) & ") ]";
report "sigArr3D_2(5) [0=> (" &
to_string(sigArr3D_2(5)(0)) & ") , 1=( " &
to_string(sigArr3D_2(5)(1)) & ") ]";
wait;
end process;
end architecture;

```

## Output

```
Report NOTE at 10 NS in design unit TB(A) from process /TB/
_P1:
-      "sigArr3D_2(0) [0=> (0001) , 1=(0010)]"
Report NOTE at 10 NS in design unit TB(A) from process /TB/
_P1:
-      "sigArr3D_2(1) [0=> (0001) , 1=(0010)]"
Report NOTE at 10 NS in design unit TB(A) from process /TB/
_P1:
-      "sigArr3D_2(2) [0=> (0001) , 1=(0010)]"
Report NOTE at 10 NS in design unit TB(A) from process /TB/
_P1:
-      "sigArr3D_2(3) [0=> (0001) , 1=(0010)]"
Report NOTE at 10 NS in design unit TB(A) from process /TB/
_P1:
-      "sigArr3D_2(4) [0=> (0001) , 1=(0010)]"
Report NOTE at 10 NS in design unit TB(A) from process /TB/
_P1:
-      "sigArr3D_2(5) [0=> (0001) , 1=(0010)]"
(simv): Simulation complete, time is 10 NS.
```

## Limitation

Following is the limitation for this feature:

- This feature is only applicable for one-dimensional array aggregate or the inner most sub-aggregate of a multidimensional array aggregate.

---

## Support for Type Conversion in VHDL 2008

VHDL 2008 extends the support for use of different element types in the source and target array expression types.

VHDL 2008 supports the following enhancements in type conversion:

- An array can be converted to another array whether the index subtype of the target and operand is closely related or not. In the case of arrays, only the element types need to be closely related.
- If the target sub-type is an array type for which the index ranges are not defined, then the bounds of the result are checked whether they belong to the corresponding index subtype of the target type. If the target subtype is an array sub-type for which the index ranges are defined, then each element is checked for the operand if there is a matching element of the target sub-type.
- A record can be converted to its sub-type.

## Usage Example

### Example for Type Conversion

*Example 23-17 Type conversion*

```

entity ent is

end entity;

architecture arch of ent is
    type exp_type is (int, undef, trap, ovf, div0);
    type exp_vector is array (exp_type) of bit;

    signal d_bitVect: bit_vector(31 downto 0);
    signal exp_reg : exp_vector;

    signal realVect :real_vector(0 to 3);
    signal intVect :integer_vector(0 to 3);
begin

    d_bitVect(31 downto 27) <= bit_vector( exp_reg );
    intVect <=integer_vector(realVect);

    process begin
        wait for 1 ns;

```

```

        exp_reg <= "10010";
realVect <= ( 0=> 1.25 , 1=>2.67 , 2=>2.85 , 3=>6.75);

        wait for 1 ns;
        report "exp_reg=" & to_string(exp_reg);
        report "d_bitVect(31 downto 27)=" &
to_string(d_bitVect(31 downto 27));
        report "realVect=( " & to_string(realVect(0)) & ", "
& to_string(realVect(1)) & "," & to_string(realVect(1)) &
", " & to_string(realVect(2)) & ")" ;
        report "intVect=( " & to_string(intVect(0)) & ", "
& to_string(intVect(1)) & "," & to_string(intVect(1)) & ","
& to_string(intVect(2)) & ")" ;
        wait;
    end process;
end architecture;

```

**The following are the commands to run the test case:**

```
% vhdlan -vhdl08 -q test.vhd
% vcs ent -q
% simv -q
```

## Output

```
2 NS
Report NOTE at 2 NS in design unit ENT(ARCH) from process /
ENT/_P2:
  "exp_reg=10010"
Report NOTE at 2 NS in design unit ENT(ARCH) from process /
ENT/_P2:
  "d_bitVect(31 downto 27)=10010"
Report NOTE at 2 NS in design unit ENT(ARCH) from process /
ENT/_P2:
  "realVect=(1.25e+0,2.67e+0,2.67e+0,2.85e+0)"
Report NOTE at 2 NS in design unit ENT(ARCH) from process /
ENT/_P2:
  "intVect=(1,3,3,3)"
(simv): Simulation complete, time is 2 NS.
```

## Limitation

Following is the limitation for this feature:

- Type conversion is not supported for unconstrained array elements and record elements.

---

## Support for Case Expression Subtype

VCS supports the following enhancements for VHDL 2008:

- The case expression need not have a locally static index range. It must have the choices that have same length as the case expression.
- An array aggregate containing others is allowed as a choice in a case statement, if the index range of the case expression is locally static.

## Example for Case Expression

The following example shows case expression with concatenation (which are not locally static), but the case choices are locally static.

### *Example 23-18 Usage of case expression*

```
entity ent is
end entity;

architecture arch of ent is
    signal sel1 : bit_vector (1 downto 0);
    signal sel2 : bit_vector (1 downto 0);
    signal out1 : bit_vector (3 downto 0);

    signal clk : bit := '0';
    signal endSim : bit := '0';
begin
```

```

process (clk) begin
    if (clk ='1') then
        case (sel1 & sel2) is
            when "0000" => out1 <= "0000";
            when "0001" => out1 <= "0001";
            when "0010" => out1 <= "0010";
            when "0100" => out1 <= "0100";
            when "1000" => out1 <= "1000";
            when others => out1 <= "1111";
        end case;
    end if;
end process;

process begin
    wait for 4 ns;
    sel1 <= "00";
    sel2 <= "01";
    wait for 10 ns;
    sel1 <= "00";
    sel2 <= "10";
    wait for 10 ns;
    sel1 <= "01";
    sel2 <= "00";
    wait for 10 ns;
    sel1 <= "10";
    sel2 <= "00";
    wait for 10 ns;
    sel1 <= "11";
    sel2 <= "11";
    wait for 10 ns;
    endSim <= '1';
    wait;
end process;

process begin
    clk <= '0';
    wait for 5 ns;
    clk <= '1';
    wait for 5 ns ;
    if (endSim = '1') then
        wait;
    end if;

```

```

    end process;

    process (out1) begin
        report "out1 =" & to_string(out1);
    end process;

end architecture;

```

**The following are the commands to run the test case:**

```

% vhdlan -vhdl08 -q test.vhd
% vcs ent -q
% simv -q

```

## **Output**

```

Report NOTE at 0 NS in design unit ENT(ARCH) from process /
ENT/_P3:
    "out1 =0000"
5 NS
Report NOTE at 5 NS in design unit ENT(ARCH) from process /
ENT/_P3:
    "out1 =0001"
15 NS
Report NOTE at 15 NS in design unit ENT(ARCH) from process
/ENT/_P3:
    "out1 =0010"
25 NS
Report NOTE at 25 NS in design unit ENT(ARCH) from process
/ENT/_P3:
    "out1 =0100"
35 NS
Report NOTE at 35 NS in design unit ENT(ARCH) from process
/ENT/_P3:
    "out1 =1000"
45 NS
Report NOTE at 45 NS in design unit ENT(ARCH) from process
/ENT/_P3:
    "out1 =1111"
(simv): Simulation complete, time is 60 NS.

```

This example shows the case choices with concatenation (not locally static), but the case expression is locally static.

```
entity ent is
end entity;

architecture arch of ent is
    signal sell1 : bit_vector (3 downto 0);
    signal out1 : bit_vector (3 downto 0);

    signal clk : bit :='0';
    signal endSim : bit :='0';
begin
    process (clk) begin
        if (clk ='1') then
            case (sell1 ) is
                when "00"&"00" => out1 <= "0000";
                when '0' & "001" => out1 <= "0001";
                when "00"&"10" => out1 <= "0010";
                when "01"&"00" => out1 <= "0100";
                when "10"&"00" => out1 <= "1000";
                when others => out1 <= "1111";
            end case;
        end if;
    end process;

    process begin
        wait for 4 ns;
        sell1 <= "0001";
        wait for 10 ns;
        sell1 <= "0010";
        wait for 10 ns;
        sell1 <= "0100";
        wait for 10 ns;
        sell1 <= "1000";
        wait for 10 ns;
        sell1 <= "1111";
        wait for 10 ns;
        endSim <= '1';
        wait;
    end process;
```

```

    end process;

    process begin
        clk <= '0';
        wait for 5 ns;
        clk <= '1';
        wait for 5 ns ;
        if (endSim = '1') then
            wait;
        end if;
    end process;

    process (out1)  begin
        report "out1 =" & to_string(out1);
    end process;
end architecture;

```

**The following are the commands to run the test case:**

```
% vhdlan -vhdl08 -q test1.vhd
% vcs ent -q
% simv -q
```

## Output

```

Report NOTE at 0 NS in design unit ENT(ARCH) from process /
ENT/_P3:
    "out1 =0000"
5 NS
Report NOTE at 5 NS in design unit ENT(ARCH) from process /
ENT/_P3:
    "out1 =0001"
15 NS
Report NOTE at 15 NS in design unit ENT(ARCH) from process
/ENT/_P3:
    "out1 =0010"
25 NS
Report NOTE at 25 NS in design unit ENT(ARCH) from process
/ENT/_P3:
    "out1 =0100"
35 NS
Report NOTE at 35 NS in design unit ENT(ARCH) from process

```

```

/ENT/_P3:
    "out1 =1000"
45 NS
Report NOTE at 45 NS in design unit ENT(ARCH) from process
/ENT/_P3:
    "out1 =1111"
(simv): Simulation complete, time is 60 NS.

```

---

## Support for Subtypes of Ports and Parameters

VCS supports the following enhancements for VHDL 2008:

- The subtype rules for scalar ports and scalar signal parameters are allowed to support runtime subtype checks.
- The subtype of the signal must be compatible with the subtype of the port for a port of mode ‘in’ or ‘inout’. Also, the subtype of the port must be compatible with the subtype of the signal for a port of mode ‘out’ or ‘inout’.

## Example for Subtypes of Ports and Parameters

### *Example 23-19 Subtypes of ports and parameters*

```

entity leaf is
    port (in1 : in integer;
          out1 : out positive );
end entity;

architecture arch of leaf is
begin

end architecture;

entity ent is
end entity;

architecture arch of ent is

```

```

        signal sigInt : positive;
        signal sigOut : integer;
begin
    intsl : entity work.leaf port map( sigInt,sigOut);

    process begin
        sigInt <= 10;
        wait for 1 ns;
        report "sigOut=" & to_string(sigOut);
        wait;
    end process;
end architecture;

```

The following are the commands to run the test case:

```

% vhdlan -vhdl08 -q test.vhd
% vcs ent -q
% simv -q

```

## Output

```

Report NOTE at 1 NS in design unit ENT(ARCH) from process /
ENT/_P0:
    "sigOut=1"
(simv): Simulation complete, time is 1 NS.

```

## Support for Static Composite Expressions

VCS provides additional support for the use of locally static expressions as defined in *VHDL 2008 LRM, Section 9.4*. A locally static expression can be of a composite type, if the subtype of each of the primaries in the expression is locally static and has a locally static subtype.

VCS expands the support for the following kinds of locally static expressions:

- Composite types, such as arrays and records.

- Allowed primaries in a locally expression can be array or record aggregates, indexed array elements, array slices, selected record elements and so on.
- The operators in an expression can either be any of the predefined operators or functions or those defined in the standard packages, such as `std_logic_1164`, `numeric_bit`, `numeric_std`, `numeric_bit_unsigned`, and `numeric_std_unsigned`.

## Usage Examples

The following examples show the use of static composite expressions in case and select statements:

### Use of Case Choice and Constant Initializations

*Example 23-20 Usage of case choice and constant initializations*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity e is
port(
    p1:in bit_vector(3 downto 0);
    p2:in std_logic_vector(3 downto 0);
    p3:in unsigned(3 downto 0));
end entity;
architecture arch_e of e is
    constant kbv1:bit_vector(3 downto 0):=(others => '0');
    constant kbv2:bit_vector(3 downto 0):='0'&(kbv1(2
downto 0) xor "001");

    constant kst1:std_logic_vector(3 downto 0):=(others
=> '0');
    constant kst2:std_logic_vector(2 downto
0):=To_StdULogicVector(kbv2(2 downto 0));

    constant kun1:unsigned(2 downto 0):=(others => '0');
    constant kun2:unsigned(3 downto
```

```

0) :=resize(unsigned'("10"),4);
      constant kun3:unsigned(3 downto
0) :=SHIFT_RIGHT("1010",1);

begin
  -- predefined types bit-vector
  process(p1)begin
    case(p1)is
      when kbv1                      => report
      "p1:bit_vector value is 0"; -- 0000
      when "00"&"01"                  => report
      "p1:bit_vector value is 1"; -- 0001
      when ("0001" sll 1)            => report
      "p1:bit_vector value is 2"; -- 0010
      when ("0111" and "1011")       => report
      "p1:bit_vector value is 3"; -- 0011
      when kbv2(1 downto 0)&"00"     => report
      "p1:bit_vector value is 4"; -- 0100
      when (2=>kbv2(0),0=>'1',others=>'0') => report
      "p1:bit_vector value is 5"; -- 0101
      when others                     => report
      "p1:bit_vector value is unknown";
      end case;
    end process;
    -- std_logic_vector
    process(p2)begin
      case(p2)is
        when kst1                      => report
        "p2:std_logic_vector value is 0"; -- 0000
        when '0'&kst2                  => report
        "p2:std_logic_vector value is 1"; -- 0001
        when ("0001" sll 1)            => report
        "p2:std_logic_vector value is 2"; -- 0010
        when ("0111" and "1011")       => report
        "p2:std_logic_vector value is 3"; -- 0011
        when kst2(1 downto 0)&"00"     => report
        "p2:std_logic_vector value is 4"; -- 0100
        when (2=>kst2(0),0=>'1',others=>'0') => report
        "p2:std_logic_vector value is 5"; -- 0101
        when others                     => report
        "p2:std_logic_vector value is unknown";
        end case;
      end process;

```

```

    end process;
    -- signed/unsigned from numeric_std
process(p3)begin
    case(p3)is
        when kun1&'0'                      => report
    "p3:unsigned value is 0"; -- 0000
        when to_unsigned(1,4)                => report
    "p3:unsigned value is 1"; -- 0001
        when kun2                          => report
    "p3:unsigned value is 2"; -- 0010
        when ("0111" and "1011")           => report
    "p3:unsigned value is 3"; -- 0011
        when SHIFT_LEFT(kun2,1)            => report
    "p3:unsigned value is 4"; -- 0100
        when kun3                          => report
    "p3:unsigned value is 5"; -- 0101
        when others                        => report
    "p3:unsigned value is unknown";
        end case;
    end process;
end arch_e;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity tb is
end entity;
architecture arch_tb of tb is
    signal p1: bit_vector(3 downto 0);
    signal p2: std_logic_vector(3 downto 0);
    signal p3: unsigned(3 downto 0);
begin
    dut_inst:entity work.e port map (p1,p2,p3);
process begin
    wait for 5 ns;
    p1 <= "0000";wait for 5 ns;
    p1 <= "0001";wait for 5 ns;
    p1 <= "0010";wait for 5 ns;
    p1 <= "0011";wait for 5 ns;
    p1 <= "0100";wait for 5 ns;
    p1 <= "0101";wait for 5 ns;
    p1 <= "0110";wait for 5 ns;

```

```

        p2 <= "0000";wait for 5 ns;
        p2 <= "0001";wait for 5 ns;
        p2 <= "0010";wait for 5 ns;
        p2 <= "0011";wait for 5 ns;
        p2 <= "0100";wait for 5 ns;
        p2 <= "0101";wait for 5 ns;
        p2 <= "0110";wait for 5 ns;

        p3 <= "0000";wait for 5 ns;
        p3 <= "0001";wait for 5 ns;
        p3 <= "0010";wait for 5 ns;
        p3 <= "0011";wait for 5 ns;
        p3 <= "0100";wait for 5 ns;
        p3 <= "0101";wait for 5 ns;
        p3 <= "0110";wait for 5 ns;
        wait;
    end process;
end architecture;

```

## Output

```

Report NOTE at 0 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P0:
    "p1:bit_vector value is 0"
Report NOTE at 0 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
    "p2:std_logic_vector value is unknown"
Report NOTE at 0 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P2:
    "p3:unsigned value is unknown"
10 NS
Report NOTE at 10 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P0:
    "p1:bit_vector value is 1"
15 NS
Report NOTE at 15 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P0:
    "p1:bit_vector value is 2"
20 NS
Report NOTE at 20 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P0:

```

```

        "p1:bit_vector value is 3"
25 NS
Report NOTE at 25 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P0:
        "p1:bit_vector value is 4"
30 NS
Report NOTE at 30 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P0:
        "p1:bit_vector value is 5"
35 NS
Report NOTE at 35 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P0:
        "p1:bit_vector value is unknown"
40 NS
Report NOTE at 40 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
        "p2:std_logic_vector value is 0"
45 NS
Report NOTE at 45 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
        "p2:std_logic_vector value is 1"
50 NS
Report NOTE at 50 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
        "p2:std_logic_vector value is 2"
55 NS
Report NOTE at 55 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
        "p2:std_logic_vector value is 3"
60 NS
Report NOTE at 60 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
        "p2:std_logic_vector value is 4"
65 NS
Report NOTE at 65 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
        "p2:std_logic_vector value is 5"
70 NS
Report NOTE at 70 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
        "p2:std_logic_vector value is unknown"
75 NS
Report NOTE at 75 NS in design unit E(ARCH_E) from process

```

```

/TB/DUT_INST/_P2:
    "p3:unsigned value is 0"
80 NS
Report NOTE at 80 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P2:
    "p3:unsigned value is 1"
85 NS
Report NOTE at 85 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P2:
    "p3:unsigned value is 2"
90 NS
Report NOTE at 90 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P2:
    "p3:unsigned value is 3"
95 NS
Report NOTE at 95 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P2:
    "p3:unsigned value is 4"
100 NS
Report NOTE at 100 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P2:
    "p3:unsigned value is 5"
105 NS
Report NOTE at 105 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P2:
    "p3:unsigned value is unknown"
(simv): Simulation complete, time is 110 NS.

```

## Use of Select Choice and Constant Initialization

*Example 23-21 Usage of select choice and constant initialization*

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity e is
port(
    p3:in unsigned(3 downto 0));
end entity;
architecture arch_e of e is
    constant kun1:unsigned(2 downto 0):=(others => '0');
    constant kun2:unsigned(3 downto

```

```

0):=resize(unsigned'("10"),4);
constant kun3:unsigned(3 downto
0):=SHIFT_RIGHT("1010",1);

signal s_out :integer:=6;
begin
    with p3 select s_out <=
        0 when kun1&'0',
        1 when to_unsigned(1,4),
        2 when kun2,
        3 when ("0111" and "1011"),
        4 when SHIFT_LEFT(kun2,1),
        5 when kun3,
        6 when others;

    process(s_out)begin report "s_out:
"&to_string(s_out);end process;
end arch_e;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity tb is
end entity;
architecture arch_tb of tb is
    signal p3: unsigned(3 downto 0);
begin
    dut_inst:entity work.e port map (p3);
process begin
    p3 <= "0000";wait for 5 ns;
    p3 <= "0001";wait for 5 ns;
    p3 <= "0010";wait for 5 ns;
    p3 <= "0011";wait for 5 ns;
    p3 <= "0100";wait for 5 ns;
    p3 <= "0101";wait for 5 ns;
    p3 <= "0110";wait for 5 ns;
    wait;
end process;
end architecture;

```

## Output

```
Report NOTE at 0 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
"s_out: 6"
Report NOTE at 0 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
"s_out: 0"
5 NS
Report NOTE at 5 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
"s_out: 1"
10 NS
Report NOTE at 10 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
"s_out: 2"
15 NS
Report NOTE at 15 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
"s_out: 3"
20 NS
Report NOTE at 20 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
"s_out: 4"
25 NS
Report NOTE at 25 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
"s_out: 5"
30 NS
Report NOTE at 30 NS in design unit E(ARCH_E) from process
/TB/DUT_INST/_P1:
"s_out: 6"
(simv): Simulation complete, time is 35 NS.
```

The following are the commands to execute:

```
% vhdlan -vhdl08 test.vhd
% vcs tb
% simv
```

## Limitations

The following are the limitations for the static composite expression feature:

1. Complete packages of `numeric_bit` and `numeric_bit_unsigned` are not supported.
2. The following function calls from `numeric_std` and `numeric_std_unsigned` packages are not supported:
  - minimum and maximum
  - `find_leftmost` and `find_rightmost`
  - `to_string` and similar string conversions
  - resize function with vector size is not supported but only scalar size is supported.
3. The string conversion function `to_string` and similar string conversions present in the `std_logic_1164` package are not supported.
4. Only `'ascending`, `'left`, `'right`, `'low`, `'high`, `'length`, and `'range` attributes are supported.
5. Aggregated and index selects from multidimensional arrays are not supported.
6. The physical data types are not supported.
7. Using aliased objects to slices of vectors is not supported.



# 24

## SAIF Support

---

The Synopsys Power Compiler enables you to perform power analysis and power optimization for your designs by entering the `power` command at the `vcs` prompt. This command outputs Switching Activity Interchange Format (SAIF) files for your design.

SAIF files support signals and ports for monitoring as well as constructs such as generates, enumerated types, and integers.

This chapter covers the following topics:

- [Using SAIF Files with VCS](#)
- [SAIF System Tasks for Verilog or Verilog-Top Designs](#)
- [The Flows to Generate a Backward SAIF File](#)
- [SAIF Calls That Can Be Used on VHDL or VHDL-Top Designs](#)
- [SAIF Support for Two-Dimensional Memories in v2k Designs](#)

- UCLI SAIF Dumping
- Criteria for Choosing Signals for SAIF Dumping
- Improving Simulation Time by Reducing the Overhead due to SAIF File Dumping

---

## Using SAIF Files with VCS

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.

---

## SAIF System Tasks for Verilog or Verilog-Top Designs

This section describes SAIF system tasks that you can use at the command line prompt.

Note that *mixedHdIScope* in the following discussion can be one of the following:

- Verilog scope
- VHDL scope
- Mixed HDL scope

Note also that a *design\_object* in the following discussion can be one of the following:

- Verilog scope or variable
- VHDL scope or variable
- Any mixed HDL scope or variable

`$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,  
                mixedHdlScope);
```

This task has a slight change in native SAIF implementation compared to PLI-based implementation. VCS considers only the arguments specified here for processing. Other arguments have no meaning.

VCS does not report signals in modules defined under the `celldefine compiler directive.

```
$read_lib_saif
```

Allows you to read in a state dependent and path dependent (SDPD) library forward SAIF file. It registers the state and path dependent information on the scope. It also monitors the internal nets of the design.

Syntax:

```
$read_lib_saif("inputFile");
```

```
$set_gate_level_monitoring
```

Allows you to turn on or off the monitoring of nets in the design.

Syntax:

```
$set_gate_level_monitoring("on" | "rtl_on");
```

The "on" and "rtl\_on" keyword arguments are called policies.

#### "rtl\_on"

Monitors all reg, tri, and trireg data objects for toggles. Monitors other types of nets for toggles if they are cell highconn (ports that connect toward the top of the design hierarchy in a module declared to be a cell).

#### "on"

Monitors all net type of objects for toggles. Monitors reg data objects if they are cell highconn. This is the default monitoring policy.

Note:

Verilog memories, Multi-dimensional arrays, and SystemVerilog data objects are supported with an extended syntax:

```
$set_gate_level_monitoring("on" | "rtl_on",
"mda" | "sv");
```

You include the `mda` argument for Verilog memories and multi-dimensional arrays, the `sv` argument for SystemVerilog data objects.

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.

---

## The Flows to Generate a Backward SAIF File

You can generate the following kinds of backward (or output) SAIF files:

- an SDPD backward SAIF file — using a library forward (or input) SAIF file
- a non-SDPD backward SAIF file — without using a library forward (or input) SAIF file.

---

### Generating an SDPD Backward SAIF File

To generate an SDPD backward SAIF file, include the SAIF system tasks in the module definition containing the

`$read_lib_saif("inputFile")` system task.

For example:

```
initial begin
    $read_lib_saif("inputFile");
    $set_toggle_region(mixedHdlsScope);
    // initialization of Verilog signals
    :
    $toggle_start;
    // testbench
    :
    $toggle_stop;
    $toggle_report("outputFile", timeUnit,mixedHdlsScope);
end
```

The `$set_toggle_region(mixedHdlsScope)` system task's scope argument must be one level higher in the design hierarchy than the scope of the module in the library forward SAIF file, for which you intend VCS to generate the backward SAIF file.

For example, if VCS monitors instance  
top.u\_dut.u\_saif\_module, the argument to the  
\$set\_toggle\_region system task is top.u\_dut, as follows:

```
$set_toggle_region(top.u_dut);
```

Enclose the modules listed in the library forward SAIF file, those from which you intend VCS to monitor and generate the backward SAIF file, between 'celldefine and 'endcelldefine compiler directives.

---

## Generating a Non-SPDP Backward SAIF File

If you are not including a library forward (or input) SAIF file, include the \$set\_gate\_level\_monitoring("on") system task with the other SAIF system tasks.

For example:

```
initial begin
    $set_gate_level_monitoring("on");
    $set_toggle_region(mixedHdScope);
    // initialization of Verilog signals, and then:
    $toggle_start;
    // testbench
    :
    $toggle_stop;
    $toggle_report("outputFile", timeUnit, mixedHdScope);
end
```

---

## SAIF Calls That Can Be Used on VHDL or VHDL-Top Designs

VHDL use model mainly consists of the `power` command and its options at the `simv` command-line.

The `power` command syntax is as follows:

```
power -enable -disable -reset -report <filename>
<synthesisTimeUnit> <mixedHd1Scope> <filename>
[<testbench_path_name>]-gate_level on|off|rtl_on
<region/signal/variable>
```

Here:

`-enable`

Enables monitoring of switching (`toggle_start`).

`-disable`

Disables monitoring of switching (`toggle_stop`).

`-reset`

Resets monitoring of switching (`toggle_reset`).

`-report`

Reports switching activity to an output file (`toggle_report`).

`-gate_level`

Turns on or off the monitoring based on the following:

`on`: Monitors both ports and signals.

`off`: Does not print ports or signals.

`rtl_on`: Monitors both ports and signals (same as on)

`<region/signal>`

Arguments for specifying the following:

**region:** Mixed HDL/VHDL region and its children to consider for monitoring.

**signal:** (hierarchical path to) signal name.

**Note:**

VHDL variables are not dumped in SAIF SDPD (VHDL gate level).

### Examples

```
# power -enable  
# power -report
```

---

## SAIF Support for Two-Dimensional Memories in v2k Designs

SAIF supports monitoring of two-dimensional memories in v2k designs.

You must pass the `mda` keyword to the `$set_gate_level_monitoring` system task to monitor two-dimensional memories in v2k designs.

**Note:**

You must pass the `+memcbk` compile-time option at vcs command-line, to dump two-dimensional wire or register.

If you want to dump through the UCLI command, you must pass the `mda` string to the `power -gate_level` command, as shown in the below section.

---

## UCLI SAIF Dumping

The following is the use model for UCLI SAIF dumping:

```
% simv -ucli  
ucli% power -gate_level on mda  
ucli% power <scope>  
ucli% power -enable  
ucli% run 100  
ucli% power -disable  
ucli% power -report <saif_filename> <timeUnit> <modulename>  
ucli% quit
```

---

## Criteria for Choosing Signals for SAIF Dumping

Verilog:

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 highconns based on the policy.

VHDL:

Signals or ports are supported for monitoring. Variables are not supported, as it is difficult to infer latches/flops at RTL level.

Constructs like generates, enumerated types, records, array of arrays integers, and so on, are also supported over and above the basic VHDL types.

The following rules are followed regarding the monitoring policy for VHDL:

|        | Port | Signals | Variables |
|--------|------|---------|-----------|
| on     | Y    | Y       | N         |
| off    | N    | N       | N         |
| rtl_on | Y    | Y       | N         |

Mixed HDL:

The rules for mixed HDL are basically the same as that of VHDL if VHDL is on top, and Verilog if Verilog is on top.

---

## Improving Simulation Time by Reducing the Overhead due to SAIF File Dumping

SAIF file dumping is enhanced to improve the simulation time by reducing the overhead due to SAIF.

---

### Use Model

At runtime, you can use the `-saif_opt` option with appropriate arguments to reduce the overhead due to SAIF file dumping as shown in the following command line:

```
% simv [simv_options] -saif_opt+option1+option2+...
```

You can specify one or more options along with the `-saif_opt` option. The options available are as follows:

`toggle_start_at_set_region`

**Use this option to implicitly call \$toggle\_start with \$set\_toggle\_region.**

toggle\_stop\_at\_toggle\_report

**Use this option to implicitly call \$toggle\_stop with \$toggle\_report.**

skip\_celldefine\_scopes

**Use this option to skip monitoring activity for the modules that are defined under the `celldefine compiler directive or are resolved using -v and -y options.**

---

## Example

Consider the following example:

```
`timescale 1ns/1ns
module top;
    wire w;
    bot b(w);
    initial begin
        #5 $set_gate_level_monitoring("rtl_on");
        $set_toggle_region(b);
        #95 $toggle_stop;
        $toggle_report("1.saif", 1e-9, b);
        $finish;
    end
endmodule

module bot(output reg p);
    initial begin
        #20 p = 1'b1;
        #20 p = 1'b0;
        #20 p = 1'b1;
        #20 p = 1'b0;
    end

```

```
endmodule
```

To run the example, use the following commands:

```
% vcs -sverilog 1.v  
% simv -saif_opt+toggle_start_at_set_region
```

It generates the following output:

```
/** The set_gate_level_monitoring command explicitly turns  
ON the internal nets monitoring **/  
(SAIFILE  
(SAIFVERSION "2.0")  
(DIRECTION "backward")  
(DESIGN)  
(VENDOR "Synopsys, Inc")  
(VERSION "1.0")  
(DIVIDER / )  
(TIMESCALE 1 ns)  
(DURATION 95.00)  
(INSTANCE top  
  (INSTANCE b  
    (NET  
      (p  
        (T0 40) (T1 40) (TX 15)  
        (TC 3) (IG 0)  
      )  
    )  
  )  
)  
)
```

---

## Limitations

The feature has the following limitations:

- The `skip_celldefine_scopes` option is supported only if library forward SAIF file is not read.

- The enhancements are not supported for UCLI `power` command.

# 25

## Encrypting Source Files

---

There are different ways to encrypt your HDL source files to deliver your IP. Of these, this chapter describes the following methods to encrypt your Verilog and VHDL source files and exchange IPs.

- “[IEEE Verilog Standard 1364-2005 Encryption](#)”
- “[IEEE VHDL Standard 1076-2008 Encryption](#)”
- “[128-bit Advanced Encryption Standard](#)”
- “[Skipping Encrypted Source Code](#)”
- “[gen\\_vcs\\_ip](#)”

---

## IEEE Verilog Standard 1364-2005 Encryption

VCS supports encryption of Verilog and SystemVerilog IP code in protected envelopes as defined by the IEEE Standard 1364-2005.

In addition, VCS supports the recommendations from the IEEE P1735 working group for encryption interoperability between different encryption and decryption tools, denoted as “version 1” by P1735.

Note:

SystemC encryption is not supported by this feature.

The following option tells VCS to encrypt the specified Verilog or SystemVerilog source files according to the “IEEE Std 1364-2005” standard for encryption envelopes.

```
-ipprotect protection_header_file
```

In this encryption mode, VCS does not compile the Verilog or SystemVerilog source files, but instead encrypts each source file into a separate encrypted Verilog or SystemVerilog file. Each encrypted file is saved under the same filename, but changes its filename extension to .vp. Using the -ipprotect option allows IP providers to specify a *protection\_header\_file* that contains various protection pragmas.

VCS encrypts:

- the source files on the vcs command line
- the source files specified in ‘include compiler directives.

**Note:**

- By default, VCS encrypts complete input files. Use the `-ipopt=partialprotect` option and argument to enable partial protection with it, VCS encrypts only the regions specified by 'protect' pragma begin-end expressions.
- All `include directives in the encrypted source files are modified by changing the extension of the included filenames from .v to .vp. The modified `include directives are left as unencrypted text. In addition, every file included by a `include directive is also encrypted and saved under the modified filename (changing the extension to .vp). Use the `-ipopt=noincludeprotect` option and argument with the `-ippprotect` option to disable processing of `include compiler directives and the source files included by it.

This section on the IEEE Std 1364-2005 encryption mode includes the following:

- “The Protection Header File” on page 3
- “Other Options for IEEE Std 1364-2005 Encryption Mode” on page 6
- “How Protection Envelopes Work” on page 9
- “VCS Public Encryption Key ” on page 10
- “Creating Interoperable Digital Envelopes Using VCS - Example” on page 11
- “Discontinued -ipkey Option” on page 15

---

## The Protection Header File

The `protection_header_file` may look like the following:

### **Example 25-1 Sample IEEE Encryption Header File**

```
`pragma protect data_method = "x-abc"  
`pragma protect encoding = (enctype = "base64")  
`pragma protect key_keyowner="IPcorp"  
`pragma protect key_method="rsa"  
`pragma protect key_keyname="IPcorp-123"  
`pragma protect key_public_key  
<content_representing_the_public_encryption_key>
```

For the VCS base64 encoded RSA public encryption key, contact Synopsys support ([vcs\\_support@synopsys.com](mailto:vcs_support@synopsys.com)).

The following `pragma protect expressions are required inside the *protection\_header\_file*:

`key_keyowner`

Identifies the owner of the key encryption key.

`key_method`

Specifies the key encryption algorithm, the asymmetric method for encrypting or decrypting.

`key_keyname`

Specifies keyowner's key name.

`key_public_key`

Specifies the public key for key encryption

The optional `pragma protect expressions that can be included are as follows:

`data_method`

Identifies the data encryption algorithm. Supported methods are as follows:

|            |            |            |
|------------|------------|------------|
| aes256-cbc | aes192-cbc | aes128-cbc |
| des-cbc    | 3des-cbc   |            |

The default `data_method`, if none is specified, is `aes256-cbc`.

`author`

Identifies the author of an envelope.

`author_info`

Specifies additional author information.

`encoding`

Specifies the coding scheme for encrypted data, you can specify either of the following:

|        |          |
|--------|----------|
| base64 | uuencode |
|--------|----------|

The default encoding scheme, if none is specified, is `base64`.

`comment`

Comment documentation string that is not encrypted.

**Note:**

These encryption pragmas are only supported inside the `protection_header_file`, which is specified by the `-ipprotect` option. If they are specified anywhere else (such as in the Verilog or SystemVerilog source files), VCS outputs a warning message and ignores the pragma.

The only `pragma protect expressions allowed in input Verilog and SystemVerilog files are `pragma protect begin and `pragma protect end, when enabled with the -ipopt=partialprotect option and argument to mark the regions to be protected.

## Unsupported Protection Pragma Expressions

The `pragma protect expressions that are not currently supported include:

|                 |                  |
|-----------------|------------------|
| data_keyowner   | data_keyname     |
| data_public_key | data_decrypt_key |
| decrypt_license | runtime_license  |
| reset           | viewpoint        |

Also unsupported are any expressions beginning with digest\_.

---

## Other Options for IEEE Std 1364-2005 Encryption Mode

In addition to the -ipprotect option, there are other options that you can use in this mode. This section describes them.

-ipopt=partialprotect

VCS encrypts complete file by default. Use this option to encrypt only regions marked by the pragmas: `pragma protect begin and `pragma protect end in the Verilog or SystemVerilog source files.

`-ipopt=noincludeprotect`

VCS in encryption mode encrypt files which are included by the `\include` compiler directive. Use this option to disable the processing of the `\include` compiler directive and files included by it.

`-ipopt=ext=ext`

Use this option to specify the filename extension for encrypted files.

`+incdir+directory+...`

Specifies the directories that VCS searches for source files specified with the `\include` compiler directive. By default VCS writes encrypted versions of these source files in the directory in which it finds the source files.

The encrypted copies have the same filename and extension of the original except that the `p` character is appended to the filename extension. So for example if it finds a SystemVerilog source file in a Verilog library with the name `dev1.sv`, the encrypted version in that library is `dev1.svp`.

You can specify multiple Verilog libraries with this option by using the plus (+) character as a delimiter, for example:

`+incdir+INTRCTR+IOMTR+/DW/SIMENV`

`-f | -F | -file filename`

Specifies a file that contains a list of Verilog or SystemVerilog source files to be encrypted. The `-f`, `-F`, and `-file` options are interchangeable in this encryption mode.

`+define+MACRO=VALUE`

Defines the specified text macro to the specified value.

A text macro so defined at encryption time (when encrypting files instead of compiling files) cannot be overridden at a subsequent compile-time (when including the encrypted files in some later compilation and also entering the `+define` option). VCS ignores the attempted override without displaying any error, warning, or informational message.

`-ipout filename.ext`

This option tells VCS to write the encrypted file for the first Verilog or SystemVerilog source file on the command line with the specified filename and extension. You can enter a pathname for the protected file.

This option only works for the first Verilog or SystemVerilog source file on the `vcs` command line, and does not work for other source files on the command line or files included with the `'include` compiler directive or in Verilog libraries.

---

## How Protection Envelopes Work

As specified in IEEE Std 1364-2005, annex H “Encryption/decryption flow,” section H.3 Digital envelopes:

“The sender encrypts the design using a symmetric key encryption algorithm and then encrypts the symmetric key using the recipient’s public key. The encrypted symmetric key is recorded in a `key_block` in the protected envelope. The recipient is able to recover the symmetric key using the appropriate private key and then decrypts the design with the symmetric key.”

Protection envelopes work as follows:

1. The encrypting tool generates a random key called "session key."
2. The encrypting tool then encrypts the design using this session key.
3. For each potential decrypting tool, information about that tool must be provided using `'pragma protect' expressions` in the encryption envelope.

This information includes `key_keyowner`, `key_keyname`, the asymmetric `key_method`, and `key_public_key` for each tool.

4. The encrypting tool then encrypts the session key multiple times, once for each decrypting tool using information provided in the encryption envelope for that tool.
5. The encrypted session key is then recorded in `key_blocks` in the protected envelope.

Multiple `key_blocks` are generated, one for each decrypting tool.

6. The decrypting tool examines the `key_blocks` in the decryption envelope to find one encrypted using a key to which the tool has access.
7. The decrypting tool is able to recover the session key from its `key_block` using the appropriate private key.
8. The decrypting tool then decrypts the design with the session key.

---

## VCS Public Encryption Key

For the VCS base64 encoded RSA public encryption key, contact Synopsys support ([vcs\\_support@synopsys.com](mailto:vcs_support@synopsys.com)). Synopsys also provides use cases and examples along with the key.

The following `pragma protect expression identifies this key:

```
`pragma protect key_keyowner="IPcorp"  
`pragma protect key_method="rsa"  
`pragma protect key_keyname="IPcorp-123"
```

VCS can decrypt and compile source files, which are encrypted by VCS or third-party tools.

To allow VCS to decrypt encrypted source files, the following snippet must be included while encrypting.

```
`pragma protect key_keyowner="IPcorp"  
`pragma protect key_method="rsa"  
`pragma protect key_keyname="IPcorp-123"  
`pragma protect key_public_key  
<content_representing_the_public_encryption_key>
```

The following example illustrates the protection envelope methodology for using this key in Verilog or SystemVerilog source code.

---

## **Creating Interoperable Digital Envelopes Using VCS - Example**

VCS allows more than one key\_block in a single protected envelope so it can be decrypted by tools from different vendors.

In the following example an IP provider created encrypted source files that can be decrypted by two different EDA tools, VCS and tools from VendorX.

An IP provider retrieves public keys for an EDA tool according to the EDA tool vendor's specific policies. For VCS, please contact Synopsys support ([vcs\\_support@synopsys.com](mailto:vcs_support@synopsys.com)).

The *protection\_header\_file* that this example specifies with the `-ipprotect` option is in [Example 25-2](#).

**Example 25-2 Example Protection Header File for Source Encryption with VCS**

```
`pragma protect author = "IPProvider"
`pragma protect data_method = "x-abc"
`pragma protect encoding = (enctype = "base64")

`pragma protect key_keyowner="IPcorp"
`pragma protect key_method="rsa"
`pragma protect key_keyname="IPcorp-123"
`pragma protect key_public_key
<content_representing_the_public_encryption_key>
```



(1) key block for VCS

(2) key block for VendorX

**Example 25-3 Verilog Source File to be Encrypted**

```
// example.v
module secret (a, b);
    input a;
    output b;
    reg b;

    initial
        begin
            b = 0;
        end
```

```

always
begin
    #5 b = a;
end
endmodule

```

The following vcs command line generate the encrypted file example.vp which can be decrypted by VCS and tools from VendorX.

```
vcs -ipprotect pragma_header_file example.v
```

**Example 25-4 example.vp generated by VCS**

```

`pragma protect begin_protected
`pragma protect version=1
`pragma protect encrypt_agent="VCS"
`pragma protect encrypt_agent_info="G-2012.09-A[D]
(ENG) Build Date Feb 18 2012 00:14:12"
`pragma protect author="IPProvider"
`pragma protect key_keyowner="IPcorp"
`pragma protect key_keyname="IPcorp-123"
`pragma protect key_method="rsa"
`pragma protect encoding = (enctype = "base64",
line_length = 76, bytes = 128 )
`pragma protect key_block
<content_representing_the_public_encryption_key>

```



(1)

```

`pragma protect key_keyowner="VendorX"
`pragma protect key_keyname="VENDORX-123"
`pragma protect key_method="rsa"
`pragma protect encoding = (enctype = "base64",
line_length = 76, bytes = 128 )
`pragma protect key_block
<content_representing_the_public_encryption_key>

```



```

`pragma protect data_method="x-abc"

`pragma protect encoding = (enctype = "base64",
line_length = 76, bytes = 176 )
`pragma protect data_block

<content_representing_the_public_encryption_key>

`pragma protect end_protected

```



- ① Key block for VCS which contains the encrypted session key.  
(encrypted using VCS public RSA key)
- ② Key block for VendorX which contains the encrypted session key.  
(encrypted using VendorX public RSA key)
- ③ Data block which contains the encrypted IP (encrypted using the session key)

To determine the session key that was used to encrypt the data\_block:

- VCS retrieves the session key from first `key_block`
- VendorX uses the second `key_block`

Consequently, both implementations could successfully decrypt the data block which contains the encrypted IP.

---

## Discontinued -ipkey Option

The `-ipkey key` option will be obsolete in future releases.

IP providers should use `-ipprotect` instead. It allows you to specify various protection pragmas (via a protection header file) which are needed while generating interoperably encrypted IPs.

VCS will no longer use the key you pass with the `-ipkey key` option. It will generate a secure key internally.

---

## IEEE VHDL Standard 1076-2008 Encryption

VCS supports the encryption and decryption of the VHDL IP source code in protection envelopes, as defined by the IEEE VHDL Standard 1076-2008.

This section describes the following topics:

- “VHDL 1076-2008 Encryption Use Model”
- “Protection Envelopes”
- “VCS Public Encryption Key ”
- “Creating Interoperable Digital Envelopes Using VCS - Example”

- “Debug Protection”
  - “Limitations”
- 

## VHDL 1076-2008 Encryption Use Model

You must specify the `-ipprotect` option on the `vhdlan` command line to use VHDL 1076-2008 encryption. The `-ipprotect` option tells VCS to encrypt the VHDL source files specified on the `vhdlan` command line according to the IEEE VHDL Standard 1076-2008 for encryption envelopes.

The syntax of the `-ipprotect` option is as follows:

```
-ipprotect <header_file>
```

Where, `header_file` is the protection header file that contains various protection directives. For more information on the protection header file, see [Using the Protection Header File](#).

In the encryption mode, VCS does not analyze VHDL source files, instead it encrypts each source file into a separate encrypted VHDL file. It saves each encrypted file with the same filename, but changes its filename extension by appending `p` at the end of it.

Example:

```
% vhdlan -ipprotect toolkeys /src/dir/ipmod1.vhd \
package5.vhdl
```

The above command line generates:

/src/dir/ipmod1.vhdp and package5.vhdlp.

The following sections describe the use models to encrypt the complete or parts of the VHDL source files:

- [Encrypting the Entire VHDL Source Files](#)
- [Encrypting the Parts of VHDL Source Files](#)

## Encrypting the Entire VHDL Source Files

VCS encrypts the entire VHDL input files, by default. The following is the use model to encrypt entire VHDL source files:

```
% vhdlan -ipprotect <header_file> <VHDL_file_list>
```

During encryption, if a file contains a decryption envelope (`protect begin\_protected), VCS generates an error message and stops the encryption.

## Encrypting the Parts of VHDL Source Files

Use the `-ipopt=partialprotect` option to enable partial protection. The `-ipopt=partialprotect` option enables VCS to encrypt only the regions of the VHDL code specified between `protect begin and `protect end directives.

The use model to encrypt the parts of VHDL source files is as follows:

```
% vhdlan -ipprotect <header_file> -ipopt=partialprotect  
<VHDL_file_list>
```

vhdlan does not stop encryption if a file does not contain `protect begin or `protect end. It generates an error message and skips the encryption for that file.

## Using the Protection Header File

The protection header file (`header_file`) specifies the protect directives as per the VHDL 1076-2008 LRM. It contains an additional directive, namely, ``protect key_public_key`, which is a base64 encoded public key.

The ``protect key_public_key` directive replaces the ``protect key_block` directive in the VHDL 2008 LRM. That is, the VHDL encryption requires that the key block (the sequence of `key_keyowner`, `key_keyname`, and `key_method` directives) ends with a ``protect key_public_key` directive.

The syntax of `header_file` is as follows:

```
'protect data_method = "data_method"  
'protect key_keyowner="owner_name"  
'protect key_keyname="owner_key_name"  
'protect key_method="encryption_method_name"  
'protect key_public_key  
<content_representing_the_public_encryption_key>
```

where,

- The following ``protect` directives are required in `header_file`:

`key_keyowner`, `key_keyname`, `key_method`,  
`key_public_key`

- The following ``protect` directives are optional in `header_file`:

`data_method`, `encoding`

- The data block following the `protect key\_public\_key directive is an example of a base64 encoded version of a public key.

[Table 25-1](#) describes the `protect directives in header\_file.

*Table 25-1 The `protect Directives in header\_file*

| Directive      | Description                                                                                                                                                                                                                 |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| data_method    | <p>Identifies the data encryption algorithm.</p> <p>The supported data encryption algorithms are as follows:</p> <p>3des-cbc, aes128-cbc, aes192-cbc, aes256-cbc, des-cbc</p> <p>The default data_method is aes256-cbc.</p> |
| encoding       | <p>Specifies the coding scheme for an encrypted data.</p> <p>VCS only supports the base64 encoding scheme.</p> <p>The default encoding scheme is base64.</p>                                                                |
| key_keyowner   | Identifies the owner of the encryption key.                                                                                                                                                                                 |
| key_keyname    | Specifies the key name of keyowner.                                                                                                                                                                                         |
| key_method     | <p>Specifies the key encryption algorithm, that is, the asymmetric method for encrypting or decrypting.</p> <p>VCS supports the rsa encryption algorithm.</p>                                                               |
| key_public_key | Specifies the public key for the key encryption.                                                                                                                                                                            |

**Note:**

- The `protect encryption directives are only supported in the header\_file, which is specified by the -ipprotect option. If they are specified anywhere else, VCS generates a warning message and ignores the directives.
- The following directives are optional in the protection header file:

author, author\_info

## Options for VHDL 1076-2008 Encryption Mode

The options for the VHDL 1076-2008 encryption mode are as follows:

-ipopt=partialprotect

VCS encrypts an entire file by default. Use this option to encrypt only regions marked by the directives, namely `protect begin and `protect end, in VHDL source files.

-ipopt=ext=<ext>

Use this option to specify the filename extension for encrypted files.

**Example:**

```
% vhdlan -ipprotect toolkeys -ipopt=ext=de config6.vhd
```

The above command line generates:

```
config6.vhdde
```

-ipopt=outdir=<dir>

Use this option to specify the target directory for encrypted files.

**Example:**

```
% vhdlan -ipprotect toolkeys -ipopt=outdir=db6 /src/dir/
ipmod1.vhd package5.vhdl
```

The above command line generates:

db6/ipmod1.vhdp and db6/package5.vhdlp

-ipout *filename*

Use this option to allow VCS to write an encrypted file for the first VHDL source file on the command line with the specified file name and extension.

Example:

```
% vhdlan -ipprotect toolkeys -ipout encrypt.enc  
config6.vhd
```

The above command line generates an encrypted file, encrypt.enc.

-ipopt=overwrite

Use this option to allow VCS to replace any existing output files.

Note:

You must specify -ipprotect while using -ipopt or -ipout options; otherwise VCS generates an error message.

---

## Protection Envelopes

The `protect` encryption directives are used to form protection envelopes, which include specification of cryptographic methods and keys to be used by a tool. This section describes the functionality of the protection envelopes.

Protection envelopes work as follows:

1. The encrypting tool generates a random key called “session key”.
2. The encrypting tool encrypts a design using the session key.
3. Use `protect directives in the encryption envelope to provide the information about each potential decrypting tool.

This information includes key\_keyowner, key\_keyname, the asymmetric key\_method, and key\_public\_key for each tool.

4. The encrypting tool encrypts the session key multiple times, once for each decrypting tool using its information provided in the encryption envelope.
5. The encrypting tool records the encrypted session in key blocks in the protected envelope. The tool then generates multiple key blocks, one for each decrypting tool.
6. The decrypting tool examines the key blocks in the decryption envelope to find the encrypted envelope using a key to which the tool has access.
7. The decrypting tool is able to recover the session key from its key block using an appropriate private key.
8. The decrypting tool decrypts the design with the session key.

---

## The VCS Public Encryption Key

For the VCS base64 encoded RSA public encryption key, contact Synopsys support ([vcs\\_support@synopsys.com](mailto:vcs_support@synopsys.com)). Synopsys also provides use cases and examples along with the key.

The following `protect directives identify this key:

```
`protect key_keyowner="IPcorp"
```

```
`protect key_method="rsa"  
`protect key_keyname="IPcorp-123"
```

VCS can decrypt and analyze the source files, which are encrypted by VCS or third-party tools. To allow VCS to decrypt encrypted source files, include the following code snippet while encrypting:

```
`protect key_keyowner="IPcorp"  
`protect key_keyname="IPcorp-123"  
`protect key_method="rsa"  
`protect key_public_key  
<content_representing_the_public_encryption_key>
```

The following example illustrates the protection envelope methodology for using this key in Verilog or SystemVerilog source code.

---

## Usage Example

VCS allows multiple key blocks in a single protected envelope, so you can decrypt it using the tools from different vendors.

## Example for Full Encryption

**Example 25-2** specifies *header\_file* with the **-ipprotect** option.

*Example 25-5 Example header\_file for Source Encryption With VCS*

```
`protect data_method = "x-abc"  
`protect encoding = (enctype = "base64")  
`protect key_keyowner="IPcorp"  
`protect key_method="rsa"  
`protect key_keyname="IPcorp-123"  
`protect key_public_key  
<content_representing_the_public_encryption_key>
```

**Example 25-6 VHDL Source File to be Encrypted:** *example.vhd*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Counter is
Port(
clk : in STD_LOGIC;
Reset : in STD_LOGIC;
Count : out STD_LOGIC_VECTOR (3 downto 0);
Carry : out STD_LOGIC
);
end Counter;

architecture Behavioral of Counter is
    signal count_int : std_logic_vector(3 downto 0); -
-- define internal register
begin
    process (reset, clk)
    begin
        if reset = '1' then
            count_int <= "0000"; -- set counter, and
            carry <= '0';--carry to zero
        elsif clk'event and clk = '1' then
            if count_int <= "1000" then -- check count
                count_int <= count_int + "1"; --increment
                carry <= '0'; -- show still below 9
            else -- else we are at 9
                count_int <= "0000"; -- roll over count
                carry<= '1';-- flag roll over
            end if;
        end if;
    end process;
    count<= count_int;
-- send value to the outside
end Behavioral;
```

## **Perform the following commands:**

The following vhdlan command line generates the encrypted file, example.vhdp, which can be decrypted by VCS:

```
% vhdlan -ipprotect header_file example.vhd  
  
% vhdlan example.vhdp  
  
% vcs Counter
```

## **Example for Partial Encryption**

Refer [Example 25-2](#) that specifies *header\_file*.

*Example 25-7 VHDL Source File to be Encrypted:* example.vhd

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;  
  
entity Counter is  
Port(  
clk : in STD_LOGIC;  
Reset : in STD_LOGIC;  
Count : out STD_LOGIC_VECTOR (3 downto 0);  
Carry : out STD_LOGIC  
);  
end Counter;  
  
'protect begin  
architecture Behavioral of Counter is  
    signal count_int : std_logic_vector(3 downto 0); -  
- define internal register  
begin  
    process (reset, clk)  
    begin  
        if reset = '1' then
```

```

count_int <= "0000"; -- set counter, and
carry <= '0';--carry to zero
elsif clk'event and clk = '1' then
if count_int <= "1000" then -- check count
    count_int <= count_int + "1"; --increment
    carry <= '0'; -- show still below 9
else -- else we are at 9
    count_int <= "0000"; -- roll over count
    carry<= '1';-- flag roll over
    end if;
end if;
end process;
count<= count_int;
-- send value to the outside
end Behavioral;
`protect end

```

### **Perform the following commands:**

The following vhdlan command line generates the partially encrypted file, example.vhdp, which can be decrypted by VCS:

```

% vhdlan -ipprotect header_file -ipopt=partialprotect \
example.vhd

% vhdlan example.vhdp
% vcs Counter

```

### **Debug Protection**

This feature only supports debug protection at the design-unit level. If any part of a design unit is encrypted, then all of it is protected against access from various debug tools.

## Combining Encrypted and Unencrypted Code

A design can be unencrypted, entirely encrypted, or partially encrypted. This section consists of the following two sub-sections:

- [Entirely-Encrypted or Unencrypted Source Files](#)
- [Partially-Encrypted Source Files](#)

### Entirely-Encrypted or Unencrypted Source Files

Generics, ports, declarations, concurrent statements in entity declarations, and architecture bodies are protected from debug as per [Table 25-2](#).

*Table 25-2 Entirely-Encrypted or Unencrypted Source Files*

| Entity | Architecture | Generics | Ports | E. Decl | A. Decl | A. Conc |
|--------|--------------|----------|-------|---------|---------|---------|
| Plain  | Plain        | N        | N     | N       | N       | N       |
| Plain  | Crypt        | N        | N     | N       | Y       | Y       |
| Crypt  | Plain        | Y        | N     | Y       | N       | N       |
| Crypt  | Crypt        | Y        | N     | Y       | Y       | Y       |

where,

**E. Decl** indicates declarations other than generics and ports

**A. Decl** indicates architecture declarations

**A. Conc** indicates architecture concurrent statements

N indicates not protected

Y indicates protected

Plain indicates plain text, not encrypted

Crypt indicates entire design unit encrypted

See [Table 25-3](#) to determine protection for packages and package bodies.

*Table 25-3 Protection for Packages and Package Bodies*

| Package | Package Body | Package Declaration | Package Body Declaration |
|---------|--------------|---------------------|--------------------------|
| Plain   | Plain        | Not protected       | Not protected            |
| Plain   | Crypt        | Not protected       | Protected                |
| Crypt   | Plain        | Protected           | Not protected            |
| Crypt   | Crypt        | Protected           | Protected                |

### Partially-Encrypted Source Files

Partially-encrypted code in architecture bodies, packages, and package bodies has the same consequences for protection as if the entire design unit is encrypted, except in the case of entity declarations, where protection is determined as per [Table 25-4](#),

*Table 25-4 Partially-Encrypted Source Files*

| Encrypted Portion                       | Generic       | Port          | Declaration   | Concurrent Statements |
|-----------------------------------------|---------------|---------------|---------------|-----------------------|
| Context clause (all design units)       | Not protected | Not protected | Not protected | Not protected         |
| Entire entity except for context clause | Protected     | Not protected | Protected     | Protected             |
| Entity generic clause                   | Protected     | Not protected | Not protected | Not protected         |
| Entity port clause                      | Not protected | Not protected | Protected     | Protected             |

## **Assertion and Report Statements**

As per IEEE VHDL Standard 1076-2008 LRM:

“If a decryption tool executes an assertion statement (see 10.3) that causes an assertion violation, or executes a report statement (see 10.4), the message shall not include the name of the design unit containing the statement, the rules of 10.3 and 10.4 notwithstanding.”

Messages from assertion and report statements exclude any hierarchy from an encrypted region.

## **Hierarchy Attributes**

As per IEEE VHDL Standard 1076-2008 LRM:

“The value of any 'INSTANCE\_NAME or 'PATH\_NAME predefined attribute (see 16.2) formed by the decryption tool shall not include any element that is a name or label defined in a decrypted portion of a VHDL description, the rules of 16.2 notwithstanding.”

## **Profiling**

Simulation time and memory profiles do not display information on encrypted design units.

## **VHPI**

As per IEEE VHDL Standard 1076-2008 LRM:

“If a decrypted portion of a VHDL description includes an instantiation of a declaration that is declared in a portion of the VHDL description that is not encrypted, a decryption tool may provide access to a representation of the design sub-hierarchy whose root is the instance, provided the means of providing access does not contradict other requirements of this sub-clause. For example, a VHPI tool may return a handle to a VHPI object representing such an instance and allow navigation of associations from that reference object, provided the target objects represent parts of the design sub-hierarchy.”

## **VPD / VCD**

Recursive VCD, VPD, or EVCD dumping does not dump objects or other information from encrypted design units.

## **Coverage**

Coverage does not dump the information related to encrypted design units.

## **Error Messages**

During analysis, if an encrypted code is encountered in the design unit, then error message is replaced with a generic message.

VCS displays error messages during elaboration and simulation, but hides the source code from the encrypted portions of a design.

---

## **Limitations**

VCS does not support the following:

- Nested encryption envelopes in the source files
- Nested decryption envelopes
- The following protect directives:

`data_keyname, data_keyowner, data_public_key,  
decrypt_license, digest_block, digest_keyname,  
digest_keyowner, digest_key_method, digest_method,  
runtime_license, viewport`

- The following encrypt data methods:

`blowfish-cbc, twofish128-cbc, twofish192-cbc,  
twofish256-cbc, serpent128-cbc, serpent192-cbc,  
serpent256-cbc, cast128-cbc`

- The following encoding types:

`quoted-printable, raw`

- The following asymmetric key methods:

`elgamal, pgp-rsa`

- VHDL 2008 encryption which specifies encryption envelopes to be present entirely in the source files.

---

## 128-bit Advanced Encryption Standard

VCS uses the 128-bit Advanced Encryption Standard (AES) to encrypt the Verilog files. The 128-bit key is generated internally by VCS. This 128-bit encryption methodology is exclusive to VCS, and can be decrypted only by VCS.

This section includes the following topics:

- “Compiler Directives for Source Protection”
  - “Using Compiler Directives or Pragmas”
  - “Automatic Protection Options”
  - “Using Automatic Protection Options”
  - “Protecting ‘include File Directive”
  - “Enabling Debug Access to Ports and Instance Hierarchy”
  - “Debugging Partially Encrypted Source Code”
- 

### Compiler Directives for Source Protection

`protect

Defines the start of protected code. Syntax: `protect

`endprotect

Defines the end of protected code. Syntax: `endprotect

`protected

Defines the start of protected code. Syntax: `protected

```
`endprotected
```

Defines the end of protected code. Syntax: `endprotected

```
`protect128
```

Defines the start of protected code. Syntax: `protect128

```
`endprotect128
```

Defines the end of protected code. Syntax: `endprotect128

---

## Using Compiler Directives or Pragmas

You can use VCS to encrypt selected parts of your source files. In order to achieve this, complete the following steps:

### **-protect128**

1. Enclose the Verilog code that you want to encrypt between the `protect128 and the `endprotect128 compiler directives.
2. Compile the files with the -protect128 option. For example:

```
% vcs -protect128 foo.v
```

When you compile the design with the -protect128 option, VCS creates a new file with the .vp extension for each Verilog file specified in the command line. For example, VCS creates foo.vp when you execute the command listed above.

In the .vp files, VCS replaces the `protect128 and `endprotect128 compiler directives with the `protected128 and `endprotected128 compiler directives and encrypts the code in between these directives.

### Note:

- If you specify the `protect` and `protect128` compile options on the same vcs command line, VCS ignores the `protect128` option and uses the `protect` option. It also reports a warning message.
- The `protect128` and `genip` options are mutually exclusive, you cannot specify both of these options on the same vcs command line.

### Example

The following Verilog file illustrates the use of '`protect128` and '`endprotect128` to mark the code that needs to be encrypted:

```
cat test.v
module counter( inp, outp);
  input [7:0] inp;
  output [7:0] outp;
  reg [7:0] count;
  always
    begin:counter
      `protect128
      reg [7:0] int;
      count = 0;
      int = inp;
      while (int)
        begin
          if (int [0]) count = count + 1;
          int = int >> 1;
        end
      `endprotect128
    end
    assign outp = count;
  endmodule

  module top;
    parameter p1 = 3;
```

```

wire mux;
reg control,dataA,dataB;
dut #(.p1(3)) d1(mux,control,dataA,dataB);
counter c1(inp,outp);
initial begin
    control=0;
    dataA=1;
    dataB=0;
#2; dataA=1;dataB=1;
#2;dataB=1'bx;
#2; dataA=0; dataB=0;
#2; dataB=1;
#2; dataB=1'bx;
#2 ;control=1; dataB=1;
#2; dataA=1;
#2;dataA=1'bx;
#2; dataA=0; dataB=0;
#2; dataA=1;
#2; dataA=1'bx;
#2; control=1'bx; dataA=0; dataB=0;
#2; dataA=1;dataB=1;
#2; $finish;
end
endmodule

primitive multiplexer(mux, control, dataA, dataB ) ;
output mux ;
input control, dataA, dataB ;
table
// control dataA dataB mux
0 1 0 : 1 ;
0 1 1 : 1 ;
0 1 x : 1 ;
0 0 0 : 0 ;
0 0 1 : 0 ;
0 0 x : 0 ;
1 0 1 : 1 ;
1 1 1 : 1 ;
1 x 1 : 1 ;
1 0 0 : 0 ;
1 1 0 : 0 ;
1 x 0 : 0 ;

```

```

x 0 0 : 0 ;
x 1 1 : 1 ;
endtable
endprimitive

module dut #(parameter p1 = 1) (output mux, input control,
dataA, dataB);
multiplexer m1(mux, control, dataA, dataB);
endmodule

```

The contents of the .vp file that are generated using the –protect128 compile option are as follows:

```

always
begin:counter
`protected128
PWXH[Q[X&;D#.->0!SIF<HI"D7X)2F-MZCTCK.R+U8;SAE3M.+ ,;N'/
3.B=6%$_5PHYD]E1G#<O,VW A_>!1S/0%XYM98MW0'OA]?PNK:[T)*_]
IRSN+R.EE#]-I JJRPA_#KZ+7$\TIAY83B8L<U0!U.GK[V?\V,
=>JF:GK6"C8=\M5MB'!2+WY/7S5_&RONPGO!LK8#25
(CO>3N7N.YG%==FF'),"J90A8OS5$E2+ &4@T2Q!U?DOS;2(O3G6G3T>
`endprotected128

```

### **-putprotect128 <Dir-name>**

By default, the encrypted .vp file is saved in the same directory as the source files. You can change this location by using the –putprotect128 compile option.

For example, the following command saves the foo.vp encrypted file in the ./out directory:

```
% vcs -putprotect128 ./out -protect128 foo.vp
```

VCS creates a protected file in the specified directory. The './out/foo.vp' protected file is created.

---

## Automatic Protection Options

-autoprotect128

For Verilog and VHDL files, VCS encrypts the module port list (or UDP terminal list) along with the body of the module (or UDP).

-auto2protect128

For Verilog and VHDL files, VCS encrypts only the body of the module or UDP. It does not encrypt port lists or UDP terminal lists. This option produces a syntactically correct Verilog module or UDP header statement.

-auto3protect128

This option is similar to the `-auto2protect128` option except that VCS does not encrypt parameters preceding the ports declaration in a Verilog module.

+autoprotect [*file\_suffix*]

Creates a protected source file; all modules are encrypted.

+auto2protect [*file\_suffix*]

Creates a protected source file that does not encrypt the port connection list in the module header; all modules are encrypted.

+auto3protect [*file\_suffix*]

Creates a protected source file that does not encrypt the port connection list in the module header or any parameter declarations that precede the first port declaration; all modules are encrypted.

+deleteprotected

Allows overwriting of existing files when doing source protection.

+pli\_unprotected

Enables PLI and UCLI access to the modules in the protected source file being created (PLI and UCLI access is normally disabled for protected modules).

+protect [*file\_suffix*]

Creates a protected source file by only encrypting `protect/`endprotect regions.

+object\_protect <sourcefile>

Debugs the partially encrypted source code.

```
vcs +protect +object_protect <sourcefile.v>
```

+putprotect+target\_dir

Specifies the target directory for protected files.

+sdfprotect [*file\_suffix*]

Creates a protected SDF file.

-Xmangle=*number*

Produces a mangled version of input, changing variable names to words from list. Useful to get an entire Verilog design into a single file. Output is saved in the tokens.v file. You can substitute -Xman for -Xmangle.

The argument *number* can be 1, 4, 12, or 28:

-Xman=1

Randomly changes names and identifiers, and removes comments, to provide more secure code.

`-Xman=4`

Preserves variable names, but removes comments.

`-Xman=12`

Does the same thing as `-Xman=4`, but also enters, in comments, the original source file name and the line number of each module header.

`-Xman=28`

Does the same thing as `-Xman=12`, but also writes at the bottom of the file comprehensive statistics about the contents of the original source file.

`-Xnomangle=.first|module_identifier,...`

Specifies module definitions whose module and port identifiers VCS does not change. You use this option with the `-Xman` option. The `.first` argument specifies the module by location (first in file) rather than by identifier. You can substitute `-Xnoman` for `-Xnomangle`.

---

## Using Automatic Protection Options

Note:

The **-auto3protect128** option takes precedence over **-auto2protect128** and **-autoprotect128** options, **-auto2protect128** takes precedence over **-autoprotect128**, and **-autoprotect128** takes precedence over **-protect128**.

### **-autoprotect128**

For Verilog and VHDL files, VCS encrypts the module port list (or UDP terminal list) along with the body of the module (or UDP).

For example, the contents of the **.vp** file that are generated using the **-autoprotect128** option are as follows:

```
module counter
`protected128
P6O # ON'-, 5&.Y)AO )WH1MLZ6=M^=MG! HNZ [ ; ]%0^2CSHD; !"DA
Y_*<7CQP.GB
P>NV,,82,G9%HZBYEBWO@D^JP*HXZR8K\))1?'OI=-Q^T(@V7^I^@T&I1
[>.3GCO@ [ PWTN(F,CSX.ZH$37A3F/8IWXML[>/JJN8P\Q)Y=\FQ$J4M>
#. (31WZ'&(5&+%/L<RP0F+$)E-U7!KA1Y!&5;S3>ID8RC) :@*V>X
YZ1NC:S"/F] !NX0NKG"K8X5&4D_#)%PV(Y%PFO?4*96PED9&SI:PGMM
(J?GOD$%XF8CV:?#A_[^<QX3-;IC1)I3\^-8C%GIDPRR$%26.$L 'OZ5B4
6-_C10X,WOMU'Y'IM'CZ*/CW=XYBE\L\,4.U =N<HY*O2I@
`endprotected128
endmodule

primitive multiplexer
`protected128
P"-9R8;C8?O\K)>&) $0%*8Q2_OQP5 (+NY%R&X+=G;QX@:=#$<CRS0A\&]
/IO&6+SFP+OK+-UK) $^ B*1NCC./ESFVG!_H2CYI3"+'T'*^-&*/#
P%<U:&I@]S=Y#2""]) I&P).;YML_ #- [&7>#5[9K@>9+L( Y8H$G\?TJ
&35W=*#-NKBM9] !HZ& (=B:;$_]FUPE@T8Q+:7*(Z+14ES2-^ZRJ(WX
#NV!6;%>UM>VL0H(T0\+TRKYZG5)G'AK1)*F'$P=9LR \&;G#.
6D":CF71O@V/:&/;O3T491+,=A5((6LN"\U*J!,7>RQX2A1*DP,2J
```

```

PK_..KR$/((1C"+/^0"MHNPQ.,;D] [?NRD_X6W._XTPGP6-,0"<47*7>
$KYQS,-S<P84)%2K^O:/,>&+0#4\]CJ)TA45&7H1$V@PJ$Q<=\PI9\5\
-3PSENY+K,)C-V.0E
`endprotected128
endprimitive

```

In this example all the module port lists and UDP terminal list along with UDP definition are encrypted.

### **-auto2protect128**

For Verilog and VHDL files, VCS encrypts only the body of the module or UDP. It does not encrypt port lists or UDP terminal lists. This option produces a syntactically correct Verilog module or UDP header statement.

The contents of the .vp file that are generated using the -auto2protect128 compile option are as follows:

```

module counter( inp, outp);
input [7:0] inp;
output [7:0] outp;
`protected128
P-62]23&H.F//I;K-%+[=
WD$[*GB:L2U<9W,03Y<B_1=DRWLJV;'OM'P];^[B5ZI
P"")X^0Y.WRTK61I+D1_3=7\0D1C!%3+"(NR'K3$HKQA[FL@^).B.(P/
'"-X;,XUJP"# 4)M:.<4R2VAUA0TZM'61%_!;=,UY3/,P(=A$RA_/$
(EBK*>X>P8K,@'*LIS0,PVGH"H+7,C4;@,.?X *HQLHHM3::F_E!
((8>BYMKVT8HA-4*N6EJU1OIU0T3]6@9!PAS43Y)QD_DI(J15%0KCEX[/+_
Y'7UC6<%D@;.0?I/-W$P[HNCB6A+X\9P</C6-$[.
`endprotected128
endmodule

primitive multiplexer(mux, control, dataA, dataB ) ;
output mux ;
input control, dataA, dataB ;
`protected128
P1T4VA5J%C(4VK!^U;R^"ND56SO3AG+*12MZ&7#<;&/_;Q1D4V >" .4-

```

```

4Q#" (@T;P-P<'^#*2WG'8T/SNA(/:2Z*HK"$@L^D&AP@E;, P$O:9#PG3]
1X >DV?TZK/, S*:PMC+T1#65A@RYU+*=XFFMS^+C(8H9XL-Z-<J"E6%V>
2N,%%:*U I>HQ2*F Z%D/QYPG32(5;P;D>X" ^^008)]%)O&3/7/P)O"??
[B@\",E<Y,N'"(&R 05300;7X%3TV]QJP[H47--_DZ .]FAAJ^! V":T=
E0#PYJL\)\Y.:OGH%VW]D=R-.K_11S)4I-CU-P=&+
`endprotected128
endprimitive
endmodule

```

In this example it encrypts only the body of the module or UDP and does not encrypt port lists or UDP terminal lists.

### **-auto3protect128**

This option is similar to the **-auto2protect128** option except that VCS does not encrypt parameters preceding the ports declaration in a Verilog module.

The contents of the **.vp** file that are generated using the **-auto3protect128** compile option are as follows:

```

module dut (mux,control, dataA, dataB);
parameter p1 = 1;
output mux;
input control;
input data;
input dataB;
`protected128
PR@#>:B8A;TV_. 4184;Y,%!E@E-P8,WL)%D+%2C@JY0L3)_%J"P;8S*
ESYV_;38PAAX3?7V=/PD$@4E*9DK2U^R_0>@2JUT:=#?D:0EX'+GLZ?8
S';N=FS!"S?D[I;E7
`endprotected128
endmodule

```

In this example it encrypts only the body of the module or UDP and does not encrypt port lists or UDP terminal lists.

```
module top;
```

```
parameter p1 = 3;
```

### +protect option

1. Enclose the Verilog and VHDL code that you want to encrypt between the '`protect`' and the '`endprotect`' compiler directives.
2. Compile the files with the `+protect` option

For example:

```
% vcs +protect foo.v
```

When you compile the design with the `+protect` option, VCS creates a new file with the `.vp` extension for each Verilog file specified at the command line. For example, VCS creates `foo.vp` when you execute the command listed above.

In the `.vp` file, VCS replaces the '`protect`' and '`endprotect`' compiler directives with the '`protected`' and '`endprotected`' compiler directives, and encrypts the code in between these directives.

The contents of the `.vp` file that are generated using the `+protect` compile option are as follows:

```
always
begin:counter
reg [7:0] int;
count = 0;
int = inp;
`protected
z370P(PNd1ZOKL9PH7?6a=LC8JB\Lf9dBes3T<#ZE58?b# [=[_&)>
_3eL6_1aY7+c,@0BZF#U;</EHfdM&I1fI-@]#?U;Gef\PX2fJ?1.HQ
:M.X_>3CYc9_QUZ2R97VA^8IT3V/,Kf<9N^-MHS (=bBbN&BDPh\?$
`endprotected
```

### **+putprotect+<Dir-name>**

By default, the encrypted .vp file is saved in the same directory as the source files. You can change this location by using the +putprotect option either at vlogan stage for three-step flow or at VCS stage for two-step flow.

For example, the following vlogan or vcs command saves the testfile.vp encrypted file in the ./out directory:

```
% vlogan -sverilog +autoprotect +putprotect+$cwd/out  
testfile.v
```

or

```
% vcs -sverilog +autoprotect +putprotect+$cwd/out testfile.v
```

VCS creates a protected file under the specified directory. The protected file is './out/testfile.vp'.

This option is not supported with -protect128, -aotuprotect128, -auto2protect128, and -auto3protect128 options.

### **+autoprotect[file\_suffix]**

For Verilog and VHDL files, VCS encrypts the module port list (or UDP terminal list) along with the body of the module (or UDP).

For example, the contents of the .vp file that are generated using the +autoprotect compile option are as follows:

```
module counter  
`protected  
.423IPYX4.Z-JJ#MF2_EDM(7RN]634+76?=U?f-  
ZVLX(1?N<2UTZ())T4I2K) fXX  
-@+EK?e=^Z\DLXU5XH0VQ19,>-9]16+gDJ7RF431EgL=7#>Y1V,9+3-
```

```

8&G>F[Q0C
4#B[FgQ#DUU<>_UR^I#D:eS(2+015=^HMTY]f<XXU6=;4RP]f>?X,5d4B&
X1T&UC
MAZQ[N=K6(>R>b2g/,HGEHMD/
+W:38b[(6Lf4f@g)_Me#b\34E7ECQMDcHJKaY?\_
cK:ZA]TbbMa]bAFX>fR&YC-MH[79#=CUUFG>:0RcKOU\bI-
&2I^_[K=LbUL9,GRF
U9)68:,CZ@Df[@(:PdEP2F)cWU7\K<[c)A?K,-9:C@c\f$`endprotected
endmodule

primitive multiplexer
`protected
T:=e^@R59Xg#P];gMBf9#>(d[ZD7J.Pa/8PSPY)=G1BaGT,//+
+QM5)T.a[/+e,D+
>g--ENRe-4GV(@7#UN0f_e/.dY.1Xbg-?9NZ0CTP-
U^D@?Ja^8AF@&R=0CHd/VKV
--NN]RIS;Q2.A2RBE5_A,PJF@7</
F6fH]AgM8N57RJ,.C>3KEWD4dN+V4B2a@<V:
5\QQJJ2O#/=f/YbF-\)/
ERc_gM(Y,_.3+?&?IGU_87ZLeYc;(SfcTePTRB]2LUR
4/,aMg?WIPS[A]+OUG]7,<]L4FP-8=_JPE)7O]&UbSdI+-_
F_+5gK[27NgXW4<0SD
Q2D>.d_;L89<Y[LFD0OME?fMA7b.5aa+^N/
F#3[<[\N_<d5+>QKYQ>+KZ,0/fbB
@ZTB7#P2RL=3Ud>e1CMa2<<7<\PIYR(S;$`endprotected
endprimitive

```

In this example all the module port lists, UDP terminal list along with UDP definition are encrypted.

### **+auto2protect[file\_suffix]**

For Verilog and VHDL files, VCS encrypts only the body of the module or UDP. It does not encrypt port lists or UDP terminal lists. This option produces a syntactically correct Verilog module or UDP header statement.

For example, the contents of the .vp file that are generated using the +auto2protect compile option are as follows:

```

module counter( inp, outp);
input [7:0] inp;
output [7:0] outp;
`protected
Y5S,#M;BJ&FL9:,U#/R;T;+G:#XZD#NUZ58-U0RB;V?9JM?FcOI/) FE
:XM0I&#3LM.D]L2X0<:,89-DQ07GWM[Gc9LRc#7#IN:#1H@+CRBU-Z?G
O/c[9B;.Q9e@30IZM]7XR0LRXF;FT4<&&M#+E6Z-].B(),ceZBDO4<fo
[Nd,O@#>a3\~DF4EL[^SgXX^:#R+0-d3MK^WF(QY\WfLK?4IVdPXJFdHg
Ld(#./_NIYKaSOOMURg@00(C[1A\eo(<9WIT1,+Q8^e>fATb6\Y2K7@9f)
SK=>\H20&86;Y6;6KD$
`endprotected
endmodule

primitive multiplexer(mux, control, dataA, dataB) ;
output mux ;
input control, dataA, dataB ;
`protected
X-NKV3Ld>NGW@?2WZKeWBaZJ:[IUV+=H[?BKE&:#@+B;fSa^YL<,)dJ-
2HF.#A89K?K4+WT11Id9R<CJ^@=Q5KF(Y^S#\L5#bEdP:ag49F;=b15
CHfW</f?Sa.9=+^Id^0WN^-IAX^.AU<^_81T2AB=4+Ce)]KFAYBRD>DT>L
#Z/7;;YC>KRBJ3GHLKT;_<V&6V?(WJa#\W//.QRcW&OCG^R#A.+0HH(>=
(((SegQ]YC0,G2^4.03B9@UF/@a_OG=-++Rc?A2/J^_;MdG-C>S_NaH\WM
f#BY6;VR_2451C2<7A].Nb^\"3BP$
`endprotected
endprimitive

module dut #(parameter p1 = 1) (output mux, input control,
dataA, dataB);
`protected
.5(K9-=#4,NGO2NY+&g&+^bN,SN4f))0]J2PC)&(?9+WKdaGS\C+))
;KP>;I_@1B3?SFLc5U&B;,?,S==2_;4K-PD,-I=1E\`a8^YC=-/)I9f--
GE24UBaD9CFGd;BHJKU$
`endprotected
endmodule

```

In this example it encrypts only the body of the module or UDP and does not encrypt port lists or UDP terminal lists.

### **+auto3protect[file\_suffix]**

This option is similar to the `+auto2protect` option except that VCS does not encrypt parameters preceding the ports declaration in a Verilog module.

The contents of the `.vp` file that are generated using the `+auto3protect` compile option are as follows:

```
module top;
parameter p1 = 3;
```

In this example it encrypts only the body of the module or UDP and does not encrypt port lists or UDP terminal lists.

### **+deleteprotected**

Allows overwriting of existing files when doing source protection using the `+protect` option.

This option is not supported with `-protect128`,  
`-autoprotect128`, `-auto2protect128`, and  
`-auto3protect128` options.

### **+pli\_unprotected**

Enables PLI and UCLI access to the modules in the protected source file being created (PLI and UCLI access is normally disabled for protected modules).

This works with both `+protect` (all variants) and `-protect128` (all variants). To enable PLI capabilities, use the `+pli_unprotected` option as follows:

```
% vcs +protect +pli_unprotected <sourcefile.v>
or
```

```
% vcs -protect128 +pli_unprotected <sourcefile.v>
```

---

## Protecting ‘include File Directive

You can use VCS to automatically protect ‘include file directive while protecting the module.

### +autoincludeprotect

For Verilog files, VCS encrypts the ‘include files using the switch +autoincludeprotect.

```
vlogan +autoincludeprotect test.v <auto-protect switches>
```

Or,

```
%vcs +autoincludeprotect test.v <auto-protect switches>
```

Consider that the source file a.v include b.v as shown below:

```
`include "b.v"  
module a();  
endmodule
```

After encryption, b.v is encrypted and is renamed to b.vp. The encrypted b.vp file along with the source file a.vp is saved in the same directory specified by –putprotect128 compile option. The directive changes to ‘include b.vp as shown below:

```
`include "b.vp"  
module a();  
endmodule
```

Note:

This option is not supported for VHDL.

---

## Enabling Debug Access to Ports and Instance Hierarchy

You can use VCS to enable debug access to port and instance hierarchy.

### **+autobodyprotect**

For Verilog files, VCS enables debug access to port and instance hierarchy using the switch **+autobodyprotect**.

```
%vcs +autobodyprotect test.v
```

Hence, port list containing parameter and instance hierarchy of each module are accessible only to FSDB for Verdi users.

Note:

This option is not supported with **+autoprotect** and **-autoprotect128** options. This option is not supported for VHDL.

---

## Debugging Partially Encrypted Source Code

The partial encrypted code is a code that has some of its part enclosed with the **'protect** and **'endprotect** macros. VCS allows you to debug the objects that are not enclosed within **'protect** and **'endprotect** while restricting access to the variables that are within **'protected** and **'endprotected** macros.

**Note:**

When you enclose a part of code using 'protect and 'endprotect, VCS converts it into 'protected and 'endprotected when you pass +protect.

To debug the partially encrypted source code, use the +object\_protect option as follows:

```
vcs +protect +object_protect <sourcefile.v>
```

You can enable partial debug capability by adding the +object\_protect option in the vcs encryption command line, so that partial encryption is applied and the encrypted file is also enabled with debug capability (-debug\_access+f+drivers -debug\_region+encrypt) for the unencrypted objects.

---

## **Skipping Encrypted Source Code**

VCS allows you to skip some portion of the code unencrypted when the complete module is encrypted with autoprotect options. You can use 'unprotect and 'endunprotect pragmas to mark a block of source code to be excluded from encryption.

All autoprotect options ignore protection when you use 'unprotect and 'endunprotect pragmas.

Enclose the Verilog code that you want to decrypt between the 'unprotect and the 'endunprotect compiler directives.  
Enclose the VHDL code that you want decrypt between the -unprotect and the -endunprotect compiler directives.

**Note:**

unprotect and `endunprotect pragmas do not work with -Xman=4 (for tokens.v file) and -Xrad=0x2 (for rad.v file) options.

---

## gen\_vcs\_ip

VCS allows you to protect a VHDL or a Verilog source file using the gen\_vcs\_ip utility as shown below:

```
% gen_vcs_ip -ipdir my_dir -e "vhdlan file1.vhd"  
% gen_vcs_ip -ipdir my_dir -e "vlogan file1.v"
```

The protected IPs are platform and release independent. You share these protected IPs with your vendors.

The protected IP files are saved under the directory specified with the option -ipdir dir\_path, and are named as file1.vhd.e, file1.v.e and so on. The gen\_vcs\_ip utility also writes the analyze.genip script, which can be later used to analyze all the protected files.

IPs protected using gen\_vcs\_ip are black box, and, therefore, are not in user readable format. Except for the ports of the protected design unit, none of the internal signals or variables can be accessed by any UI, GUI or PLIs. These black box IPs do not allow the following:

- Access by XMR paths to any object within or through the generated IP.
- PLI access (acc, tf, vpi, vhpi) to objects that reside in generated IP.

- Dumping (vcd or vpd files) any objects (signals or variables) that reside in generated IP.

You can use the `-debug_access+f+drivers - debug_region+encrypt` option to create the protected modules, whose ports are visible, and the internal signals and variables can be accessed using Synopsys UI, GUI or PLIs.

For example:

```
% gen_vcs_ip -ipdir my_dir -debug_access+f+drivers -  
debug_region+encrypt "vhdlan file1.vhd"  
% gen_vcs_ip -ipdir my_dir -debug_access+f+drivers -  
debug_region+encrypt "vlogan file1.v"
```

The IP protected using the `-debug_access+f+drivers - debug_region+encrypt` option is a grey box and using VCS, UCLI, GUI, VHPI, VPI or MHPI, IP consumer can:

- View the ports at the boundary of the IP
- View the complete design hierarchy
- View all the internal signals or variables
- Query the value of signals or variables
- Set callbacks on value changes of the signal
- Use the `force` command to change the value of the signal
- Monitor the loads and drivers of the signal

Along with the specified design files, the `gen_vcs_ip` utility also protects the Verilog library files specified using `'include`, `-v` and `-y` options.

For example:

```
% gen_vcs_ip -ipdir VCSIP_DIR -e "vlogan top.v -v lib1/sub.v"
```

In the above example, the `gen_vcs_ip` utility protects both `top.v` and `sub.v`, and the protected files are saved under the `VCSIP_DIR` directory.

---

## Syntax

```
% gen_vcs_ip -ipdir [ipdir_name] -debug_access+f+drivers -  
debug_region+encrypt \  
-e "[analysis_command/script]" -parse -noencrypt  
-top_name <IP Module/Entity Prefix>
```

## Analysis Options

`-ipdir [ipdir_name]`

Physical directory where IP files are generated.

`-debug_access+f+drivers -debug_region+encrypt`

Generates binary IP files, whose ports are visible, and whose internal signals and the variables can be accessed using Synopsys UI, GUI or PLIs.

`-e`

Specify `vhdlan/vlogan` command line. You can also specify a make command or a run script.

`-parse`

Parses and does semantic checking of the design files.

`-noencrypt`

Used to package design files without encryption.

```
-top_name <IP Module/Entity Prefix>
```

It is possible that different IPs can use the same module name and can create module name conflicts. To resolve these conflicts, you need to make sure that all module/entity names inside an IP are unique. The `-top_name<IP Module/Entity Prefix>` option can be used to provide a prefix for all the IP modules being encrypted.

For example,

```
% gen_vcs_ip -top_name IP1 .....
```

All the modules in this IP will be prefixed with “IP1”.

```
% gen_vcs_ip -top_name IP2 .....
```

All the modules in this IP will be prefixed with “IP2.”

Thus, there is a lesser possibility that the module names from different IPs conflict.

Note that this switch is not intended to specify the top level module.

Note:

- VCS protects the library files specified with the `-y` and `-v` options and places in the directory where the IP model is generated.
- If you specify multiple `-y [lib_dir]` options, and if multiple files with the same file name exist in different library directories, the file that exists in the last directory you specify overwrites the others. In this case, VCS issues a warning message indicating from which library the module is picked up.

---

## Exporting The IP

After protecting the IP, you can tar the generated IP directory and ship it to the IP consumer. To use the IP, the IP consumer should extract the IP directory and execute the `analyze.genip` script to analyze the protected files.

---

## Use Model

### IP Vendor

Synopsys recommends you analyze, elaborate and simulate the design before you protect them. This ensures that you are protecting the right set of source files.

### Analysis

Always analyze Verilog before VHDL.

```
% vlogan [vlogan_options] file1.v file2.v  
% vhdlan [vhdlan_options] file3.vhd file4.vhd
```

(The VHDL bottom-most entity first, then move up in order)

### Elaboration

```
% vcs [elab_options] top_module/entity/config
```

### Simulation

```
% simv [run_options]
```

## IP Generation

```
% gen_vcs_ip -ipdir ip_dir -e "analyze.csh"
```

**Note:**

analyze.csh contains vlogan, and vhdlan command lines to analyze the Verilog and VHDL design files.

## IP User

The usage model to use the protected IP is shown below:

### Analysis

```
% ip_dir/analyze.genip
```

### Elaboration

```
% vcs [elab_options] top_module/entity/config
```

### Simulation

```
% simv [run_options]
```

---

## Licensing

You require a “DW-Developer” license to protect an IP. However, a license is not required to use the protected IPs.

# 26

## VCS Fine-Grained Parallelism Technology

---

This chapter explains how to use the new performance features in this release, in the following sections:

- [Introduction](#)
- [Use Model](#)
- [Profiling to Detect Design Suitability for Parallelism](#)
- [Limitation](#)

---

## Introduction

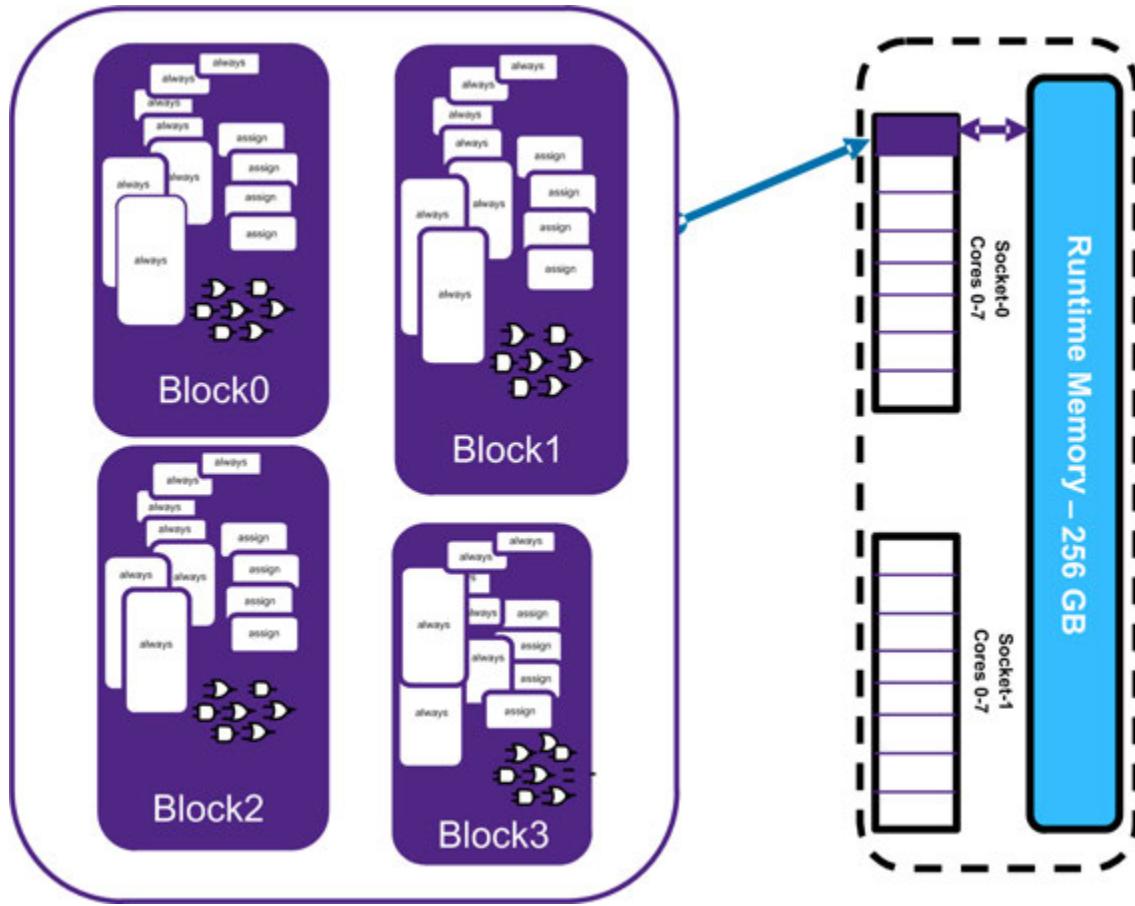
Traditionally, simulation performance has been a function of CPU performance, memory efficiency, and serial algorithm. However, advancements in processor hardware and instruction set architectures are providing the opportunity for order-of-magnitude simulation performance gains.

The rapid growth in the number of available cores per processor, in addition to the fast evolving vectorization support, enables next-generation simulation technologies, such as VCS' new Fine-Grained Parallelism (FGP). With this technology, you can gain the complete advantage of the newly available compute power to reduce the total turnaround time for simulation.

VCS FGP technology is designed to take advantage of the heterogeneous environments that consists of any combination of processors and vector cores. It improves the performance by adapting its algorithms dynamically to completely utilize all the available cores. This optimizes the simulation for latency sensitive tests.

Consider the following scenario, shown in [Figure 26-1](#).

Figure 26-1 Serial Simulation

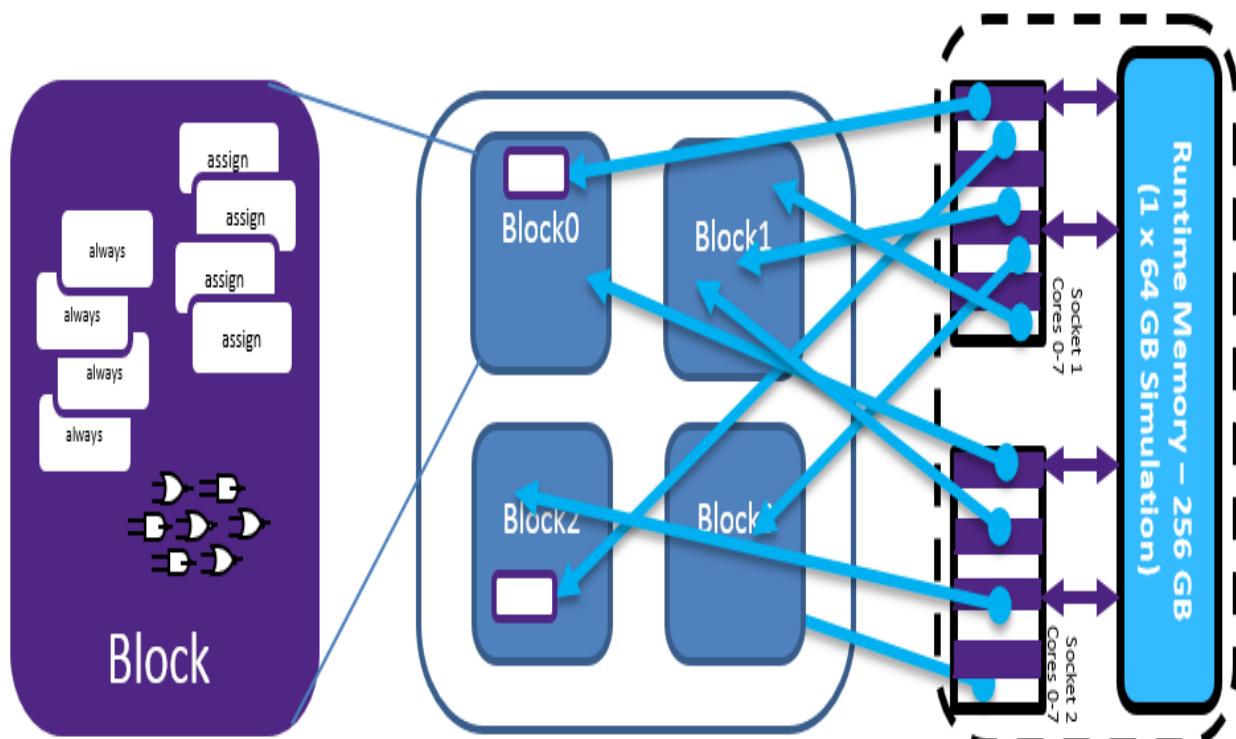


In Figure 26-1, entire block is executed on a single core in a single thread. Depending on the available memory, this might lead to available cores remaining unused, which might result in decreased utilization and efficiency.

Now, a HDL block is comprised of many events, such as `always` blocks, gates and assignments (shown in Figure 26-2). This enables you to parallelize events within a block on many CPU cores. While this can achieve significant performance gains, sometimes the gains are limited, if the processes are too large. To break these limitations, large processes are broken down further into parallel micro tasks to ensure well-balanced activity on each CPU core.

This method of partitioning a design into parallel micro tasks and events and scheduling them on multiple threads is the underlying principle of VCS FGP. In addition to this, VCS FGP further optimizes the simulation performance by adapting its algorithms for task scheduling, load balancing, and cache efficiency for target processor architectures. This enables you to take full advantage of the provisioned compute power.

Figure 26-2 Block Composition and Fine-Grained Parallelism



VCS FGP technology is inherently part of VCS M-2017.03 simulation engines. Therefore, it natively and transparently supports the crucial simulation features that you can rely on, such as Native Low Power (NLP), X-Propagation simulation, SDF, coverage (code coverage and functional coverage), and fully parallelized FSDB dumping for Verdi

Debug (fully parallelized FSDB dumping is essential for parallel simulation technology to enable full performance speedup while maintaining the debug accuracy).

These simulation features continue to work as before with no changes needed to the design, or testbench and no disruption to the existing VCS simulation environment. Furthermore, VCS FGP delivers these performance gains with minimal impact to the memory usage and compile time.

In summary, VCS FGP technology provides breakthrough simulation performance improvements using the existing x86 hardware.

---

## Use Model

VCS FGP use model is simple and is easy to deploy with no changes needed to the design and minimal changes required to the VCS verification environment.

Designs that get the most benefit from using VCS FGP are those that have a high-level of design activity. VCS provides a full range of profiling, analysis, and reporting tools to help determine the design suitability for FGP.

FGP is supported in both VCS two-step and three-step flows.

For two-step flow and three-step flow, add the option `-fgp` in VCS command line. At runtime, you need to specify the number of cores that you want to utilize for the parallel engines. So, if you specify 12 threads for the parallel engines and run the top design, you can see 13 cores being utilized for your simulation as one extra core is used by the master for running serial and parallel simulation.

You do not need to recompile to change the number of cores used at runtime.

### **Two-step flow:**

```
% vcs -fgp -full64 <otherOptions>  
% simv -fgp=[FGP-OPTIONS]
```

### **Three-step flow:**

```
% vlogan -full64 <otherOptions>  
% vhdlan -full64 <otherOptions> (for Mixed-HDL design only)  
% vcs -fgp -full64 <otherOptions>  
% simv -fgp=num_threads:<value>
```

Where, the `-fgp` option is used to enable FGP at compile-time or at runtime.

If you specify the `-fgp` option at compile time, and do not specify the option at runtime, `simv` runs on a single core.

#### **Note:**

The `-fgp` option works only in 64-bit mode.

You can use the `num_threads:<value>` runtime option to specify the number of cores. `<value>+1` cores are picked that includes one master core and `<value>` child cores.

`simv` collects a list of free cores. A core is determined to be loaded or free based on the load threshold. If the number of free cores is less than the requested number of threads ( $N+1$ ), then VCS generates an error. Otherwise, VCS uses  $N+1$  cores out of the list

(first use the cores on one socket) and bind threads to them. If enough free cores are not available, VCS generates an error message.

For example, for `auto_affinity` and `num_threads=15`, VCS generates the following error message, where the assumption is that 15 cores are available in the system whereas only 14 cores are available:

```
Error-[FGP_AFFINITY_FAILED] cpu_affinity/auto_affinity
failed
Failed to allocate CPU cores: Simulation threads requested
15 cores but only 14 core(s) are available in system.
For more details on status of CPU cores, please use runtime
option -fgp=diag:ruse.
```

Several sub-options are available under the `-fgp` option that allows you to control various functionality. The following are the sub-options supported under the `-fgp` option during compilation:

`multisocket`

When you use this option, the memory binding for the thread takes into consideration the socket that is running the thread. This helps in further optimizing the performance.

The following are the additional runtime arguments that are optional:

`sync:<scheme_value>`

Use this option to specify thread synchronization scheme. The default synchronization scheme used for optimal performance is `-fgp=sync:busywait`, which is best for optimizing simulation performance. You can also select any of the following synchronization schemes to override the default scheme:

- **mutex** - Specify this scheme to use `pthread_mutex` based synchronization.
- **serial** - Specify this scheme to run the threads serially. This is used to debug any potential thread race issue.

#### `cpu_affinity`

Allows you to specify fixed list of cores that can be used. For example, consider the following configuration:

```
...
node0 CPU(s): 0-4,10-14
node1 CPU(s): 5-9,15-19
```

When you use the following runtime command:

```
% simv -fgp= num_threads:8,cpu_affinity:\(6-9,16-19\)
```

9 cores are selected from the 10 available node1 CPUs.

#### `single_socket_mode`

Use this option to use all the available cores on the given socket. If you enable dumping, some cores are shared with the FSDB dumping threads.

When simulation starts, VCS detects the number of cores that are available on the current socket. All cores are treated as available irrespective of the current CPU load. When the simulation threads are created, VCS distributes these threads to the available cores. If the simulation threads do not use all the assigned cores, the simulation continues, but VCS generates a warning message.

#### `num_cores:N`

Use this option to specify the number of cores.  $N$  cores are picked that includes one master core and  $N-1$  child cores.

**Note:**

This option is used for the `single_socket_mode` option.

`num_fsdb_threads:M`

Use this option to indicate VCS that dumping is involved. This option is mandatory when FSDB dumping is enabled. VCS allocates as many cores as specified by `M` for dumping.

When `M = 0`, FSDB dumping runs on the master core (core that runs all the non-parallel portion of the design) and VCS generates a warning message. The remaining cores of the socket are used for FGP.

When `M > 0`, additional cores are allocated for dumping.

When the `num_fsdb_threads:M` option is not specified, VCS generates the following error:

```
Error-[FGP_NO_FSDB_DUMPING_THREADS] No cores specified  
for FSDB dumping  
Exclusive cores for FSDB dumping must be specified in  
non-mutex synchronization mode.  
Specify exclusive cores for FSDB dumping with -  
fgp=num_fsdb_threads option.
```

`dynamictoggle`

When there are no sufficient events for parallelization, simulation may slow down when VCS FGP is turned on. With this option, VCS FGP is turned off when simulation hits low activity region. This allows you to optimize the FGP engine while handling low event simulation.

`diag:ruse`

Use this option to print the thread resource usage statistics at the end of the simulation. It generates the `fgp_diag_profile.txt` file with the details for each thread. You can find the data with thread divergence, imbalance, master time, and total time in this file.

## Profiling to Detect Design Suitability for Parallelism

Designs that get the most benefit from using VCS FGP are those that have a high-level of design activity.

Designs that do not get benefit include designs that spend a significant portion of the overall time in the procedural testbench code, or have a high PLI or DPI content. Other examples include design that have low activity. The following are the profiling tools to detect the design suitability:

- Simulation Profile: Tells details on the time spent in DUT. shown in [Figure 26-3](#).

*Figure 26-3 Simulation Profile*

| Time Summary View        |            |            |
|--------------------------|------------|------------|
| Component                | Time       | Percentage |
| VERILOG                  | 45199.29 s | 70.98 %    |
| Module                   | 44942.50 s | 70.57 %    |
| Interface                | 147.36 s   | 0.23 %     |
| Package                  | 109.33 s   | 0.17 %     |
| Function Coverage Kernel | 80.12 ms   | 0.00 %     |
| Functional Coverage      | 20.03 ms   | 0.00 %     |
| HSIM                     | 10036.67 s | 15.76 %    |
| KERNEL                   | 8289.98 s  | 13.02 %    |
| Garbage Collection       | 2.69 s     | 0.00 %     |
| PLI/DPI/DirectC          | 138.07 s   | 0.22 %     |
| CONSTRAINT               | 13.67 s    | 0.02 %     |
| ASSERTION_KERNEL         | 2.98 s     | 0.00 %     |
| PLI/DPI                  | 20.03 ms   | 0.00 %     |
| total                    | 63680.69 s | 100%       |

This profile report shows that this is a good design for parallelism because less time is spent on the testbench code and on the PLI/DPI content.

For more information on simulation profile, see “The Unified Simulation Profiler” section in *VCS User Guide*.

- Timeline Profile: Tells event activity of the simulation.

The `-Xdprof=timeline` runtime option in the `simv` command line generates the DProf (event-counter based profiler) report under the current simulation working directory, shown in [Figure 26-4](#).

*Figure 26-4 Timeline Profile*

|               | Events         | Events% | Time(s)  | Time% | SimCycles   | AveTimePerCycle | (micro-seconds) |
|---------------|----------------|---------|----------|-------|-------------|-----------------|-----------------|
| EPC>=100K     | 5,189,573,399  | 20.6    | 111.83   | 6.0   | 37,265      | 3000.83         |                 |
| EPC[10K,100K] | 17,981,521,541 | 71.3    | 1,428.68 | 76.7  | 464,480     | 3075.86         |                 |
| EPC[1K,10K]   | 451,249,881    | 1.8     | 50.26    | 2.7   | 207,012     | 242.80          |                 |
| EPC[100,1K]   | 232,112,673    | 0.9     | 44.42    | 2.4   | 636,677     | 69.76           |                 |
| EPC[10,100]   | 772,581,317    | 3.1     | 72.29    | 3.9   | 17,736,179  | 4.08            |                 |
| EPC<10        | 603,898,949    | 2.4     | 156.29   | 8.4   | 240,325,678 | 0.65            |                 |
| Time-0        | 107,977,656    | NA      | 434.41   | NA    | 1           | 434407374.94    |                 |

Note:

If the top three rows of EPC (events per cycle) contribute to 80% and above, then it is better for parallelism. If the EPC of the last two rows dominate over other rows, this implies that it is a very low activity design. Such designs can actually slowdown with FGP when run with higher number of threads. The log file name where the EPC is generated is `dprof.txt`. Using the `diag:ruse` option at runtime, the switching activity is dumped in the report file. Refer to the runtime option `dynamictoggle` to dynamically turn off the VCS FGP for low activity regions.

---

## Limitation

The feature has the following limitation:

- Analog (Spice and Verilog-AMS) is not supported.
- For multiple socket optimization, the `cpu_affinity` option is not supported. Only the `auto_affinity` option is supported.

# 27

## Integrating VC Formal With Coverage and Planner

---

This feature is Limited Customer Availability (LCA). Limited Customer Availability (LCA) features are features available with select functionality. These features will be ready for a general release, based on customer feedback and meeting the required feature completion criteria. LCA features does not need any additional license keys.

This chapter provides a brief description on the VC Formal tool and how it works with Coverage and Planner. This chapter consists of the following sections:

- “Introduction to VC Formal”
- “VC Formal Coverage With Verdi Coverage and Planner”

---

## Introduction to VC Formal

VC Formal is a comprehensive high performance and high capacity solution for functional verification. Formal mathematical techniques are used to prove properties or assertions to ensure correct operations of RTL design.

VC Formal can be used throughout the design and verification process, from specification validation, white box and black box property checking, to pre-silicon and post-silicon debugging. To use the tool for property checking, the assertion-based verification methodology is recommended.

VC Formal consists of the following major components:

- GUI activity viewer
- VC Formal schematic viewer
- Front-end parser to build a netlist model that is common to all VC Static platform applications

Additionally, there is a formal model builder, formal engines, engine orchestration, and a built-in netlist simulator used to set up the initial condition and trace replay. A result database is also an essential part of the components, where results are accessible either from the activity view or from the `vcf` Tcl command interface.

---

# VC Formal Coverage With Verdi Coverage and Planner

This section explains how VC Formal coverage is integrated with the Verdi coverage reporting flow. The two primary links between Verdi and VC Formal display VC Formal results in Verdi, and link VC Formal results into your verification plan using Verdi Planner.

---

## Use Model

The section describes how Verdi coverage can be used to display and link to VC Formal results. This section consists of the following subsections:

- “[Collecting VC Formal Results in the Coverage Database](#)”
- “[Measuring VC Formal Assert Status in HVP](#)”

## Collecting VC Formal Results in the Coverage Database

To display VC Formal results in the coverage report, you must first have set the `VC_STATIC_HOME` environment variable. For example:

```
setenv VC_STATIC_HOME /tools/synopsys/vcst
```

Here, `VC_STATIC_HOME` is an environment variable that must be set to point to the installation directory for VC Formal.

To start the VC Formal tool, use the following command:

```
vcf -f test.tcl -verdi
```

where, `vcf` is a command to start the VC Formal tool with an interactive shell. The `vcf` shell is a new shell that calls `vc_static_shell` internally. The `vcf` shell supports all the options that `vc_static_shell`

supports. The `vcf` shell automatically runs in the 64-bit mode, unless you explicitly specify the `-mode32` option. For details on `vc_static_shell`, see the [VC Formal Verification User Guide](#).

`-f` indicates your VC Formal execution script

You provide your own tcl script to run VC Formal. To enable collection and display of coverage data in Verdi, there are the following two commands that must be included in that script:

1. The command to run VC Formal must include the `-cov all` flag. If you wish assertions to be targeted (not just cover properties), you should also include `-cm assert` as a flag to VCS:

```
read_file -cov all -format verilog -sva -top $top -vcs "-cm assert -sverilog $testDir/test.v -sva"
```

2. You must include a command to save the results to the coverage database, for example, `my_covdb`:

```
save_covdb -name my_covdb -cov assert+cover
```

This section consists of the following two subsections:

- “[Verdi GUI for VC Formal](#)”
- “[VC Formal Coverage in Verdi](#)”

## Verdi GUI for VC Formal

To display the Verdi GUI for VC Formal (see [Figure 27-1](#)), use the `vcf` command with the `-verdi` option.

*Figure 27-1 Verdi GUI for VC Formal*

| Verification Targets: ALL |        |       |                      |         |         |        |        |
|---------------------------|--------|-------|----------------------|---------|---------|--------|--------|
|                           | status | depth | name                 | vacuity | witness | engine | type   |
| 1                         | ✗      | 65    | fsm.a2               | ●       |         | b1     | assert |
| 2                         | ✓      |       | fsm.a_complete_frame | ●       |         | t1     | assert |
| 3                         | ✗      | 81    | fsm.a_loop_break     | ●       |         | b1     | assert |
| 4                         | ✓      |       | fsm.a_onehot         | ●       |         | e2     | assert |
| 5                         | ✓      |       | fsm.c1               | ●       |         | e2     | cover  |
| 6                         | ✓      | 79    | fsm.c2               |         |         | b1     | cover  |
| 7                         | ✓      | 2     | fsm.c_blk_cnt        |         |         | s1     | cover  |
| 8                         | ✓      | 1     | fsm.c_onehot         |         |         | s1     | cover  |

The properties are categorized according to usage fields in VC Formal as:

- `assert`: The property specifies an assertion to be solved. Its values are proven, inconclusive, vacuous, and falsified.
- `assume`: The property specifies as a constraint.
- `cover`: The property specifies as a cover property. Its values are covered, inconclusive, and uncoverable.
- `unused`: The property is disabled.

## VC Formal Coverage in Verdi

To load the coverage database generated with VC Formal in Verdi, use the following command:

```
verdi -cov -covdir my_covdb.vdb
```

where,

-cov: Starts Verdi in the coverage mode.

-covdir: Opens the coverage database.

my\_covdb.vdb: The name of the coverage database that gets opened. This is specified in the `test.tcl` script with the `save_covdb` command. The command generates the following window:

Figure 27-2 Verdi Coverage Assert Pane

| Assert                  | Type         | Success/N | Attempt | Failure | Incomplete | Vacuous | Category | AllMatch | Severity | FirstMatch | FVtype | FVstatus     | FVdepth |
|-------------------------|--------------|-----------|---------|---------|------------|---------|----------|----------|----------|------------|--------|--------------|---------|
| [A]fsm.a1               | Assertion    | 1         | 0       | 0       | 0          | ...     | 0        | ...      | 0        | ...        | assert | proven       |         |
| [A]fsm.a2               | Assertion    | 0         | 0       | 0       | 0          | ...     | 0        | ...      | 0        | ...        | assert | falsified    | 65      |
| [A]fsm_a_complete_frame | Assertion    | 1         | 0       | 0       | 0          | ...     | 0        | ...      | 0        | ...        | assert | proven       |         |
| [A]fsm_a_loop_break     | Assertion    | 0         | 0       | 0       | 0          | ...     | 0        | ...      | 0        | ...        | assert | inconclusive | 70      |
| [A]fsm_a_onehot         | Assertion    | 1         | 0       | 0       | 0          | ...     | 0        | ...      | 0        | ...        | assert | proven       |         |
| [A]fsm_a_trans          | Assertion    | 0         | 0       | 0       | 0          | ...     | 0        | ...      | 0        | ...        | assume |              |         |
| [P]fsm.c1               | ... Property | 1         | 0       | ...     | 0          | 0       | 0        | ...      | 0        | ...        | assert | proven       |         |
| [P]fsm.c2               | ... Property | 0         | 0       | ...     | 0          | 0       | 0        | ...      | 0        | ...        | cover  | inconclusive | 70      |
| [P]fsm_c_blk_cnt        | ... Property | 1         | 0       | ...     | 0          | 0       | 0        | ...      | 0        | ...        | cover  | covered      | 1       |
| [P]fsm_c_onehot         | ... Property | 1         | 0       | ...     | 0          | 0       | 0        | ...      | 0        | ...        | cover  | covered      | 1       |
| [P]fsm_c_trans          | ... Property | 0         | 0       | ...     | 0          | 0       | 0        | ...      | 0        | ...        |        |              |         |

The VC Formal information is displayed in the Assert tab in the columns: FVtype, FVstatus, and FVdepth. This information should be same as VC Formal results. The green color represents Proven or Covered, the red color represents Falsified, Uncoverable, or Vacuous, and the yellow color represents Inconclusive.

FVtype indicates the usage field. The value in this column can be either Assert, Cover, or Assume.

FVstatus indicates the run status of VC Formal. Its possible values depend on FVtype:

- If `FVtype` is `Assert`, the value of `FVstatus` can be `Proven`, `Inconclusive`, `Falsified`, or `Vacuous`.
- If `FVtype` is `Cover`, the value of `FVstatus` can be `Covered`, `Inconclusive`, or `Uncoverable`.

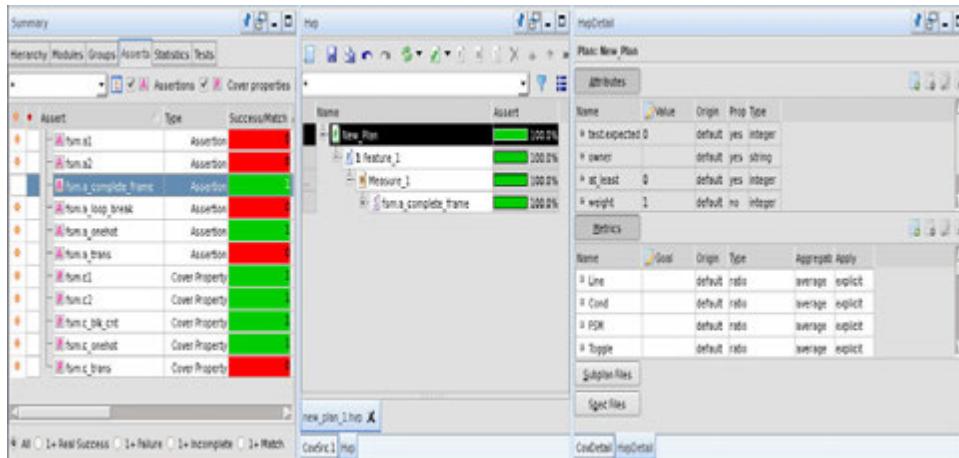
The `FVdepth` column is interpreted as follows:

- If `FVstatus` is `Proven`, `Vacuous`, or `Uncoverable` and its value is `-1`, it represents infinite depth.
- For other `FVstatus`,  $N \geq 0$  represents the depth of the trace.

## Measuring VC Formal Assert Status in HVP

VC Formal results can be annotated automatically onto features in your verification plan. Using these results, you can measure your expectations with `FVstatus`. [Figure 27-3](#) shows VC Formal results in a verification plan with attributes and metrics.

*Figure 27-3 VC Formal Results in HVP*



To measure the VC Formal Assert status in HVP, perform the following steps:

1. Set the expected FV Assert status with FV attribute values using `fvassert_expected_status` as shown here:

```

feature f_default;
    //These values are same as default value, no need
    //assignments again here.
    // fvassert_expected_status = proven;
    // fvassert_expected_mindepth = -1;
    // fvcover_expected_status = covered;
    // fvcover_expected_maxdepth = 0;
measure Assert, FVAssert m1;
    source = "property: fsm.*";
endmeasure
endfeature
feature f_expected_assigned;
    fvassert_expected_status = inconclusive;
    fvassert_expected_mindepth = 50;
    fvcover_expected_maxdepth = 50;
measure Assert, FVAssert m1;
    source = "property: fsm.a2", "property:
                fsm.a_loop_break", "property: fsm.c2";
endmeasure
endfeature

```

2. Add measures with property type sources and reference the FVAssert metric.

The new FV Assert built-in metric is added to keep the score of FV Assert status against users' expectation. If `FVstatus` is same as `fvassert_expected_status`, then the assertion/property is considered to be covered.

For example:

```

measure Assert, FVAssert m1;
    source = "property: top.a1*";
endmeasure

```

3. Calculate FVAssert metric scores.

After finding the matching region in the coverage database, the covered/coverable status is extracted from that database and added as a ratio for the FVAssert metric score. For the FVAssert metric, you can get `FVstatus` from FV annotations in the coverage database and compare `fvassert_expected_status`, which is set in `fvassert*`

attributes.

If FVstatus is same as fvassert\_expected\_status, then the assertion/property is considered to be covered.

For example, the matching property `top.a11: FVtype=assert` and `FVstatus=Proven`. If expectation is

`fvassert_expected_status = Proven`, the FVA assert metric score becomes 1/1. If expectation is

`fvassert_expected_status = inconclusive` and

`fvassert_expected_depth = 60`, the FVA assert metric score becomes 0/1.

Also, consider the following figure that shows six cases to measure the VC Formal Assert status in your verification plan:

| Assert                   | Type           | Success/N | FVtype | FVstatus     | FVdepth | A     |
|--------------------------|----------------|-----------|--------|--------------|---------|-------|
| [A] fsm.a1               | Assertion      | 1         | assert | proven       |         |       |
| [A] fsm.a2               | Assertion      | 0         | assert | falsified    | 65      | case1 |
| [A] fsm.a complete frame | Assertion      | 1         | assert | proven       |         |       |
| [A] fsm.a_loop_break     | Assertion      | 0         | assert | inconclusive | 70      | case2 |
| [A] fsm.a_onehot         | Assertion      | 1         | assert | proven       |         |       |
| [A] fsm.a_trans          | Assertion      | 0         | assume |              |         | case3 |
| [P] fsm.c1               | Cover Property | 1         | assert | proven       |         |       |
| [P] fsm.c2               | Cover Property | 0         | cover  | inconclusive | 70      | case4 |
| [P] fsm.c_blk_cnt        | Cover Property | 1         | cover  | covered      | 1       |       |
| [P] fsm.c_onehot         | Cover Property | 1         | cover  | covered      | 1       |       |
| [P] fsm.c_trans          | Cover Property | 0         |        |              |         | case5 |

**Case1:** `fsm.a2`: FVtype is "assert", FVstatus is "falsified", FVdepth is "6" and it is excluded. In the `top.f_default.m1` feature, the "property: fsm.\*" source matches the excluded property, therefore, no score is added.

**Case2:** `fsm.a_loop_break`: FVtype is "assert", FVstatus is "inconclusive", and FVdepth is "70". In the `top.f_expected_assigned` feature, `fvassert_expected_status` is "inconclusive" and `fvassert_expected_mindepth` is "50" that are set by users. It is met with users' expectations and the FVA assert metric score becomes 1/1.

**Case3:** `fsm.a_trans`: FVtype is "assume". In the `top.f_assume` feature, `fvassume_status` is "reviewed" that is set by users. The FVAssert metric score becomes 1/1.

**Case4:** `fsm.c_blk_cnt`: FVtype is "cover", FVstatus is "covered", and FVdepth is "1". In the `top.f_default` feature, with default attribute values, `fvcover_expected_status` is "covered" and `fvcover_expected_maxdepth` is "0". Compare FVstatus to expected attributes values and if the expectation does not meet, the FVAssert metric score becomes 0/1.

**Case5:** `fsm.c_trans`, no FV annotation. In the `top.f_default.m_no_fv_ann` measure, matching `fsm.c_trans` without FV annotation, the FVAssert metric becomes 0/1 because the FVAssert metric is referred explicated in measure.

**Case6:** dummy source. In the `top.f_default.m_dummy` measure, for all metrics for "no matching region", the score becomes all "0".

# 28

## Integrating VCS With Certitude

---

This chapter provides a brief description on the Certitude tool and how VCS works with Certitude.

---

### Introduction to Certitude

Certitude is a functional qualification tool. This tool enhances simulation-based functional verification by providing measures and feedback to assist in improving the quality of the verification environment (VE).

To detect errors, the functional verification environment must ensure that each error is activated, propagated, and then detected.

Mutation-based techniques used by Certitude helps improve the testbench by identifying the flaws in the testbench. Generating the coverage metrics using simulation and performing the testbench qualification are sequential activities. This necessitates the need to

perform redundant steps to setup the tool, one time for simulation and secondly for functional qualification of the testbench, which involves generation of database and fault-aware simulation executable.

---

## VCS and Certitude Integration

With VCS and Certitude integration, you can generate VCS and Certitude databases with the same compilation. By eliminating the dependency on different parsers, you can generate the fault-aware simulation executable without having to prepare the design for testbench qualification separately. This seamless integration:

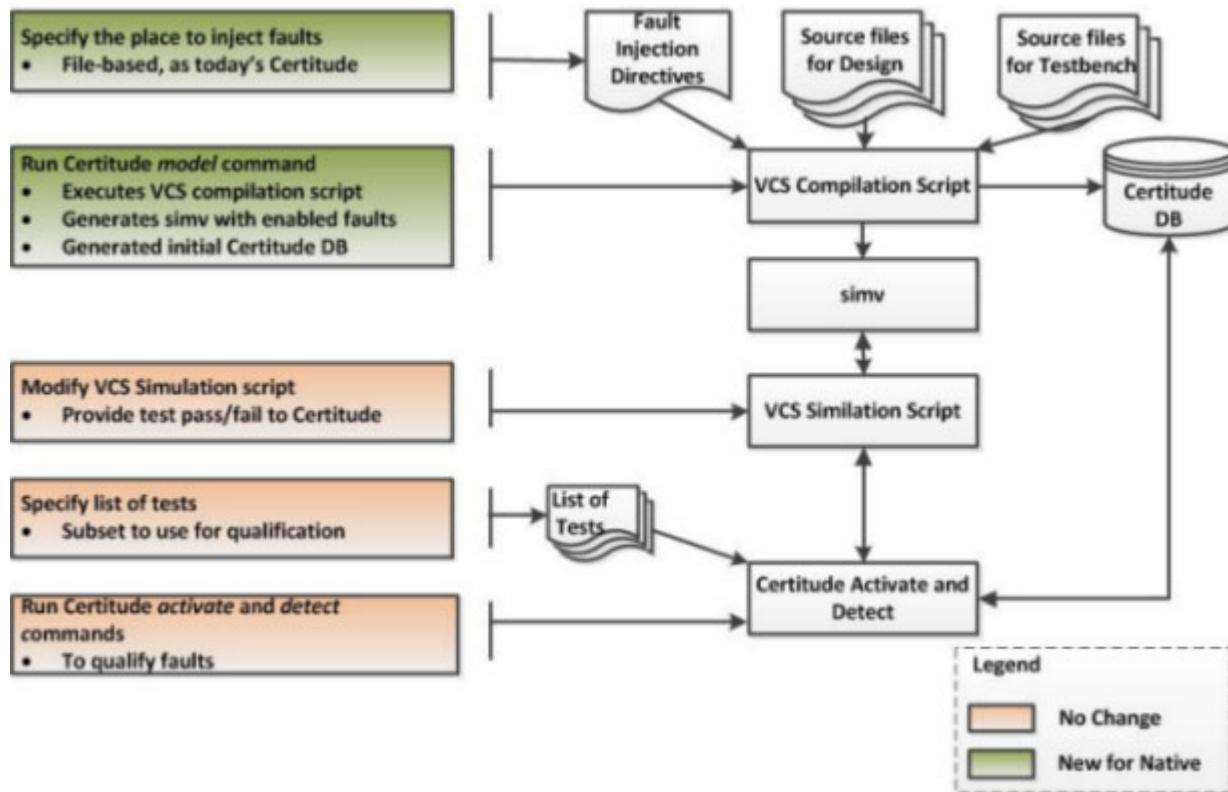
- Simplifies the setup requirements of Certitude environment with VCS
- Generates one or more simulation executables that contain all the instrumentation necessary to activate and detect faults.

With this integration, VCS generates the following database and executables:

- A Certitude database that corresponds to the Certitude model command.
- One or more simulation executables that contain all the instrumentation necessary to run activate, detect, regress and testscript commands.

[Figure 28-1](#) shows the Certitude integration with VCS.

*Figure 28-1 Certitude Integration with VCS*



The Certitude Functional Qualification System requires a specific set of configuration files for a qualification run. These configuration files describe the verification environment. You must create these files before running a qualification phase. For more information about configuration files and use model, see the *Certitude User Manual*.

Note:

Certitude communicates with VCS internally through a combination of environment variables and the Certitude database. This process is transparent.

---

# Loading Designs Automatically in Verdi with Native Certitude

With the integration of Certitude, VCS, and Verdi, you can load designs automatically in the Verdi system without setting the Certitude VerdiInitCommand configuration option.

This section consists of the following subsections:

- “[Use Model](#)”
- “[Points to Note](#)”

---

## Use Model

To use this feature, perform the following steps:

1. Specify the Native mode using the following setting in the certitude\_config.cer configuration file:  
setconfig -Simulator=native
2. Specify the –kdb option in the certitude\_compile configuration file.

In the two-step flow, specify the –kdb option in the command line as follows:

```
#!/bin/sh -e  
# VCS compile script  
% vcs -kdb -sverilog tb_top.sv dut_top.sv dut_bot.sv -  
debug_access+r
```

In the UUM flow, specify the –kdb option in all the vcs/vlogan/vhdlan command lines as follows:

```
#!/bin/sh -e
```

```
# VCS compile script  
% vlogan -kdb -sverilog tb_top.sv dut_top.sv dut_bot.sv  
% vcs -kdb -debug_access+r top
```

3. Leave the VerdiInitCommand configuration option as empty (default value).
- 

## Points to Note

The following points must be noted for using this feature:

- During the model phase, the certitude\_compile file is executed once and information of the KDB design is collected. The KDB design is then automatically loaded when the Verdi system is launched by Certitude. The KDB design cannot be loaded automatically if the model phase has not been executed. The feature is effective only after the model phase.
- If the design is not loaded automatically in the Verdi system, it may be due to one of the following reasons:
  - The -kdb option is not applied correctly in the certitude\_compile file.
  - The -kdb option is applied but the KDB design is not compiled and generated correctly.
  - The VerdiInitCommand configuration option is set by the user, and Certitude applies the user setting.

---

## Dumping and Comparing Waveforms in Verdi for SystemC Designs

With the integration of Certitude, VCS, Verdi, and CBug, the following benefits are available with this seamless integration:

- Dump the waveform for SystemC designs run on a specific testcase with or without an injected fault.
- Compare the reference waveform with the faulty waveform for SystemC designs.
- Generate Runtime Information Database (RIDB) for loading SystemC designs in Verdi.

This section consists of the following subsections:

- “[Use Model](#)”
- “[Points to Note](#)”

## Use Model

To use this feature, perform the following steps:

1. Specify VCS as the simulator using the following setting in the `certitude_config.cer` configuration file:

```
setconfig -Simulator=vcs
```

In the `certitude_compile` configuration file, compile the design using VCS. For example,

```
#!/bin/sh -e
# VCS compile script
syscan $CER_SYSSCAN_OPTIONS $SRC/top.cpp
vcs -sysc sc_main $CER_VCS_SC_OPTIONS
```

2. Set the `WaveUseEmbeddedDumper` configuration option to `true` in the `certitude_config.cer` configuration file to use the embedded dumper for dumping waveforms:

```
setconfig -WaveUseEmbeddedDumper=true
```

3. Invoke Certitude and execute commands for the model, activation, and detection phases.

```
>certitude  
cer> model  
cer> activate  
cer> detect
```

4. Execute the `dumpwaves` and `verdiwavedebug` commands accordingly to dump and compare waveforms.

For example,

```
cer> dumpwaves -fault=10 -testcaselist=fir_rtl  
cer> verdiwavedebug -fault=10 -testcase=fir_rtl
```

Note:

For more details on dump and compare waveforms with Certitude, see the *Certitude User Manual*.

5. Generate an RIDB file with the original source code and load the design automatically in Verdi with the `verdistart` or the `verdisourcedebug` command.

For example,

```
verdidumpridb -testcase=fir_rtl
```

---

## Point to Note

Simulation executed by the `dumpwaves` command is killed if simulation CPU timeout is reached. However, simulation is not killed if the `dumpwaves` command is executed immediately after executing the `model` command.

---

## Reducing Compilation Time in Native Certitude With VCS Partition Compile Flow

This feature is Limited Customer Availability (LCA). Limited Customer Availability (LCA) features are features available with select functionality. These features will be ready for a general release based on customer feedback and meeting the required feature completion criteria. LCA features do not need any additional license keys.

With the integration of Native Certitude with VCS Partition Compile flow, you can improve the turnaround time for successive compilations in Native Certitude.

Partition Compile is used primarily to improve the turnaround time in successive compilations. In Native Certitude, you have various phases of compilation. Therefore, when you integrate Native Certitude along with the Partition Compile flow, you are able to reduce the compilation time at each step. This results in improved performance during compilation time.

---

### Use Model

The Native Certitude use model remains the same as that of the previous use model. For information about the use model, see the *Native Integration of Certitude and VCS* section in the *Certitude User Manual*.

Native Certitude consists of three phases in the following order:

- Model
- Activation
- Detection

The three different phases always run in the same order.

The model phase and the activation phase occur in a single compilation. The model phase analyzes the DUT and creates a list of faults. The activation phase generates the `simv` executable, which is used to determine the activated faults. The detection phase performs the new compilation that generates the `simv` executable to determine the propagation and the detection status of the faults. All the phases target the DUT part of the design and they do not affect the testbench.

If you want to fine-tune the testbench and recompile the design at activation or detection phase, you must recompile the entire design including the DUT.

Similarly, after you are completed with the model and activation phases compilation and you want to perform compilation in the detection phase, you must recompile the entire design including the testbench that has not changed.

By integrating with Partition Compile, you can avoid the recompilation of unchanged testbench part of the design. The compilation of testbench partition takes place only when the testbench is changed.

---

## Example

Consider the following test cases:

```
//topcfg.v
config topcfg;
    design top;
        partition instance top.dut use dut;
endconfig

//top.v
`noinline
module top;
    dut dut();

```

```

        Test test();
endmodule

module dut;
    reg clk;
    reg data;
    initial begin
        clk=1'b0;
        #25
        $finish();
    end

    always
    #7 clk = ~clk;
Core core1(clk,data);
endmodule

module Core(input clk,output data);
    reg data;
    always @(posedge clk)
        data = ~data;
    initial begin
        $display("%m");
    end
    sub sub_core(data);
endmodule

module Test;
    initial begin
        #4 $display("%m");
    end
endmodule

module sub(input data);
    initial begin
        $display("Data: %d",data);
    end
endmodule

```

The following is the VCS compilation script that is located in the certitude\_compile configuration file:

```
% vlogan -sverilog ./topcfg.v ./top.v
```

```
% vcs topcfg -partcomp -partcomp_dir=". /partitionlib"
```

Invoke Certitude and execute commands for the model, activation and detection phases as follows:

1. Run model using the following command:

```
cer> model
```

This command compiles all partitions.

2. Run activate using the following command:

```
cer> activate -compile
```

This command compiles only the DUT partition.

3. Run detect using the following command:

```
cer> detect -compile
```

This command recompiles only the DUT partition.

Note:

You can use all default partition compile options along with the mentioned limitation. For more information on partition compile options, see *VCS/VCS LCA Features Guide*.

---

## Limitation

The feature has the following limitation:

- The DUT must be present entirely in a single partition.



# 29

## Integrating VCS with Vera

---

Vera® is a comprehensive testbench automation solution for module, block and full system verification. The Vera testbench automation system is based on the OpenVera™ language. This is an intuitive, high-level, object-oriented programming language developed specifically to meet the unique requirements of functional verification.

You can use Vera with VCS to simulate your testbench and design. This chapter describes the required environment settings and usage model to integrate Vera with VCS.

---

## Setting Up Vera and VCS

To use Vera, you must set the Vera environment as shown below:

```
% setenv VERA_HOME Vera_Installation  
% setenv PATH $VERA_HOME/bin:$PATH  
% setenv LM_LICENSE_FILE license_path:$LM_LICENSE_FILE  
or  
% setenv SNPSLMD_LICENSE_FILE license_path:$SNPSLMD_LICENSE_FILE
```

Note:

If you set the SNPSLMD\_LICENSE\_FILE environment variable, then VCS ignores the LM\_LICENSE\_FILE environment variable.

Set the VCS environment as shown below:

```
% setenv VCS_HOME VCS_Installation  
% setenv PATH $VCS_HOME/bin:$PATH  
% setenv LM_LICENSE_FILE license_path:$LM_LICENSE_FILE  
or  
% setenv SNPSLMD_LICENSE_FILE license_path:$SNPSLMD_LICENSE_FILE
```

Note:

If you set the SNPSLMD\_LICENSE\_FILE environment variable, then VCS ignores the LM\_LICENSE\_FILE environment variable.

For more information on VCS installation, see “[Setting Up the Simulator](#)” .

---

## Using Vera with VCS

The usage model to use Vera with VCS includes the following steps:

- Compile your OpenVera code using Vera

This will generate a `.vro` file and a `filename_vshell.v` file.  
The `filename_vshell.v` is a Verilog file.

The following table lists the Vera options to generate a shell file based on your design topology:

**Table 0-1.**

| Option               | Description                                                                                                               |
|----------------------|---------------------------------------------------------------------------------------------------------------------------|
| <code>-vlog</code>   | Generates a Verilog shell file, <code>filename_vshell.v</code> . Use this option if your design is a Verilog-only design. |
| <code>-sro</code>    | Generates a VHDL shell file, <code>filename_vshell.vhd</code> . Use this if your design is a VHDL-only design.            |
| <code>-sro_mx</code> | Generates a VHDL shell file, <code>filename_vshell.vhd</code> . Use this if your design top is in VHDL.                   |
| <code>-vcs_mx</code> | Generates a Verilog shell file, <code>filename.vshell</code> . Use this if your design top is in Verilog.                 |

- Analyze all Verilog files including the vshell file generate in the above step.
- Analyze all VHDL files.
- Compile/elaborate your design and the `filename_vshell.v` file using the `-vera` option. This option is required to use Vera with VCS.

- Simulate the design by specifying the `.vro` file created in the first step using the `+vera_load` runtime option. You can also specify this `.vro` file in the `vera.ini` file in your working directory as shown in the following example:

```
vera_load = tb_top.vro
```

See the *Vera User Guide* for more information.

---

## Usage Model

Use the following usage model to compile OpenVera code using Vera:

```
% vera -cmp [Vera_options] OpenVera_files
```

See the *Vera User Guide* for a list of Vera compilation options.

## Two-Step Flow

### Compilation

```
% vcs [compile_options] -vera verilog_filelist
filename_vshell.v
```

### Simulation

```
% simv [simv_options] +vera_load=file.vro
```

## Three-Step Flow

### Analysis

```
% vlogan [vlogan_options] Verilog_files filename.vshell
% vhdlan [vhdlan_options] VHDL_files
```

## **Elaboration**

```
% vcs [elab_options] -vera top_entity/module/config  
filename_vshell.v
```

## **Simulation**

```
% simv [simv_options] +vera_load=file.vro
```



# 30

## VCS Mixed-Signal Simulation

---

This chapter describes Synopsys CustomSim™ simulator and VCS mixed-signal mixed-HDL environment setup and usage model for better understanding. Supported analog simulators also include HSIM and FineSim. You can use any one of the simulators to do mixed-signal simulations.

Before reading the subsequent topics in these sections, you must be familiar with the:

- SPICE, Verilog, and VHDL languages
- CustomSim and VCS usage

This section consists of the following topics:

- “[Introduction to VCS and CustomSim](#)”
- “[Setting up the Environment](#)”

- “Scheduling Analog-to-Digital Events in the NBA Region”

For more information about CustomSim, see the Discovery AMS: *Mixed-Signal Simulation User Guide*. For more information about CustomSim HSIM, see the CustomSim HSIM documentation. For more information about CutomSim FineSim, see the *FineSim User Guide: Pro and SPICE Reference*.

---

## Introduction to VCS and CustomSim

The VCS and CustomSim cosimulation feature provides mixed-signal mixed-HDL verification solution. This feature enables you to simulate a design, which is described in SPICE (or other transistor-level description language that FineSim supports), Verilog-HDL (Verilog), and VHDL.

VCS and CustomSim cosimulation supports:

- Verilog-top, VHDL-top, and SPICE-top netlist configurations.
- Verilog and VHDL as digital modeling languages.
- Donut design configuration, which is the interleaved instantiations of SPICE sub-circuits and Verilog or VHDL digital cells in the design hierarchy.
- The use of cell-based partitioning.

In the VCS and CustomSim cosimulation flow, if a SPICE cell is instantiated under a VHDL block, a dummy Verilog wrapper is required for the instantiated SPICE cell. For successful SPICE instantiation, this wrapper file must be analyzed like any other Verilog file.

Note:

CustomSim and VCS cosimulation uses Direct Kernel Interface to exchange data between CustomSim and VCS.

The VCS and CustomSim mixed-signal simulation process involves the following three phases:

1. [Analyzing a Design](#)
  2. [Elaborating a Design](#)
  3. [Running the Simulation](#)
- 

## Analyzing a Design

During design analysis, the syntax of Verilog and VHDL files is verified and intermediate files are generated. The generated intermediary files are later used during the elaboration phase. Any syntax errors in Verilog or VHDL netlists are flagged at this phase.

---

## Elaborating a Design

During elaboration, the design hierarchy is built based on the information obtained from the analysis phase. In this phase, incorrect port connectivity or missing definitions for instantiated blocks in Verilog, VHDL, or SPICE are identified and flagged, if they exist. If no error is encountered, at the end of the Elaboration phase, the binary executable is generated.

---

## **Running the Simulation**

To start the mixed-signal simulation, run the executable generated during the elaboration phase.

A working installation of VCS and a matching version of CustomSim are required to run VCS and CustomSim mixed-signal mixed-HDL simulation that can be found at: <https://solvnet.synopsys.com/retrieve/1463626.html>

---

## **Setting up the Environment**

You must set the following environment variables, before running the VCS and CustomSim simulation:

### **Licenses**

Either `LM_LICENSE_FILE` or `SNPSLMD_LICENSE_FILE` can be used to specify the license file location:

```
% setenv LM_LICENSE_FILE license_file_path  
or
```

```
% setenv SNPSLMD_LICENSE_FILE license_file_path
```

### **Required UNIX Paths and Variable Settings**

To set the paths for CustomSim and VCS, do the following:

#### **For CustomSim**

```
% source XA_install_directory/CSHRC_xa
```

## **For CustomSim HSIM**

```
% setenv VCS_HOME VCS_Installation  
% setenv HSIM_HOME HSIM_Installation  
% setenv HSIM_64 1
```

Unset the variable `HSIM_64`, if you are using in 32-bit mode.

Note:

If you set the `SNPSLMD_LICENSE_FILE` environment variable, then VCS ignores the `LM_LICENSE_FILE` environment variable.

## **For CustomSim FineSim**

```
% source FineSim_install_directory/finesim.cshrc
```

## **For VCS**

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

---

## **Use Model**

Using VCS and CustomSim involves the following three phases:

### **1. Analysis**

Execute the following during analysis:

```
% vlogan [vlogan_options] Verilog_files  
% vhdlan [vhdlan_options] VHDL_files
```

### **2. Elaboration**

Execute the following during elaboration:

```
% vcs -ad=initFile [elab_options] top_entity/module/  
config
```

### 3. Simulation

To enable mixed-signal simulation, use the `-ad=initFile` elaboration option. If you use `-ad` without specifying the `initFile`, VCS assumes the mixed-signal setup filename as `vcsAD.init`.

```
% simv [simv_options]
```

---

## Example

The following example shows a sample compilation script that contain commands to analyze and elaborate a design with VHDL, Verilog, and SPICE components. In this example, the files `tb.vhd` and `blk_1.vhd` contain the VHDL netlist, files `blk_2.v` and `blk_3.v` contain the Verilog netlist, and the file `all_spice.spi` contains the SPICE netlist.

**Example:**

```
% vlogan blk_2.v blk_3.v
% vhdlan tb.vhd blk1.vhd
% vcs -ad=setup.init testbench
% simv
```

In this example, `testbench` is the name of the top-level entity. The mixed-signal setup file `setup.init` is shown in the following code:

### For CustomSim

```
choose xa -n all_spice.spi;
use_spice -cell counter ddr_flop;
set bus_format <%d>;
```

## **Limitation**

If a testbench contains an array of instances, SPICE substitution is not supported for the elements of the array.

For example, in the following command, if `top.dut5[1]` is an instance of the array, `addr2`, SPICE substitution does not occur for the `top.dut5[1]` instance.

```
use_spice -cell addr2 -inst top.dut5[1];
```

## **For CustomSim HSIM**

```
choose hsim all_spice.spi;
use_spice -cell counter ddr_flop;
set bus_format <%d>;
```

## **For CustomSim FineSim**

```
choose finesim -n all_spice.spi;
use_spice -cell counter ddr_flop;
set bus_format <%d>;
```

Where, `counter` and `ddr_flop` are the names of the multi-view cells. The SPICE view of these cells are used in this design.

---

## **Scheduling Analog-to-Digital Events in the NBA Region**

VCS enables you to schedule a2d events in the nonblocking assignment (NBA) region. With this feature you will have more flexibility on data latching and more control to handle data race conditions thereby bringing in more predictability in mixed signal designs.

---

## Use Model

To schedule all a2d events on a given SPICE node in the NBA region, add the `queue=blocking|nonblocking` option to the `a2d` command in the mixed signal setup file (`vcsAD.init`).

```
a2d node=<spice_node> queue=nonblocking;
```

The default value is `blocking` and all a2d events on `<spice_node>` to the digital field are scheduled in the active region. When the value is `nonblocking`, the a2d events on the given node are scheduled in the NBA region.

Note:

If a SPICE node for which `queue=nonblocking` is assigned turns to be a digital-to-digital (d2d) node, then this feature will have no effect on the SPICE node.

---

## Support of Verilog Force and Release Assignments on Wreal Nets

The force and release on wreal nets allows you to force and release wreal nets from the testbench code written in UCLI, MHPI, or VPI and allows you to use `$hdl_xmr_force/$hdl_xmr_release` system tasks. It is useful when there are large number of designs and instances.

It does not support deposit functionality. It generates an error message if you try to deposit values on wreal nets. This applies to all sources of deposit including:

- `$deposit`

- UCLI force -deposit
  - \$hdl\_xmr
  - vpi\_put\_value without vpiForceFlag
- 

## Usage Example

The following example supports usage of force and release assignments on wreal nets:

### *Example 30-1 top.v*

```
module top;
    mid m();
endmodule

module mid;
    wreal wr[1:2];
    bot b();
    always @(wr[1]) $display ("wr[1]=%g at %0t", wr[1], $time);
endmodule

module bot;
    real r;
    assign top.m.wr[1] = r;

    initial begin
        r = 0.1;
        repeat (5) #10 r = r * 2.0;
        $finish;
    end
endmodule
```

### *Example 30-2 force.tcl*

```
catch {force {top.m.wr[1]} 1.0 -deposit} err
puts $err
force {top.m.wr[1]} 8.0
```

```
run 1
get {top.m.wr[1]}
release {top.m.wr[1]}
run 3
get {top.m.wr[1]}
quit
```

***Command line:***

```
% vcs top.v -sverilog -wreal res_sum
-debug_access+all
```

```
% ./simv -ucli -i force.tcl
```

***The following output is generated:***

```
ucli% catch {force {top.m.wr[1]} 1.0 -deposit} err
1
ucli% puts $err
Error-[UCLI-FORCE-ERR-WREAL] NYI force on WREAL
    Force command failed on object 'top.m.wr[1]' with type
    'vpiRealWire'.
    Command force deposit on wreal net is not yet supported.
    Please refer to UCLI user guide for more documentation on
    force commands.
ucli% force {top.m.wr[1]} 8.0
ucli% run 1
ucli% get {top.m.wr[1]}
8.000000
ucli% release {top.m.wr[1]}
ucli% run 3
ucli% get {top.m.wr[1]}
0.100000
ucli% quit
```

---

## Limitations

The feature has the following limitations:

- The `$deposit` system task on the wreal net is not supported.
- The force/release on wreal arrays that are present in modules instantiated in VHDL is not supported.
- The force/release on wreal arrays are not supported when wreal scalarization is disabled using `-wreal noscalarize` option.

## Support for Wreal Nets in Verilog-AMS Flow

Connect modules are inserted based on the specified connect rules. Connect rules are created to control which connect modules must be used and where they must be inserted. The connect modules are also searched as per the connect rules defined and first matched connect module is selected.

To select the connect module, the discipline of the `wreal` net or port should also match along with the `wreal` type.

## Usage Example

The following example explains how to select the connect module that matches the discipline of the `wreal` net or port along with the `wreal` type:

```
module bot(output wreal p1);
    myLogical p1;
    assign p1 = 1.0;
    anaMod ANAMOD (p1);
endmodule

// Analog module
module anaMod (e1);
    output e1;
```

```

    electrical e1;
endmodule

// Following connect module has output port with myLogical
discipline with wreal port type.
connectmodule elect_to_wreal (ain, dout);
    input ain;
    electrical ain;

    output dout;
    wreal dout;
    myLogical dout;
    .....
endmodule

// Following connect module has output port with myLogical
discipline with wire type.
connectmodule elect_to_logic (ain, dout);
    input ain;
    electrical ain;

    output dout;
    myLogical dout;
    .....
endmodule

connectrules mixedsignal;
connect elect_to_logic input electrical, output myLogical;
connect elect_to_wreal input electrical, output myLogical;
.....
endconnectrules

```

In the above example, a connect module insertion is required inside the bot module because p1 wreal signal is connected to the e1 electrical port of the anaMod module. To select the connect module, there are two connect rules that refer to two different connect modules, that is, elect\_to\_logic and elect\_to\_wreal respectively.

The `elect_to_wreal` connect module has digital output port of type `wreal` with `myLogical` discipline. Therefore, the `elect_to_wreal` connect module is selected as per the connect module selection algorithm.

---

## Support for SystemC Designs in Verilog-AMS

Verilog-AMS is supported with SystemC in the design topologies using the same option `-sysc=ams`.

Note:

The `-ams` option must be provided along with the `-sysc=ams` option.

---

## Use Model

This section provides the use model to support Verilog-AMS and SPICE with SystemC designs.

The other options of VCS elaboration for mixed signal simulations remain the same.

If the `-sysc=ams` option is not used for Verilog-AMS elaboration, then an error message is generated that mentions Verilog-AMS is not supported.

---

## Usage Example

The following example illustrates the usage of SystemC designs in Verilog-AMS and SPICE:

## **test.v**

```
`include "connectmodules.vams"
`timescale 1ns/1ps

// Function to check values of 1, 0, x or z
function check_10xz (input logic orig_logic, input logic ref_logic, input integer tag);
    if(orig_logic === ref_logic) $display("Passed - %-d", tag);
    else $display("Failed - %-d : Orig_value = %-d : Ref_value = %-d", tag, orig_logic, ref_logic);
    return 0;
endfunction

module amp (out, in);
    input in; output out;

    electrical out, in;
    parameter real Gain = 2;

    analog
        V(out) <+ Gain * V(in);
endmodule

module testbench();
    electrical out,in;
    bit sysc_out, clk;

    amp a(out,in);

    initial clk = 1'b0; initial #1000 $finish();
    always #10 clk = ~clk;
    analog begin
        V(in) <+ transition((clk == 1) ? 3.3 : 0.0, 0.1n,
        0.1n ,0.1n);
    end

    sysc_inv SYS_INV (out, sysc_out);

    initial begin
        #6 check_10xz(sysc_out, 1'b1, 1);
    end

```

```

#6 check_10xz(sysc_out, 1'b0, 2);
end
endmodule

```

### *disciplines.vams*

```

discipline logical
domain discrete;
enddiscipline

nature Current
  units = "A";
  access = I;
  idt_nature = Charge;
  abstol = 1e-12;
endnature

nature Charge
  units = "coul";
  access = Q;
  ddt_nature = Current;
  abstol = 1e-14;
endnature

nature Voltage
  units = "V";
  access = V;
  idt_nature = Flux;
  abstol = 1e-6;
endnature

nature Flux
  units = "Wb";
  access = Phi;
  ddt_nature = Voltage;
  abstol = 1e-9;
endnature

discipline electrical
  potential Voltage;
  flow Current;

```

```
enddiscipline
```

### ***connectmodules.vams***

```
`include "disciplines.vams"
`timescale 1ps/1ps

connectmodule elect_to_logic(el, cm);
    input el; electrical el;
    output cm; logical cm;
    reg cm_1;
    real vel;
    assign cm = cm_1;

    always @(above(V(el) - 1.65)) begin
        cm_1 = 1;
    end
    always @(above(1.65 - V(el))) begin
        cm_1 = 0;
    end
endmodule

connectmodule logic_to_elect(l2e_cm,l2e_el);
    input l2e_cm; logical l2e_cm;
    output l2e_el; electrical l2e_el;

    assign l2e_cm = l2e_cm;
    analog begin
        V(l2e_el) <+ transition((l2e_cm == 1) ? 3.3 : 0.0,
0.1n, 0.1n ,0.1n);
    end
endmodule

connectrules mixedsignal;
    connect elect_to_logic input electrical, output logical;
    connect logic_to_elect input logical, output electrical;
endconnectrules
```

### ***sysc\_inv.cpp***

```

#include "sysc_inv.h"

sysc_inv.h

#ifndef _sysc_inv_h_
#define _sysc_inv_h_

#include "systemc.h"

SC_MODULE(sysc_inv) {
    sc_in<bool> ina;
    sc_out<bool> outx;

    SC_CTOR(sysc_inv) : ina("ina"), outx("outx") {
        SC_METHOD(action);
        sensitive << ina;
    }

    void action() {
        outx.write(!ina.read());
    }
};

#endif

```

### *test.init*

```
choose xa
```

To run the example, use the following commands:

```
%> vlogan -sverilog test.v -ams
%> syscan sysc_inv.cpp:sysc_inv
%> vcs -timescale=1ns/1ps -sysc=ams testbench -
  ams_discipline logical -ams -ams_iereport -full64 -sverilog
  -ad=test.init
%> simv
```

This plugs-in the preverified SystemC models into Verilog-AMS simulation.

---

## Support for Wildcard Character and the -exclude Option in the Mixed Signal Control Command

The `-exclude` option is supported in the `use_verilog mixed` signal control command, which can be used to provide the instances that must not be partitioned. Also, the "\*" wildcard character is supported in the hierachal path of the `-inst` and `-exclude` options in the `use_verilog` mixed signal control command. This feature is useful when there are large number of instances in the design.

---

### Use Model

The syntax of the `-exclude` option with the wildcard character is as follows:

```
use_verilog -module inverter -inst top.*.*  
-exclude top.a0.*;
```

**Note:**

The "\*" wildcard character in the instance name is only supported if the `-adopt wildcard` option is added to the VCS command line as follows:

```
% vcs -adopt wildcard ...
```

For example:

```
% vcs -sverilog example.v -ad=example.init -adopt  
wildcard -full64
```

**Note:**

The use of the `-adopt wildcard` option and the "\*" wildcard character in the `use_verilog` command increases (sometimes almost doubles) the compilation time.

---

## Usage Example

The following example illustrates usage of `-exclude` option in the `use_verilog` command.

*Example 30-3 Example of usage of -exclude option in the use\_verilog command*

### *Example.spi*

```
.subckt inv i zn
xi0 i zn inverter
.ends

.subckt inverter in out
mn1 out in gnd gnd n w=1u l=1u
mp1 out in vdd vdd p w=4u l=1u
.ends
```

### *Example.init*

```
choose xa -n example.spi;

use_spice -cell inv;
use_verilog -module inverter -inst top.*.* -exclude
top.a0.i0;
resolve_x_inst_prefix enable;
```

---

## Limitation

The feature has the following limitation:

- Wildcard character is restricted to only one layer of design hierarchy.  
For example if `top.*.b1` design is searched in the following design hierarchies  
`top.a1.b1`  
`top.p0.a1.b1`  
then the wildcard character is applied only to the design  
`top.a1.b1.`

# 31

## Integrating VCS with Specman

---

The VCS ESI Adapter integrates VCS with the Specman Elite. This chapter describes how to prepare a stand-alone VHDL/Verilog design or mixed VHDL/Verilog design for use with the ESI interface. See the *Specman Elite User Guide* for further information.

VCS has two ESI adapters, one for Verilog and the other for VHDL. You can use both the adapters together for mixed HDL simulation. VHDL adapter is implemented as a VHPI foreign architecture, while the Verilog adapter is implemented as a Verilog PLI application.

VHDL adapter is called as `specman_vcsmx.vhd`. This file is generated using the `specman` command as explained when using VCS H-2013.06 release with Specman 12.2 version. Generate the VHDL stub file using the following `specman` command:

```
% specman -c "load xor_verify.e ; write stubs -  
vcsmx_vhdl"
```

Analyze the VHDL stub file using the following command:

```
% vhdlan -nc specman_vcsmx.vhd
```

This chapter includes the following topics:

- “[Type Support](#)”
- “[Usage Flow](#)”
- “[Using specrun and specview](#)”
- “[Version Checker for Specman](#)”
- “[Version Checker for Specman](#)”

---

## Type Support

The VCS ESI adapter supports the following VHDL types:

- Predefined types
  - bit
  - Boolean
  - std\_logic/std\_ulogic
  - character
  - array
- User-defined enum types
- VHDL memory
- in/out/inout/buffer ports

- Access to elements of the following composite types supported:
  - Access to individual elements of any of the supported scalar types
  - Predefined types based on any of the supported scalar types such as string, bit\_vector, integer, and so on.

Note:

Calling VHDL procedure or functions through e code is not supported.

The VCS ESI adapter supports the following Verilog Types:

- nets
- wires
- registers
- integers
- array of registers (verilog memory)

Other Verilog support:

- Verilog macros
- Verilog tasks
- Verilog functions
- Verilog events
- in/out/inout ports

---

## Usage Flow

The Specman usage model for VCS depends upon whether the e code can access both VHDL and Verilog, or just one language. If the e code can access just one language, then you do not have to specify the unused part.

This section explains how to integrate Specman with VCS flow.

---

## Setting Up The Environment

To set up the environment to run Specman with VCS:

- Set your `VCS_HOME` and `VRST_HOME` environment variables:

```
% setenv VCS_HOME [vcs_installation_path]
% set path = ($VCS_HOME/bin $path)
% setenv VRST_HOME [specman installation]
```

- Source your `env.csh` file for Specman:

```
% source ${VRST_HOME}/env.csh
```

For 64-bit simulation, source your `env.csh` file as shown below:

```
% source ${VRST_HOME}/env.csh -64bit
```

- Source the `environ.csh` file for VCS:

```
% source $VCS_HOME/bin/environ.csh
```

- Set your environment for the VCS Specman ESI adapter:

```
% setenv SPECMAN_VCSMX_VHDL_ADAPTER ${VCS_HOME}/${ARCH}/
lib/libvhdl_sn_adapter.so
```

---

## Specman e code accessing VHDL only

Instantiate `SPECMAN_REFERENCE` in the top-level VHDL code as follows:

```
component comspec
end component;
for all: comspec use entity work.SPECMAN_REFERENCE(arch);

I: comspec;
```

Note:

In a Verilog-top design, instantiate `SPECMAN_REFERENCE` in one of the top-level VHDL files underneath the Verilog-top code.

Analyze Verilog design files as shown below:

```
% vlogan [vlogan_options] -f Verilog_filename_list
```

Analyze the VHDL stub file and then VHDL design files as shown below:

```
% vhdlan specman_vcsmx.vhd
% vhdlan [vhdlan_options] file1.vhd file2.vhd
```

Elaborate the design as given in the following table:

| <b>Elaboration Mode</b> |                      | <b>Commands</b>                                                                                                                      | <b>Generated Executable</b> |
|-------------------------|----------------------|--------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| Compile                 | Execution with -o    | "% sn_compile.sh -sim vcs \ -sim_flags "[compile-time_options] \ -debug_access top_cfg/entity/module" -o <exe_name> <top_e_file>.e " | vcs_<exe_name>              |
|                         | Execution without -o | "% sn_compile.sh -sim vcs \ -sim_flags "[compile-time_options] \ -debug_access top_cfg/entity/module" <top_e_file>.e "               | vcs_<top_e_file>            |
| Loaded                  | Execution with -o    | "% sn_compile.sh -sim vcs \ -sim_flags "[compile-time_options] \ -debug_access top_cfg/entity/module" -o <exe_name> "                | <exe_name>                  |
|                         | Execution without -o | "% sn_compile.sh -sim vcs \ -sim_flags "[compile-time_options] \ -debug_access top_cfg/entity/module" "                              | vcs_specman                 |

Simulate the design as given below:

- In Compiled mode:

```
% vcs_simv -ucli [simv_options]
ucli% sn "test"
ucli% run
ucli% quit
```

**Note:**

Notice the use of the `-o` option with this script in compile mode to change the name of the executable generated to `vcs_simv` from the default name given by the script which is `vcs_<top_e_file>`.

- **In Loaded mode:**

```
% simv -ucli [simv_options]  
ucli% sn "load <top_e_file>; test"  
ucli% run  
ucli% quit
```

**Note:**

Notice the use of the `-o` option with this script in loaded mode to change the name of the executable generated to `simv` from the default name given by the script which is `vcs_specman`.

---

## Specman e Code Accessing Verilog Only

Create the Verilog stub file `specman.v` and analyze/compile all Verilog files including `specman.v` as shown below:

**Compile step:**

```
% specman -c "load [top_e_file]; write stubs -verilog;"  
% vlogan [vlogan_options] -f Verilog_filename_list specman.v
```

**Analyze all VHDL design files as shown below:**

```
% vhdlan [vhdlan_options] file1.vhd file2.vhd
```

Elaborate/Compile the design as given in the following table:

| <b>Elaboration Mode</b> |                      | <b>Commands</b>                                                                                                                                          | <b>Generated Executable</b> |
|-------------------------|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| Compile                 | Execution with -o    | "% sn_compile.sh -sim vcs \<br>-sim_flags "[compile-<br>time_options] \<br>-debug_access top_cfg/<br>entity/module" -o<br><exe_name> <top_e_file>.e<br>" | vcs_<exe_name>              |
|                         | Execution without -o | "% sn_compile.sh -sim vcs \<br>-sim_flags "[compile-<br>time_options] \<br>-debug_access top_cfg/<br>entity/module" <top_e-<br>file>.e<br>"              | vcs_<top_e_file>            |
| Loaded                  | Execution with -o    | "% sn_compile.sh -sim vcs \<br>-sim_flags "[compile-<br>time_options] \<br>-debug_access top_cfg/<br>entity/module" -o<br><exe_name><br>"                | <exe_name>                  |
|                         | Execution without -o | "% sn_compile.sh -sim vcs \<br>-sim_flags "[compile-<br>time_options] \<br>-debug_access top_cfg/<br>entity/module"<br>"                                 | vcs_specman                 |

Simulate the design as given below:

- In compiled mode:

```
% vcs_simv -ucli [simv_options]
ucli> sn "test"
```

```
ucli> run  
ucli> quit
```

Note:

Notice the use of the `-o` option with this script in compile mode to change the name of the executable generated to `vcs_simv` from the default name given by the script which is `vcs_<top_e_file>`.

- In loaded mode:

```
% simv -ucli [simv_options]  
ucli% sn "load <top_e_file>; test"  
ucli% run  
ucli% quit
```

Note:

Notice the use of the `-o` option with this script in loaded mode to change the name of the executable generated to `simv` from the default name given by the script which is `vcs_specman`.

---

## Specman e code accessing both VHDL and Verilog

Instantiate `SPECMAN_REFERENCE` in the top-level VHDL code as follows:

```
component comspec  
end component;  
for all: comspec use entity work.SPECMAN_REFERENCE(arch);  
  
I: comspec;
```

**Note:**

In a Verilog-top design, instantiate **SPECMAN\_REFERENCE** in one of the top-level VHDL files underneath the Verilog-top code.

Create the Verilog stub file `specman.v` and analyze all Verilog files including `specman.v` as shown below:

```
% specman -c "load [top_e_file]; write stubs -verilog;"  
% vlog [vlogan_options] -f Verilog_filename_list specman.v
```

Create the VHDL stub file `specman_vcsmx.vhd` and analyze all VHDL files including `specman_vcsmx.vhd` as shown below:

```
% specman -c "load xor_verify.e ; write stubs -vcsmx_vhdl"
```

Analyze the VHDL stub file and then VHDL design files as shown below:

```
% vhdlan specman_vcsmx.vhd  
% vhdlan [vhdlan_options] file1.vhd file2.vhd
```

Elaborate the design as given in the following table:

| <b>Elaboration Mode</b> |                      | <b>Commands</b>                                                                                                                                          | <b>Generated Executable</b> |
|-------------------------|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| Compile                 | Execution with -o    | "% sn_compile.sh -sim vcs \<br>-sim_flags "[compile-<br>time_options] \<br>-debug_access top_cfg/<br>entity/module" -o<br><exe_name> <top_e_file>.e<br>" | vcs_<exe_name>              |
|                         | Execution without -o | "% sn_compile.sh -sim vcs \<br>-sim_flags "[compile-<br>time_options] \<br>-debug_access top_cfg/<br>entity/module" <top_e_-<br>file>.e<br>"             | vcs_<top_e_file>            |
| Loaded                  | Execution with -o    | "% sn_compile.sh -sim vcs \<br>-sim_flags "[compile-<br>time_options] \<br>-debug_access top_cfg/<br>entity/module" -o<br><exe_name><br>"                | <exe_name>                  |
|                         | Execution without -o | "% sn_compile.sh -sim vcs \<br>-sim_flags "[compile-<br>time_options] \<br>-debug_access top_cfg/<br>entity/module"<br>"                                 | vcs_specman                 |

Simulate the design as given below:

- In Compiled mode:

```
% vcs_simv -ucli [simv_options]
# sn "test"
# run
# quit
```

**Note:**

Notice the use of the `-o` option with this script in compile mode to change the name of the executable generated to `vcs_simv` from the default name given by the script which is `vcs_<top_e_file>`.

- In Loaded mode:

```
% simv -ucli [simv_options]
# sn "load <top_e_file>; test"
# run
# quit
```

**Note:**

Notice the use of the `-o` option with this script in loaded mode to change the name of the executable generated to `simv` from the default name given by the script which is `vcs_specman`.

---

## **Guidelines for Specifying HDL Path or Tick Access with VCS-Specman Interface**

The guidelines to specify HDL path or tick access with VCS-Specman interface are as follows:

- You cannot mix `[]` and `("()")` in a single tick access or HDL path.
- HDL path or tick access notation should use `[]` on `e` side through VHDL generate. If you do not use `[]`, an adapter error is generated, to specify that the signal is not found. Apparently, `()` conflicts with the computed names in `e` code.
- Specman generates an error, if you use `("()")` in HDL path.

- In the tick access notation, you must use [ ] or (" () ") , instead of () . Apparently, () conflicts with the computed names in e code.
- You cannot use : , as a starting delimiter in the absolute HDL path in e code.

**Example:** ~:test\_top"m1.b

---

## Using specrun and specview

VCS allows you to use the following Specman utilities to simulate your design:

- specrun
- specview

specrun invokes Specman in batch mode, while specview invokes the Specman GUI. The usage model is shown below:

### Using specrun

- In Compiled Mode:

```
% specrun -p "test -seed=1;" simv [simv_options]
```

- In Loaded Mode:

```
% specrun -p "load [top_e_file]; test -seed=1;" \
simv [simv_options]
```

### Example 31-1 To Invoke UCLI Using specrun

The following command invokes the simulation in UCLI mode:

```
% specrun -p "test -seed=1;" simv -ucli -i include.cmd
```

## Using specview

Set the environment variable `SPECMAN_OUTPUT_TO_TTY` as shown below:

```
% setenv SPECMAN_OUTPUT_TO_TTY 1
```

- In Compiled Mode:

```
% specview -p "test -seed=1;" -sio simv -ucli
```

- In Loaded Mode:

```
% specview -p "load [top_e_file]; test -seed=1;" \  
-sio simv -ucli
```

You can also specify VCS runtime options with `specview` or `specrun`.

### *Example 31-2 To Invoke UCLI Using specview*

You can use `-ucli` with `specview` to invoke simulation in UCLI mode.

```
% specview -p "test -seed=1;" -sio simv -ucli
```

---

## Version Checker for Specman

This section describes how to check the compatibility version of Specman with VCS. If non-compatible version of Specman is used, then VCS generates a warning message at compile-time/elaboration-time.

---

## Use Model

- **Two-step Flow**

Through Command-line Options

```
% vcs +warn=V2V_CHECK_SPECMAN
```

To convert warning to error:

```
% vcs -error=V2V_CHECK_SPECMAN
```

Enabling at Runtime:

```
%simv +warn=V2V_CHECK_SPECMAN
```

You can use the `+warn=noV2V_CHECK_SPECMAN` option to turn off the warning message. In this option, `no` specifies disabling warning messages .

- **Three-step Flow**

Through command-line options:

```
% vlogan
```

```
% vhdlan
```

```
% vcs +warn=V2V_CHECK_SPECMAN
```

```
%simv +warn=V2V_CHECK_SPECMAN
```

To convert warning to error:

```
% vcs -error=V2V_CHECK_SPECMAN
```

You can use the `+warn=noV2V_CHECK_SPECMAN` option to turn off the warning message. In this option, `no` specifies disabling warning messages.

- Through `synopsys_sim.setup` file for VCS flow:

```
V2V_CHECK_SPECMAN=TRUE/FALSE
```

- Through new environment variable for VCS flow:

```
% setenv V2V_CHECK_SPECMAN TRUE/FALSE
```

## Precedence Order

1. Command-line
2. Setup file
3. Environment variable

In VCS flow, command-line will have the highest priority compared to setup file and environment variable. Also, runtime enabling is automatically done, when enabled using environment variable or setup file.

# 32

## Integrating VCS with Denali

---

Denali is a third-party Memory Modeler - Advanced Verification (MMAV) tool, which you can integrate with VCS using a set of APIs. Denali provides a complete solution for memory modeling and system verification. It automatically monitors all the timing and protocol requirements specified by the memory vendor.

---

### Setting Up Denali Environment for VCS

To use Denali with VCS, set your Denali environment using the following commands:

```
% setenv DENALI [installation_path_of_DENALI]  
% setenv LM_LICENSE_FILE [Denali_license]:$LM_LICENSE_FILE  
% setenv LD_LIBRARY_PATH $DENALI/vhci:$LD_LIBRARY_PATH
```

---

## Integrating Denali with VCS

The generic functionality of various memory architectures are captured in a set of highly-optimized C models. The vendor-specific features and the timing for any particular memory device are defined within the specification of memory architecture (SOMA) file. After the Denali model objects are linked into the simulation environment, modeling any type of memory is as simple as referencing the appropriate SOMA file for that particular memory device.

To access a particular SOMA file, include the following declaration in the source code:

For VHDL portions of designs:

```
GENERIC (
    memory_spec: string := soma_file_path;
    init_file: string := "";
);
```

For Verilog portions of designs:

```
parameter memory_spec = soma_file_path;
parameter init_file = "";
```

**Note:**

`memory_spec` and `init_file` are keywords.

---

## Use Model

Denali provides you both Verilog and VHDL memory models. However, for mixed HDL designs, Synopsys recommends you to use either Verilog or VHDL memory model for the whole design. The use model does not allow mixing of PLI and VHPI calls.

This section describes the following topics:

- Use Model for VHDL Memory Models
- Use Model for Verilog Memory Models
- Execute Denali Commands at UCLI Prompt

---

## Use Model for VHDL Memory Models

The VHDL memory models must be integrated with VCS using VHPI calls in the VHDL design code as shown in the following command:

```
attribute foreign of [architecture_name]: architecture is
    "vhpi:[library_name]:[elaboration_function_name]:
     [initialisation_function_name]:[model_name]";
```

For example,

```
attribute foreign of behavior: architecture is
    "vhpi:denvhpi:flashElabVHPI:flashInitVHPI:mobilesdram";
```

You can use VHDL memory models with the following types of design topologies:

- VHDL DUT and VHDL Testbench
- VHDL DUT and Verilog Testbench
- Verilog DUT and VHDL Testbench

The use model is as follows:

### Analysis

```
% vlogan [vlogan_options] file2.v file3.v
% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd \
  [memory_model.vhd] [memory_wrapper.vhd]
```

## **Elaboration**

```
% vcs [vcs_options] top_entity/module/config
```

## **Simulation**

```
% simv [simv_options]
```

---

## **Use Model for Verilog Memory Models**

You can integrate Verilog memory models with VCS using PLIs. To use Verilog memory models, specify the `pli.tab` file and `denverlib.o` during compilation/elaboration.

You can use Verilog memory models with the following types of design topologies:

- Verilog DUT and Verilog Testbench
- VHDL DUT and Verilog Testbench
- Verilog DUT and VHDL Testbench

The use model is as follows:

## **Two-Step Flow**

### **Compilation**

```
% vcs -debug_access+f [vcs_options] verilog_filelist\  
-P $DENALI/verilog/pli.tab $DENALI/verilog/denverlib.o
```

### **Note:**

To compile the design in 64-bit mode, use the `-lpthread` option.

### **Simulation**

```
% simv [simv_options]
```

## Three-Step Flow

### Analysis

```
% vlogan [vlogan_options] file2.v file3.v \
[memory_model.v] [memory_wrapper.v]

% vhdlan [vhdlan_options] file3.vhd file2.vhd file1.vhd
```

### Elaboration

```
% vcs -debug_access+f [vcs_options] top_entity/module/
config -P $DENALI/verilog/pli.tab $DENALI/verilog/
denverlib.o
```

### Note:

To elaborate the design in 64-bit mode, use the `-lpthread` option.

### Simulation

```
% simv [simv_options]
```

---

## Execute Denali Commands at UCLI Prompt

VCS allows you to execute Denali commands at the UCLI prompt.

For example,

```
% simv -ucli
ucli% mmload :top:I_dut:I_denali_model data_file
```

This UCLI command loads Denali memory in the `I_denali_model` instance with the data specified in the `data_file`.

For more information on invoking UCLI, see “[Using UCLI](#)” .



# 33

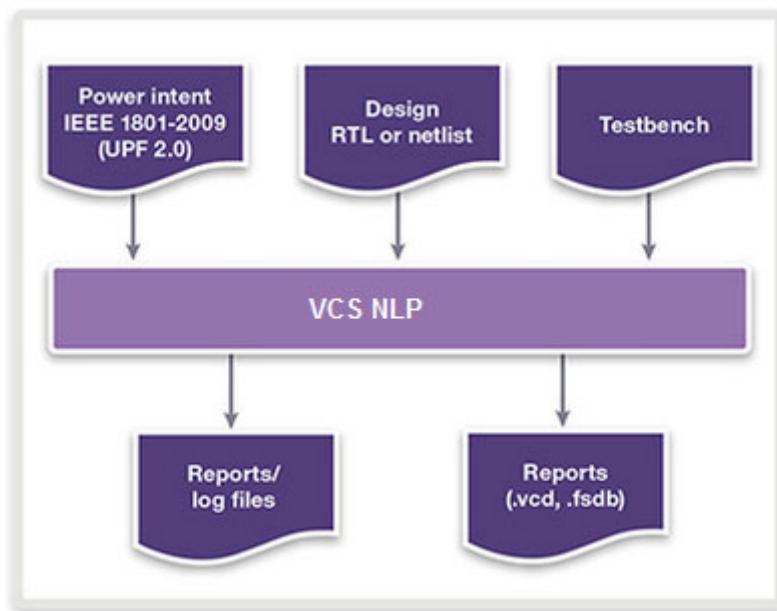
## Integrating VCS with Native Low Power (NLP)

The VCS NLP add-on for VCS enables you to specify the UPF based power-intent of your design directly to VCS and generate a simulation model, which contains all power-objects directly instrumented in it.

VCS NLP equips VCS to natively perform voltage-level aware simulation with a complete understanding of the UPF-defined power network, including at RTL prior to implementation flows. This uniquely allows engineers to comprehensively verify correct behavior of designs that use advanced voltage control techniques for power management, and catch potentially design-killing low power bugs very early in the design process.

VCS NLP equips VCS to read this UPF, model the entire power network described in the UPF, and accurately understand the low power policies and voltage events. VCS in native low power mode generates a log file and an error and warnings report for all violations related to multi-voltage checks, as illustrated in [Figure 33-1](#).

*Figure 33-1 VCS NLP Low Power Simulation Flow*



For more details about VCS NLP add-on for VCS, refer to the *VCS Native Low Power (NLP) User Guide*.

# 34

## Unified UVM Library for VCS and Verdi

---

The unified UVM library integrates the instrumented UVM libraries available in VCS and Verdi. With the introduction of the unified UVM library, VCS and Verdi transaction recorder and message catcher coexist and are compiled together. You can directly use the Unified UVM library with Verdi recording mechanism during simulation and for debugging with Verdi. You do not need to set `VCS_UVM_HOME` pointing to `VERDI_HOME` for Verdi transactions.

Single compilation, UUM and UVM-VMM interoperability flows are supported in the unified UVM library. The unified UVM library can also be qualified and validated using Synopsys VIPs.

The UVM libraries are available in the following location:

- `$VCS_HOME/etc/uvm-1.1`: The unified UVM 1.1d library.
- `$VCS_HOME/etc/uvm-1.2`: The unified UVM 1.2 library.

- `$VCS_HOME/etc/uvm`: This path is symbolically linked to the `$VCS_HOME/etc/uvm-1.1` directory.
- 

## Transaction/Message Recording in Verdi with VCS

The following sections describe how to use the unified UVM library with Verdi transaction recorder and message catcher for the VCS simulator:

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

### Compilation

The compile flow is same for any GUI transaction recording. It is not required to point `VCS_UVM_HOME` to UVM library in Verdi installation and recompile.

### Enabling FSDB Transaction Recording

Unified UVM library shipped with VCS has additional features that allow you to take advantage of FSDB transaction recording and Verdi transaction debugging capabilities.

For FSDB transaction recording, set `VERDI_HOME` and `VCS_HOME` environment variables. With the `-debug_access+r` option, and NOVAS tab and PLI files, the Verdi transaction recorder is compiled.

You can enable the FSDB dumper using the following command:

```
% vcs -sverilog -ntb_opts uvm -debug_access
```

## **Recommended Use Model for FSDB Transaction Dumping**

To enable FSDB transaction recorder with unified UVM library, it is recommended to use the `-debug_access` option, as follows:

```
% vcs -sverilog -debug_access+all -ntb_opts uvm-1.1  
[compile_options]
```

Note:

- You must use the `-ntb_opts uvm-1.2` option for UVM-1.2 code.
- You must set `VERDI_HOME` and `VCS_HOME` environment variables.

To compile your UVM-1.1d or UVM-1.2 code, no extra compile-time option is needed. VCS transaction recorder, Verdi settings and recorder, `novas.tab` and `pli.a` files of FSDB dumper are automatically included.

---

## **Simulation**

The following section describe how to perform simulation using the unified UVM library.

## **Dumping Transactions or Messages in Verdi Flow**

Add the following runtime options to enable Verdi transaction recorder and message catcher:

- `+UVM_VERDI_TRACE=<Argument>`

Enables the Verdi flow when added during simulation.

You can use any of the following values as an input to the <Argument> parameter:

UVM\_AWARE | RAL | TLM | MSG | HIER | PRINT

For more details, see the Verdi Application Note -  
*V3\_new\_transaction\_debug\_platform.doc*

- +UVM\_TR\_RECORD  
Enables Verdi transaction recorder.
- +UVM\_LOG\_RECORD  
Enables Verdi message catcher.

Note:

If you do not use +UVM\_VERDI\_TRACE in the simv command line, transactions get dumped in the VPD file.

For example,

```
%> ./simv +UVM_VERDI_TRACE +UVM_TR_RECORD \
+UVM_LOG_RECORD
```

# 35

## Debugging with Verdi

---

This chapter describes debugging your designs with Verdi in the following sections:

- [Introduction](#)
- [Generating Verdi KDB](#)
- [Dumping FSDB File for Various Flows](#)
- [Interactive and Post-Processing Debug](#)

---

## Introduction

In a complex design, it is difficult to find the bugs associated with the logic at the HDL or testbench level. The process of debugging includes locating the design logic that is associated with an error, isolating the cause, and understanding how the design is supposed to behave and why it is not behaving that way. Design and verification engineers need sophisticated tools to find errors in the data produced by the simulator.

Verdi debug tool enables comprehensive debug for all design and verification flows. It includes powerful technology that helps you to comprehend complex and unfamiliar design behavior, and automate difficult and tedious debug processes.

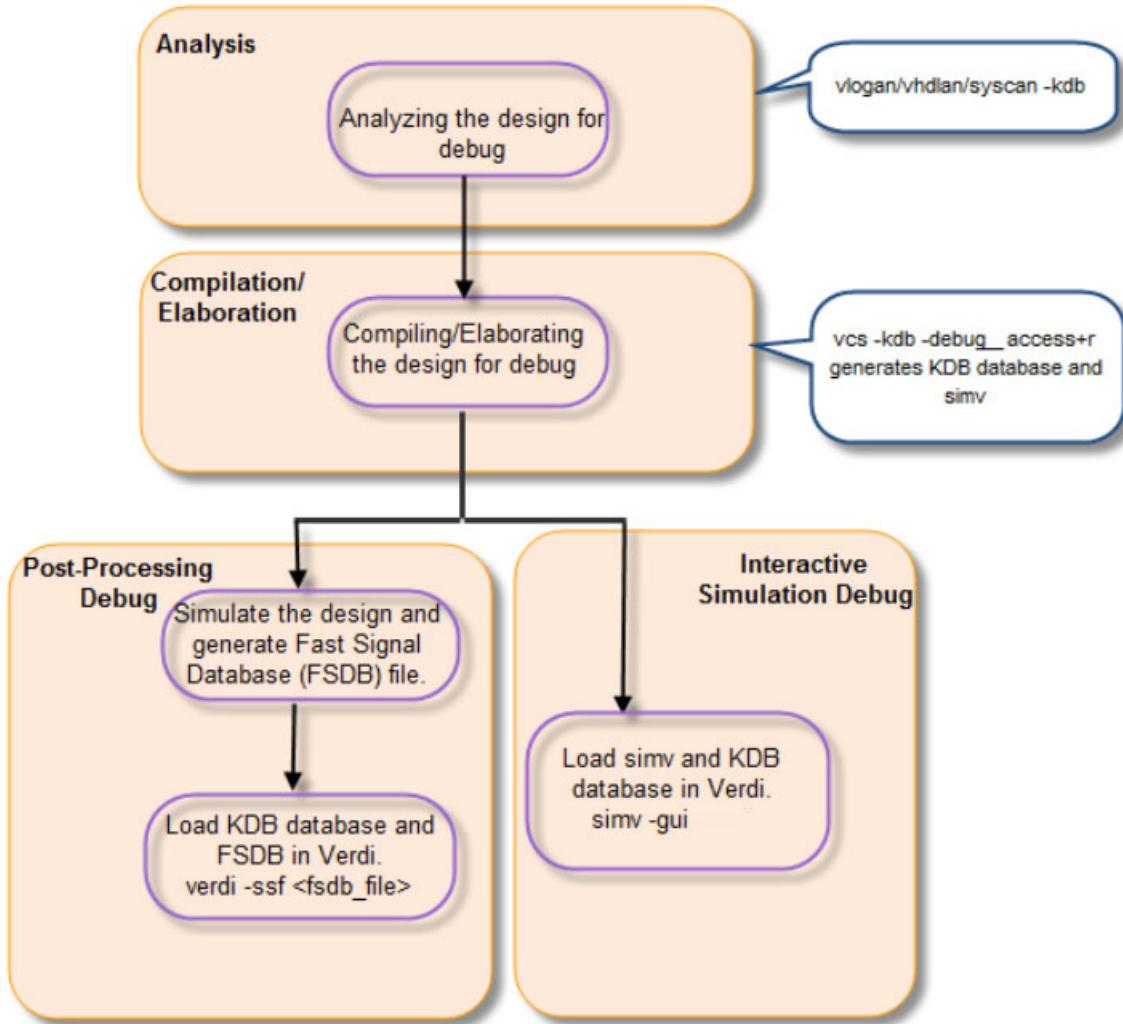
In addition to the standard features of a source code browser, schematics, waveforms, state machine diagrams, and waveform comparison, the Verdi tool includes advanced features for automatic tracing of signal activity using temporal flow views, assertion-based debug, and debug and analysis of transaction and message data.

You can debug your simulation using Verdi in the following modes:

- Interactive Simulation Debug Mode: Allows you to debug your design interactively.
- Post-Processing Debug Mode: Allows you to debug your design using a database.

The following flowchart shows the Verdi debug flow:

*Figure 35-1 Verdi Debug Flow*



---

## Generating Verdi KDB

Verdi Knowledge Database (KDB) is supported in both VCS two-step and three-step flows. In the VCS two-step flow, add the `-kdb` option to the command line to generate Verdi Knowledge Database (KDB). In case of VCS three-step flow, add the `-kdb` option in all the `vlogan/vhdlan/vcs` command lines.

When you specify the `-kdb` option, Verdi creates the KDB and dumps the design into the libraries specified in the `synopsys_sim.setup` file.

For example,

```
// Compile the design using VCS and generate both VCS  
database and Verdi KDB //
```

```
// -kdb in VCS two-step flow
```

```
%> vcs -kdb <compile_options> <source files>
```

```
// -kdb in VCS three-step flow
```

```
% vlogan -kdb <vlogan_options> <source files>
```

```
% vhdlan -kdb <vhdlan_options> <source files>
```

```
% vcs -kdb <top_name>
```

To generate only the Verdi KDB and skip the simulation database generation, specify the following argument with the `-kdb` option:

`-kdb=only`

Generates only the Verdi KDB that is required for both post-process and interactive simulation debug with Verdi.

This option is supported only in VCS two-step flow. It is not supported in VCS three-step flow.

In VCS two-step flow, this option does not generate the VCS compile data/executable, and does not disturb the existing VCS compile data/executables.

For example,

```
% vcs -kdb=only <compile_options> <source files>
```

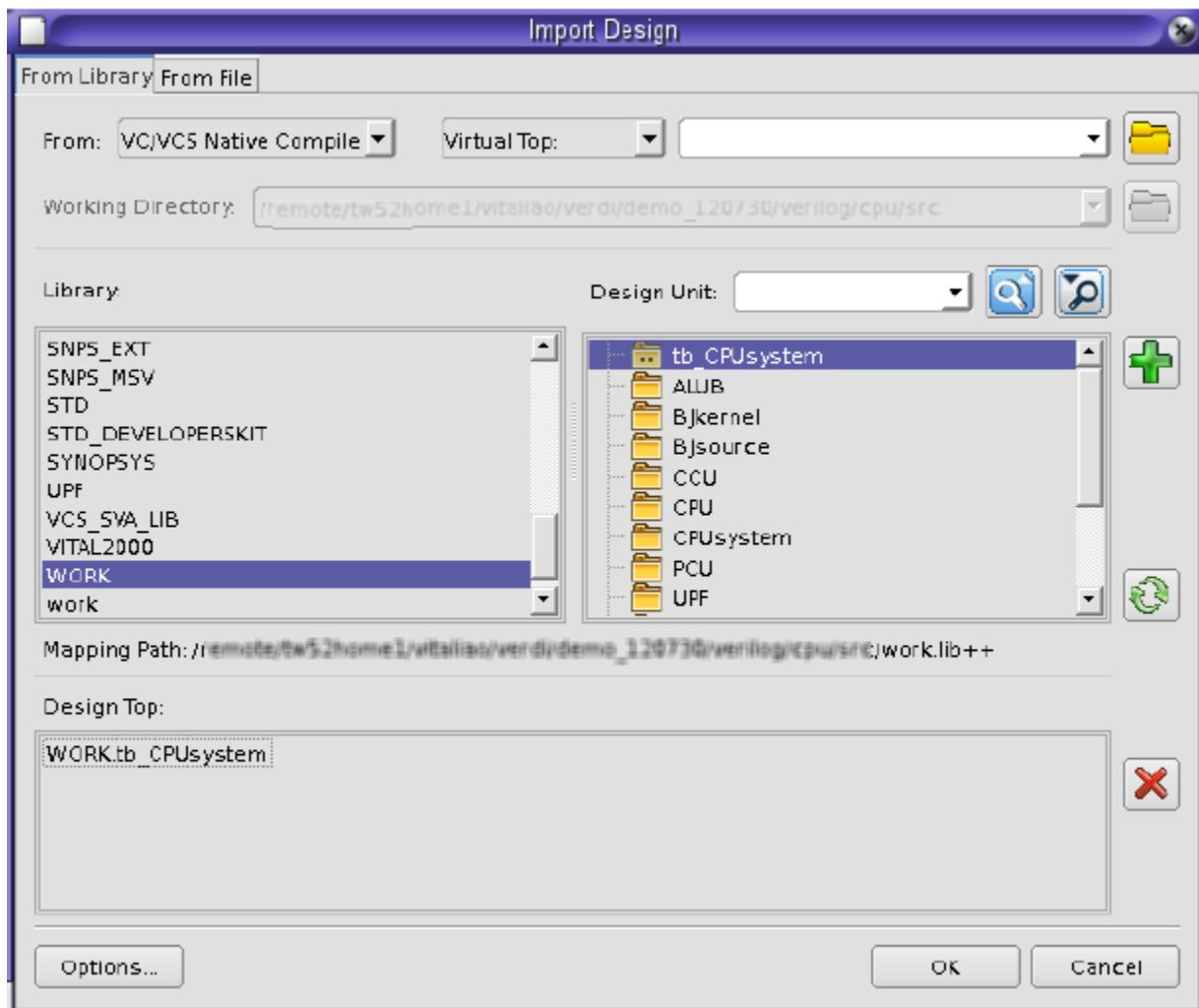
## Reading Compiled Design with Verdi

To read a compiled design, add the `-simflow` option to the Verdi command line. This imports the KDB and enables Verdi and its utilities to use the library mapping from the `synopsys_sim.setup` file. It is also used to import the design from the KDB library paths.

You can perform the same operations through the Verdi GUI as follows:

1. Click **File > Import Design** option.
2. In the **Import Design** form, select the **From Library** tab.
3. In the **From** field, select the **VC/VCS Native Compile** option, as shown in [Figure 35-2](#).

Figure 35-2 Import Design Form



You can also add the `-dbmdir <path>` option to the Verdi command line to ensure that VCS and Verdi use the same data from the `synopsys_sim.setup` file. The `<path>` argument points to the library directory from where VCS is compiled. Use this option if you want to invoke Verdi from a working directory that is different from the VCS working directory.

You can also use the `-top` option with the `-dbmdir` option to specify the top module in the specified library directory. For example,

```
%> verdi -simflow -dbdir [<path>] -top [<top module>]
```

If the `-top` option is not specified, the design top is used by default.

## Example

Consider the following testcase, `top.v`:

```
module top;
    reg a,b;
    wire c;
    reg [0:1] mem1 [0:10];
    dut d1(a,b,c);
    initial begin
        a=0; b=0;
        #1 a=0; b=1;
        #1 a=1; b=0;
        #1 a=1; b=1;
        #1 $finish;
    end
endmodule

module dut(input a,b,output c);
    assign c=a&b;
    count cnt(a,b,c);
endmodule

module count(input a,b,output c);
    reg d;
    initial $monitor("A=%0d,B=%0d,C=%d,D=%0d",a,b,c,d);
endmodule
```

## Compiling Designs

VCS Two-step Flow Compilation:

```
% vcs -debug_access+r -sverilog -kdb top.v
```

VCS Three-step Flow Compilation:

```
% vlogan top.v -kdb
```

```
% vcs top -debug_access+r -kdb
```

## Setting Up Verdi

To dump an FSDB file, you must set the following environment variables:

```
% setenv VERDI_HOME verdi_installation  
% setenv LM_LICENSE_FILE[verdi_license]:$LM_LICENSE_FILE
```

## Import KDB

```
% verdi -simflow -dbdir <path> -top top -nologo &
```

## Invoke Verdi in Interactive Mode

```
% simv -gui
```

## Invoke Verdi in Post-Processing Mode

```
% verdi -ssf novas.fsdb -nologo
```

Note:

Invoking Verdi in interactive and post-processing modes loads KDB automatically.

## Key Points to Note

- The `vericom` utility exists in Verdi. For VCS users, it is recommended to use the `-kdb` option at compile time to generate the KDB database for data consistency and better performance. For third-party simulator users, the compile flow does not change and continues to use `vericom`. When loading the compiled design library (KDB) from the GUI (loading from the command line stays the same), ensure that the `vericom` option is selected in the **From** field under the **From Library** tab of the *Import Design* form.

- As VCS and `vericom` are different Verilog compilers, there are some behavioral differences between them. In such cases, VCS generated KDB follows the behavior of VCS for consistency reasons. The supported language subset also follows the supported subset of VCS.
- All the compilation information including compile log of Verdi KDB is logged to the regular VCS compiler log file.
- The library mapping information is obtained from the `synopsys_sim.setup` file in VCS three-step flow. The library mapping information in the `novas.rc` resource file is ignored in the VCS three-step flow.
- VCS generated KDB does not apply to the import-from-file flow of Verdi. The import-from-file flow continues to use the `vericom` parser to read in the Verilog source code directly. It uses the library mapping information from the `novas.rc` resource file, which is similar to the Verdi behavior.
- In the VCS two-step flow, the VCS generated KDB (`kdb.elab++`) is saved under the `simv.daidir/` directory (like `simv.daidir/kdb.elab++`).
- In the VCS three-step flow, the `vlogan -work <work>` generated KDB (`<work>.lib++`) is saved in the same working directory as `AN.DB` and the physical directory path of the library is picked as per the mapping present in the `synopsys_sim.setup` file. You can use the `verdi -simflow -lib` option to specify the working directory to load the KDB.

## Limitations

The following are the limitations with VCS generated KDB:

- Parallel compilation is not supported.

- Fault tolerance compilation is not supported.

---

## Dumping FSDB File for Various Flows

This section describes the use model to set up Verdi and dump an FSDB file using VHDL procedures, Verilog system tasks, or UCLI in the following topics:

- [Setting Up Verdi](#)
- [Use Model for FSDB Dumping](#)
- [Examples](#)

---

### Setting Up Verdi

To dump an FSDB file, you must set the following environment variables:

```
% setenv VERDI_HOME verdi_installation  
% setenv LM_LICENSE_FILE[verdi_license]:$LM_LICENSE_FILE
```

---

### Use Model for FSDB Dumping

This topic describes the use model to dump an FSDB file using VHDL procedures, Verilog system tasks, or UCLI.

- Using VHDL Procedures

The following are the two ways to dump an FSDB file using VHDL procedures:

- You can use the VHDL procedures `fsdbDumpfile()` and `fsdbDumpvars()` in your VHDL code to dump an FSDB file.

**Note:**

To use these procedures, you should include `SYNOPSYS` library in your VHDL file as follows:

```
--Your VHDL file
library SYNOPSYS;
use SYNOPSYS.novas.all;

entity test is
...
end test;

architecture arch of test is
...
end arch;
```

- You can use the Verdi provided VHDL file: compile the Verdi provided VHDL file `$VERDI_HOME/$VERDI_LIB/novas.vhd` using the VCS analyzer and `vhdlan`, and save it in the same directory where the design is saved. The `novas.vhd` VHDL file contains the definitions of the FSDB foreign functions.

Use the `novas` package in any VHDL design file that invokes FSDB foreign functions.

### **Example:**

```
use work.novas.all; --using novas package.
entity testbench is end;
architecture blk testbench is Begin
...
Process begin: dump
  fsdbDumpvars(0, : , +fsdbfile+signal.fsdb ); -- call
  VHDL procedure wait;
end process end;
```

Then recompile the VHDL files you have modified.

- **Using Verilog System Tasks**

You can use the Verilog system tasks `$fsdbDumpfile()` and `$fsdbDumpvars()` in your Verilog design to dump an FSDB file (see “[Using VHDL Procedures or Verilog System Tasks](#)” ).

- **UCLI**

At a UCLI prompt, you can use either of the following commands to dump an FSDB file.

- Using the UCLI `dump` command.

```
dump [-file <filename>] [-type FSDB]
```

```
dump -add <list_of_nids> [-fid <fid>] [-depth <levels>] [-aggregates]
```

For complete use model of the UCLI `dump` command, see the *Unified Command Line Interface User Guide*.

- Using the UCLI commands `fsdbDumpfile` and `fsdbDumpvars`.

For information on the Verdi FSDB dumping commands for Verilog, see *Linking Novas Files with Simulators and Enabling FSDB Dumping Manual*. To access Verdi documentation, log in to the SolvNet, click the **Documentation** tab, and select **Verdi**.

You must use the `-debug_access` option to enable FSDB dumping.

# Using VHDL Procedures or Verilog System Tasks

## Analysis

```
% vlogan [vlogan_options] file1.v file2.v  
% vhdlan [vhdlan_options] file3.vhd file4.vhd
```

## Elaboration

```
% vcs -debug_access [elab_options] top_module/  
entity/cfg
```

For VHDL procedural flow, you must use the `-vhpi` option at runtime as follows:

```
% simv -vhpi novas:FSDBDumpCmd
```

## Simulation

```
% simv [run_options]
```

## Using UCLI

### Simulation

```
% simv [run_options] -ucli  
ucli> fsdbDumpfile your_fsdb_dumpfile  
ucli> fsdbDumpvars level module/entity
```

#### Note:

The default FSDB file name is `novas.fsdb`.

---

## Examples

### *Example 35-1 Using Verilog System Tasks*

This example demonstrates the use of Verilog system tasks, \$fsdbDumpfile and \$fsdbDumpvars.

```
'timescale 1ns/1ns
module test;
initial
begin
$fsdbDumpfile("test.fsdb");
$fsdbDumpvars(0,test);
end

...
endmodule
```

Now, the use model to compile/elaborate and simulate the above design is as follows:

### **VCS Two-step Flow**

Compilation:

```
% vcs -debug_access test.v
```

Simulation:

```
% simv
```

### **VCS Three-step Flow**

Analysis:

```
% vlogan test.v
```

**Elaboration:**

```
% vcs -debug_access test
```

**Simulation:**

```
% simv
```

The above set of commands dumps all the instances in `test` into the `test.fsdb` file.

### *Example 35-2 Using UCLI*

This example demonstrates the use of UCLI commands `fsdbDumpfile` and `fsdbDumpvars` at the UCLI prompt to dump an FSDB file:

Consider the following Verilog file:

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

The use model to compile/elaborate the design to use UCLI commands is as follows:

### **VCS Two-step Flow**

**Compilation:**

```
% vcs -debug_access test.v
```

**Simulation:**

```
% simv -ucli
```

| <b>Using UCLI dump command</b>                                                                                              | <b>Using UCLI Command Line Option</b>                                                              |
|-----------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <pre>ucli&gt; dump -file test.fsdb -type FSDB ucli&gt; dump -add test -depth 0 -file FSDB0 ucli&gt; run ucli&gt; quit</pre> | <pre>ucli&gt; fsdbDumpfile test.fsdb ucli&gt; fsdbDumpvars 0 test ucli&gt; run ucli&gt; quit</pre> |

## VCS Three-step Flow

Analysis:

```
% vlogan test.v
```

Elaboration:

```
% vcs -debug_access test
```

Simulation:

```
% simv -ucli
```

| <b>Using UCLI dump command</b>                                                                                              | <b>Using UCLI Command Line Option</b>                                                              |
|-----------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <pre>ucli&gt; dump -file test.fsdb -type FSDB ucli&gt; dump -add test -depth 0 -file FSDB0 ucli&gt; run ucli&gt; quit</pre> | <pre>ucli&gt; fsdbDumpfile test.fsdb ucli&gt; fsdbDumpvars 0 test ucli&gt; run ucli&gt; quit</pre> |

The above command dumps the whole design `test` into the `test.fsdb` file.

---

## Interactive and Post-Processing Debug

After the Verdi Knowledge Database (KDB) is generated, you can invoke Verdi with the KDB in a single step for the following debug modes respectively:

- Interactive Simulation Debug Mode

Verdi can be automatically invoked with the KDB through the simulator command line option to perform interactive simulation debugging without other configurations.

- Post-Processing Debug Mode

The KDB and the `synopsys_sim.setup` file information can be automatically loaded into Verdi through a command line option to perform post-processing debug. There is no need to manually specify the compiled design. VCS and Verdi will have the same information from the `synopsys_sim.setup` file.

For detailed information on using the Verdi platform, see *Verdi and Siloti Command Reference Manual*. To access Verdi documentation, log in to the SolvNet, click the **Documentation** tab, and select **Verdi**.

---

## Prerequisites

The following are the prerequisites to perform interactive simulation debug and post-process debug:

- Generate Verdi KDB

- Specify the `-debug_access+<option>` compile time option on the VCS command line. This option automatically picks Verdi tab file and Verdi PLI file, and there is no need to pass these files explicitly during compilation. For more information on the `-debug_access` option, see “[Optimizing Simulation Performance for Desired Debug Visibility With the `-debug\_access` Option](#)” section.
- Enable FSDB file dumping using the dumping tasks present in the source file or at runtime using `fsdbDumpvars` from the UCLI command line.

## Interactive Simulation Debug Flow

When executing the `simv` simulator executable, perform one of the following steps to invoke Verdi within the interactive simulation debug mode:

- Use the `-gui` option to invoke Verdi.

For example,

```
// invoke Verdi
%> simv <simv_options> -gui [-verdi_opts
"<verdi_options>"]
```

- Set the `SNPS_SIM_DEFAULT_GUI` environment variable to `verdi` to set the default debug tool as Verdi and the default dump type as FSDB.

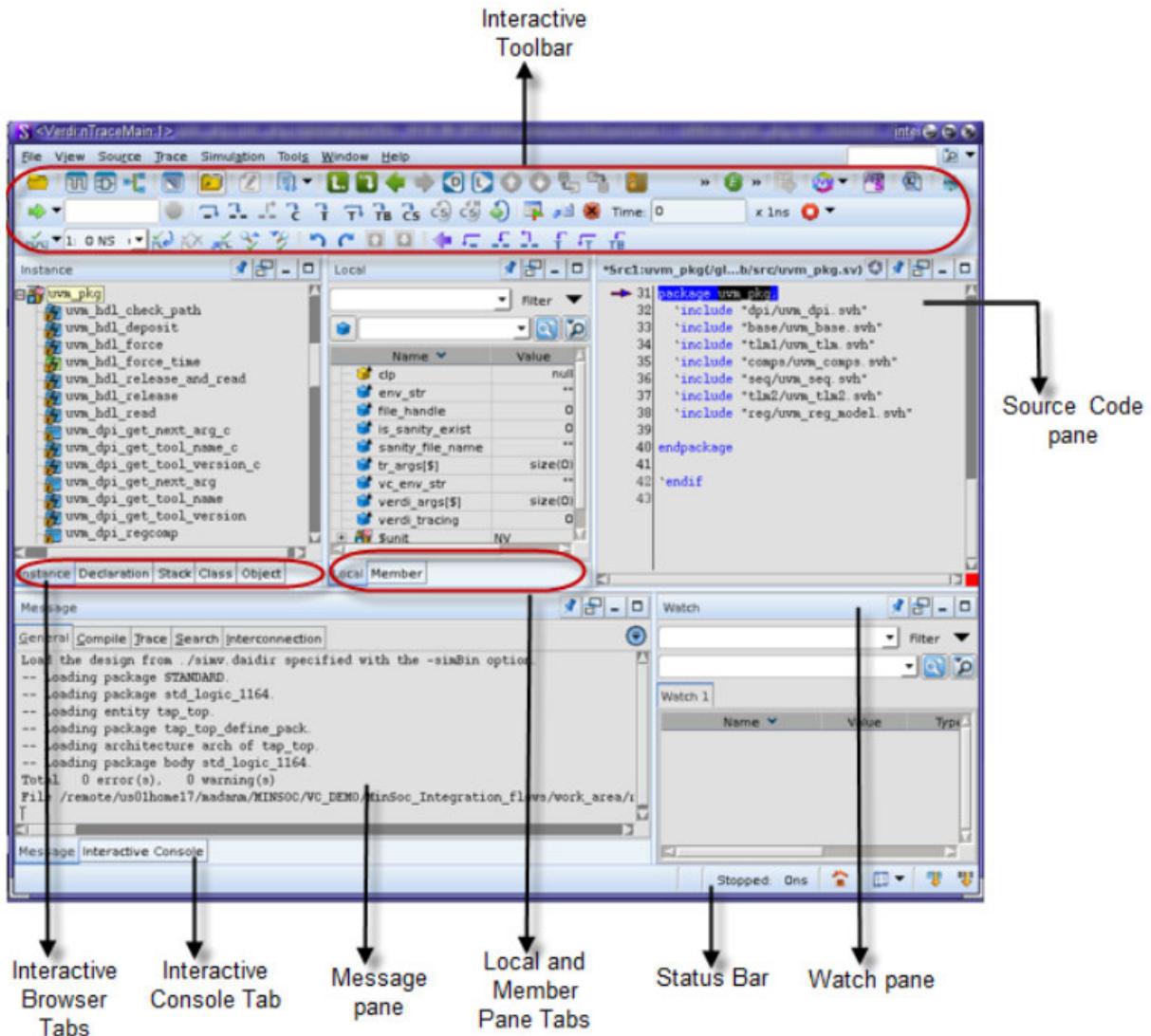
## Examples

```
//setting the default debug tool as Verdi and the default
dump type as FSDB

%> setenv SNPS_SIM_DEFAULT_GUI verdi
```

```
%> simv <simv_options> -gui [-verdi_opts
"<verdi_options>"]
```

*Figure 35-3 Verdi GUI*

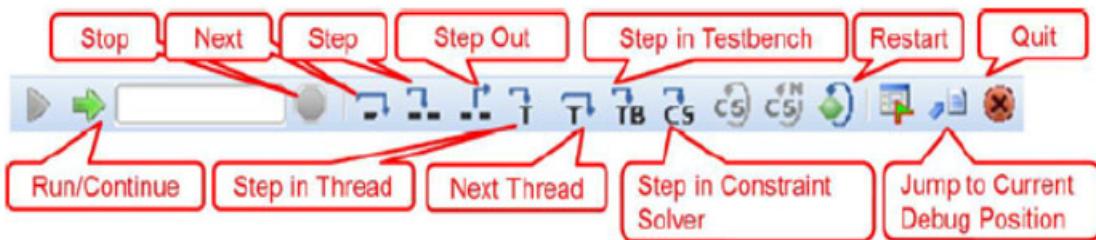


You can view the following panes in the Interactive Simulation Debug mode:

- Click *Instance* tab to view the *Instance pane*. This pane displays the static instance tree. The names in the instance tree are in the “instance name (definition name)” format. Top modules (or scopes) are at the top-level of the tree.
- Click *Stack* tab to view the *Stack pane*. This pane displays the testbench dynamic hierarchy tree along with all the testbench threads and their status.
- Click *Class* tab to view the *Class pane*. This pane displays all the classes defined in the design in a hierarchical view.
- Click *Object* tab to view the *Object pane*. This pane displays current dynamic objects and its values.
- Click *Local* tab to view the *Local pane*. This pane displays variables in a selected scope in the stack pane. This view is tied to the *Stack pane*, and the default view shows variables of the current active thread.
- Click *Member* tab to view the *Member pane*. This pane displays members and values of a class object selected in the *Object pane*.
- The *Interactive Console* frame provides the interface to input the Unified Command Line Interface (UCLI) commands and shows the results of simulator execution.
- The *Watch pane* allows you to monitor the status of your variables during simulation.
- The simulator status is shown in the *Status bar*.

## Simulation Control Commands

The frequently used simulation control commands are listed in the Interactive toolbar as shown in the following figure to direct the simulator how to execute the program.



You can advance the simulation to the statement where a function is called using the **Step** command (by clicking the icon or selecting the **Simulation > Step/Next > Step** command). The Local and Watch tabs are refreshed accordingly.

When the **Next** or **Step** simulation control commands are used, and the scope of the current simulation belongs to a testbench or defined as a class object, all variables of the current simulation scope are shown automatically in the Local tab.

The *Watch tab* is used to observe the current values of variables in the current simulation scope. Both the values and current scopes will be updated as simulation time changes.

Values can be annotated to the variables on the source code frame using the **Source > Active Annotation** command. The values of the variables will be displayed in the *Source Code* frame.

When you run a simulation interactively, the line where the simulation stopped is marked by the blue arrow in the *Source Code* frame.

You can set breakpoints to stop the simulation. Breakpoints can be set by single-clicking or double-clicking the line number in the **Source Code** frame. Breakpoints execute each time a specified line is reached during simulation. A green flag  indicates that a breakpoint is set. Use **Simulation > Manage Breakpoints** command to open the Manage Breakpoints dialog box. This dialog box allows you to edit the breakpoints with more options.

The **Run Simulation** icon  or **Simulation > Run/Continue** command starts the simulator. If the simulator is invoked and stopped at a preset breakpoint (a flag and arrow combination indicates the current breakpoint and simulator position), invoke this command again to continue the simulation run.

For more information, see *Verdi SVTB Interactive Simulation Debug User Guide* and *Verdi and Siloti Command Reference Manual*.

## Key Points to Note

- Use the `-verdi_opts` option to specify other Verdi specific options.
- The UVM Interactive Debug in Verdi is enabled by default while using the Unified Debug solution.
- If the design includes SystemC and the `default.ridb` is not available under the `simv.daidir/` directory, Verdi generates it automatically.
- In SystemC designs, for SystemC debug flow, you must create a `.ridb` file and set the `SNPS_VERDI_CBUG_LCA` environment variable.

---

## Post-Processing Debug Flow

To automatically load the KDB, use one of the following Verdi command line options:

- `verdi -ssf <fsdb_file>`

To use this command line option, compile your design with `-kdb` and generate FSDB.

- `verdi -simflow -dbdir <path> -top <top_name> <other_verdi_options>`

Where,

`-simflow`

Enables Verdi and its utilities to use the library mapping from the `synopsys_sim.setup` file and also import the design from the KDB library paths.

`-dbdir <path>`

Specifies the path of the library directory when you want to invoke Verdi from a working directory that is different from the VCS working directory. For more information, see “[Reading Compiled Design with Verdi](#)” .

## Reducing Disk Space for Post-Process Only Debug

You can use the `trim_daidir` script file to delete the files that are not required for the post-process debug flow. This feature helps you to significantly reduce the `simv.daidir` disk space.

## Use Model

Perform the following steps:

1. Compile the testcase. For example,

```
% vcs -sverilog -kdb -lca -debug_access+all file.sv
```

2. Run the simulation once and dump the FSDB file.

```
% simv <simv_options>
```

3. Use the `trim_daidir` script file to delete the files that are not required for the post-process debug flow.

Example: `trim_daidir simv.daidir`

4. Execute the following command to invoke Verdi:

```
% verdi -simflow -dbdir simv.daidir -ssf  
<fsdb_file>
```

# A

## VCS Environment Variables

---

This appendix covers the following topics:

- “[Simulation Environment Variables](#)”
- “[Setup Variables](#)”
- “[Optional Environment Variables](#)”
- “[Using Environment Variables in Verilog Source Code](#)”

---

### Simulation Environment Variables

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

`$VCS_HOME`

When you or someone at your site installed VCS, the installation created a directory that is called 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` or `SNPSLMD_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
```

or

```
setenv SNPSLMD_LICENSE_FILE 7182@serveroh:/u/net/  
server/eda_tools/license.dat
```

Note:

- You can use `SNPSLMD_LICENSE_FILE` environment variable to set licenses explicitly for Synopsys tools.

- If you set the `SNPSLMD_LICENSE_FILE` environment variable, then VCS ignores the `LM_LICENSE_FILE` environment variable.
- 

## Setup Variables

You can configure the compilation and simulation behavior of VCS by assigning values to setup variables in the `synopsys_sim.setup` file. The variable assignment statements have the following syntax:

```
variable_name = value
```

This section lists the setup variables that affect VCS simulation. In addition to these variables, the setup file can contain other variable assignments that apply to other Synopsys tools. VCS ignores setup variables related to other products, but generates a warning for the unrecognized variables.

The setup variables described in this section are organized into the following four parts:

- “[Analysis Setup Variables](#)”
- “[Compilation/Elaboration Setup Variables](#)”
- “[Simulation Setup Variables](#)”
- “[C Compilation and Linking Setup Variables](#)”
- “[Timescale Implementation](#)”

---

## Analysis Setup Variables

The setup variables that configure the analysis behavior of VCS are listed here in alphabetical order.

### `IGNORE_BINDING_HOMOGRAPHS`

Controls the generation of warning messages when encountering homographs while doing component binding. When set to `TRUE`, VCS suppresses all component binding homograph messages. The default value of `IGNORE_BINDING_HOMOGRAPHS` is `FALSE`.

### `LIBRARY_SCAN`

When set to `TRUE`, it checks and searches for a matching entity in all libraries defined in the `synopsys_sim.setup` file to resolve a component instantiation. If one is not found, an error message is issued. The default value of `LIBRARY_SCAN` is `FALSE`.

### `LICENSE_WAIT_TIME`

Enables license queuing and specifies the timeout time in minutes before `vhdlan` gives up waiting for a license.

The timeout time should be an integer greater than zero; any decimal part of the number is ignored.

With the `LICENSE_WAIT_TIME` variable in the setup file set to an integer, you do not have to specify the `-licwait` option. However, if you do specify the `-licwait` option, it overrides the setting in the setup file.

This variable affects analysis, compilation, and simulation steps. This variable is not set by default.

## OPTIMIZE

When set to TRUE, the VCS analyzer optimizes the compiled event code by eliminating VHDL checks for:

- Arithmetic overflow
- Constraint checks
- Array size compatibility at assignment
- Subscripts out of bounds
- Negative exponents to integer

The `-optimize` option to the `vhdlan` command overrides the `OPTIMIZE` value. The default value of `OPTIMIZE` is TRUE.

## Note:

If a VHDL error occurs when `OPTIMIZE` is TRUE, you may receive erroneous results or it can cause VCS to fail in an unpredictable way. If you have not completely debugged your design, it is recommended to temporarily set `OPTIMIZE` to FALSE.

## RELAX\_CONFORMANCE

When set to TRUE, the VCS analyzer relaxes any VITAL conformance violation error into a warning when analyzing VITAL models. The default value of `RELAX_CONFORMANCE` is FALSE.

## SPC

When set to TRUE, the VCS analyzer performs synthesis policy checking while analyzing VHDL design files. The analyzer checks the VHDL design files against the VHDL subset supported by Synopsys synthesis tools. The analyzer does not check for synthesis elaboration errors.

To make the synthesis policy checking work correctly, you must install the synthesis software correctly and the \$SYNOPSYS variable must point to your synthesis installation. The –spc option of the vhdlan command overrides the SPC value. The default value of SPC is FALSE.

#### `IEEE_1076_1987`

When set to TRUE, VHDL analyzer allows you to use VHDL-87 syntax. The default value of `IEEE_1076_1987` is FALSE.

#### `XLRM_TIME`

When set to TRUE, VCS (vlogan) relaxes timescale restriction, and issues a warning message when a module does not have timescale at analysis phase. For more information, refer to “[Timescale Implementation](#)” .

---

## Compilation/Elaboration Setup Variables

The following setup variables configure the compilation behavior of VCS.

#### `ERROR_WHEN_UNBOUND`

Set this variable to TRUE to change a warning message to an error message issued due to an unbound design unit. By default, VCS issues a warning message if there are any unbound design units.

#### `IGNORE_BINDING_HOMOGRAPHHS`

See “[IGNORE\\_BINDING\\_HOMOGRAPHHS](#)” on page 4 for more information.

#### `LIBRARY_SCAN`

See “[LIBRARY\\_SCAN](#)” on page 4 for more information.

#### LICENSE\_WAIT\_TIME

See “[LICENSE\\_WAIT\\_TIME](#)” on page 4 for more information.

#### NUM\_COMPILERS

Specifies the number of compilers used in parallel compilation. When PARALLEL\_COMPILE\_OFF is FALSE, NUM\_COMPILERS is set to 4. You can override the default value by specifying another integer value. If PARALLEL\_COMPILE\_OFF is TRUE, NUM\_COMPILERS is set to 1, that is, serial compilation. The default value of NUM\_COMPILERS is 4.

#### PARALLEL\_COMPILE\_OFF

Speeds up the compilation of generated C files by controlling the parallelism between code generation and compilation, and between compilation of different files.

When set to TRUE, elaboration step uses serial compilation instead of parallel compilation. The default value of PARALLEL\_COMPILE\_OFF is FALSE.

#### TIMEBASE

Specifies the basic unit of time used in simulating the design. All units of time used and understood by VCS are non-negative, whole-number multiples of the timebase unit. Valid TIMEBASE values are fs, ps, ns, us, ms, and sec.

The -time option to the vcs command overrides the TIMEBASE value. The default value of TIMEBASE is NS.

#### TIME\_RESOLUTION

Specifies the VCS time resolution. It basically sets the precision or the number of simulation ticks per base time unit.

```
TIME_RESOLUTION = [1 | 10 | 100] [fs | ps | ns |  
us | ms | sec]
```

If no numeric value (1, 10, or 100) is provided, then the default value is 1. For example:

```
TIME_RESOLUTION = ps
```

If a value beside 1, 10, or 100 is provided, a warning is issued during elaboration and a default setting of 1 <unit> is used (where unit is the specified time unit (fs, ps, etc.)).

Time resolution value cannot be higher than the time base value. An error is issued if this happens.

- The `-time_resolution` option to the `vcs` command overrides the `TIME_RESOLUTION` value. The default value is `TIME_RESOLUTION = 1NS`.

#### ELAB\_EXPAND\_ENV

When set to TRUE, this environment variable supports the expansion of UNIX environment variable, which is used with VHDL string generic.

#### Example

```
% cat test.v  
  
module memory_module (input data);  
    parameter memoryfile = "";  
    initial  
        $display(" memoryfile is = %s " ,memoryfile);  
endmodule
```

```
% cat test.vhd

library IEEE;
use IEEE.std_logic_1164.all;

entity top is
    generic (memoryfile : string := "$MEMORYFILE");
end entity;

architecture arch of top is
component memory_module is
    generic (memoryfile : string );
    port (data : in std_logic);
end component;

signal data : std_logic;

begin
    inst : memory_module generic map (memoryfile)
port map (data);
end architecture;
```

The following steps describe the use model:

1. Set the value of environment variable ELAB\_EXPAND\_ENV to TRUE in synopsys\_sim.setup file, along with other library mappings or environment variables.

```
ELAB_EXPAND_ENV = TRUE
```

2. Set the UNIX environment variable which is used in VHDL file test.vhd, as shown below:

```
setenv MEMORYFILE memory.txt
```

3. Run the design

```
vlogan test.v
vhdlan test.vhd
```

```
vcs top  
simv
```

The following is the output from Verilog file:

```
memoryfile is = memory.txt
```

**Note:**

As per Verilog LRM, you cannot change the value of parameter from one value to another, after elaboration or compilation.

For example:

*First Run:*

1. Set generic to value
2. vcs
3. simv

*Second Run:*

1. Set generic to some other value
2. simv

Therefore, you must set the value of this environment variable before the elaboration of the design, that is, before vcs.

## **Limitations**

The following are the limitations of the ELAB\_EXPAND\_ENV environment variable :

- This variable supports only string generic. It does not support variables or constants.
- This variable supports only unconstrained generics. This variable is not supported if the generic `memoryfile` in the above example is declared as follows:

```
generic (memoryfile : string(1 to 11) :=  
        "$MEMORYFILE");  
.  
.  
component memory_module is  
    generic (memoryfile : string(1 to 11) );  
    port (data : in std_logic);  
end component;
```

## Simulation Setup Variables

The following setup variables configure the simulation behavior of VCS.

### `ASSERT_IGNORE`

Controls the generation of messages in response to VHDL assertion violations or report statements. The possible values for this variable are `NOTE`, `WARNING`, `ERROR`, `FAILURE`, `NOIGNORE`, or `NOTSET`.

`ASSERT_IGNORE` has higher precedence than the individual assertion variable settings. If `ASSERT_IGNORE` equals `NOTSET`, simulation proceeds to check the values of the individual assertion variable settings, `ASSERT_IGNORE_NOTE`, `ASSERT_IGNORE_WARNING`, `ASSERT_IGNORE_ERROR`, and `ASSERT_IGNORE_FAILURE`. If `ASSERT_IGNORE` is set to any other value, the individual assertion variable settings are ignored.

If `ASSERT_IGNORE` equals `NOIGNORE`, the simulation prints messages for all assertion violations. The other values prevent simulation from printing a message unless the assertion violation is of greater severity than the value specified.

`ASSERT_IGNORE` has higher precedence than `ASSERT_STOP`. This means that when `ASSERT_IGNORE` is set, the simulator does not stop on `ASSERT_STOP` assertions. The default value of `ASSERT_IGNORE` is `NOTSET`.

#### `ASSERT_IGNORE_NOTE`

Controls the generation of messages in response to VHDL assertion violations of severity `NOTE`. If set to `TRUE`, all assertions of severity `NOTE` are ignored. VHDL assertions of severity other than `NOTE` are not affected by this variable.

`ASSERT_IGNORE` has higher precedence than `ASSERT_IGNORE_NOTE`. If `ASSERT_IGNORE` is set to any value other than `NOTSET`, the value of `ASSERT_IGNORE_NOTE` is ignored. The default value of `ASSERT_IGNORE_NOTE` is `FALSE`.

#### `ASSERT_IGNORE_WARNING`

Controls the generation of messages in response to VHDL assertion violations of severity `WARNING`. If set to `TRUE`, all assertions of severity `WARNING` are ignored. VHDL assertions of severity other than `WARNING` are not affected by this variable.

`ASSERT_IGNORE` has higher precedence than `ASSERT_IGNORE_WARNING`. If `ASSERT_IGNORE` is set to any value other than `NOTSET`, the value of `ASSERT_IGNORE_WARNING` is ignored. The default value of `ASSERT_IGNORE_WARNING` is `FALSE`.

#### `ASSERT_IGNORE_ERROR`

Controls the generation of messages in response to VHDL assertion violations of severity ERROR. If set to TRUE, all assertions of severity ERROR are ignored. VHDL assertions of severity other than ERROR are not affected by this variable.

`ASSERT_IGNORE` has higher precedence than `ASSERT_IGNORE_ERROR`. If `ASSERT_IGNORE` is set to any value other than NOTSET, the value of `ASSERT_IGNORE_ERROR` is ignored. The default value of `ASSERT_IGNORE_ERROR` is FALSE.

#### `ASSERT_IGNORE_FAILURE`

Controls the generation of messages in response to VHDL assertion violations of severity FAILURE. If set to TRUE, all assertions of severity FAILURE are ignored. VHDL assertions of severity other than FAILURE are not affected by this variable.

`ASSERT_IGNORE` has higher precedence than `ASSERT_IGNORE_FAILURE`. If `ASSERT_IGNORE` is set to any value other than NOTSET, the value of `ASSERT_IGNORE_FAILURE` is ignored. The default value of `ASSERT_IGNORE_FAILURE` is FALSE.

#### `ASSERT_IGNORE_OPTIMIZED_LIBS`

Defines the maximum severity level of an assertion to be ignored in the built-in packages during simulation. For global scope, the value of `ASSERT_IGNORE` is used. For built-in simulation packages, the value of the higher severity level between `ASSERT_IGNORE` and `ASSERT_IGNORE_OPTIMIZED_LIBS` takes precedence. These built-in packages include all the Synopsys and IEEE packages included with VCS.

**Valid values for this variable are ERROR, NOTE, WARNING, FAILURE, or NOIGNORE. The default value of ASSERT\_IGNORE\_OPTIMIZED\_LIBS is WARNING.**

#### ASSERT\_STOP

**Determines whether simulation stops in response to VHDL assertion violations. The possible values for this variable are NOTE, WARNING, ERROR, FAILURE, or NOSTOP.**

**If ASSERT\_STOP equals NOSTOP, simulation never stops for assertion violations. The other values cause simulation to stop when it encounters assertion violations of severity equal to, or greater than, the value specified. The default value of ASSERT\_STOP is ERROR.**

#### CS\_ASSERT\_STOP\_NEXT\_WAIT

**Controls the response of the compiled-code simulation mode to VHDL ASSERT statements. If set to TRUE, a failed VHDL assertion causes VCS to continue until the next WAIT statement, then stop. If not set, or set to FALSE, VCS prompts you to choose whether to stop immediately or to continue until the next WAIT statement.**

**For example:**

```
Assertion ERROR at 30 NS in design unit E(A) from process /E/_P0: "Assertion violation."  
An ASSERT STOP is currently pending in compiled code, and CS_ASSERT_STOP_NEXT_WAIT is not set to TRUE in synopsys_sim.setup.  
Continue until next wait (y), or stop simulation immediately (n)? [y/n]:
```

**If you choose to stop at the next WAIT statement, you can then continue the simulation by executing the VCS run command.**

If you choose to stop immediately, you cannot continue the current simulation. You must either restart the simulation with the VCS `restart` command or quit VCS and start it again.

The `CS_ASSERT_STOP_NEXT_WAIT` has no effect on debug mode simulations. The default value of `CS_ASSERT_STOP_NEXT_WAIT` is TRUE.

#### `CS_ASSERT_STOP_PROMPT`

If set to TRUE when running batch mode simulation, this variable causes the simulation to stop immediately without the possibility of continuing if an assertion of severity equal or higher than `ASSERT_STOP` occurs. The default value of `CS_ASSERT_STOP_PROMPT` is FALSE.

#### `LICENSE_WAIT_TIME`

See “[LICENSE\\_WAIT\\_TIME](#)” for more information.

#### `MAX_DELTA`

Specifies the maximum number of delta cycles in a simulation time step. When `MAX_DELTA` is set to a positive value, `simv` monitors the delta cycle number and stops the simulation when it reaches the `MAX_DELTA` limit. `simv` then issues a warning and prints a list of signals with pending zero-delay transactions. Additionally, `simv` may print a list of processes with pending wait for 0 timeouts. With that information, you can immediately start debugging possible infinite zero-delay cycles.

If you decide there is nothing wrong, you can disable delta cycle monitoring by setting `MAX_DELTA` to zero, or to a negative value. The default value of `MAX_DELTA` is 0.

## `MONITOR_TIME_DISPLAY`

If set to FALSE, the monitor command does not display time information. The default value of `MONITOR_TIME_DISPLAY` is TRUE.

## `USE`

Specifies the list of directories, separated by spaces, that VCS searches for VHDL source files. This information is used for viewing the VHDL source code of a design during a simulation.

The settings for the `USE` variable are not cumulative. For example, if there is a `synopsys_sim.setup` file in your home directory with `USE = ./ ./asic_lib`, and in your design directory, the `USE variable is set to USE = ./my_lib ./temp_lib, the final value for the USE variable is USE = ./my_lib ./temp_lib.`

The default value of `USE` is:

```
USE = . $VCS_HOME/packages/synopsys/src \
        $VCS_HOME/packages/IEEE/src \
        $VCS_HOME/packages/IEEE_asic/src \
        $VCS_HOME/packages/gtechnox/src \
        $VCS_HOME/packages/gtech/src \
        $VCS_HOME/packages/gscomp/src \
        $VCS_HOME/packages/dware/src \
        $VCS_HOME/dw/dw01/src \
        $VCS_HOME/dw/dw02/src \
        $VCS_HOME/dw/dw03/src \
        $VCS_HOME/dw/dw04/src \
        $VCS_HOME/dw/dw05/src \
        $VCS_HOME/dw/dw06/src \
        $VCS_HOME/dw/dw07/src \
        $VCS_HOME/dw/dw08/src
```

#### VCD\_IMMEDIATE\_FLUSH

When set to TRUE, every time you issue a new VCD dump command, the VCD file is immediately updated with the correct header and signal information. By default, all VCD file information is flushed when you exit VCS.

Setting this variable to TRUE may slow down the simulation performance when tracing design objects. The default value of VCD\_IMMEDIATE\_FLUSH is FALSE.

#### VCD\_OUTFILE

Specifies the output filename for the VCD file. To create a VCD file, use the dump command during simulation. The VCD file contains traced data that is used for post-processing. For example, you can set VCD\_OUTFILE = my\_vcd\_file.vcd.

#### VPD\_DELTA\_CAPTURE

Enables delta-cycle capturing in interactive simulation. The default value of VPD\_DELTA\_CAPTURE is OFF.

#### WAVEFORM\_UPDATE

When set to TRUE, objects in the Wave View are refreshed with every simulation time step. By default, the Wave View is refreshed when each simulation command is completed. Setting this variable to TRUE slows down the simulation performance when tracing design objects. The default value of WAVEFORM\_UPDATE is FALSE.

---

## C Compilation and Linking Setup Variables

Following are the setup variables that configure the C compilation of the C code that VCS generates.

`CS_CCFLAGS_$ARCH`

Specifies the C compiler flags used to compile the VCS generated C code on the specific platform.

One reason to use this variable is to specify a different compiler optimization level, such as `-O3`.

To get a listing of flags for your C compiler, use the `UNIX man` utility.

The `CS_CCFLAGS` variable is still supported and it has higher precedence than the platform specific `CS_CCFLAGS_$ARCH` variables.

The `-ccflags` option to the `vhdlan` and `vcs` commands overrides the `CS_CCFLAGS_$ARCH` value.

The default value of `CS_CCFLAGS_$ARCH` is different for each platform. Default values for SparcOS5, Linux, and RS6000 are as follows:

- SparcOS5

`CS_CCFLAGS_SPARCOS5 = -c -O`

- Linux

`CS_CCFLAGS_LINUX = -c -O`

- RS6000

```
CS_CCFLAGS_RS6000 = -c -qchars=signed -O -qmaxmem=2048000  
CS_CCPATH_$ARCH
```

**Specifies the C compiler used to compile VCS generated C code on the specific platform.**

The GCC compiler is incorporated in the VCS image for Sun SPARC operating systems (Solaris). This is the recommended compiler for the Solaris platform. VCS is optimized for performance with the GCC C compiler.

**Note:**

**CS\_CCPATH variable is supported and it has higher precedence than the platform specific CS\_CCPATH\_\$ARCH variables.**

**The -ccpath option to the vhdlan and vcs commands overrides the CS\_CCPATH\_\$ARCH value.**

**The default value of CS\_CCPATH\_\$ARCH is different for each platform. Default values for SparcOS5, Linux, and RS6000 are as follows:**

- SparcOS5

```
CS_CCPATH_SPARCOS5 = $VCS_HOME/sparcOS5/gcc/gcc-2.6.3/  
bin/gcc
```

- Linux

```
CS_CCPATH_LINUX = cc
```

- RS6000

```
CS_CCPATH_RS6000 = cc
```

**Note:**

It is your responsibility to set up the proper path for the C compiler on HPUX10, LINUX, and RS6000 platforms. This can be done in many different ways, for example:

- At tool's initial installation time, by editing the master `synopsys_sim.setup` file (from `/admin/setup`) and setting the proper C compiler path.
- For each user in their home directory, by having own `synopsys_sim.setup` file with proper C compile path.
- By setting the `PATH` environment variable to pick up the proper C compiler by default.

---

## Timescale Implementation

VCS supports the timescale implementation as defined in the IEEE 1800 standard. For information on timescale directives, see the *Verilog Language Reference Manual*.

This section describes the following topics:

- “[Understanding ‘timescale’](#) on page 21
- “[Verilog only and Verilog Top Mixed Design](#)” on page 25
- “[VHDL only and VHDL Top Mixed Designs](#)” on page 25
- “[Setting up Simulator Resolution From Command Line](#)” on page 26
- “[Other Useful Timescale Related Options](#)” on page 28
- “[Non-Compatible Options](#)” on page 30

## Understanding `timescale

In Verilog, all delays are governed by `timescale directive in the source file. The behavior is precisely defined in the *1364-1995 Verilog Language Reference Manual*. Now, there can be multiple `timescale compiler directives across multiple files. According to LRM:

*The `timescale compiler directive specifies the unit of measurement for time and delay values, and the degree accuracy for delays in all modules that follow this directive until another `timescale compiler directive is read.*

Consider the following three files:

| a.v                                           | b.v                                           | c.v                    |
|-----------------------------------------------|-----------------------------------------------|------------------------|
| `timescale 10ns/1ns<br>module a;<br>endmodule | `timescale 10ps/1ps<br>module b;<br>endmodule | module c;<br>endmodule |

You can see that the file c.v does not contain any timescale information, so it inherits the timescale from last encountered one during parsing.

### Scenario 1:

```
% vlogan a.v b.v c.v
```

In this case, a.v and b.v have their own timescale, so they follow it. But for c.v, the last encountered timescale is from b.v (10ps/1ps), so the simulator assigns the same to c.v.

## **Scenario 2:**

```
% vlogan a.v c.v b.v
```

In this case, *a.v* and *b.v* follow their own well-defined timescale. However, *c.v* inherits timescale from *a.v*, as it is the latest one as far as *c.v* is concerned.

## **Scenario 3:**

```
% vlogan c.v a.v b.v
```

In this case, it is not very clear which timescale *c.v* gets, as no timescale is parsed before *c.v*.

Situation becomes more complex when you go for mixed language simulation, involving both Verilog and VHDL.

Therefore, VCS came up with well-defined set of rules for all the above scenarios. This new implementation is under a variable defined in *synopsys\_sim.setup* file. The syntax for the same is as follows:

```
XLRM_TIME = TRUE  
TIMEBASE=time_base  
TIME_RESOLUTION=time_resolution
```

where,

```
time_number ::= 1 | 10 | 100  
time_unit ::= s[ec] | ms | us | ns | ps | fs  
time_base ::= time_unit  
time_resolution ::= time_number time_unit
```

If you specify only `XLRM_TIME=TRUE` without `TIME_RESOLUTION`, then it will be set to the value of `TIMEBASE`. There is a default `TIMEBASE` defined in default `synopsys_sim.setup` (from `$VCS_HOME/bin`).

It is recommended that the `time_unit` for `TIMEBASE` and `TIME_RESOLUTION` should be the same. If the `TIMEBASE` is finer than `TIME_RESOLUTION`, then it is an error condition. You can resolve this error condition by correcting the `TIMEBASE` entry in `synopsys_sim.setup`.

The following are the new terms which you will be using for rest of the section:

### **ana module:**

Verilog modules which get the timescale during the analysis phase (during vlogan time) is termed as “*ana module*”. Out of the three scenarios mentioned above, in Scenario 1 and Scenario 2, `module c` does not have its own timescale, but inherits it from other modules (`module b` in Scenario 1 and `module a` in Scenario 2) because of the parsing order. Since you know the timescale for all three modules now, all three modules are classified as “*ana modules*” in Scenario 1 and Scenario 2.

### **elab module:**

Verilog module which does not have any timescale after analysis phase is termed as “*elab module*”. In the above mentioned Scenario 3, module c neither has its own timescale nor has inherited from the previous modules, as there is none. Therefore, module c is treated as “*elab module*”, whereas module a and module b are treated as “*ana module*”. To make it clear, remove timescale from file b.v, hence it is rewritten as follows:

```
module b;  
endmodule
```

Consider the same command line again:

```
% vlogan c.v a.v b.v
```

In this case, c.v does not have any timescale (by its own or by inheritance), a.v has its own, and b.v gets the one from a.v by inheritance.

Hence, module c is treated as “*elab module*”, whereas module a and module b are treated as “*ana module*”.

During elaboration phase, VCS assigns timescale to all “*elab modules*”. All it does is to calculate simulator precision and use it as a timescale for all “*elab modules*”. This means

Timescale for all elab modules = time\_unit/  
time\_precision

## Verilog only and Verilog Top Mixed Design

For this topology of the design, simulator precision is determined by the finest of time resolution from all “*ana* modules”. If none of the Verilog modules in the design has timescale, then it is determined by `TIME_RESOLUTION` mentioned in the `synopsys_sim.setup` file.

VHDL world is also governed by `simulator_precision`. For example, reconsider Scenario 3. Also, consider the following `synopsys_sim.setup` file:

```
XLRM_TIME = TRUE  
TIMEBASE=fs  
TIME_RESOLUTION=1fs
```

Only module `a` and module `b` have timescales, and the finest resolution comes from module `b`, such as “1ps”. Hence it is treated as `simulator_precision`. Therefore, timescale assigned to module `c` is “1ps/1ps”. Note that `TIME_RESOLUTION` from the setup file is not considered here. Also, delays in VHDL files are rounded to resolution of “1ps” and not to “1fs” (from the `synopsys_sim.setup` file).

## VHDL only and VHDL Top Mixed Designs

In this case, `simulator_precision` is determined by `TIME_RESOLUTION` in `synopsys_sim.setup` file irrespective of the finest time precision from all “*ana* modules”. If the finest time precision from all “*ana* modules” is finer than `TIME_RESOLUTION` in

*synopsys\_sim.setup* file, then it is an error condition, and therefore VCS issues a proper error message. Consider the above given Verilog files (*a.v*, *b.v*, and *c.v*) and VHDL top given below:

```
library work;
use work.all;

entity top is
end top;

architecture top_arch of top is
component a is
end component;
component b is
end component;
component c is
end component;

begin
    U1:a;
    U2:b;
    U3:c;
end top_arch;
```

Now, `simulator_precision` is taken from *synopsys\_sim.setup file*, that is “1fs”, and timescale given to module `c` is “1fs/1fs” (and not “1ps/1ps” as in case of Verilog top design).

## Setting up Simulator Resolution From Command Line

You can set the simulator resolution from the command line irrespective of the design topology using the `-sim_res` option. The syntax of `-sim_res` is as follows:

```
-sim_res=<time_resolution>
```

Where,

```
time_resolution ::= time_number time_unit
time_number ::= 1 | 10 | 100
time_unit ::= s[ec] | ms | us | ns | ps | fs
```

This option supersedes the setting from *synopsys\_sim.setup* file (in case of VHDL top designs) or finest resolution from Verilog “*ana* modules” (in case of Verilog only or Verilog top designs).

Also, the same is used to construct the timescale for all *elab* modules.

For example, if you pass `-sim_res=1fs`, then the timescale for *elab* module is “1fs/1fs”. Also, the overall simulator resolution is “1fs”.

Note:

- With the current implementation of `XLRM_TIME`, if `-sim_res` is coarser than `TIME_RESOLUTION` in *synopsys\_sim.setup* (for VHDL top designs) or the finest time resolution from “*ana* modules” (for Verilog top designs), VCS issues an error message.
- For Verilog top designs, it is an error if the time resolution from the design is coarser than the time base from setup file.
- While using the `-sim_res` option, you must add the `XLRM_TIME=TRUE` in the *synopsys\_sim.setup* file. Otherwise, the `-sim_res` option does not have any effect during simulation.

## Other Useful Timescale Related Options

**-timescale=<time\_unit/time\_resolution>**

This is analysis time option. If present on the vlogan command line, it is applied to all files which have no timescale of their own, or not yet hit any timescale directive from other files during parsing order.

For example, consider following three files:

| a.v                                                | b.v                                                | c.v                            |
|----------------------------------------------------|----------------------------------------------------|--------------------------------|
| <pre>'timescale 10ns/1ns module a; endmodule</pre> | <pre>'timescale 10ps/1ps module b; endmodule</pre> | <pre>module c; endmodule</pre> |

The command line is:

```
% vlogan -timescale=1fs/1fs a.v b.v c.v
```

In this case, a.v and b.v have their own timescale and c.v inherits it from b.v, so *timescale* has no effect in this case. Alter c.v to add `resetall in it, as follows:

```
`resetall
module c;
endmodule
```

`resetall nullifies all compiler directives hit so far during parsing. Therefore, c.v instead of inheriting timescale from b.v, now takes it from the command line option. This is same as having the following command line:

```
% vlogan -timescale=1fs/1fs c.v a.v b.v
```

It is recommended to have the `-timescale` option accompanied with every `vlogan` command line to avoid any ambiguity at later stage.

#### **`-override_timescale=<time_unit/time_resolution>`**

If applied at the analysis time, this option overrides the timescale of all analyzed modules into the same work library from all previous analysis commands.

**Example:**

```
% vlogan -work lib1 a.v b.v
```

```
% vlogan -work lib1 -override_timescale=10fs/1fs  
c.v d.v
```

In this case, timescale from `a.v` and `b.v` are replaced with the one from `-override_timescale` (from the second `vlogan` command) and `c.v` and `d.v` are replaced with the one from `-override_timescale`. Hence, `-override_timescale` replaces timescale of all the modules that are analyzed so far into the work library `lib1`. Any other modules that are analyzed later into the work library `lib1` through other `vlogan` commands are not affected by this option, and have their own timescale, that is, derived from the command line.

If applied at the elaboration time, this option is applied to all the modules in the design, irrespective of how they are analyzed.

Also, simulator precision is determined by `time_resolution` part of `-override_timescale`. This supersedes the `-sim_res` option.

## Non-Compatible Options

Under this implementation, all older timescale related switches are ignored and appropriate warning is issued.

The following elaboration time options are ignored:

- `-t [ime]`
- `-time_res [olution]`
- `-timescale (At elab time)`

## Limitations

- SystemC designs are not supported
- Separate compile flow is not supported

---

## 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
```

`MATCH_ENUM_VECTOR`

A port map aspect associates signals or values with the formal ports of a block.

When the `MATCH_ENUM_VECTOR` variable is set to `TRUE`, VCS allows types to be mappable in port maps as long as the element type of the array is a matching enumeration type. However, this does not comply with *VHDL LRM*. By default, this variable is not set.

It is recommended that the code be fixed to make it LRM compliant.

#### PERSISTENT\_FLAG

When set to 1, VCS disables the checks enabled by the `persistent` specification in the tab file. It also disables similar checks that are enabled by the `-debug_access` option. See the section “[PLI Table File](#)” .

#### SYSTEMC\_OVERRIDE

Specifies the location of the SystemC simulator used with the VCS/SystemC co-simulation interface. See [Using SystemC](#).

#### TMPDIR

Specifies the directory used by VCS and the C compiler to store temporary files during compilation.

#### VCS\_CC

Indicates the C compiler to be used. To use the `gcc` compiler, specify the following:

```
setenv VCS_CC gcc
```

#### VCS\_COM

Specifies the path to the VCS compiler executable named `vcs1`, not the compile script. If you receive a patch for VCS, you might need to set this environment variable to specify the patch. This variable is used for solving problems that require patches from VCS and should not be set by default.

#### `VCS_LIC_EXPIRE_WARNING`

By default, VCS displays a warning message 30 days before a license expires. You can specify that this warning message begin fewer days before the license expires with this environment variable, for example:

```
VCS_LIC_EXPIRE_WARNING 5
```

To disable the warning, enter the 0 value:

```
VCS_LIC_EXPIRE_WARNING 0
```

#### `VCS_LOG`

Specifies the runtime log file name and location.

#### `VCS_TARGET_ARCH`

Enables elaboration and simulation in 64-bit mode architecture. When running 64-bit simulations, there is no need to use `-full64` option whenever this environment variable is set. For example,

```
% setenv VCS_TARGET_ARCH linux64
```

#### `VCS_NO_RT_STACK_TRACE`

Tells VCS not to return a stack trace when there is a fatal error, and instead dump a core file for debugging purposes.

#### `VCS_SWIFT_NOTES`

Enables the `printf` PCL command. PCL is the Processor Control Language that works with SWIFT microprocessor models. To enable it, set the value of this environment variable to 1.

#### VCS\_DIAGTOOL

Generates valgrind data for `vcs1`, if you set this environment variable as follows:

```
% setenv VCS_DIAGTOOL "valgrind --tool=memcheck"
```

Once you set this environment variable, any subsequent invocation of `vcs1` generates valgrind data.

#### OVERWRITE\_PREDECLARATION\_IN\_LIB

By default, if the same `vlogan` command encounters more than one implementation bearing the same module name, VCS generates the following error message:

```
Error-[MPD] Module previously declared
      The module was previously declared at:
      "child_ccf_.v", 1
      It is redeclared later at:
      "child_ccf_.v", 1: token is 'child_ccf'
      module child_ccf (clk );
                           ^
Please remove one of the declarations and compile again.
```

To get a similar message for the scenario where several `vlogan` commands write to the same logical library, you should add `OVERWRITE_PREDECLARATION_IN_LIB = TRUE` to the `synopsys_sim.setup` file.

Consider the following example:

```
%cat child_ccf.v
```

```

module child_ccf ();
    initial $display("from child_ccf.v" );
endmodule

%cat child_ccf_.v

module child_ccf ();
    initial $display("from child_ccf_.v");
endmodule

%cat synopsys_sim.setup

rtl_one_lib : rtl_one_lib
OVERWRITE_PREDECLARATION_IN_LIB = TRUE

```

**Run the following commands that parse two different files (one with an underscore before the file extension) to the same library:**

```
%vlogan -sverilog child_ccf.v -work rtl_one_lib
%vlogan -sverilog child_ccf_.v -work rtl_one_lib
```

**VCS generates the following warning message:**

```
Parsing design file 'child_ccf_.v'

Warning-[OPDL] Overwrite pre-declaration in library
  child_ccf_.v, 1
  child_ccf
  A previously declared module 'child_ccf' at
  "child_ccf.v,1" is found in library 'RTL_ONE_LIB'. It
  will be Overwritten by current module with the same name.
```

---

## Using Environment Variables in Verilog Source Code

To make the Verilog source code reusable, environment variables are often used in the source code instead of providing the complete file path.

For example:

```
module test();
initial
    bank_preload("$cwd/f.txt");
endmodule
```

The SystemVerilog LRM does not support using environment variables in Verilog source code. Therefore, by default, VCS does not support using environment variables in Verilog source code. You should either use the `-xlrn env_expand` option or use the `ELAB_EXPAND_ENV` variable in the `synopsys_sim.setup` file to use environment variables in the Verilog source code.

For example:

```
% vlogan t.sv -sverilog -xlrn env_expand
```

Content of `synopsys_sim.setup` file:

```
ELAB_EXPAND_ENV=TRUE
```



# B

## Analysis Utilities

---

This chapter describes the following utilities, which you can use during the VCS analysis process:

- “[The vhdlan Utility](#)”
- “[Using Smart Order](#)”
- “[The vlogan Utility](#)”
- “[The syscan Utility](#)” - For information about the `syscan` utility, see the *VCS SystemC User Guide*.

---

## The vhdlan Utility

The `vhdlan` utility analyzes VHDL source files and produces intermediate files for simulation. It checks for syntactic errors and if it finds any, generates error messages for them. The `vhdlan` utility uses the `synopsys_sim.setup` file to determine the logical-to-physical mapping of VHDL libraries.

### Syntax

```
vhdlan [vhdlan_options] <VHDL_filename_list>
```

where, *vhdlan\_options* are as follows:

`-help`

Prints usage information for `vhdlan`.

`-nc`

Suppresses the Synopsys copyright message.

`-q`

Suppresses compiler messages.

`-version`

Prints the version number of `vhdlan` and exits without running analysis.

`-4state`

Turns on Compact Data Representation (CDR) optimization. This option benefits designs that use std logic/ulogic vectors as 4state (for example, X, Z, 0, 1). Values other than X, Z, 0, 1 are reduced to the following:

- 'H' is converted to '1'
- 'L' is converted to '0'
- 'W' and '-' are converted to 'X'

If `-verbose` mode is specified, a warning will be issued about the values conversions performed if the information is statically visible in the design during analysis.

Performance benefits are seen because internally these values are represented in a compact form allowing for better data locality.

#### Note:

`-4state` optimizes the code and hence debugging is turned off under this mode.

#### `-work library`

Maps a design library name to the logical library name `WORK`, which receives the output of `vhdlan`. Mapping with the command-line option overrides any assignment of `WORK` to another library name in the setup file.

`library` can also be a physical path that corresponds to a logical library name defined in the setup file.

#### `-vhdl87`

Lets you analyze non-portable VHDL code that contains object names that are now, by default, VHDL-93 reserved words. VCS is VHDL-93 compliant.

#### `-vhdl02`

Lets you analyze the VHDL 2002 protected types. For more information, see chapter “[Support for VHDL 2002 and 2008](#)”

`-vhdl08`

Lets you analyze the VHDL 2008 constructs. For more information, see chapter “[Support for VHDL 2002 and 2008](#)” .

`-output outfile`

Redirects standard output from VCS analysis (that usually goes to the screen) to the file you specify as *outfile*.

`-list`

Creates a list file (`.lis`) containing the VHDL source code of the analyzed files, the names of the analyzed design units, and warning or error messages produced during analysis.

`-sva`

Enables SVAs inlined in the VHDL source code.

`-sv_opts "vlog_opts_to_SVAs"`

Specifies Verilog options for SVAs inlined in the VHDL source code.

`-optimize`

It improves simulation performance by generating optimized code, eliminating the following VHDL checks:

- Arithmetic overflow
- Constraint checks
- Array size compatibility at assignment
- Subscripts out of bounds
- Negative exponents to integer

This option overrides the value of the `OPTIMIZE` variable specified in the `synopsys_sim.setup` file. Use this option after you have successfully debugged the design and want to achieve better simulation performance. This option is on by default. The `-no_opt` option takes precedence over the `-optimize` option on the `vhdlan` command line.

`-no_opt`

Enables all VHDL language checks by canceling the effect of the `-optimize` option. Use this option while debugging the VHDL source files in your design.

The `-no_opt` option takes precedence over the `-optimize` option on the `vhdlan` command line.

`-ccpath path`

Specifies the C compiler that the Analyzer must use for compiling the code from VHDL to C. This option has already been set for the SPARC OS5 platform to use the C compiler included with this software. We recommend that you do not change this value. This option overrides the value of the `CS_CCPATH_<ARCH>` variable specified in the `synopsys_sim.setup` file.

`-ccflags "flags"`

Specifies the flags that `vhdlan` passes to the C compiler. The default flags are set in the `synopsys_sim.setup` file. This option overrides the value of the `CS_CCFLAGS_<ARCH>` variable specified in the `synopsys_sim.setup` file.

`-x1rm`

Enables VHDL features beyond those described in LRM.

**-f *optionsfile***

Specifies an *optionsfile* that expands the `vhdlan` command-line options.

**-functional\_vital**

Specifies generating code for functional VITAL simulation mode.

**-full64**

Enables compilation and simulation in 64-bit mode.

**-no\_functional\_vital**

Specifies generating code for full-timing VITAL simulation mode.

**-keep\_vital\_ifs**

Turns off some of the aggressive functional VITAL optimizations related to `if` statements in Level 0 VITAL cells.

**-keep\_vital\_path\_delay**

Preserves the calls to `VitalPathDelay`. Use this option if non-zero assignments to the outputs is required to preserve correct functionality.

**-keep\_vital\_wire\_delay**

Preserves the calls to `VitalWireDelay`. Use this option if delays on the inputs are required to preserve correct functionality.

**-keep\_vital\_signal\_delay**

Preserves the calls to `VitalSignalDelay`. Use this option if delays on signals are required to preserve correct functionality.

**-keep\_vital\_timing\_checks**

Preserves the timing checks within the VITAL cell.

**-keep\_vital\_primitives**

Preserves calls to VITAL primitive subprograms.

**-sva**

Enables SVAs inlined in your VHDL code.

**-sv\_opts "vlog\_opts\_to\_SVAs"**

Specifies Verilog options like timescale, +define+macro to SVAs inlined in your VHDL code.

For example:

```
% vhdlan -sva -sv_opts "+define+SVA1" file1.vhd
```

*VHDL\_filename\_list*

Specifies the VHDL source file names to be analyzed. If you do not provide an extension, .vhd is assumed.

Note:

The maximum identifier name length is 250 for package, package body, and configuration names. The combined length of an entity name plus architecture name must not exceed 250 characters as well. All other VHDL identifier names and string literals do not have a limitation.

---

## Using Smart Order

The `smart_order` option with `vhdlan`, allows you to automatically identify the file order dependencies internally and then do file-by-file analysis of all VHDL files passed to it. They are ordered as per the dependencies of design units contained within them.

Identifying the dependencies between design units, establishing an order for design files that contain them, and then running `vhdlan` to analyze these files is a difficult and time-consuming process in most cases.

According to *IEEE Standard VHDL 1076-2008 LRM*, Section 11.4, VHDL design units must be analyzed in the order of their dependency, that is, before analyzing a particular unit, its dependent unit must be analyzed. For example, if `unit1` is dependent on `unit2`, then `unit2` must be analyzed before analyzing `unit1`.

**Note:**

By default, design files that you input to `vhdlan` are analyzed in the order in which they are listed in the command line.

---

## Use Model

Order-independent analysis of VHDL files using the `smart_order` option:

Specify the `-smart_order` option in the `vhdlan` command line or set `SMART_ORDER=TRUE` in the `synopsys_sim.setup` file.

**Syntax:**

```
vhdlan -smart_order [vhdlan_options]
VHDL_filelist
```

**Example:**

```
vhdlan -smart_order -work lib bottom.vhd mid.vhd top.vhd
vhdlan -smart_order -work lib *.vhd
vhdlan -smart_order -work lib t*.vhd
vhdlan -smart_order -f flist
```

- Using the `smart_script` option along with `smart_order`:

When used along with the `-smart_order` option, the `-smart_script` option generates a re-analysis script, which is a complete `vhdlan` command line. The script includes an ordered file list and all options (except for the `-file` option since it is expanded and replaced) specified in the original `vhdlan` command line.

Specify `-smart_script` followed by a user-specified file name in the `vhdlan` command line. The `-smart_script` option must be used with the `-smart_order` option to generate the re-analysis script.

**Syntax:**

```
vhdlan -smart_order -smart_script script_name
[vhdlan_options] VHDL_filelist
```

**Example:**

```
vhdlan -smart_order -smart_script ana.sh -work
lib bottom.vhd mid.vhd top.vhd

vhdlan -smart_order -smart_script ana.sh -work
lib *.vhd
```

Note:

The ordered file list dumped by `smart_script` can be re-used directly with the `vhdlan` as the ordered file list, thereby avoiding the need to use `-smart_order -smart_script` often.

---

## Limitations

Following are the limitations of the `smart_order` option:

- You cannot resolve a design unit that was analyzed into one logical library, but referenced with another logical library prefix (these two libraries point to a same UNIX path) when using the `smart_order` option. For example:

```
%vhdlan -work lib1 leaf.vhd top.vhd  
leaf is referred in top as follows:
```

```
Library lib2;  
Use lib2.leaf;
```

- If there is no explicit configuration for a component instance, then this component instance must have a port map clause when it is defined.
- Identifying file order dependencies across different logical libraries is not supported.

Note:

- The primary design units (package, entity, and configuration) in the listed design files must have unique names, otherwise, `vhdlan` generates an error message and aborts sorting of the design files.

- For Mixed HDL Designs (Verilog + VHDL), you need to analyze all Verilog files that are instantiated in VHDL first, otherwise, `vhdlan` generates warning messages for unresolved references. This is a general flow for Mixed HDL designs, and is not specific when `smart_order` is used. The `smart_order` option does not identify Verilog dependencies.
- 

## The `vlogan` Utility

VCS uses the `vlogan` utility to analyze Verilog portions of a design instantiated within a VHDL design.

The syntax of the `vlogan` command line is as follows:

```
vlogan [vlogan_options] Verilog_source_filename
```

Here, the `vlogan_options` are:

`-help`

Displays a succinct description of the most commonly used compile-time and runtime options.

`-nc`

Suppresses the Synopsys copyright message.

`-q`

Suppresses compiler messages.

`-f filename`

Specifies a file that contains a list of path names to source files and required analysis options.

You can use Verilog comment characters such as `//` and `/* */` to comment out entries in the file.

Note that the following restrictions apply to the contents of this file:

- You can only specify the following analysis options that begin with a minus(-) character:

|                 |                 |                 |
|-----------------|-----------------|-----------------|
| <code>-f</code> | <code>-l</code> | <code>-y</code> |
| <code>-u</code> | <code>-v</code> |                 |

- You cannot specify escape characters and meta characters like `$`, `'`, and `! .`.

Note:

The maximum line length in the specified file *filename* should be less than 1024 characters. VCS truncates the line exceeding this limit, and issues a warning message.

`-full64`

Enables compilation and simulation in 64-bit mode.

`-ID`

Displays the hostid or dongle ID for your machine.

`-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.**

`-l filename`

**Specifies a log file where VCS records compilation messages and runtime messages if you include the `-R` option.**

`-liblist logical_lib1+logical_lib2+...logical_libn`

**It specifies the library search order for resolving imported package definitions. The `vlogan -liblist` option restricts the libraries in which `vlogan` should search for resolving package references found while analyzing.**

**If the `-liblist` option is not included, `vlogan` searches all logical libraries listed in the `synopsys_sim.setup` file.**

**When using multiple `vlogan` commands with same `-work` logical library, run the commands sequentially, and if one command uses `-liblist`, then ensure that all the remaining `vlogan` commands are using the same `-liblist` argument list, as shown in the following example:**

```
%vlogan a.v -work shared_lib -liblist  
shared_lib+ovm_lib+common_lib
```

```
%vlogan b.v -work shared_lib -liblist  
shared_lib+ovm_lib+common_lib
```

```
%vlogan c.v -work shared_lib -liblist
```

`shared_lib+ovm_lib+common_lib`

`-location`

Displays the location of the vlogan installation.

`-libmap filename`

Specifies a library mapping file.

`-notice`

Enables verbose diagnostic messages.

`-ntb`

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

`-ntb_define macro`

Specifies any OpenVera macro name on the command line. You can specify multiple macro names using the plus (+) character.

`-ntb_fileext .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_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. Enter this argument with the `dep_check` argument.

`rvm`

Use `rvm` when RVM or VMM is used in the testbench.

**Example:** `vlogan vmm_test.sv -sverilog -ntb_opts rvm`

For more information, refer to the “[Using VMM with VCS](#)” section.

`tb_timescale=value`

**Specifies an overriding timescale for the testbench, whenever the required testbench timescale is different from that of the design. It must be used in conjunction with the `-timescale` option that specifies the timescale for the design.**

If the required testbench timescale is different from the design or DUT timescale, then both the testbench timescale and the DUT timescale must be passed during VCS compilation.

**Example:**

The following command specifies a required testbench timescale of 10ns/10ps and a design timescale of 1ns/1ps:

```
%> vcs -ntb_opts tb_timescale=1ns/1ps  
-timescale=10/10ns file.sv
```

`tokens`

**Preprocesses the OpenVera files to generate two files, tokens.vr and tokens.vrp. The tokens.vr file contains the preprocessed result of the non-encrypted OpenVera files, while the tokens.vrp file contains the preprocessed result of the encrypted OpenVera files. If there is no encrypted OpenVera file, VCS sends all the OpenVera preprocessed results to the tokens.vr file.**

`use_sigprop`

**Enables the signal property access functions. For example, `vera_get_ifc_name()`.**

`vera_portname`

Specifies the following:

- The Vera shell module name is named `vera_shell`.
- The interface ports are named `ifc_signal`.
- Bind signals are named, for example, as: `\ifc_signal[3:0]`.

`-platform`

Returns the name of the platform directory in your VCS installation directory.

`-sv_pragma`

Analyzes SystemVerilog Assertions that follow the `sv_pragma` keyword in a single line or multi-line comment.

`-p1800_macro_expansion`

This option is used for LRM compliance to support macro expansion. The option produces result that is more LRM-compliant and accurate especially for SystemVerilog macros.

The syntax is:

```
% vlogan [vlogan_options] t.sv -sverilog  
-p1800_macro_expansion
```

For example, consider the following test case, `test.sv`:

```
module top;  
logic [3:0] addr0_for_bank0='d10;  
`define VAR(ANUM,BNUM) addr``ANUM``_for_bank``BNUM  
`define NAME(STR) $display(```\``"STR`\``" is %d\n``",STR);  
`define ARG addr0_for_bank0
```

```
initial begin
  `NAME(`VAR(0,0));
  `NAME(`ARG)
end
endmodule
```

If you run the test case without the `-p1800_macro_expansion` option, VCS generates the following output:

```
"`VAR(0,0)" is 10
"addr0_for_bank0" is 10
```

If you run the test case with the `-p1800_macro_expansion` option, VCS generates the following output:

```
"addr0_for_bank0" is 10
"addr0_for_bank0" is 10
-timescale=time_unit/time_precision
```

This option enables you to specify the timescale for the source files that do not contain `'timescale` compiler directive and precede the source files that do.

Do not include spaces when specifying the arguments to this option as shown in the following example:

```
% vlogan -timescale=1ns/1ns file1.v file2.v file3.v
-overrides_timeunit_timeprecision
Overrides the time unit and precision unit for all the 'timescale compiler directives in the source code and, like -timescale, provides a timescale for all module definitions that do not have a 'timescale compiler directive.
```

`-noinherit_timescale`

This is a VCS or vlogan option that allows you to specify a global timescale within the compilation unit of any source file that does not have an explicit `'timescale` directive.

If you specify the `-noinherit_timescale` option in the command line and there is no time unit/time precision specification present inside the module, program, interface, or package definition, then the time unit/time precision is determined using the following rules:

- If the module or interface definition is nested, then the time unit/time precision shall be inherited from the enclosing module or interface.
- Else, if `'timescale` directive is previously specified within the file, then the time unit/time precision shall be set to that of the `'timescale` directive.
- Else, if the time unit/time precision is specified outside all other declarations within a compilation unit, then the time unit/time precision shall be set to that specification.
- Else, the global time unit/time precision specified by the `-noinherit_timescale` option shall be used.

Similarly in presence of other timescale options specified in this document, the following is the precedence:

`-override_timescale > -noinherit_timescale > -timescale/-unit_timescale`

Therefore, when the combination of options are used, the behavior is as follows:

- The `-override_timescale` option has the highest precedence. Therefore, the timescale specified by the `-noinherit_timescale` option is overridden by the `-override_timescale` option.
- If the `-timescale` (or the `-unit_timescale`) and the `-noinherit_timescale` options are used in the same command line, timescale from the `-noinherit_timescale` option shall be used and `-timescale` (or `-unit_timescale`) option is ignored.
- In case of three-step flow, if the `-timescale` (or `-unit_timescale`) and the `-noinherit_timescale` options are used in different command line, such as one is in `vlogan` and another is in `vcs` command line, timescale specified at `vlogan` stage cannot be overridden at `vcs` stage (except for the case when you use the `-override_timescale` option).

**Example B-1 Example for `-noinherit_timescale` Option**

| a.v                                                         | b.v                                                                                               | c.v                                                                                            |
|-------------------------------------------------------------|---------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <pre>module A; ... endmodule  module B; ... endmodule</pre> | <pre>`timescale 1ns/1ns module X; ... endmodule  `timescale 1ns/1ns module Y; ... endmodule</pre> | <pre>module P; timeunit 100ps; timeprecision 1ps; ... endmodule  module Q; ... endmodule</pre> |

You can use the following `vcs` and `vlogan` command lines:

```
% vcs -noinherit_timescale="10ps/1ps" a.v b.v c.v
```

Or

```
% vcs -noinherit_timescale="10ps/1ps" b.v c.v a.v
```

Or

```
% vlogan -noinherit_timescale="10ps/1ps" c.v a.v c.v
```

Any of the above commands result into the following timescale:

- module A - 10ps/1ps
- module B - 10ps/1ps
- module P - 100ps/1ps
- module Q - 10ps/1ps
- module X - 1ns/1ns
- module Y - 1ns/1ns

### Key Points to Note

- The files that are read using a `include directive inherit the timescale of the file containing the `include directive. That is, the global timescale is only applied to files that are not read using an `include directive.
- Once the file is compiled or analyzed, the timescale of a module is not changed. This means that the timescale of a logic library is determined when the library is compiled, not when one of its module is incorporated using the -y option.

+delay\_mode\_path

Uses only the delay specifications in module path delays in specify blocks. Overrides all the delay specifications on all gates, switches, and continuous assignments to zero.

`+delay_mode_zero`

Removes delay specifications on all gates, switches, continuous assignments, and module paths.

`+delay_mode_unit`

Overrides all the delay specifications in module path delays in specify blocks to zero delays. Overrides 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`

Overrides all the delay specifications in module path delays in specify blocks to zero delays. Uses only the delay specifications on all gates, switches, and continuous assignments.

`-u`

Changes all characters in identifiers to uppercase.

`-v[t]`

Enables warning messages and displays the time used by each command.

`-v library_file`

Specifies a Verilog library file to search for module definitions.

`-y library_directory`

Specifies a Verilog library directory to search for module definitions. Use this option with `+libext+extension`. See below for the description of `+libext+extension`.

`-work VHDL_logical_library`

Specifies creating the VEROLOG directory and writing intermediate files in the physical directory associated with this logical library.

`+define+macro`

Defines a text macro. Test for this definition in your Verilog source code using the `'ifdef` compiler directive.

`+libext+extension+`

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

`+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.

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.

Additionally, some of the messages of these options 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

`+incdir+directory`

Specifies the directories that contain the files you specified with the `'include` compiler directive. You can specify more than one directory, separating each path name with the “`+`” character.

`+nowarnTFMPC`

Suppress the “Too few module port connections” warning messages during Verilog Compilation.

`-sverilog`

Enables the analysis of SystemVerilog source code.

`+systemverilogext+ext`

Specifies a file name extension for SystemVerilog source files. If you use a different file name extension for the SystemVerilog part of your source code and you use this option, the `-sverilog` option has to be omitted.

Note:

This compile-time option also works similar to the `-sverilog` option in which it enables SystemVerilog LRM (IEEE Std 1800-2012) rules for all the source files on the vcs command line and not only the files with the specified extension.

`+verilog2001ext+ext`

Specifies a file name extension for Verilog 2001 source files.

`+verilog1995ext+ext`

Specifies a file name extension for Verilog 1995 files. Using this option allows you to write Verilog 1995 code that would be invalid in Verilog 2001 or SystemVerilog code, such as using Verilog 2001 or SystemVerilog keywords, like `localparam` and `logic`, as names.

**Note:**

**Do not specify the `+systemverilogext+ext`,  
`+verilog2001ext+ext`, and `+verilog1995ext+ext`  
options on the same command line.**

**-extinclude**

If a source file for one version of Verilog contains the `'include` compiler directive, `vlogan` by default compiles the included file for the same version of Verilog, even if the included file has a different filename extension. If you want `vlogan` to compile the included file with the version specified by its extension, enter this option. The following code examples show using this option:

If source file `a.v` contains the following:

```
`include "b.sv"  
module a();  
reg ar;  
endmodule
```

and if source file `b.sv` contains the following:

```
module b();  
logic ar;  
endmodule
```

`vlogan` compiles `b.sv` for SystemVerilog with the following command line:

```
vlogan a.v +systemverilogext+.sv -extinclude
```

```
+warn=[no] ID|none|all
```

Uses warning message IDs to enable or disable display of warning messages. In the following warning message, the text string **TFIPC** is the message ID:

Warning-[TFIPC] Too few instance port connections

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 <b>no</b> and the ID.                    |
| none | Specifies disabling all warning messages. IDs that follow, in a comma-separated list, specifies exceptions.                                   |
| all  | Specifies enabling all warning messages. IDs that follow preceded by the keyword <b>no</b> , in a comma-separated list, specifies exceptions. |

The following are examples that show how to use this option:

|                       |                                                                     |
|-----------------------|---------------------------------------------------------------------|
| +warn=noIPDW          | Enables all warning messages except the warning with the IPDW ID.   |
| +warn=none, TFIPC     | Disables all warning messages except the warning with the TFIPC ID. |
| +warn=noIPDW, noTFIPC | Disables the warning messages with the IPDW and TFIPC IDs.          |
| +warn=all             | Enables all warning messages. This is the default.                  |

*Verilog\_source\_filename*

Specifies the name of the Verilog source file.

# C

## Compilation/Elaboration Options

---

The `vcs` command performs compilation/elaboration of your design and creates a simulation executable. Compiled event code is generated and used by default. The generated simulation executable, `simv`, can then be used to run multiple simulations.

This section describes the `vcs` command and related options.

### Syntax for two-step flow:

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

Here:

*source\_files*

The Verilog or OVA source files for your design separated by spaces.

`source_or_object_files`

Optional C files (`.c`), object files (`.o`), or archived libraries (`.a`). These are DirectC or PLI applications that you want VCS to link into the binary executable file along with the object files from your Verilog source files. When including object files include the `-cc` and `-ld` options to specify the compiler and linker that generated them.

`options`

Compile-time options that control how VCS compiles your design.

### Syntax for three-step flow:

`% vcs [libname.]design_unit [options]`

Here:

`[libname.]design_unit`

Specifies the `design_unit` you want to simulate, with an optional logical library name. By default, the `WORK` library is assumed.

The `design_unit` can be one of the following:

`cfgname`

Name of the top-level event configuration to be simulated.

`entname[____archname]`

Name of the entity and architecture to be simulated. By default, `archname` is the most recently analyzed architecture.

`module`

- Name of the top-level Verilog module to be simulated
- options
- Elaboration options that control how VCS elaborates your design.
- This appendix lists the following:
- "Option for Code Generation"
  - "Options for Accessing Verilog Libraries"
  - "Options for Incremental Compilation"
  - "Options for Help"
  - "Options for SystemVerilog Assertions"
  - "Options to Enable Compilation of OVA Case Pragmas"
  - "Options for Native Testbench"
  - "Options for Different Versions of Verilog"
  - "Option for Initializing Verilog Variables, Registers and Memories with Random Values"
  - "Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design"
  - "Options for Selecting Register or Memory Initialization"
  - "Options for Using Radiant Technology"
  - "Options for Starting Simulation Right After Compilation"
  - "Options for Specifying Delays and SDF Files"
  - "Options for Compiling an SDF File"

- "Options for Specify Blocks and Timing Checks"
- "Options for Pulse Filtering"
- "Options for Negative Timing Checks"
- "Options for Profiling Your Design"
- "Options to Specify Source Files and Compilation/Elaboration Options in a File"
- "Options for Compiling Runtime Options Into the Executable"
- "Options for PLI Applications"
- "Options to Enable the VCS DirectC Interface"
- "Options for Flushing Certain Output Text File Buffers"
- "Options for Simulating SWIFT VMC Models and SmartModels"
- "Options for Controlling Messages"
- "Option to Run VCS in Syntax Checking Mode"
- "Options for Cell Definition"
- "Options for Licensing"
- "Options for Controlling the Linker"
- "Options for Controlling the C Compiler"
- "Options for Source Protection"
- "Options for Mixed Analog/Digital Simulation"
- "Options for Changing Parameter Values"
- "Unified Option to Change Generic and Parameter Values"

- "Checking for x and z Values in Conditional Expressions"
- "Options for Detecting Race Conditions"
- "Options to Specify the Time Scale"
- "Option to Exclude Environment Variables During Timestamp Checks"
- "Options for Overriding Parameters"
- "Options for Overriding Parameters"
- "Global -check\_all Option"
- "Option to Enable Bounds Check at Compile-Time"
- "Option to Enable Bounds Check at Runtime"
- "Option to Enable Extra Runtime Checks in VHDL"
- "General Options"

---

## Option for Code Generation

`-mcg`

Enables mixed code generation model in VCS backend. Part of code is aggressively optimized by the available C compilers.

---

## Options for Accessing Verilog Libraries

`-v filename`

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

`-y directory`

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

Note:

If multiple `-y` options are on the `vcs` command line, VCS starts searching the directory passed with the first `-y` option, then second and so on.

For example:

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

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

VCS picks `cell.v` from `rev1`.

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

```
//top.v
```

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

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

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

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

#### `+libext+extension+`

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

#### `+liborder`

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

#### `+librescan`

Specifies always searching libraries for module definitions for unresolved module instances beginning with the first library on the `vcs` command line.

**Note:**

+libborder and +librescan switches on elaboration command line will have impact only when the user specifies -y/-v on elaboration 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.

VCS also displays verbose information of the module instances while resolving library modules under the +libverbose option. If you do not want this verbose information to be displayed while resolving library modules, then use the +libverbose+1 option to avoid messages related to the library modules.

---

## Options for Incremental Compilation

-Mdirectory=*directory*

Specifies the incremental compile directory. The default name for this directory is `csrc`, and its default location is your current directory. You can substitute the shorter `-Mdir` for `-Mdirectory`.

`-Mlib=dir`

This option provides VCS with a central place to look for the descriptor information before it compiles a module and a central place to get the object files when it links together the executable. This option allows you to use the parts of a design that have been already tested and debugged by other members of your team without recompiling the modules for these parts of the design.

You can specify more than one place for VCS to look for descriptor information and object files by providing multiple arguments with this option.

### **Example:**

```
% vcs design.v -Mlib=/design/dir1 -Mlib=/design/  
dir2
```

Or, you can specify more than one directory with this option, using a colon (:) as a delimiter between them, as shown below:

```
% vcs design.v -Mlib=/design/dir1:/design/dir2
```

`-Mupdate [=0]`

By default, VCS overwrites the makefile between compilations. If you wish to preserve the makefile between compilations, enter this option with the 0 argument.

Entering this argument without the 0 argument, specifies the default condition, incremental compilation and updating the makefile.

`-Mmakep=make_path`

Specifies the make path.

`-noIncrComp`

Disables incremental compilation.

`-parallel_compile_off`

Turns off parallel compilation and uses serial compilation.

---

## Options for Help

`-h or -help`

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

---

## Option for SystemVerilog

`-sverilog`

Enables SystemVerilog constructs specified in the IEEE Standard of SystemVerilog, IEEE Std 1800-2009.

---

## Options for SystemVerilog Assertions

`-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.**

You can tell VCS to report errors for both unique and priority violations with the `-error` compile-time option as shown below:

`-error=UNIQUE`

VCS reports unique violations as error conditions.

`-error=PRIORITY`

VCS reports priority violations as error conditions.

`-error=UNIQUE, PRIORITY`

VCS reports unique and priority violations as error conditions.

`-assert keyword_argument`

This runtime option is enabled only when the `-assert enable_diag` option is used at compile/elaboration time.

The following is the list of `keyword_argument` that are enabled when the `-assert enable_diag` compilation/elaboration option is used:

- `-assert success`
- `-assert summary`

- `-assert maxcover`
- `-assert maxsuccess`

The following is the list of assertion options that are enabled when the `-assert enable_hier` compilation/elaboration option is used:

- `-assert hier`
- `-assert maxfail=N`
- `-assert finish_maxfail=N`

The following is the list of assertion options that do not require the `-assert enable_diag` or `-assert enable_hier` option:

- `-assert dumpoff`
- `-assert nocovdb`
- `-assert nopostproc`
- `-assert quiet`
- `-assert quiet1`
- `-assert no_fatal_action`
- `-assert report`
- `-assert vacuous`
- `-assert global_finish_maxfail=N`

```
-assert sync_disable
```

Converts disabled signal to its sampled value. Use this option when there is a clock and reset changes at the same time, and you have to use the sample value of the disable signal. Therefore, if you have an assertion assert property (@(posedge clk) disable iff(rst) a) and if you apply the -assert sync\_disable option, this assertion behaves like assert property (@(posedge clk) disable iff(\$sampled(rst)) a).

For example,

Consider the following test.v file.

```
module test;
bit clk;
always #5 clk = ~clk;
logic [2:0] a;
logic reset;
initial begin
    reset =1'b1;
    #10
    a = 3'b011;
    #5
    reset =1'b0;
    a = 3'b000;
    #5
    a = 3'b111;
    #20
    $finish;
end
// clk is posedge at 5,15,25
// reset value 1,0, 0
// sampled reset 1,1, 0
LOL Assert: assert property (@(posedge clk) disable
iff(reset) 0);
endmodule
```

Run the `test.v` file with the following commands:

- % vcs -sverilog test.v -R
- % vcs -sverilog test.v -assert async\_disable -R

When you run the `test.v` file without the `-assert` `async_disable` option, you will get a failure at 15 s. However, with the `-assert` `async_disable` option, you will not get a failure at 15 s.

`enable_diag`

Enables further control of results reporting with runtime options. The runtime assert options are enabled only if you compile the design with this option.

`func hier`

Enables enhanced reporting for assertions in functions.

`hier=file_name`

You can use the `-assert hier=file_name` compile-time option to specify the configuration file for enabling and disabling SystemVerilog assertions. You can either enable or disable:

- Assertions in a module or in a hierarchy.
- An individual assertion.

This option works at runtime only for mixed HDL designs.

Note:

If you pass an empty assert hier file at compile-time or runtime, VCS generates the CM-ASHR-EF error, as shown below:

```
Error-[CM-ASHR-EF] Empty file
The file 'foo.txt' given to the assertion hier control
option was found, but
it is empty.
Please fix the file and try again.
```

You can convert this error message to a warning message, as shown below, using the `-error=noCM-ASHR-EF` option at compile-time or runtime:

```
Warning-[CM-ASHR-EFW] Empty file
The file 'foo.txt' given to the assertion hier control
option was found, but
it is empty.
Please fix the file and try again.
```

**Note:**

If the assertion filter used in assert hier file does not match any assertion in the design, VCS generates the SVA-FILTUNUSED warning message, as shown below, at compile-time or runtime:

```
Warning-[SVA-FILTUNUSED] Unused filter in hier file
'-assert (.)*GLITCH_CBO_TAP_holdinreset' in hier file
cbo_basic_1213083628.hier does not match any module/
instance
hierarchy/assertion.
```

You can convert this warning message to an error message, as shown below, using the `-error=SVA-UNUSEDFLT` option at compile-time or runtime:

```
Error-[SVA-UNUSEDFLT] Unused filter in hier file
'-assert (.)*GLITCH_CBO_TAP_holdinreset' in hier file
cbo_basic_1213083628.hier does not match any module/
instance
hierarchy/assertion.
```

The types of entries that you can specify in the file are as follows:

```
-assert <assertion_name> or  
-assert <assertion_hierarchical_name>
```

If *assertion\_name* is provided, VCS disables the assertions based on wildcard matching of the name in the full design. If *assertion\_hierarchical\_name* is provided, VCS disables the assertions based on wildcard matching of the name in the particular hierarchy given.

## Examples

```
-assert my_assert
```

Disables all assertions with name `my_assert` in the full design.

```
-assert A*
```

Disables all assertions whose name starts with `A` in the full design.

```
-assert *
```

Disables all assertions in the full design.

```
-assert top.INST2.A
```

Disables all assertions whose names start with `A` in the hierarchy `top.INST2`. If assertions whose name starts with `A` exists in inner scopes under `top.INST2`, they are not disabled. This command has effect on assertions only in scope `top.INST2`.

```
+assert <assertion_name> or  
+assert <assertion_hierarchical_name>
```

If *assertion\_name* is provided, VCS enables the assertions based on wildcard matching of the name in the full design. If *assertion\_hierarchical\_name* is provided, then VCS enables the assertions based on wildcard matching of the name in the given hierarchy.

## Examples

```
+assert my_assert
```

Enables all assertions with name `my_assert` in the full design.

```
+assert A*
```

Enables all assertions whose name starts with `A` in the full design.

```
+assert *
```

Enables all assertions in the full design.

```
+assert top.INST2.A
```

Enables assertion `A` in the hierarchy `top.INST2`.

```
+tree <module_instance_name> or  
+tree <assertion_hierarchical_name>
```

If `module_instance_name` is provided, VCS enables assertions in the specified module instance and all module instances hierarchically under that instance. If `assertion_hierarchical_name` is provided, VCS enables the specified SystemVerilog assertion. Wildcard characters can also be used for specifying the hierarchy.

## Examples

```
+tree top.inst1
```

Enables the assertions in module instance `top.inst1` and all the assertions in the module instances under this instance.

```
+tree top.inst1.a1
```

Enables the SystemVerilog assertion with the hierarchical name `top.inst1.a1`.

```
+tree top.INST*.A1
```

Enables assertion `A1` from all the instances whose names start with `INST` under module `top`.

```
-tree <module_instance_name> or  
-tree <assertion_hierarchical_name>
```

If `module_instance_name` is provided, VCS disables the assertions in the specified module instance and all module instances hierarchically under that instance. If `assertion_hierarchical_name` is provided, VCS disables the specified SystemVerilog assertion. Wildcard characters can also be used for specifying the hierarchy.

## Examples

```
-tree top.inst1
```

Disables the assertions in module instance `top.inst1` and all the assertions in the module instances under this instance.

```
-tree top.inst1.a1
```

Disables SystemVerilog assertion with the hierarchical name `top.inst1.a1`.

```
-tree top.INST*.A1
```

Disables assertion `A1` from all the instances whose names start with `INST` under module `top`.

```
+module module_identifier
```

VCS enables all the assertions in all instances of the specified module, for example:

```
+module dev
```

VCS enables the assertions in all instances of `module dev`.

```
-module module_identifier
```

VCS disables all the assertions in all instances of the specified module, for example:

```
-module dev
```

VCS disables the assertions in all instances of `module dev`.

The specifications are applied serially as they appear in file `file_name` except for `+/-assert`. The `+/- assert` patterns are always serially applied after all other patterns are done. The result of applying the specifications in this file is that a group of assertions get excluded. The remaining assertions are available for further exclusion by other means, such as the `$assertoff` system task in the source code. However, the following should be noted:

- The first specification denotes the default exclusion for interpreting the file. If the first specification is a minus(`-`), then all assertions are included before applying the first and the following specifications. Conversely, if the first specification is a plus(`+`), then all assertions are excluded prior to applying the first and the following specifications.
- Unlike `-/+module` and `-/+tree` specifications, any assertion excluded by applying `-assert` specification cannot be included by the later specifications in the file.

#### `enable_hier`

Enables the use of the runtime option `-assert hier=file.txt`, which allows turning assertions on or off.

#### `filter_past`

For assertions that are defined with the `$past` system task, ignore these assertions when the past history buffer is empty. For instance, at the very beginning of the simulation, the past history buffer is empty. Therefore, the first sampling point and subsequent sampling points should be ignored until the past buffer has been filled with respect to the sampling point.

#### `offending_values`

Enables the reporting of the values of all variables used in the assertion failure messages. For more information, see ["Reporting Values of Variables in the Assertion Failure Messages"](#).

`disable`

Disables all SystemVerilog assertions in the design.

`disable_cover`

When you include the `-cm assert` compile-time and runtime option, VCS includes information about cover statements in the assertion coverage reports. This keyword prevents cover statements from appearing in these reports.

`disable_assert`

Disables only the `assert` and `assume` directives without affecting the `cover` directives. It complements the existing control options which allows you to disable only cover directives or all of the assertions such as `assert/assume/cover`.

`disable_rep_opt`

Specifying a delay or a repetition value greater than 200 in the assertion expression will affect both compile-time and runtime performance. Therefore, VCS optimizes expression and issues a warning message as shown below:

```
Warning-[SVA-LDRF] Large delay or repetition found.  
Large delay or repetition found (data ##[0:200] (!data)).VCS will  
optimize compile time. However it may affect runtime.  
"test.v", 8
```

`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.

+lint=PWLNT:<max\_count>

Enables the PWLNT lint messages when \$past is used in the code with the number of clock ticks exceeding 5. You can restrict the number of PWLNT lint messages for a particular compilation using the `max_count` argument.

For example, `+lint=PWLNT:10` restricts the number of PWLNT lint messages to a maximum of 10 for one compilation.

---

## Options to Enable Compilation of OVA Case Pragmas

-ova\_enable\_case

Enables the compilation of OVA case pragmas only, when used without `-Xova` or `-ova_inline`. All inlined OVA assertion pragmas are ignored.

---

## Options for Native Testbench

-ntb

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

`-ntb_define macro`

Specifies any OpenVera macro name on the command line. You can specify multiple macro names using the plus (+) character.

The macro can also be defined to be a fixed number. For example, in the following:

```
program test
{
    integer x;
    x =12345;
printf ("DEBUG==> my value = %d and x = %d\n", MYVALUE,
x);
}
```

When you compile and run:

```
% vcs -ntb -ntb_define MYVALUE=10000 myprog.vr -R
```

This outputs are:

```
DEBUG==> my value = 10000 and x = 12345
```

`-ntb_fileext .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`

Does a bounds check on dynamic type arrays (dynamic, associative, queues) and issues an error at runtime.

`check=dynamic`

Same as `check`. Does a bounds check on dynamic type arrays (dynamic, associative, queues) and issues an error at runtime.

`check=fixed`

Does a bounds check only on fixed size arrays and issues an error at runtime.

`check=all`

Does a bounds check on both fixed size and dynamic type arrays and issues errors at runtime.

The following error messages are displayed during runtime:

- ERROR- [ DT-OBAE ] Out-of-bound access for queues

This error message is displayed, if a queue element is accessed with an out-of-bounds index condition.

For example,

```
module tb();
    int temp; // temp signal
    int int_queue[$] = { 1, 2, 3}; //Queue
    initial
        begin
            //Queue
            temp = int_queue[9];
        end
    endmodule
```

The following error message is displayed:

```
Error-[DT-OBAE] Out of bound access
test2.sv, 9
Out of bound access on smart queue (size:3, index:9)
Simulation time = 0
Please make sure that the index is positive and less
than size.
```

- ERROR-[DT-OBAE] Out-of-bound access for dynamic arrays

This error message is displayed, if a dynamic array element is accessed with an out-of-bounds index condition.

For example,

```
module tb();
    reg some_bit; // temp signal
    reg nibble[]; // Dynamic array
    int int_queue[$] = { 1, 2, 3}; //Queue
    initial
        begin
            //Dynamic array
```

```

        nibble = new[3]; // Create a 3-element array.
        some_bit = nibble[3];
    end
endmodule

```

**The following error message is displayed:**

```

Error-[DT-OBAE] Out of bound access
test2.sv, 11
Out of bound access on dynamic array (size:3,
index:3)
Simulation time = 0
Please make sure that the index is positive and less
than size.

```

- **ERROR- [OBA] Out-of-bound access for fixed size  
unpacked array**

**This error message is displayed, if a fixed size unpack array  
is accessed with an out-of-bounds index condition.**

**For example,**

```

module tb();
reg [1:0] fifo_data[2:0]; // fifo memory
reg [1:0] some_signal; // temp signal
initial
begin
    //Unpacked dimension
    some_signal = fifo_data[3];
end
initial $display("some_signal = %0d", some_signal);
endmodule

```

**The following error message is displayed:**

```

Error-[OBA] Out of bound access
test2.sv, 9
Out of bound access on array (index value: 3)
Simulation time = 0

```

- ERROR-[DT-IV] Out-of-bound access for fixed size unpacked array

This error message is displayed, if a fixed size unpacked array element is accessed with an index value X or Z.

For example,

```
module tb();
reg [17:0] fifo_data[255:0]; // fifo memory
reg [7:0] rd_ptr = 8'bxxxxxxxxx;
wire [7:0] some_signal; // temp signal

assign some_signal = fifo_data[rd_ptr];
initial $display("some_signal = %0d", some_signal);
endmodule
```

The following error message is displayed:

```
Error-[DT-IV] Illegal value
test2.sv, 6
Illegal index value specified for array
Simulation time = 0
Please make sure that the value is properly
initialized with none of the bits set to x or z.
```

`dep_check`

Enables dependency analysis and incremental compilation. Detects files with circular dependencies and issues an error message when VCS cannot determine which file to compile first.

`no_file_by_file_pp`

By default, VCS does file-by-file preprocessing on each input file, feeding the concatenated result to the parser. This argument disables this behavior.

`print_deps`

Tells VCS to display the dependencies for the source files on the screen. Enter this argument with the `dep_check` argument.

`rvm`

Use `rvm` when RVM or VMM is used in the testbench. For more information, refer to the "[Using VMM with VCS](#)" section.

`sv_fmt`

The default padding used in displayed or printed strings is right padding. The `sv_fmt` option specifies left padding. For example, when `-ntb_opts sv_fmt` is used, the result of

```
$display("%10s", "my_string");
```

is to put 10 spaces to the left of `my_string`.

To specify right padding when `-ntb_opts sv_fmt` is used, put a dash before the number of spaces. For example, the result of

```
$display("%-10s", "my_string");
```

is to put 10 spaces to the right of `my_string`.

`tb_timescale=value`

Specifies an overriding timescale for the testbench, whenever the required testbench timescale is different from that of the design. It must be used in conjunction with the `-timescale` option that specifies the timescale for the design.

If the required testbench timescale is different from the design or DUT timescale, then both the testbench timescale and the DUT timescale must be passed during VCS compilation.

## **Example:**

The following command specifies a required testbench timescale of 10ns/10ps and a design timescale of 1ns/1ps:

```
%> vcs -ntb_opts tb_timescale=1ns/1ps  
      -timescale=10/10ns file.sv
```

**tokens**

Pre-processes 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 pre-processed result of the encrypted OpenVera files. If there is no encrypted OpenVera file, VCS sends all the OpenVera pre-processed results to the `tokens.vr` file.

**use\_sigprop**

Enables the signal property access functions. For example, `vera_get_ifc_name()`.

**vera\_portname**

Specifies the following:

- The Vera shell module name is named `vera_shell`.
- The interface ports are named `ifc_signal`.
- Bind signals are named, for example, as: `\ifc_signal[3:0]`.

**-ntb\_shell\_only**

Generates only a `.vshell` file. Use this option when compiling a testbench separately from the design file.

`-ntb_sfname filename`

Specifies the file name of the testbench shell.

`-ntb_sname module_name`

Specifies the name and directory where VCS writes the testbench shell module.

`-ntb_spath`

Specifies the directory where VCS writes the testbench shell and shared object files. The default is the compilation directory.

`-ntb_vipext .ext`

Specifies an OpenVera encrypted-mode file extension to mark files for processing in OpenVera encrypted IP mode. Unlike the `-ntb_fileext` option, the default encrypted-mode extensions `.vrp` and `.vrhp` are not overridden and will always be in effect. You can pass multiple file extensions at the same time using the plus (+) character.

---

## Options for Different Versions of Verilog

`-v95`

Specifies not recognizing Verilog 2001 keywords.

`+systemverilogext+ext`

Specifies a file name extension for SystemVerilog source files. If you use a different file name extension for the SystemVerilog part of your source code and you use this option, the `-sverilog` option has to be omitted.

**Note:**

This compile-time option also works similar to the `-sverilog` option in which it enables SystemVerilog LRM (IEEE Std 1800-2012) rules for all the source files on the vcs command line and not only the files with the specified extension.

`+verilog2001ext+ext`

Specifies a file name extension for Verilog 2001 source files.

`+verilog1995ext+ext`

Specifies a file name extension for Verilog 1995 files. Using this option allows you to write Verilog 1995 code that would be invalid in Verilog 2001 or SystemVerilog code, such as using Verilog 2001 or SystemVerilog keywords, like `localparam` and `logic`, as names.

**Note:**

Do not enter all three of these options on the same command line.

`-extinclude`

If a source file for one version of Verilog contains the `'include` compiler directive, VCS by default compiles the included file for the same version of Verilog, even if the included file has a different filename extension. If you want VCS to compile the included file with the version specified by its extension, enter this compile-time option. The following code examples show using this option.

If source file `a.v` contains the following:

```
'include "b.sv"  
module a();  
reg ar;  
endmodule
```

and if source file `b.sv` contains the following:

```
module b();
  logic ar;
endmodule
```

VCS compiles `b.sv` for SystemVerilog with the following command line:

```
vcs a.v +systemverilogext+.sv -extinclude
```

---

## Option for Initializing Verilog Variables, Registers and Memories with Random Values

`+vcs+initreg+random`

Initializes all bits of the Verilog variables, registers defined in sequential UDPs, and memories including multi-dimensional arrays (MDAs) in your design to random value 0 or 1, at time zero. The default random seed is used.

The supported data types are:

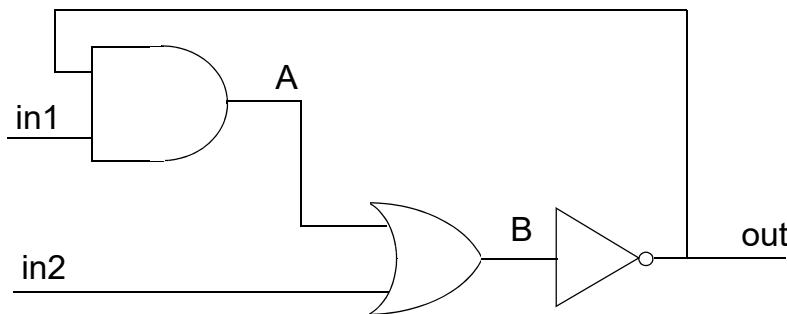
- reg
- bit
- integer
- int
- logic

For Example:

```
% vcs +vcst+initreg+random [other_vcs_options] file1.v  
file2.v file3.v
```

The initialization option may expose an infinite simulation loop at time zero in combinational logic with a feedback loop, as shown in [Figure C-1](#).

*Figure C-1 Combinational Logic With a Feedback Loop*



In [Figure C-1](#), the `reg` variables `in1`, `in2`, `A` and `B` have the default initial values `x`. Assigning value 0 or 1 to `in1` or `in2` does not alter the value of `A`, `B` and `out`. The feedback loop is stabilized and the simulation advances. Some combinations of initial values assigned to these `reg` variables trigger a continuous re-evaluation of the combinational logic which results in an infinite simulation loop.

When the initialization option is used, the initialized values of variables may conflict with the initial variable assignments specified in a design.

The following are steps to prevent potential race conditions:

- Avoid assigning initial values to `reg` variables in the variable declarations when the assigned values are different from the values specified with the `+vcs+initreg+random` option.

For example:

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

- Avoid assigning values to registers or memory elements at simulation time 0 when the assigned values are different from the values specified with the `+vcs+initreg+random` option.

For example:

```
reg [7:0] mem [7:0][15:0];
initial
begin
    mem[1][1]=8'b00000001;
```

- Avoid initializing state variables to an unknown, `x`, state.
- Avoid inconsistent states in the design due to randomization.

The initialization option can potentially be used to reduce the amount of time spent on initialization related issues in gate-level simulations. At time 0, all uninitialized `reg` variables are assigned the default value `x`, which is a non-deterministic and unknown state of a design. The value `x` can propagate during a gate-level simulation and cause unexpected behaviors. You can use the `+vcs+initreg+random` option to initialize all bits of Verilog variables, registers and memories to prevent propagation of `x` values in a gate-level simulation.

Note:

The initialization option is targeted for initializing variables in gate level simulations (including UDP variables). Initialization of variables in RTL constructs such as named blocks, structures, or in user-defined tasks or functions is not supported.

The `+vcs+initreg+random` option only applies to the Verilog portion of a mixed language design.

The `+vcst+initreg+0` and `+vcst+initreg+1` compile-time options are no longer supported. You must use the `+vcs+initreg+random` option at compile-time.

---

## Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design

`+vcs+initreg+config+config_file`

Specifies a configuration file for initializing Verilog variables, registers defined in sequential UDPs, and memories including multi-dimensional arrays (MDAs) in your design, at time zero. In the configuration file, you can define the parts of a design to apply the initialization and the initialization values of the variables.

The syntax of the configuration file entries are:

```
defaultvalue x|z|0|1|random|random seed_value  
instance instance_hierarchical_name [x|z|0|1|random|  
random seed_value]  
tree instance_hierarchical_name depth [x|z|0|1|random|  
random seed_value]
```

```
module module_name [x|z|0|1|random|random seed_value]  
modtree module_name depth [x|z|0|1|random|random seed_value]
```

#### Note:

VCS supports initializing parts of the design with *x|z|0|1* values in initreg configuration file.

### The **defaultvalue** entry

```
defaultvalue x|z|0|1|random|random seed_value
```

A **defaultvalue** entry starts with the keyword **defaultvalue** and should be the first entry in a configuration file. This entry specifies the default values for all Verilog variables, registers and memories in the design. The keyword **random** specifies initializing with random values. You can specify a **seed value** for the VCS random value generator. Only one **defaultvalue** entry is allowed in a configuration file.

### The **instance** entry

```
instance instance_hierarchical_name [x|z|0|1|random|random seed_value]
```

An **instance** entry starts with the keyword **instance**. This entry specifies a module instance and the initial values for the Verilog variables, registers and memories in this instance.

### The **tree** entry

```
tree instance_hierarchical_name depth [x|z|0|1|random|random seed_value]
```

A `tree` entry starts with the keyword `tree`. This entry specifies a sub-hierarchy and the initial values for the Verilog variables, registers and memories in this sub-hierarchy. When a hierarchical name of a module instance is specified, the initialization applies to the specified instance and the module instances that are hierarchically beneath the specified instance. You can specify a depth value to limit the levels down the hierarchy for applying the initialization.

#### **Depth Value    Level of Initialization**

|          |                                                                                              |
|----------|----------------------------------------------------------------------------------------------|
| 0        | Initialize all levels down the sub-hierarchy to the leaf level instances.                    |
| 1        | Initialize only the specified instance                                                       |
| 2 and up | Initialize the specified number of levels down the sub-hierarchy from the specified instance |

### **The `module` entry**

```
module module_name [x|z|0|1|random|random seed_value]
```

A `module` entry starts with the keyword `module`. This entry specifies initial values for all instances of the specified module.

### **The `modtree` entry**

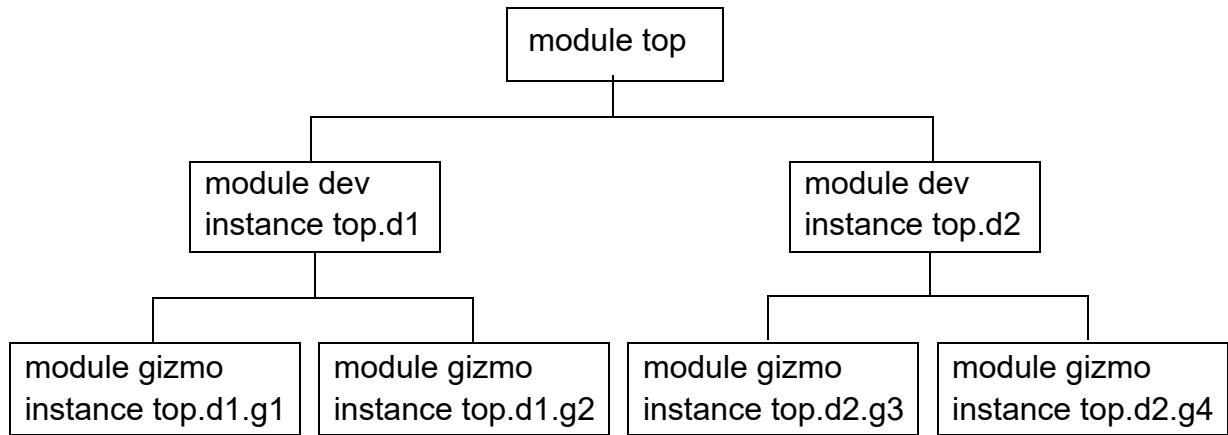
```
modtree module_name depth [x|z|0|1|random|random seed_value]
```

A `modtree` entry starts with the keyword `modtree`. This entry specifies initial values for all instances of the specified module and all instances that are hierarchically beneath those instances.

## **Configuration File Example**

Figure C-2 is a hierarchical diagram of a small design.

*Figure C-2 Design Hierarchy for Initializing from a Configuration File*



The following are the example entries in a configuration file for the small design shown in [Figure C-2](#) (with a default value x).

```
instance top.d1 0
```

Initializes the variables, registers and memories in the instance `top.d1` to value 0.

```
tree top 0 0
tree top.d1 0 x
```

The first entry initializes all variables, registers and memories in the design to value 0. The second entry changes the initial values from 0 to x for the variables, registers and memories in the instance `top.d1` and all instances beneath `top.d1`, namely `top.d1.g1` and `top.d1.g2`.

```
module gizmo 1
```

Initializes the variables, registers and memories in all instances of the module `gizmo` to value 1, namely `top.d1.g1`, `top.d1.g2`, `top.d2.g3`, and `top.d2.g4`.

```
modtree dev 0 random
```

Initializes the variables, registers and memories in both instances of module `dev` and all four instances beneath those instances with random values. Module `top` is not initialized.

```
modtree dev 0 random  
instance top.d1.g2 x
```

The first entry is described in the previous example. The second entry changes the initial values from random values to `x` for the variables, registers and memories in the instance `top.d1.g2`.

---

## Options for Selecting Register or Memory Initialization

`+vcs+initreg+random+nomem`

Disables initialization of memories or multi-dimensional arrays (MDAs). This option allows initialization of variables that do not have a dimension. This option can only be used when the `+vcs+initreg+random` or `+vcs+initreg+config+config_file` option is specified at compile-time.

`+vcs+initreg+random+noreg`

Disables initialization of variables that do not have a dimension. This option allows initialization of memories or MDAs. This option can only be used When the `+vcs+initreg+random` or `+vcst+initreg+config+config_file` option is specified at compile-time.

---

## Options for Using Radiant Technology

`+rad`

Performs Radiant Technology optimizations on your design.

Note:

These optimizations are also enabled for SystemVerilog part of the design.

`+optconfigfile+filename`

Specifies a configuration file that lists the parts of your design you want to optimize (or not optimize) and the level of optimization for these parts. You can also use the configuration file to specify ACC write capabilities. See "[Compiling With Radiant Technology](#)".

---

## Options for Starting Simulation Right After Compilation

`-R`

Runs the executable file immediately after VCS links it together.

---

## Options for Specifying Delays and SDF Files

`-sdf min|typ|max:instance_name:file.sdf`

Enables SDF annotation. Minimum, typical, or maximum values specified in *file.sdf* are annotated on the instance, *instance\_name*.

+alllmtm

Specifies compiling separate files for minimum, typical, and maximum delays when there are min:typ:max delay triplets in SDF files. If you use this option, you can use the +mindelays, +typdelays, or +maxdelays options at runtime to specify which compiled SDF file VCS uses. Do not use this option with the +maxdelays, +mindelays, or +typdelays compile-time options.

+charge\_decay

Enables charge decay in trireg nets. Charge decay does not work if you connect the trireg to a transistor (bi-directional pass) switch such as tran, rtran, tranif1, or rtranif0.

+delay\_mode\_path

Uses only delay specifications in module-path delays in specify blocks. Overrides all the delay specifications on all gates, switches, and continuous assignments to zero.

+delay\_mode\_zero

Removes delay specifications on all gates, switches, continuous assignments, and module paths.

+delay\_mode\_unit

Overrides all the delay specifications in module-path delays in specify blocks to zero delays. Overrides 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

Overrides all the delay specifications in module-path delays in specify blocks to zero delays. Uses only the delay specifications on all gates, switches, and continuous assignments.

-add\_seq\_delay

Use the -add\_seq\_delay option to set the delays of the sequential User-Defined Primitives (UPDs) without delays. Its syntax is as follows:

-add\_seq\_delay <n>

Where, n is the delay specification argument. It can be a real number or a real number followed by a time unit. The time unit can be fs, ps, ns, us, ms, or s. If no time unit is specified, then the simulation time\_unit is used. For example, if simulation time\_unit/time\_precision is 1ns/1ps, then -add\_seq\_delay 3 means 3ns.

The delay specification argument is applied to all sequential UDPs (without delays) in the design.

## Examples

For example, consider that the simulation time\_unit/time\_precision is 1ns/1ps.

- The following option assigns a 1ns delay to all sequential UDP paths.

```
-add_seq_delay 1ns
```

- The following option assigns a 0.7ns delay to all sequential UDP paths.

```
-add_seq_delay 0.7
```

```
-add_seq_delay 0.7ns
```

```
-add_seq_delay 700ps
```

## Key Points to Note

- If sequential UDPs already have delay specified (#(delay), including #0), then -add\_seq\_delay is ignored. That is, -add\_seq\_delay only supports sequential UDPs without delays.
- The -add\_seq\_delay option does not affect IOPATH delay such as:

```
specify
    (posedge ck => q +: d) = (10,11);
endspecify
```

The above IOPATH delay (10,11) remains the same even when -add\_seq\_delay <n> is specified.

- If you use `+delay_mode_zero` and `-add_seq_delay` on the same command line, then the UDP is considered, as mentioned below:

With `+delay_mode_zero`: The `+delay_mode_zero` option takes precedence.

Without `+delay_mode_zero`: The `-add_seq_delay` option takes precedence if IOPATH delay is smaller than it.

- The `-add_seq_delay` option overrides `+delay_mode_unit`. For example, if you specify `-add_seq_delay 15ps +delay_mode_unit`, then still you see 15ps delay.
- The `-add_seq_delay` option overrides `+delay_mode_distributed`. For example, if you specify `-add_seq_delay 15ps +delay_mode_distributed`, then still you see 15ps delay.

`+maxdelays`

Specifies using the maximum timing delays in the min:typ:max delay triplets when compiling the SDF file. The `mtm_spec` argument to the `$sdf_annotate` system task overrides this option.

#### Note:

If you intend to include the `+maxdelays` option during elaboration, then it is mandatory to include this option during analysis as well. The syntax is as follows:

#### Analysis

```
% vlogan [vlogan_options] file2.v file3.v +maxdelays
```

## Elaboration

```
% vcs -sdf min|typ|max:instance_name:file.sdf \
[elab_options] top_cfg/entity/module +maxdelays
```

## Simulation

```
% simv [run_options]
+mindelays
```

Specifies using the minimum timing delays in the min:typ:max delay triplets when compiling the SDF file. The *mtm\_spec* argument to the `$sdf_annotate` system task overrides this option.

### Note:

If you intend to include the `+mindelays` option during elaboration, then it is mandatory to include this option during analysis as well. The syntax is as follows:

## Analysis

```
% vlogan [vlogan_options] file2.v file3.v +mindelays
```

## Elaboration

```
% vcs -sdf min|typ|max:instance_name:file.sdf \
[elab_options] top_cfg/entity/module mindelays
```

## Simulation

```
% simv [run_options]
+typdelays
```

Specifies using the typical timing delays in min:typ:max delay triplets when compiling the SDF file. The *mtm\_spec* argument to the \$sdf\_annotation system task overrides this option.

#### Note:

If you intend to include the +typdelays option during elaboration, then it is mandatory to include this option during analysis as well. The syntax is as follows:

#### Analysis

```
% vlogan [vlogan_options] file2.v file3.v +typdelays
```

#### Elaboration

```
% vcs -sdf min|typ|max:instance_name:file.sdf \
[elab_options] top_cfg/entity/module +typdelays
```

#### Simulation

```
% simv [run_options]
```

```
+multisource_int_delays
```

Enables the multisource INTERCONNECT feature, including transport delays with full pulse control.

```
+nbaopt
```

Removes all intra-assignment delays in all the non-blocking assignment statements in the design. Many users enter a #1 intra-assignment delay in non-blocking 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 non-blocking assignment statements in the design, not just the #1 delays.

`+sdf_nocheck_celltype`

For a module instance to which an SDF file back-annotates delay data, disables comparing the module identifier in the source code with the `CELLTYPE` entry in the SDF file.

`+transport_int_delays`

Enables transport delays for delays on nets with a delay back-annotated from an INTERCONNECT entry in an SDF file. The default is inertial delays.

`+transport_path_delays`

Enables transport delays for module-path delays.

`-sdfretain`

Enables timing annotation as specified by a RETAIN entry on IOPATH delays. By default, VCS ignores RETAIN entries with the following warning message:

```
Warning-[SDFCOM_RCI] RETAIN clause ignored
SDF_filename, line_number
module: module_name, "instance: hierarchical_name"
    SDF Warning: RETAIN clause ignored, but IOPATH
    annotated,
    Please use -sdfretain switch to consider RETAIN
```

The syntax for RETAIN entries is as follows:

```
(IOPATH port_spec port_instance (RETAIN  
delval_list)* delval_list)
```

For example:

```
(IOPATH RCLK DOUT[0] (RETAIN (40)) (100.1)  
(100.2))
```

-sdfretain=warning

If the RETAIN entry values are larger than the delay values, VCS displays the following warning message at runtime:

```
Warning-[SDFRT_IRV] RETAIN value ignored  
RETAIN value is ignored as it is greater than IOPATH  
delay
```

If you want to see a warning message at compile time, enter this option along with the -sdfretain option. The following is an example of this warning message:

```
Warning-[SDFCOM_RLTPD] RETAIN value larger than IOPATH  
delay  
SDF_filename, line_number  
module: module_name, "instance: hierarchical_name"  
SDF Warning: RETAIN value (value) is larger than IOPATH  
delay, RETAIN will be ignored at runtime
```

+iopath+edge+sub-option

This option is used when edge sensitivity is used in IOPATH SDF file entries. The different sub-options used with

+iopath+edge+option and their descriptions are as follows:

+iopath+edge+strict

This option is used for LRM compliance. When edge sensitivity is specified for the input port in the SDF file and corresponding arc is not found in Verilog model, VCS by default does not give the warning message, you should use the `+iopath+edge+strict` switch to display the warning message. After the warning message is displayed, the data from SDF is not back-annotated to the Verilog model.

`+iopath+edge+ignore`

This option can be used to make the annotation work by ignoring the edge in SDF.

`+iopath+edge+max`

This option is used for annotating higher delays.

`+iopath+edge+min`

This option is used for annotating smaller delays.

---

## Options for Compiling an SDF File

`+csdf+precompile`

Precompiles your SDF file into a format that VCS can parse when it compiles your Verilog code. See "["Precompiling an SDF File"](#)".

---

## Options for Specify Blocks and Timing Checks

`+pathpulse`

Enables the search for `PATHPULSE$ specparam` in specify blocks.

`+nospecify`

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

`+notimingcheck`

Tells VCS to ignore timing check system tasks when it compiles your design. This option can moderately improve simulation performance. The extent of this improvement depends on the number of timing checks that VCS ignores. You can also use this option at runtime to disable these timing checks after VCS has compiled them into the executable. However, the executable simulates faster if you include this option at compile time so that the timing checks are not in the executable. If you need the delayed versions of the signals in negative timing checks but want faster performance, include this option at runtime. The delayed versions are not available if you use this option at compile time.

Note:

- VCS recognizes `+notimingchecks` to be the same as `+notimingcheck` when you enter it on the vcs or simv command line.
- The `+notimingcheck` option has higher precedence than any tcheck command in UCLI.

`+no_notifier`

Disables toggling of the notifier register that you specify in some timing check system tasks. This option does not disable the display of warning messages when VCS finds a timing violation that you specified in a timing check.

`+no_tchk_msg`

Disables display of timing violations, but does not disable the toggling of notifier registers in timing checks. This is also a runtime option.

---

## Options for Pulse Filtering

`+pulse_e/number`

Displays an error message and propagates an x value for any path pulse whose width is less than or equal to the percentage of the module-path delay specified by the *number* argument, but is still greater than the percentage of the module-path delay specified by the *number* argument to the `+pulse_r/number` option.

`+pulse_r/number`

Rejects any pulse whose width is less than *number* percent of the module-path delay. The *number* argument is in the range of 0 to 100.

`+pulse_int_r`

Same as the existing `+pulse_r` option, except it applies only to INTERCONNECT delays.

`+pulse_int_e`

Same as the existing `+pulse_e` option, except it applies only to INTERCONNECT delays.

`+pulse_on_event`

Specifies that when VCS encounters a pulse shorter than the module-path delay, VCS waits until the module-path delay elapses and then drives an x value on the module output port and displays an error message. It drives that x value for a simulation time equal to the length of the short pulse or until another simulation event drives a value on the output port.

+pulse\_on\_detect

Specifies that when VCS encounters a pulse shorter than the module-path delay, VCS immediately drives an x value on the module output port, and displays an error message. It does not wait until the module-path delay elapses. It drives that x value until the short pulse propagates through the module or until another simulation event drives a value on the output port.

---

## Options for Negative Timing Checks

-negdelay

Enables the use of negative values in IOPATH and INTERCONNECT entries in SDF files.

To consider a negative INTERCONNECT delay, one of the following should be true:

- Sum of INTERCONNECT and PORT delays should be greater than zero
- Sum of INTERCONNECT and IOPATH delays should be greater than zero
- Sum of INTERCONNECT and DEVICE delays should be greater than zero

Otherwise, the negative INTERCONNECT delay is ignored, and a warning message is generated for the same.

Similarly, to consider a negative IOPATH delay, the sum of IOPATH and DEVICE delays should be greater than zero. Otherwise, the negative IOPATH delay is ignored, and a warning message is generated for the same.

## Limitations

This option is not supported in the following scenarios:

- RETAIN on negative IOPATH
- INCREMENT delay

+neg\_tchk

Enables negative values in timing checks.

+old\_ntc

Prevents the other timing checks from using delayed versions of the signals in the \$setuphold and \$recrrem timing checks.

+NTC2

In \$setuphold and \$recrrem 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.

---

## Options for Profiling Your Design

-simprofile time | mem

Specifies the type of simulation profiling you want done, see [The Unified Simulation Profiler](#).

---

## Options to Specify Source Files and Compilation/Elaboration Options in a File

**-f** *filename*

Specify a file that contains a list of source files and compile-time options, including C source files and object files.

The following are the features of **-f** option:

- You can use Verilog comment characters // and /\* \*/ to comment out entries in the file. The comment character // used in between file path is treated as a file path only and not considered as a comment.

For example, consider /abc/def//xyz.v. The comment character // used here is not treated as a comment.

- You can use this option inside the *file* to point to another file.
- You can specify all compile-time options that begin with a plus (+) character. However, you can only specify the following compile-time options that begin with a minus (-) character:

-f                  -y                  -l                  -u                  -v                  -sverilog

The **-f** option is not supported in the UUM flow.

**-file** *filename*

Specify a file that contains a list of source files and VCS compilation/elaboration options, including C source files and object files.

You can use this option to overcome the limitation of `-f` option related to compile-time options that begin with `(-)` character.

The `-file` option is supported in the UUM flow.

`-F filename`

This compile-time option is similar to the `-f` option, but you can also specify a file list and a path to search for the files. Following is the syntax:

```
%vcs top.v -F <path_to_file>/filelist
```

When you specify this option, the path to the file list gets added as a prefix to the content of the file list.

Consider that the `<filelist>` consisting of files `a.v` and `b.v` exists in the previous directory of the current working directory. With the following syntax, the path to the `<filelist>` is added as a prefix to the content of the `<filelist>`. When parsed, VCS searches for files `../a.v` and `../b.v`.

```
%vcs top.v <source_files> -F ../<filelist>
```

You can also specify an absolute path name using the following syntax:

```
%vcs top.v -F <absolute_path>/filelist
```

The syntax allows you to search for files `<absolute_path>/a.v` and `<absolute_path>/b.v`.

The following are the features of `-F` option:

- You can use Verilog comment characters such as // and /\* \*/ to comment out entries in the file.
- You can use this option inside the *file* to point to another file.
- You can specify all compile-time options that begin with a plus (+) character. However, you can only specify the following compile-time options that begin with a minus (-) character:

```
-CC      -f      -F      -gen_asm      -gen_obj      -l      -line
-P       -u      -v      -Y
```

The -F option is not supported in the UUM flow.

### **Limitations of -f, -file and -F options**

- These options do not support the -full164 option in the file. You must enter that option on the vcs command-line.
- You cannot specify escape characters in the file.
- You cannot use meta characters in the file, except \* and \$.
- At UUM flow, -f option is allowed only with vlogan/vhdlan command line. With vcs elaboration command, -file option should be used instead of the -f option.
- The options do not support the +vcs+loopreport option.
- You cannot use Verilog comment characters such as // and /\* \*/ to comment out entries in the file with the -file option.

## **Options for Compiling Runtime Options Into the Executable**

+plusarg\_save

Some runtime options must be preceded by the `+plusarg_save` option for VCS to compile them into the executable.

`+plusarg_ignore`

Tells VCS not to compile the following runtime options into the `simv` executable. This option is used to counter the `+plusarg_save` option on a previous line.

---

## Options for PLI Applications

`+acc+level_number`

Enables PLI ACC capabilities for the entire design using the following options:

`+acc+3`

Enables all capabilities except value change callbacks. This option also enables module path delay annotation.

`+acc+4`

Enables all capabilities except value change callbacks. This option also enables gate delay annotation.

`+applylearn+filename`

Recompiles your design to enable only the ACC capabilities that you needed for the debugging operations you did during a previous simulation of the design.

`-e new_name_for_main`

Specifies the name of your `main()` routine. You write your own `main()` routine when you are writing a C++ application or when your application does some processing before starting the simv executable.

Note:

Do not use the `-e` option with the VCS/SystemC Cosimulation Interface.

`-P pli.tab`

Compiles a user-defined PLI definition table file.

`+vpi`

Enables the use of VPI PLI access routines.

`+vpi+1`

Allows you to reduce the runtime memory by reducing the information storage for VPI interface at runtime. This option limits the behavioral information at compile-time, but preserves the structural information.

This option allows you to do the following:

- Browse the design hierarchy and read the values of variables. This facilitates debugging.
- Write over or force values on variables using `vpi_put_value()`. This allows a foreign language testbench to drive a stimulus to a Verilog design.

- Register VPI callbacks. This facilitates the waveform dumping features. However, certain advance debugging features (such as Line stepping, Driver/Loads information, and so on) will not be available.

### **Limitations:**

- You cannot use this option to browse, enable, or disable SV and RT assertions.

#### **Note:**

The `+vpi+1+assertion` option allows you to browse, enable, and disable SV and RT assertions to the base features of `+vpi+1`.

- If you use `+vpi+1` with the `debug_access` option and try to use UCLI commands, then some of the commands may fail. No diagnostics or error messages will be generated to suggest that those commands are failing due to existence of `+vpi+1` option.

`+vpi+1+assertion`

Allows you to browse, enable, and disable SV and RT assertions to the base features of `+vpi+1`.

`-load shared_library:registration_routine`

Specifies the registration routine in a shared library for a VPI application.

## **Options to Enable the VCS DirectC Interface**

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

The `+vc` option enables extern declarations of C/C++ functions and calling these functions in your source code. See the VCS *DirectC Interface User Guide*. The optional suffixes to this option are as follows:

`+abstract`

Enables abstract access through `vc_handles`.

`+allhdrs`

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

`+list`

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

---

## Options for Flushing Certain Output Text File Buffers

When VCS creates a log, VCD, or text file specified with the `$fopen` system function, VCS writes the data for the file in a buffer and periodically dumps the data from the buffer to the file on disk. The frequency of these dumps varies depending on many factors including the amount of data that VCS has to write to the buffer as simulation or compilation progresses. If you need to see or use the latest information in these files more frequently than the rate at which VCS normally flushes this data, these options tell VCS to flush the data more often during compilation or simulation.

`+vcs+flush+log`

Increases the frequency of flushing both the compilation and simulation log file buffers.

`+vcs+flush+dump`

Increases the frequency of flushing all VCD file buffers.

`+vcs+flush+fopen`

Increases the frequency of flushing all the buffers for the files opened by the `$fopen` system function.

`+vcs+flush+all`

Shortcut option for entering all three of the `+vcs+flush+log`, `+vcs+flush+dump`, and `+vcs+flush+fopen` options.

These options do not increase the frequency of dumping other text files, including the VCDE files specified by the `$dumports` system task or the simulation history file for LSI certification specified by the `$lsi_dumports` system task.

These options can also be entered at runtime. Entering them at compile-time modifies the `simv` executable so that it runs as if these options were always entered at runtime.

---

## Options for Simulating SWIFT VMC Models and SmartModels

`-lmc-swift`

Includes the LMC SWIFT interface.

`-lmc-swift-template`

Generates a Verilog template for a SWIFT Model.

---

## Options for Controlling Messages

**-error**

Revises the `+lint` and `+warn` options, to control error and warning messages. With them you can:

- Disable the display of any lint, warning, or error messages
- Disable the display of specific messages
- Limit the display of specific messages to a maximum number that you specify

For more details, See "["Error/Warning/Lint Message Control"](#)".

Note:

The `-error` option is also a runtime option. However, only the following feature is supported at runtime:

`-error=[no]message_ID[:max_number],...`

**-nc**

Suppresses the Synopsys copyright message.

**-suppress**

Disables the display of error and warning messages. For details, see "["Error/Warning/Lint Message Control"](#)".

**+sdfverbose**

By default, VCS displays no more than ten warning and ten error messages about back-annotating delay information from SDF files. This option enables the display of all back-annotation warning and error messages.

This default limitation on back-annotation messages applies only to messages displayed on the screen and written in the simulation log file. If you specify an SDF log file in the `$sdf_annotate` system task, this log file receives all 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 as follows:

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.

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. Additionally, these options have in common 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

**-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:

`-no_error UPIMI+IOPCWM`

This option does not work with the ID for any other error message.

**-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. VCS treats all SDF error messages as warning messages so including `+warn=none` disables SDF error messages.
- all      Specifies enabling all warning messages, IDs that follow preceded by the keyword `no`, in a comma separated list, specify exceptions.

The following are examples that show how to use this option:

|                      |                                                                     |
|----------------------|---------------------------------------------------------------------|
| +warn=noIPDW         | Enables all warning messages except the warning with the IPDW ID.   |
| +warn=none,TFIPC     | Disables all warning messages except the warning with the TFIPC ID. |
| +warn=noIPDW,noTFIPC | Disables the warning messages with the IPDW and TFIPC IDs.          |
| +warn=all            | Enables all warning messages. This is the default.                  |

In cases where both `-error` and `+warn` for the same ID are used on the command line in order to downgrade the error to warning and at the same time suppress warning, the order in which they are specified on the command line also impact the compilation. VCS processes the options based on the order they are specified in the command line. If the `+warn=no<ID>` option follows the `-error=no<ID>` option, then the `+warn=no<ID>` option can take effect because the `-error=no<ID>` has already downgraded it to warning. Otherwise, the option has no use. So, if `-error=no<ID>` follows the `+warn=no<ID>`, you might not see the error and the warning is suppressed.

To suppress an error, always use `-suppress=ID`, which actually combines the functionality of `-error=no<ID>` and `+warn=no<ID>` together.

`+error+count`

Enables you to increase the VCS elaboration error count limit. By default, VCS stops elaboration after reaching 10 errors.

When you use the `+error+count` compile-time option, VCS stops elaboration once the number of errors reaches the count limit with the following note:

```
Note-[MAX_ERROR_COUNT] Maximum error count reached  
Current number of errors has reached the default maximum  
error count (12).  
Use +error+<count> to increase the limit.
```

**Note:**

This option is available only for Vlogan and VCS commands. Also note that, when VCS encounters a fatal error (for example, syntax error), it exits immediately irrespective of the count that you have specified in the `+error+count` option.

---

## Option to Run VCS in Syntax Checking Mode

VCS runs in multiple stages, such as parsing, and compilation/elaboration stage.

To make sure VCS quits normally before all syntax and semantic issues are checked, you need to enable the parsing stage using the option `-parse_only`.

```
% vcs -parse_only <other options>
```

This enables the VCS to run only during the parsing stage to check for syntax errors. Any syntax errors found is reported. Regardless of whether any syntax error is reported or not, VCS stops at the end of the parsing stage. The link or elaboration stages are not run. Hence, no link or elaboration errors are reported when the `-parse_only` option is used. Also, `simv` executable is not generated.

For example, consider the following the testcase `test.v`:

```
module top;  
  
task one(input string fldName, input int bus = 0, input string  
fldName2);
```

```

$display("In one");
endtask

task two(input int bus = 0,input string fldName);
    $display("In two");
endtask

initial
begin
    one(.bus(1));
    one(,1);
    two(.bus(1));
end
endmodule

```

If there are errors, VCS exits with the following error message:

```

Parsing design file 'test.v'

Error-[TFAFTC] Too few arguments to function/task call
error.v, 9
"one(.bus(1));"
The above function/task call is not done with sufficient
arguments.

```

If there are no errors, then VCS exits normally as follows:

```

Parsing design file 'test.v'
Top Level Modules:
top
No TimeScale specified

Note-[PARSE-ONLY] VCS Parse-Only Mode
No syntax or semantic error detected after parsing the
complete design, VCS exits normally without generating
executable.

```

## **Limitations**

The option has the following limitations:

- When `-error=<Warn_ID>` option is used, VCS quits prior to the normal quit point because VCS upgrades the warning to error. Also, when `-error=<Warn_ID>` option is used, VCS may still complete compilation and quit normally even if the specified error exists, as the error is downgraded to warning.
- 

## Options for Cell Definition

The `-debug_access` and `-debug_region` options internally handle the options for cell definition. If you compile your design with the `-debug_access` and `-debug_region` options along with any of the options for cell definition, a warning message is issued and the options for cell definition are ignored.

Following are the options for cell definition:

`+nolibcell`

Does not define as a cell modules defined in libraries unless they are under the `'celldefine` compiler directive.

`+nocelldefinepli+0`

Enables recording in VPD files, the transition times and values of nets and registers in all modules defined under the `'celldefine` compiler directive or defined in a library that you specify with the `-v` or `-y` options. This option also enables full PLI access to these modules.

`+nocelldefinepli+1`

Disables recording in VPD files, the transition times and values of nets and registers in all modules defined under the `'celldefine` compiler directive. This option also disables full PLI access to these modules. Modules in a library file or directory are not affected by this option unless they are defined under the `'celldefine` compiler directive.

`+nocelldefinepli+2`

In VPD files, disables recording the transition times and values of nets and registers in all modules defined under the `'celldefine` compiler directive or defined in a library that you specify with the `-v` or `-y` options, whether the modules in these libraries are defined under the `'celldefine` compiler directive or not. This option also disables PLI access to these modules.

Disabling recording of transition times and values of the nets and registers in library cells can significantly increase simulation performance.

Note:

Disabling recording transitions in library cells is intended for batch simulation only and not for interactive debugging with GUI.

`+nocelldefinepli+1+ports`

Removes the PLI capabilities from `'celldefine` modules but allows PLI access to port nodes and parameters.

`+nocelldefinepli+2+ports`

Removes the PLI capabilities from library and `'celldefine` modules and allows PLI access to port nodes and parameters.

---

## Options for Licensing

-licwait timeout

Enables VCS to retry for a license until timeout expires, where timeout is the time in minutes.

You can set this license option as follows:

```
% vcs -licwait 10 <other compile options>
```

Here, VCS waits for the license for 10 minutes.

-licqueue

Tells VCS to try for the license till it finds the license. If there are multiple jobs asking for a license, then any one of those jobs get the license (similar to the older or the deprecated option +vcs+lic+wait).

You can set this license option as follows:

```
% vcs -licqueue <other compile options>
```

VCS\_LICENSE\_WAIT

You must set the VCS\_LICENSE\_WAIT variable to 1 and use the -licqueue option, which enables the license wait. Thus, the first job to enter the queue gets the license when the license is available. This option must be used along with the -licqueue option.

-ID

Returns useful information about a number of things: the version of VCS that you have set the VCS\_HOME environment variable to, the name of your work station, your workstation's platform, the host ID of your workstation (used in licensing), the version of the VCS compiler (same as VCS) and the VCS build date.

---

## Options for Controlling the Linker

`-ld linker`

Specifies an alternate front-end linker. Only applicable in incremental compile mode, which is the default.

`-LDFLAGS options`

Passes flag options to the linker. Only applicable in incremental compile mode, which is the default.

`-c`

Tells VCS to compile the source files, generate the intermediate C, assembly, or object files, and compile or assemble the C or assembly code, but not to link them. Use this option if you want to link by hand.

`-lname`

Links the *name* library to the resulting executable. Usage is the letter `l` followed by a name (no space between `l` and *name*). For example: `-lm` (instructs VCS to include the math library).

`-Marchive=number_of_module_definitionst`

By default, VCS compiles module definitions into individual object files and sends all the object files in a command line to the linker. Some platforms use a fixed-length buffer for the command line, and if VCS sends too long a list of object files, this buffer overflows and the link fails. A solution to this problem is to have the linker create temporary object files containing more than one module definition so there are fewer object files on the linker command line. With this option, you enable creating these temporary object files and specify how many module definitions are in these files.

Using this option briefly doubles the amount of disk space used by the linker because the object files containing more than one module definition are copies of the object files for each module definition. After the linker creates the `simv` executable, it deletes the temporary object files.

`-picarchive`

VCS can fail during linking due to the following two reasons:

- Huge size of object files: VCS compiles the units of your design into object files, then calls the linker to combine them together. Sometimes the size of a design is large enough that the size of text section of these object files exceeds the limit allowed by the linker. If so, the linker fails and generates the following error:

```
relocation truncated to fit:....
```

- Large number of object files: By default, VCS compiles module or entity definitions into individual object files and sends this list of object files in a single command line to the linker. Some platforms use a fixed-length buffer for the command line. If VCS sends a long list of object files, this buffer overflows and the link fails, generating errors such as:

```
make: execvp: gcc: Argument list too long
```

```
make: execvp: g++: Argument list too long
```

You can use the `-picarchive` option to deal with the above linker errors. The `-picarchive` option does the following:

- 1 . Enables Position Independent Code (PIC) object file generation along with linking the shared object version of VCS libraries.
- 2 . Archives generated PIC code into multiple shared objects inside `simv.daidir` or `simv.db.dir` directory.
- 3 . Links the Shared objects at runtime to the final executable, instead of linking all the objects statically into final executable in a single step at compile-time.

---

## Options for Controlling the C Compiler

`-cc compiler`

Specifies an alternate C compiler.

`-CC options`

Passes options to the C compiler or assembler.

`-CFLAGS options`

Passes options to C compiler. Multiple `-CFLAGS` are allowed. Allows passing of C compiler optimization levels. For example, if your C code, `test.c`, calls a library file in your VCS installation under `$VCS_HOME/include`, use any of the following `CFLAGS` option arguments:

```
%vcs top.v test.c -CFLAGS "-I$VCS_HOME/include"
```

or

```
%setenv CWD `pwd`  
%vcs top.v test.c -CFLAGS "-I$CWD/include"
```

or

```
%vcs top.v test.c -CFLAGS "-I../include"
```

Note:

The reason to enter ".../include" is because VCS creates a default `csrc` directory where it runs gcc commands. The `csrc` directory is under your current working directory. Therefore, you need to specify the relative path of the `include` directory to the `csrc` directory for gcc C compiler. Further, you cannot edit files in the `csrc` because VCS automatically creates this directory.

`-cpp`

Specifies the C++ compiler.

Note:

If you are entering a C++ file or an object file compiled from a C++ file on the `vcs` command line, you must tell VCS to use the standard C++ library for linking. To do this, enter the `-lstdc++` linker flag with the `-LDFLAGS` elaboration option.

For example:

```
vcs top.v source.cpp -P my.tab \  
-cpp /net/local/bin/c++ -LDFLAGS -lstdc++
```

`-j number_of_processes`

Specifies the number of processes that VCS forks for parallel compilation. There is no space between the "j" character and the number. You can use this option when generating intermediate C files (-gen\_c) and their parallel compilation.

-C

Stops after generating the C code intermediate files.

-O0

Suppresses optimization for faster compilation (but slower simulation). Suppresses optimization for how VCS both writes intermediate C code files and compiles these files. This option is the uppercase letter "O" followed by a zero with no space between them.

-O*number*

Specifies an optimization level for how VCS both writes and compiles intermediate C code files. The number can be in the 0-4 range; 2 is the default, 0 and 1 decrease optimization, 3 and 4 increase optimization. This option is the uppercase letter "O" followed by 0, 1, 2, 3 or 4 with no space between them. See above for additional information regarding the -O0 variant.

-override-cflags

Tells VCS not to pass its default options to the C compiler. By default, VCS has a number of C compiler options that it passes to the C compiler. The options it passes depends on the platform, whether it is a 64-bit compilation, whether it is a mixed HDL design, and other factors. VCS passes these options and then passes the options you specify with the -CFLAGS compile-time option.

---

## Options for Source Protection

For information about source protection options, see chapter "[Encrypting Source Files](#)".

---

## Options for Mixed Analog/Digital Simulation

Following are the options for mixed analog/digital simulation:

`-ad=[initfile]`

Enables the mixed-signal feature. If `-ad` is used alone, the mixed-signal control file name is `vcsAD.init`, by default. If the file name is different, it must be given with the `=initFile` option.

The mixed-signal simulation control file contains all the commands to configure mixed-signal simulation.

`-ams_discipline discipline_name`

Specifies the default discrete discipline in Verilog AMS.

`-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.

```
+verilogamsext+vams
```

To avoid keyword conflicts between Verilog-AMS and SystemVerilog, it is preferable that Verilog-AMS and SystemVerilog code each get parsed separately using their own language parsers. Create all Verilog-AMS and SystemVerilog files with distinct extensions.

For example, "`*.vams`" can be used for Verilog-AMS files and "`*.v`", "`*.sv`", or "`*.svh`" for SystemVerilog. The following VCS options can be used to identify these file extensions as the differentiation between SystemVerilog and Verilog-AMS contexts:

```
% vcs -ams -ad +verilogamsext+vams \
+systemverilogext+sv+v+svh ...
```

---

## Unified Option to Change Generic and Parameter Values

`-gfile cmdfile`

Overrides the default values for design generics and parameters by using values from the file `cmdfile`. The `cmdfile` file contains assign commands targeting design generics and parameters.

The syntax for a line in the file is as follows:

```
assign value path_to_parameter/generic
```

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

---

## Options for Changing Parameter Values

`-pvalue+parameter_hierarchical_name=value`

Changes the specified parameter to the specified value.

`-parameters filename`

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

`assign value path_to_parameter`

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

---

## Checking for x and z Values in Conditional Expressions

`-xzcheck [nofalseneg]`

Checks all the conditional expressions in the design and displays a warning message every time VCS evaluates a conditional expression to have an x or z value.

`nofalseneg`

Suppress the warning message when the value of a conditional expression transitions to an x or z value and then to 0 or 1 in the same simulation time step.

---

## Options for Detecting 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. For more information, see "[The Dynamic Race Detection Tool](#)".

Note:

The `-race` compilation/elaboration option supports dynamic race detection for both pure Verilog and SystemVerilog data types.

`-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. For more information, see "[The Dynamic Race Detection Tool](#)".

The `-racecd` compilation/elaboration option supports dynamic race detection for both pure Verilog and SystemVerilog data types.

`+race=all`

Analyzes the source code during compilation to look for coding styles that cause race conditions. For more information, see "[The Static Race Detection Tool](#)".

The `+race=all` option supports only pure Verilog constructs.

---

## Options to Specify the Time Scale

You can use the following options to specify the time scale:

`-timescale=time_unit/time_precision`

Occasionally, some source files contain the `timescale compiler directive and others do not. In this case, if you specify the source files that do not contain the `timescale compiler directive on the command line before you specify the ones that do, this is an error condition and VCS halts compilation, by default. This option enables you to specify the timescale for the source files that do not contain this compiler directive and precede the source files that do. Do not include spaces when specifying the arguments to this option.

`-unit_timescale[=<default_timescale>]`

The `-unit_timescale` option enables you to specify the default time unit for the compilation-unit scope. You must not include spaces when specifying arguments to this option.

The IEEE Standard 1800-2005 SystemVerilog LRM explains the time unit declaration, as follows:

*"The time unit of the compilation-unit scope can only be set by a time unit declaration, not a `timescale directive. If it is not specified, then the default time unit shall be used."*

Since the `-timescale` option does not affect the compilation-unit scope, you must use the `-unit_timescale` option to specify the default time unit for the compilation-unit scope.

The `default_timescale` value should be in the same format as the ``timescale` directive. If the default timescale is not specified, then `1s/1s` is taken as the default timescale of the compilation-unit.

`-override_timescale=time_unit/time_precision`

Overrides the time unit and precision unit for all the `timescale compiler directives in the source code, and, similar to the -timescale option, provides a timescale for all module definitions that precede the first `timescale compiler directive. Do not include spaces when specifying the arguments to this option.

`-time base_time`

Sets the time base for the simulation. This option overrides the default TIMEBASE variable value in the synopsys\_sim.setup file. The default value for *base\_time* is ns.

`-time_res value`

Sets the time resolution for the simulation. This option overrides the default TIME\_RESOLUTION variable value in the synopsys\_sim.setup file.

`-noinherit_timescale`

This is a VCS or vlogan option that allows you to specify a global timescale within the compilation unit of any source file that does not have an explicit `timescale directive.

If you specify the -noinherit\_timescale option in the command line and there is no time unit/time precision specification present inside the module, program, interface, or package definition, then the time unit/time precision is determined using the following rules:

- If the module or interface definition is nested, then the time unit/time precision shall be inherited from the enclosing module or interface.

- Else, if `timescale directive is previously specified within the file, then the time unit/time precision shall be set to that of the `timescale directive.
- Else, if the time unit/time precision is specified outside all other declarations within a compilation unit, then the time unit/time precision shall be set to that specification.
- Else, the global time unit/time precision specified by the –noinherit\_timescale option shall be used.

Similarly in presence of other timescale options specified in this document, the following is the precedence:

`-override_timescale > -noinherit_timescale > -timescale/-unit_timescale`

Therefore, when the combination of options are used, the behavior is as follows:

- The `-override_timescale` option has the highest precedence. Therefore, the timescale specified by the `-noinherit_timescale` option is overridden by the `-override_timescale` option.
- If the `-timescale` (or the `-unit_timescale`) and the `-noinherit_timescale` options are used in the same command line, timescale from the `-noinherit_timescale` option shall be used and `-timescale` (or `-unit_timescale`) option is ignored.

- In case of three-step flow, if the `-timescale` (or `-unit _timescale`) and the `-noinherit _timescale` options are used in different command line, such as one is in `vlogan` and another is in `vcs` command line, timescale specified at `vlogan` stage cannot be overridden at `vcs` stage (except for the case when you use the `-overridde _timescale` option).

**Example C-1 Example for `-noinherit _timescale` Option**

| a.v                                                         | b.v                                                                                               | c.v                                                                                            |
|-------------------------------------------------------------|---------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------|
| <pre>module A; ... endmodule  module B; ... endmodule</pre> | <pre>`timescale 1ns/1ns module X; ... endmodule  `timescale 1ns/1ns module Y; ... endmodule</pre> | <pre>module P; timeunit 100ps; timeprecision 1ps; ... endmodule  module Q; ... endmodule</pre> |

You can use the following `vcs` and `vlogan` command lines:

```
% vcs -noinherit _timescale="10ps/1ps" a.v b.v c.v
```

Or

```
% vcs -noinherit _timescale="10ps/1ps" b.v c.v a.v
```

Or

```
% vlogan -noinherit _timescale="10ps/1ps" c.v a.v c.v
```

Any of the above commands result into the following timescale:

- module A - 10ps/1ps

- module B - 10ps/1ps
- module P - 100ps/1ps
- module Q - 10ps/1ps
- module X - 1ns/1ns
- module Y - 1ns/1ns

### **Key Points to Note**

- The files that are read using a `include directive inherit the timescale of the file containing the `include directive. That is, the global timescale is only applied to files that are not read using an `include directive.
- Once the file is compiled or analyzed, the timescale of a module is not changed. This means that the timescale of a logic library is determined when the library is compiled, not when one of its module is incorporated using the -y option.

### **Option to Exclude Environment Variables During Timestamp Checks**

`-vts_ignore_env=ENV1,ENV2,...`

You can use the `-vts_ignore_env=ENV1,ENV2,...` compile-time option to exclude certain environment variables from incremental compilation during VCS timestamp checks.

Consider the following testcase test.v:

```
module test
endmodule
```

Run the following commands:

```
%setenv myenv1 1  
%setenv myenv2 2  
% vcs test.v -vts_ignore_env=myenv1,myenv2  
  
//Incremental compilation: There are no source code  
changes. Only the environment variables "myenv1" and  
"myenv2" are changed  
  
%unsetenv myenv1  
%unsetenv myenv2  
  
% vcs test.v -vts_ignore_env=myenv1,myenv2
```

Following is the output:

The design hasn't changed and need not be recompiled.  
If you really want to, delete file simv.daidir/.vcs.timestamp and run VCS again.

Note:

The `-vts_ignore_env=ENV1,ENV2,...` option is not supported in three-step (UUM) flow. It is only supported in two-step flow.

---

## Options for Overriding Parameters

`-gfile`

You can use the `-gfile` compile-time option to override parameter and generic values through a file for both Verilog and VHDL respectively.

You must specify the file name, which contains the list of all generics and parameters that should be overridden, with the `-gfile` option.

The syntax for `-gfile` option is as follows:

```
% vcs top_level_entity_or_module -gfile  
parameters_or_generics_file other_options
```

The syntax for the `parameters_or_generics_file` is as follows:

```
assign val path
```

Where,

`val`: The value that overrides the specified parameter/generic.

`path`: Specifies the absolute hierarchical path to the parameter/generic value which is to be overridden.

Note:

The `-gfile` supports only VHDL syntax for hierarchical path representation.

All escaped identifiers in the Verilog path must be converted into VHDL extended identifiers. If the escaped identifier contains ‘\’ characters, they must be escaped with another ‘\’ character.

For example, consider the following Verilog hierarchical path for the parameter ‘P1’.

```
top.dut.\inst1_\cpu .inst2.P1
```

The corresponding `generics_file` entry is as follows:

```
assign    'hffffffff /top/dut/\inst1_\\cpu\\  
inst2/P1
```

All ‘for-generate’ and ‘instance-array’ parentheses must be round parentheses, and the path delimiter must be ‘/’. All instance paths for VHDL-Top and Verilog-Top designs must start with ‘/’.

### **Example:**

You can override the parameter and generic values using the `-gfile` option as follows:

```
% vcs vh_top -gfile overrides.txt
```

Where, `overrides.txt` contains the following entries:

```
assign      'hfffffffff /top/dut/\inst1_\\cpu\\/
inst2/P1

assign      "DUMMY" /top/dut/\inst1_\\cpu\\inst2/
P2

assign      10.34 /top/dut/\inst1_\\cpu\\inst2/P3
```

### **Supported Data Types:**

The following data types are supported with the `-gfile` option:

- Integer
- Real
- String

The `-gfile` option ignores other data types with a suitable warning message.

`-pvalue`

You can use the `-pvalue` compile-time option for changing the parameter values from the `vcs` command line.

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

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

### **Example:**

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

#### **Note:**

The `-pvalue` option does not work with a `localparam` or a `specparam`.

`-pvalue_nonlocal`

When `param_assignments` appear in a `module_parameter_port_list` and you specify the `-pvalue_nonlocal`, then any `param_assignments` that appear in the module become local parameters and shall not be overridden by any method (such as `-pvalue`).

Consider the following example (`test.v`):

```
module sub1 ();
    parameter dly = 2; // Non - Local Parameter
endmodule

module sub2 #(parameter delay = 42) ();
    parameter dly = 12; // Local Parameter As Per Verilog LRM
endmodule

module top;
    sub1 s1 ();
    sub2 #(delay(35)) s2 ();
endmodule
```

When you compile the example as follows:

```
% vcs -sverilog -pvalue+dly=20 test.v
```

VCS overrides the dly parameters from both sub1 and sub2 modules.

When you compile the example as follows:

```
% vcs -sverilog -pvalue+dly=20 -pvalue_nonlocal  
test.v
```

VCS overrides the dly parameters from sub1 module only.

*-pvalue+pkg1::WIDTH=3 -pvalue+pkg2::WIDTH=4*

You can override the package parameters during design elaboration using the -pvalue option and package scope resolution.

Consider the following example (test.v):

```
package pkg1;  
    parameter WIDTH = 0;  
endpackage  
  
package pkg2;  
    parameter WIDTH =1;  
endpackage  
  
module top;  
    initial  
    begin  
        $display("pkg1 WIDTH = %0d",pkg1::WIDTH);  
        $display("pkg2 WIDTH = %0d",pkg2::WIDTH);  
    end  
endmodule
```

When you compile the example as follows:

```
% vcs -sverilog -pvalue+pkg1::WIDTH=3  
-pvalue+pkg2::WIDTH=4 test.v  
  
. /simv
```

VCS generates the following output:

```
pkg1 WIDTH = 3  
pkg2 WIDTH = 4
```

**-gv | -gvalue generic=value**

Overrides the generic value defined in the source code with the value specified in the command line.

### **Example:**

```
vcs work.top -gvalue /TOP/LEN=1
```

### **Note:**

The **-gv | -gvalue** option overrides the generic value defined in the source code only if the generic is of type integer or real.

**-g | -generics cmdfile**

Overrides the default values for the design generics by using values from the file **cmdfile**. The file **cmdfile** is an include file that contains assign commands targeting design generics.

---

## Global -check\_all Option

You can use the `-check_all` option to run all of the additional semantic checks that are traditionally run by `-check` option for VHDL and `-boundscheck` and `-ntb_opts check=all` options for SystemVerilog.

The `-check_all` option enables the available checks namely `-check` at runtime, `-boundscheck` at compile-time, `-boundscheck` at runtime, and `-ntb_opts check=all` at compile-time. Thus, warning messages are generated for fixed-size arrays and error messages are generated for variable-sized arrays.

## Use Model

The option is supported in both VCS two-step and three-step flows.

For two-step flow and three-step flow, add the option `-check_all` in the VCS command line.

Consider the following test case:

```
module test;
  logic arr[10];
  logic index,sel;
  logic [3:0] in1,in2;
  wire [3:0] out;
  initial
  begin
    in1 = 4'b1100; in2 = 4'b1000;
    #1;
    $display(arr[index]);
    $displayb(out);
  end
  DUT inst (sel,in1,in2,out);
endmodule : test
```

```

module DUT (sel,in1,in2,out);
  input sel;
  input [3:0] in1,in2;
  output logic [3:0] out;
  always @(sel,in1,in2)
    if(sel)
      out = in1;
    else
      out = in2;
endmodule : DUT

```

If you run the test case with `-check_all` option, VCS generates the following warning message:

```

Warning-[AAII] Array access with indeterminate index
test.v, 10
Index value bits set to x or z. The array read "arr[1'bx]"
has indeterminate index value.
Simulation time = 1

```

The warning message can be upgraded to an error message using the `-error=AAII` option.

## **Limitation**

The feature has the following limitation:

- If you run the test case with the `-check_all` and `-xprop` options, the `-check_all` option is ignored and VCS generates the following warning message:

```

Warning-[XPROP-IGNORE-CHECK_ALL] -check_all is ignored in
presence of Xprop switches

```

---

## Option to Enable Bounds Check at Compile-Time

-boundscheck

Enables the compile-time check for two-dimensional or three-dimensional arrays with packed dimensions. The following warning message is displayed during compile-time:

Warning-[SIOB] Select index out of bounds

This compile-time warning message is displayed in case of out-of-bounds condition.

### Example

```
module tb();

reg [1:0][1:0] fifo_data[2:0]; // FIFO memory
reg some_bit; // temp signal

reg nibble[]; // Dynamic array

initial
begin
//Packed dimension
some_bit = fifo_data[1][1][2];

end
endmodule
```

The following warning message is displayed:

```
Warning-[SIOB] Select index out of bounds
test2.sv, 11 "fifo_data[1][1][2]"
The select index is out of declared bounds : [1:0] in
module : tb.
```

---

## Option to Enable Bounds Check at Runtime

You can use the following option to enable bounds check at runtime:

-boundscheck

This option enables the runtime checks for the out-of-bounds and intermediate index access of fixed size and variable size unpacked arrays, dynamic arrays, and queues. Warning messages are generated for fixed size arrays, and error messages are generated for variable size arrays.

The following error messages or warning messages are displayed during runtime out-of-bounds index access:

- Error- [DT-OBAE] Out of bound access for queues
- Error- [DT-OBAE] Out of bound access for dynamic arrays
- Warning- [AOOBAW] Out of bound access for fixed size unpacked arrays
- Warning- [AOOBAW] Out of bound access for fixed size packed arrays

### Error-[DT-OBAE] Out of Bounds Access for Queues

This runtime error message is displayed, if a queue element is accessed with an out-of-bound index.

#### Example

```
module tb();  
  
    int      temp;          // temp signal  
    int int_queue[$] = { 1, 2, 3}; //Queue
```

```

initial
begin
  //Queue
  temp = int_queue[9];
end
endmodule

```

The following error message is displayed:

```

Error-[DT-OBAE] Out of bound access
test2.sv, 9
Out of bound access on smart queue (size:3, index:9).
Simulation time = 0
Please make sure that the index is positive and less than
size.

```

## Error-[DT-OBAE] Out of Bounds Access for Dynamic Arrays

This runtime error message is displayed, if a dynamic array element is accessed with an out-of-bound index.

### Example

```

module tb();

reg      some_bit;           // temp signal
reg nibble[]; // Dynamic array
int int_queue[$] = { 1, 2, 3}; //Queue

initial
begin
  //Dynamic array
  nibble = new[3];          // Create a 3-element array.
  some_bit = nibble[3];
end
endmodule

```

The following error message is displayed:

```
Error-[DT-OBAE] Out of bound access
test2.sv, 11
Out of bound access on dynamic array (size:3, index:3).
Simulation time = 0
Please make sure that the index is positive and less than
size.
```

## **Warning-[AOOBAW] Array Out of Bounds Access for Fixed Size Unpacked Arrays**

This runtime warning message is displayed, if a fixed size unpacked array element is accessed with an out-of-bound index.

### **Example**

```
module tb();

reg [1:0] fifo_data[2:0]; // fifo memory
reg [1:0] some_signal; // temp signal

initial
begin
//Unpacked dimension
some_signal = fifo_data[3];
end
initial $monitor("some_signal = %b ",some_signal);
endmodule
```

The following warning message is displayed:

```
Warning-[AOOBAW] Array out of bounds access
test2.sv, 7
Array read "fifo_data[3]" is out of bounds.
Simulation time = 0
Please make sure index is within range. To disable this error
message, please remove '-boundscheck' at compile time.
```

## **Warning-[AOOBW] Array Out of Bounds Access for Fixed Size Packed Arrays**

This runtime warning message is displayed, if a fixed size packed array element is accessed with an out-of-bound index.

### **Example**

```
module tb();
reg [7:0] rd_ptr;
reg [29:0] nibble;
reg some_signal3;
initial
begin
rd_ptr = 40;
#1 $finish();
end
assign some_signal3 = nibble[rd_ptr];
initial $monitor("some_signal3 = %b",some_signal3);
endmodule
```

The following warning message is displayed:

```
Warning-[AOOBW] Array out of bounds access
test2.sv, 11
Array read "nibble[40]" is out of bounds.
Simulation time = 0
Please make sure index is within range. To disable this error
message, please remove '-boundscheck' at compile time.
```

The following error messages or warning messages are displayed during indeterminate index access, where array index has value X or Z:

- Error-[DT-OBAD] Intermediate access for dynamic arrays
- Warning-[AAIIW] Array access with intermediate index

- Warning-[AAIIW] Array access with intermediate index for fixed size packed arrays

## Error-[DT-OBAE] Intermediate Access for Dynamic Arrays

This runtime error message is displayed, if a dynamic array element is accessed with an index value X or Z.

### Example

```
module tb();
reg [7:0] rd_ptr = 8'bxxxxxxxxx;
reg nibble[]; // Dynamic array
reg some_signal3; // temp signal

initial
begin
//Dynamic array
nibble = new[3]; // Create a 3-element array.
some_signal3 = nibble[rd_ptr];
end
endmodule
```

The following error message is displayed:

```
Error-[DT-OBAE] Out of bound access
test2.sv, 10
Out of bound access on dynamic array (size:3, index:255)
Simulation time = 0
Please make sure that the index is positive and less than
size.
```

## Warning-[AAIIW] Array Access with Intermediate Index

This runtime warning message is displayed, if a fixed size unpacked array element is accessed with an index value X or Z.

## Example

```
module tb();
reg [17:0] fifo_data[255:0]; // fifo memory
reg [7:0] rd_ptr;
wire [7:0] some_signal; // temp signal
initial begin
    rd_ptr = 8'dx;
    #1 $finish();
end

assign some_signal = fifo_data[rd_ptr];
initial $monitor("some_signal = %b", some_signal);

endmodule
```

The following warning message is displayed:

```
Warning-[AAIIW] Array access with indeterminate index
test2.sv, 9
Index value bits set to x or z. The array read
"fifo_data[1'bx]" has indeterminate index value.
Simulation time = 0
To disable this warning message, please remove '-boundscheck'
at compile time. To upgrade this warning to error, add "-
error=AAII" to simv runtime command.
```

## Warning-[AAIIW] Array Access with Intermediate Index for Fixed Size Packed Arrays

This runtime warning message is displayed, if a fixed size packed array element is accessed with an index value X or Z.

## Example

```
module tb();
reg [7:0] rd_ptr;
reg [29:0] nibble;
reg some_signal3;
initial
```

```

begin
    rd_ptr = 8'bxxxxxxxx;
    #1 $finish();
end
assign some_signal3 = nibble[rd_ptr];
initial $monitor("some_signal3 = %b", some_signal3);
endmodule

```

The following warning message is displayed:

```

Warning-[AAIIW] Array access with indeterminate index
test2.sv, 10
Index value bits set to x or z. The array read "nibble[1'bx]"
has indeterminate index value.
Simulation time = 0
To disable this warning message, please remove '-boundscheck'
at compile time. To upgrade this warning to error, add "-
error=AAII" to simv runtime command.

```

## Option to Enable Extra Runtime Checks in VHDL

You can use the following option to enable extra runtime checks in VHDL:

`-check`

This option enables the extra runtime checks in the VHDL code. The following error messages are displayed during runtime:

- Error-[SIMERR\_FPEXCEPTION] Floating point exception
- Error-[SIMERR\_FDIVZERO\_SCOPE] Divide by Zero Error
- Error-[SIMERR\_NEGTIME] Wrong time format
- Error-[SIMERR\_CONSTERR\_INDEXEDRANGE] Range Violation Error

- Error-[SIMERR\_CONSTERR] Array constraint error
- Error-[SIMERR\_INCONSISTENTIS] Subtype constraints inconsistencies
- Error-[SIMERR\_INCONSISTENTIC] Incorrect binding range
- Error-[SIMERR\_SIZEMISMATCH] Array Size Mismatch

## **Error-[SIMERR\_FPEXCEPTION] Floating Point Exception**

This runtime error message is displayed when the value corresponding to the mathematical result is overflow or underflow.

### **Example**

```
entity e is
end;

architecture a of e is
begin
process
    variable r1 : real;
begin
    r1:=real'high;
    r1:=r1 + r1;
    wait;
end process;
end a;
```

The following error message is displayed:

```
Error-[SIMERR_FPEXCEPTION] Floating point exception
floating_point_overflow.vhd, 10 simulation-Sim r1:= r1 + r1;

Floating point exception during a floating point operation
at 0 NS from process /E/_P0.
Please make sure the result of the operation is not overflow
```

or underflow

## Error-[SIMERR\_FDIVZERO\_SCOPE] Divide by Zero Error

This runtime error message is displayed when the divisor is zero in floating point operation.

### Example:

```
entity e is
end;

architecture a of e is
begin
process
variable v1,v2,v : real;
begin
-- real divide by Zero :
v1 := 1.1 ;
v2 := 0.0;
v := v1/v2;
end process;
end a;
```

The following error message is displayed:

```
Error-[SIMERR_FDIVZERO_SCOPE] Divide by Zero Error
divide_by_zero.vhd, 12 v:= v1/v2;
Error in floating point operation, divide by zero in scope:/E/_P0 at 0 NS from process /E/_P0.
```

## Error-[SIMERR\_NEGTIME] Wrong Time Format

This runtime error message is displayed when a waveform element evaluates to a negative value.

## **Example**

```
entity e is
end e;

architecture a of e is
    signal s : bit;
begin
    process (s)
        variable d : time := 10 ns;
    begin
        report time'image(now);
        s <= not s after d;
        d := d - 11 ns;
    end process;
end a;
```

The following error message is displayed:

```
Error-[SIMERR_NEGTIME] Wrong time format negwav.vhd, 11
simulation-Sim s <= not s after d;
```

A negative time specification is not allowed in this context  
at 10 NS from process /E/\_P0.

## **Error-[SIMERR\_CONSTERR\_INDEXEDRANGE] Range Violation Error**

To execute a return statement, the expression (if any) is first evaluated and then converted to the result subtype. The execution of the return statement is completed, if the conversion succeeds. The runtime error message is displayed when the conversion fails.

## **Example**

```
entity e is
    generic (N : integer := 4);
end entity;

architecture a of e is
```

```

subtype t is integer range 1 to N;
function f return t is
begin
    return 5;
end function;
begin
    c: process
        variable x : t;
    begin
        x := f;
        wait;
    end process;
end architecture;

```

The following error message is displayed:

```

Error-[SIMERR_CONSTERR_INDEXEDRANGE] Range Violation Error
c2.vhd, 9 return 5;
Value 5 is out of range 1 to 4 at 0 NS from process /E/C.

```

Please extend the range in the type/subtype definition or ensure that the value is within the defined range.

## **Error-[SIMERR\_CONSTERR] Array Constraint Error**

After applying any conversion function or type conversion present in the actual part of the applicable association element, the value of the actual parameter should belong to the subtype that is denoted by the subtype indication of the formal value. The runtime error message is displayed, if the value of the actual parameter does not belong to the subtype.

### **Example**

```

entity e is
    generic (N : integer := 4);
end entity;

architecture a of e is

```

```

        subtype t is integer range 1 to N;
begin
    c: process
        variable x : t;
    begin
        x := 5;
        wait;
    end process;
end architecture;;

```

The following error message is displayed:

```

Error-[SIMERR_CONSTERR] Array constraint error
c1.vhd, 11 /E/C x:= 5;
Array constraint error in scope: /E/C at 0 NS from process
/E/C.

```

Either change the bounds of the object declaration so that the values you assign to it fall within the specified range, or change the source code so that correct values get assigned to the object.

## Error-[SIMERR\_INCONSISTENTIS] Subtype Constraints Inconsistencies

This runtime error message is displayed, if the constraints of the two subtypes do not match exactly.

### Example

```

entity e is
    generic (N : integer := 4);
end entity;

architecture a of e is
    subtype t is string(N downto 1);
    type a is access t;
begin
    P: process
        variable x : a;

```

```

begin
  x := new t;
end process P;
end architecture;

```

The following error message is displayed:

```

Error-[SIMERR_INCONSISTENTIS] Subtype constraints
inconsistencies c4_1.vhd, 12 x:= new t;
Two subtypes whose constraints must match exactly differed.
One is (4 downto 1), the other one is (1 to 4). The mismatch
occurs at 0 NS from process /E/P.

```

Please make sure that the constraints of the two subtypes match exactly.

## **Error-[SIMERR\_INCONSISTENTIC] Incorrect Binding Range**

This runtime error message is displayed due to incorrect binding range.

### **Example**

```

entity e is
end entity;

architecture a of e is
  procedure p (l, r : integer) is
    variable x : string(l to r);
  begin
    end procedure;
begin
  c: process
  begin
    p(0, 3);
    wait;
  end process;
end architecture;

```

The following error message is displayed:

```
Error-[SIMERR_INCONSISTENTIC] Incorrect binding range
c3.vhd, 6 variable x: string (l to r);

Binding range (0 to 3) is not consistent with declared range
(1 to 2147483647) at 0 NS from process /E/C.
```

## **Error-[SIMERR\_SIZEMISMATCH] Array Size Mismatch**

This runtime error message is displayed in a variable or signal assignment with zero delay.

**Scenario 1:** The error message is displayed:

- When the RHS is concatenated expression with dynamic sizes and LHS does not appear on RHS, and
- When the size of the LHS and the RHS does not match

### **Example**

```
entity e is
    generic (N : integer := 4);
end entity;

architecture a of e is
    subtype t is string(1 to N-1);
    function f return t is
    begin
        return "111";
    end function;
begin
    process
        variable x : string(1 to N);
    begin
        x := '1' & f & '1';
    end process;
end architecture;
```

The following error message is displayed:

```
Error-[SIMERR_SIZEMISMATCH] Array Size Mismatch  
c5.vhd, 15 x:= '1' & f & '1';
```

```
Array size of 4 on LHS does not match the array size of 5  
on RHS at 0 NS from process /E/_P0.
```

**Scenario 2:** The runtime error message is displayed:

- When the RHS is a function call with constraint array return type, and
- When the size of the LHS and the RHS does not match

### **Example**

```
entity e is  
    generic (N : integer := 4);  
end entity;  
  
architecture a of e is  
    subtype t is string(1 to N-1);  
    function f return t is  
    begin  
        return "111";  
    end function;  
begin  
    process  
        variable x : string(1 to N);  
    begin  
        x := f;  
    end process;  
end architecture;
```

The following error message is displayed:

```
Error-[SIMERR_SIZEMISMATCH] Array Size Mismatch
```

```
c5.vhd, 15 x:= f;  
  
Array size of 4 on LHS does not match the array size of 3  
on RHS at 0 NS from process /E/_P0.
```

---

## General Options

### Specifying Directories for ‘include’ Searches

+incdir+*directory*+

Specifies the directory or directories in which VCS searches for include files used in the `include compiler directive.

Files to be included and specified with the `include compiler directive are called included files. VCS searches for included files in the following order:

1. In the current directory
2. In the directories specified with this +incdir compile-time option.

You can specify more than one directory separated by the plus (+) character. For example:

+incdir+dir1+dir2

In this example subdirectories dir1 and dir2 are in the current directory.

+incdir+/file\_sys/server/design\_group/design\_lib

You can also specify an absolute path name.

## **Enable the VCS/SystemC Cosimulation Interface**

`-sysc`

Enables SystemC cosimulation engine.

`-sysc=adjust_timeres`

Determines the finer time resolution of SystemC and HDL in case of a mismatch, and sets it as the simulator's timescale. VCS may be unable to adjust the time resolution if you elaborate your HDL with the `-timescale` option or use the `sc_set_time_resolution()` function call in your SystemC code. In such cases, VCS reports an error and does not create simv.

**Note:**

You must use this option along with the `-sysc` option.

The `-sysc=adjust_timeres` option is not supported in two-step flow. It is only supported in three-step (UUM) flow.

## **TetraMAX**

`+tetramax`

Enables splitting of TetraMAX's large testbench to improve VCS capability and to reduce compile time.

## **Suppressing Port Coersion to inout**

`+noportcoerce`

Prevents VCS from coercing ports to inout ports, which is the default condition. This option is the equivalent of the `'noportcoerce` compiler directive.

## 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.

## Enabling Dumping

+vcs+vcdpluson

A compile-time substitute for the \$vcdpluson system task, the +vcs+vcdpluson option enables recording in the VPD file transition times and values for the entire design (except SystemVerilog memories and multi-dimensional arrays (MDAs) that have unpacked dimensions). If -debug\_access is not specified, then this option adds -debug\_access.

## Enabling Identifier Search

You can use the following elaboration options to enable and control the Search Identifiers feature:

- -genid\_db
- -nogenid\_db

- `-debug_access+idents_db`
- `-debug_access+all`

Use the `-genid_db` option in combination with a debug option, for example, as shown below, to enable Search Identifiers feature and prepare the internal search database.

```
% vcs -genid_db -debug_access+idents_db top.v
```

If you use `-genid_db` without a debug option, VCS issues a warning message saying that the feature is not enabled.

If you elaborate your design with `-debug_access+all`, but without `-genid_db`, then VCS creates the database during the first search query. This postpones most of the disk space and CPU overhead.

Specify `-nogenid_db`, if you want to completely avoid any disk space and CPU time overhead caused by Search Identifiers. You must use this option in combination with `-debug_access+all`.

## **Memories and Multi-Dimensional Arrays (MDAs)**

`+memcbk`

Enables callbacks for memories and multidimensional arrays (MDAs). Use this option if your design has memories or MDAs, and you are doing any of the following:

- Using the VCS/SystemC Interface.
- Writing an FSDB file for Verdi.

- 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.

Note:

The `+memcbk` option is enabled by default when the `-debug_access` option is used at compile-time. You can disable this option using the `-debug_access -nomemcbk` option.

## Specifying a Log File

`-l filename`

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

`-a logfilename`

Captures simulation output and appends the log information in the existing log file. If the log file doesn't exist, then this option would create a log file.

## Changing Source File Identifiers to Upper Case

`-u`

Changes all the characters in identifiers to uppercase. It does not change identifiers in quoted strings such as the first argument to the `$monitor` system task.

## 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.

Note:

The =value argument is optional.

For example:

```
vcs design.v +define+USETHIS
```

The macro is used inside the source file using the `ifdef compiler directive. If this macro is not defined using the +define option, then the else portion in the code takes priority.

```
`ifdef USETHIS
    package p1;
    endpackage
`else
    package p2;
    Endpackage
`endif
```

## Option for Macro Expansion

```
-p1800_macro_expansion
```

This option is used for LRM compliance to support macro expansion. This option produces results that are more LRM-compliant and accurate especially for SystemVerilog macros.

The syntax is:

```
% vcs [elab_options] t.sv -sverilog  
-p1800_macro_expansion
```

For example, consider the following testcase test.sv:

```
module top;  
logic [3:0] addr0_for_bank0='d10;  
`define VAR(ANUM,BNUM) addr`ANUM`_for_bank`BNUM  
`define NAME(STR) $display(`"\`"STR`\``" is %d\n`",STR);  
`define ARG addr0_for_bank0  
  
initial begin  
  `NAME(`VAR(0,0));  
  `NAME(`ARG)  
end  
endmodule
```

If you run the testcase without `-p1800_macro_expansion` option, VCS generates the following output:

```
"`VAR(0,0)" is 10  
"addr0_for_bank0" is 10
```

If you run the testcase with `-p1800_macro_expansion` option, VCS generates the following output:

```
"addr0_for_bank0" is 10  
"addr0_for_bank0" is 10
```

## Specifying the Name of the Executable File

`-o name`

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

## **Returning The Platform Directory Name**

`-platform`

Returns the name of the *platform* directory in your VCS installation directory. For example, when you install VCS on a Solaris version 5.4 workstation, VCS creates a directory named, `sun_sparc_solaris_5.4`, in the directory where you install VCS. In this directory are subdirectories for licensing, executable libraries, utilities, and other important files and executables. You need to set your path to these subdirectories. You can do so by using this option:

```
set path=($VCS_HOME/bin\  
$VCS_HOME/'$VCS_HOME/bin/vcs -platform'/bin$path)
```

## **Maximum Donut Layers for a Mixed HDL Design**

`-maxLayers value`

Sets the maximum number of donut layers for a mixed HDL design. The default value is 8.

## **Enabling feature beyond VHDL LRM**

`-x1rm`

Enables VHDL features beyond those described in *IIEEE Standard VHDL 1076-2008 LRM*.

## **Enabling Loop Detect**

`+vcs+loopreport+number`

It is mandatory to include the `+vcs+loopreport+number` option at compile-time, though the threshold number can be overridden at runtime.

When `+vcs+loopreport+number` is specified at compile time, VCS does the following based on the option specified at runtime:

- If *number* is not specified at runtime, VCS checks if the simulation event loops for 2,000,000 times (by default) in the same simulation time tick, and issues a runtime warning message. VCS also terminates the simulation and generates a report when a zero delay loop is detected.
- If `+vcs+loopreport+N` is specified at runtime, VCS checks if the simulation event loops for 'N' times instead of 2,000,000. VCS then issues a runtime warning message, and terminates the simulation.

For information about using the `+vcs+loopreport+number` option during runtime, see Section "[Enabling Loop Detect](#)" in Chapter "Simulation Options".

`+vcs+loopdetect+number`

When `+vcs+loopdetect+number` is specified at compile time, VCS does the following based on the option specified at runtime:

- If *number* is not specified at runtime, VCS checks if the simulation event loops for 2,000,000 times (by default) in the same simulation time tick, and issues a runtime error message. VCS also terminates the simulation.

- If `+vcs+loopdetect+N` is specified at runtime, VCS checks if the simulation event loops for 'N' times instead of 2,000,000. VCS then issues a runtime error message, and terminates the simulation.

For information about using the `+vcs+loopdetect+number` option during runtime, see Section "[Enabling Loop Detect](#)" in Chapter "Simulation Options".

## **Changing the Time Slot of Sequential UDP Output Evaluation**

`-nonbaudpsched`

By default, VCS evaluates the output terminals of the sequential UDP (user-defined primitive) in the NBA region. If the design is compiled with this switch, the output of sequential UDPs is scheduled in the active region of the scheduler.

## **Gate-Level Performance**

`-hsopt=gates`

Improves runtime performance on gate-level designs (both functional and timing simulations with SDF). You may see some compile-time degradation when you use this switch.

## **Option to Omit Compilation of Code Between Pragmas**

`-skip_translate_body`

Tells VCS to omit compilation of Verilog/SystemVerilog/VHDL code between the following:

```
the //synopsys translate_off or
/* synopsys translate_off */ pragma
```

and

```
the //synopsys translate_on or  
/* synopsys translate_on */ pragma
```

### **Example of SystemVerilog Code with Translate off**

The following SystemVerilog code example shows what this option can do:

```
module test;  
initial begin  
$display("\n before translate_off");  
//synopsys translate_off  
$display("\n after translate_off before translate_on");  
//synopsys translate_on  
$display("\n after translate_on before translate_off");  
//synopsys translate_off  
$display("\n 2nd after translate_off before  
translate_on");  
//synopsys translate_on  
$display("\n after translate_on\n");  
end  
endmodule
```

Without the `-skip_translate_body` option, VCS displays the following:

```
before translate_off  
  
after translate_off before translate_on  
  
after translate_on before translate_off  
  
2nd after translate_off before translate_on  
  
after translate_on
```

VCS compiles and executes all the `$display` system tasks.

With the `-skip_translate_body` option, VCS displays the following:

```
before translate_off  
after translate_on before translate_off  
after translate_on
```

VCS does not compile and execute the `$display` system tasks between the `//synopsys translate_off` and `//synopsys translate_on` pragmas.

### Example of VHDL code with Translate off

```
entity E1 is  
end entity E1;  
  
architecture A1 of E1 is  
begin  
    assert false report "before translate off"  
severity note;  
    -- synopsys translate_off  
    assert false report "after translate off before  
translate on" severity note;  
    -- synopsys translate_on  
    assert false report "after translate on before  
2nd time translate off" severity note;  
    -- synopsys translate_off  
    assert false report "after translate off before  
translate on" severity note;  
    -- synopsys translate_on  
    assert false report "after 2nd time translate  
on" severity note;  
end architecture A1;
```

### Command Line

```
% vhdlan E1.vhdl -skip_translate_body  
% vcs E1 -R
```

With the `-skip_translate_body` option, VCS displays the following message:

```
Assertion NOTE at 0 NS in design unit E1(A1) from process
  /\test/E1_inst\/_P0:
    "before translate off"
Assertion NOTE at 0 NS in design unit E1(A1) from process
  /\test/E1_inst\/_P1:
    "after translate on before 2nd time translate off"
Assertion NOTE at 0 NS in design unit E1(A1) from process
  /\test/E1_inst\/_P2:
    "after 2nd time translate on"
```

## Generating a List of Source Files

`-bom top-level_module -bfl filename`

Generates a file that contains a list of absolute path names to the source files of all the module definitions in a design or IP block.

The `-bom` option must be accompanied by the `-bfl` option.

The argument to the `-bom` option is the module name of the top-level module in the design or IP block.

The argument to the `-bfl` option is the filename that contains the list. VCS adds the `.bfl` extension to the filename you specify.

If a module definition is in a Verilog source file in a Verilog library directory, the name of the directory and source file is included in the path names. If a module definition is in a Verilog library file, the pathname of the library file is included in the list.

The following is an example of the output pathname file:

```
/file_system/design_group/LIBDIR/dev.v
```

```
/file_system/user_name/design1/top.v  
/file_system/design_group/libfile
```

## Option for Dumping Environment Variables

-diag env

Enables you to dump all environment variables that are set before starting the compilation and the simulation process. The list of environment variables that are set in the terminal is stored in the log file, which can be used to debug the environment related issues when the verification setup is complex and multiple and when nested scripts are used.

To dump all the environment variables, use the `-diag env` option with `vlogan/vcs` command line or `simv` command line.

### Syntax

The following is the syntax for `-diag env` option:

```
% vlogan -diag env
```

Dumps all the environment variables in the `vlogan_env_diag_<pid>.log` log file that is generated in the AN.DB directory.

```
% vcs -diag env
```

Dumps all the environment variables in the `vcs_env_diag_<pid>.log` log file that is generated in the simv.daidir directory.

```
% simv -diag env
```

Dumps all the environment variables in the  
simv\_env\_diag\_<pid>.log log file that is generated in the  
current working directory.



# D

## Simulation Options

---

This appendix describes the options and syntax associated with the `simv` executable. These runtime options are typically entered in the `simv` command line, however, some of them can be compiled into the `simv` executable at compile time.

This appendix describes the following runtime options:

- “[Options for Simulating Native Testbenches](#)”
- “[Options for SystemVerilog Assertions](#)”
- “[Options to Control Termination of Simulation](#)”
- “[Options for Enabling and Disabling Specify Blocks](#)”
- “[Options for Specifying When Simulation Stops](#)”
- “[Options for Recording Output](#)”
- “[Options for Controlling Messages](#)”

- “Options for VPD Files”
- “Options for VCD Files”
- “Options for Specifying Delays”
- “Options for Flushing Certain Output Text File Buffers”
- “Options for Licensing”
- “Option to Specify User-Defined Runtime Options in a File”
- “Option for the Support of Reading Gzipped Files”
- “Option for Initializing Verilog Variables, Registers and Memories at Runtime”
- “Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design at Runtime”
- “General Options”

---

## Options for Simulating Native Testbenches

`-cg_coverage_control`

Enables or disables the coverage data collection for all the coverage groups in your NTB-OV or SystemVerilog testbench.

Note:

The `$cg_coverage_control` system task takes precedence over this option.

Syntax: `-cg_coverage_control=value`

The valid values for `-cg_coverage_control` are 0 and 1. A value of 0 disables coverage collection and a value of 1 enables coverage collection.

**Note:**

You can also use this runtime option with the `coverage_control()` system task. The `coverage_control()` system task enables or disables data collection for one or more coverage groups at the program level. The runtime option takes precedence over the system task. For more information on this system task, see the *OpenVera Language Reference Manual: Native Testbench*.

`+ntb_cache_dir`

Specifies the directory location of the cache that VCS maintains as an internal disk cache for randomization.

`+ntb_delete_disk_cache=value`

Specifies whether VCS deletes the disk cache for randomization before simulation. The valid values are:

0 - do not delete (the default condition)

1 - delete the disk cache

`+ntb_disable_cnst_null_object_warning [=value]`

VCS produces the following warning when a null object handle is encountered in an object being randomized. Allowed values are 0 and 1.

0 - Do not disable null object warning (this is the default)

1 - Disable null object warning

Following is an example for the null object warning:

```
Warning- [CNST-PPRW] Constraint randomize NULL  
object warning test.sv, <line number>. Null  
object found during randomization. Please make  
sure all random variables/arrays/function calls  
being randomized are allocated fully and  
properly.
```

The null handle might be intentional or the result of an oversight. If you want to randomize objects that contain null handles, you can use this switch to disable the runtime warning.

```
+ntb_enable_checker_trace=0|1
```

In-line constraint checker using `randomize(null)` returns 1 if all constraints are satisfied and 0 otherwise. This option controls whether the constraint checker trace is enabled or not. The valid arguments are as follows:

0 - does not display the constraint checker trace (default)

1 - displays the constraint checker trace

If `+ntb_enable_solver_trace` is specified without an argument, the default value is 1. If it is not specified, the default value is 0.

```
+ntb_enable_checker_trace_on_failure[=value]
```

Enables a mode that displays trace information only when the randomize returns 0. Allowed values are 0, 1, and 2.

- |   |                  |
|---|------------------|
| 0 | Disables tracing |
| 1 | Enables tracing  |

- 2       Enables more verbose message in trace
- 3       In addition to the message in trace with option 2, the checker reports all the earlier solved constraints, which could have lead to the current failing constraint.

If `ntb_enable_checker_trace_on_failure` is specified without an argument, the default value is 1. If the `ntb_enable_checker_trace_on_failure` is not specified on the command line, then the default value is 0.

`+ntb_enable_solver_trace_on_failure[=0|1|2|3]`

Displays trace information when the VCS constraint solver fails to compute a solution. The valid argument values are as follows:

- 0       Disables displaying trace information
- 1       Enables displaying trace information
- 2       Enables more verbose trace information
- 3       In addition to the more verbose trace information specified with 2, the solver reports all the earlier solved constraints, which could have lead to the current failing constraint.

`+ntb_exit_on_error[=value]`

Causes VCS to exit when the value is less than 0. The value can be:

0 - continue

1 - exit on first error (default value)

N - exit on nth error

When the value is 0, the simulation finishes regardless of the number of errors.

`+ntb_random_seed=value`

Sets the seed value to be used by the top-level random number generator at the start of simulation. The `srandom(seed)` system function call overrides this setting. The value can be any integer. The default random seed value is 1.

`+ntb_random_seed_automatic`

Picks a unique value to supply as the first seed used by a testbench. The value is determined by combining the time of day, host name and process id. This ensures that no two simulations have the same starting seed.

The `+ntb_random_seed_automatic` seed appears in both the simulation log and the coverage report. When you enter both `+ntb_random_seed_automatic` and `+ntb_random_seed` VCS displays a warning message and uses the `+ntb_random_seed` value.

`+ntb_random_reseed`

Enables the re-seeding of the value the top-level random number generator uses after a save and restore of the simulation.

You enter this option with the `+ntb_random_seed_automatic` or `+ntb_random_seed=value` options. The seed value after the restore is the same as the one specified or generated by these other options.

If you omit these other options, VCS ignores the `+ntb_random_reseed` option and displays the following informational message:

```
Info-[RNG-SEED-MISSING] New seed was not specified for  
reseeding.
```

```
Please use runtime option +ntb_random_seed= or  
+ntb_random_automatic to specify new seed.
```

The `srandom(seed)` system function overrides this re-seeding.

`+ntb_solver_array_size_warn=value`

Specifies the array size warning limit (default is 10000) for constrained array sizes.

`+ntb_solver_debug=keyword_argument`

Tells VCS to give you more information so you can debug the constraints for the `randomize()` calls in batch mode. The keyword arguments are as follows:

`extract`

Tells VCS to extract a standalone test case in SystemVerilog for the specified `randomize()` call(s). To use this keyword argument, also enter the `+ntb_solver_debug_filter` runtime option.

`profile`

Enables constraint profiling in VCS. You can view the constraint profile report in `simv.cst/html/profile.xml` using a web browser (`simv` is the default name of the VCS `simv` executable).

This keyword argument also writes a file with a listing of the top randomize calls in `simv.cst/serial2trace.txt` (`simv` is the default name of the VCS `simv` executable).

`serial`

**Displays the randomize serial number at the end of each randomize() completion.**

**trace**

**Displays the solver trace to show how VCS solved the constraints for the random variables in specified randomize() call(s). To use this argument, also enter the +ntb\_solver\_debug\_filter runtime option.**

**trace\_all**

**Displays the solver trace for all randomize() calls. The +ntb\_solver\_debug=trace\_all option is the equivalent of entering the following options and arguments together:**

**+ntb\_solver\_debug=trace  
+ntb\_solver\_debug\_filter=all**

**You can enter multiple the keyword arguments using a plus (+) as a delimiter. For example:**

```
vcs source.sv +ntb_solver_debug=serial+extract+profile \  
+ntb_solver_debug_filter=12
```

**However, you cannot enter multiple +ntb\_solver\_debug options.**

**+ntb\_solver\_debug\_dir=pathname**

**Directs VCS to place profiles and extracted testcases in the specified directory. The default directory name is simv.cst, after the simv executable with the .cst extension.**

**+ntb\_solver\_debug\_filter=  
serial\_num [.partition\_num] | file[:filename] |  
all**

Specifies a list of `randomize()` calls that VCS displays debug information about. You can specify this list in the following ways:

- A comma separated list, for example:

```
+ntb_solver_debug_filter=1.5,4,20
```

This example specifies: the 5th partition of 1st call, and all partitions of the 4th and 20th call.

- In a file. The default filename is:

`simv.cst/serial2trace.txt`.

You need to enter the keyword argument file if the file is the default file name and location.

- The keyword `all` as in:

```
+ntb_solver_debug_filter=all
```

Specifying `all` means you want debug information about all `randomize()` calls.

Note:

The `all` argument can result in a large amount of solver trace information or extracted test cases.

```
+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 mode is 3.

In the default mode, based on the test case, the solver picks up one of the above solver mode. In case of any difficulty to solve with the initially chosen solver mode, it tries to switch to the other mode automatically.

+ntb\_stop\_on\_constraint\_solver\_error=0|1

Specifies whether VCS continues or exits after a constraint solver failure due to constraint inconsistency.

- 0      VCS continues to run after a constraint solver failure (default).
- 1      VCS exits on the first constraint solver error

+gc+high\_threshold [+1 | 2 | 3 | 4 | 5]

Sets a higher memory threshold for garbage collection (GC) invocation. It improves simulation time by reducing the number of GC invocations at the cost of additional memory. If GC overhead is higher due to high amount of dynamic memory usage and high number of GC invocations, this option provides improvement. You can know the GC time and the GC memory overheads using the unified simulation profiler time and memory reports.

The valid argument values are as follows:

- 1      Sets the higher memory threshold to at least 20% higher than the actual dynamic memory usage.
- 2      Sets the higher memory threshold to at least 25% higher than the actual dynamic memory usage.
- 3      Sets the higher memory threshold to at least 33% higher than the actual dynamic memory usage.
- 4      Sets the higher memory threshold to at least 50% higher than the actual dynamic memory usage.

- |   |                                                                                               |
|---|-----------------------------------------------------------------------------------------------|
| 5 | Sets the higher memory threshold to at least 67% higher than the actual dynamic memory usage. |
|---|-----------------------------------------------------------------------------------------------|

If you do not provide any argument value, the default value 3 is used.

---

## Options for SystemVerilog Assertions

`-assert keyword_argument`

Note:

- All the `-assert keyword_argument` runtime options, except the `-assert maxfail` and `-assert finish_maxfail` options are enabled only when the `-assert enable_diag` option is used at compile time.
- To enable the `-assert maxfail` and `-assert finish_maxfail` options at runtime, you must use the `-assert enable_hier` option at compile time.

The keyword arguments are as follows:

dbgopt

Enables the assertion optimization in the presence of higher debug caps. This compile time option is useful in the presence of `-debug_access` option and helps in generating optimal code for runtime performance. All the optimizations that are not part of the full debug flow are enabled under this switch. When this option is used at compile time, success callback is disabled and only failures are dumped. This option improves performance when the design is compiled with `-debug_access+f` or CLI level greater than 2 or `-debug_access+all`. You can enable success dumping by giving the switch `-assert dumpsuccess` at runtime, which can potentially degrade runtime performance.

dbsopt=*N*

Creates groups of *N* assertions with non-temporal expressions under the same clock, thereby localizing the evaluation of such assertions due to changes in a smaller set of signals.

disable=<value>

Disables the concurrent, immediate, and deferred assertions in a particular design. The options for <value> can be concurrent, deferred, and immediate.

dumpoff

Disables the dumping of SVA information in the VPD file during simulation.

finish\_maxfail=*N*

Terminates the simulation if the number of failures for any assertion reaches  $N$ . You must supply  $N$ , otherwise no limit is set.

`global_finish_maxfail=N`

Terminates the simulation when the total number of failures from all SystemVerilog assertions reaches  $N$ .

`maxcover=N`

Disables the collection of coverage information for cover statements after the cover statements are covered  $N$  number of times.  $N$  must be a positive integer; it cannot be 0.

`maxfail=N`

Limits the number of failures for each assertion to  $N$ . When the limit is reached, VCS disables the assertion. You must supply  $N$ , otherwise no limit is set.

`maxsuccess=N`

Limits the total number of reported successes to  $N$ . You must supply  $N$ , otherwise no limit is set. VCS continues to monitor assertions even after the limit is reached.

`nocovdb`

Tells VCS not to write the `program_name.db` database file for assertion coverage.

`noDefSuccRpt`

Improves the performance of deferred assertions during runtime. When the `-debug_access` option or higher debug capabilities are used, VCS tracks both successes and failures, hence results in reduced runtime performance. When this switch is used, VCS tracks only failures of deferred assertions and drops the successes. This improves the performance because successes happen more frequently. This option has no effect when the design is compiled without the `-debug_access` switch.

`nopostproc`

Disables the display of the SystemVerilog `assert` and `cover` statement summary at the end of simulation.

This begins with the `assert` and `cover` statements that started but did not finish in the following format:

```
"source_filename.v", line_number:  
assert_or_cover_statement_hierarchical_name:  
started at simulation_time not finished
```

If the `assert` or `cover` statement does not start, this summary also reports about this in the following format:

```
**** Following assertions did not fire at all  
during simulation. ****  
"source_filename.v", line_number:  
assert_or_cover_statement_hierarchical_name:  
No attempt started
```

This is followed by a `cover` statement summary in the following format:

```
"source_filename.v", line_number:  
cover_statement_hierarchical_name, number  
attempts, number match
```

`no_fatal_action`

Excludes failures on SVA assertions with fail action blocks for computation of failure count in the `-assert [global_] finish_maxfail=N` runtime option.

`no_default_msg [=SVA|OVA|PSL]`

Disables the display of default failure messages for SVA assertions that contain a fail action block, and OVA and PSL assertions that contain user messages.

novp<sub>i</sub>

Disables all vpi based callback access for assertions and dumping of assertion. This compile time option is useful in the presence of `-debug_access` option. It helps to reduce assertion debug overhead, when the design is compiled with higher debug capabilities.

Note:

If a combination of `-assert dbgopt+novpi` is given, you can have the access to perform the following:

- Ability to iterate over all the assertions.
- Mark the assertions for dynamically enabling or disabling them.

quiet

Disables the display of messages when assertions fail.

quiet1

Disables the display of messages when assertions fail, however, enables the display of summary information at the end of simulation. For example,

Summary: 2 assertions, 2 with attempts, 2 with failures

report [=path/filename]

- Generates a report file in addition to printing results on your screen. By default, the report file name and location is `./assert.report`, however, you can change it by entering the `path/filename` argument. The report file name can start with a number or letter.
- Generates a report of all assertions that are disabled using any one of the following mechanisms:
  - System tasks `$asserton/off/kill`
  - `assert hier` at compile time or runtime

The report is categorized based on:

- Disabled assertions on a module level (compile time)
- Assertions disabled through the `-assert hier` option
- Disabled assertions at the end-of-simulation

Note:

- If the file name is specified by the user, it is dumped as `<user_file>.disablelog`.
- If the file name is not specified by the user, it is dumped as `assert.report.disablelog`

The following special characters are acceptable in the file name: %, ^, and @. Using the following unacceptable special characters: #, &, \*, [ ], \$, ( ), or ! has the following consequences:

- A file name containing # or & results in a file name truncation to the character before the # or &.

- A file name containing \* or [ ] results in a No match message.
- A file name containing \$ results in an Undefined variable message.
- A file name containing () results in a Badly placed ()'s message.
- A file name containing ! results in an Event not found message.

`success`

Enables reporting of successful matches, and successes on cover and assert statements respectively, in addition to failures. The default is to report only failures.

`vacuous`

Enables reporting of vacuous successes on assert statements in addition to the failures. By default, VCS reports only failures.

`verbose`

Adds more information to the end of the report specified by the report keyword argument, and a summary with the number of assertions present, attempted, and failed.

`hier=file_name`

Specifies a file to enable and disable SystemVerilog assertions when you simulate your design. This feature enables you to control which assertions are active and VCS records in the coverage database, without having to recompile your design.

The types of entries you can make in the file are as follows:

```
-assert <assertion_name> or  
-assert <assertion_hierarchical_name>
```

If `<assertion_name>` is provided, VCS disables the assertions based on wildcard matching of the name in the complete design. If `<assertion_hierarchical_name>` is provided, VCS disables the assertions based on wildcard matching of the name in the particular hierarchy given.

## Examples

```
-assert my_assert
```

Disables all assertions with name `my_assert` in the full design.

```
-assert A*
```

Disables all assertions whose name starts with `A` in the full design.

```
-assert *
```

Disables all assertions in the full design.

```
-assert top.INST2.A
```

Disables all assertions whose names start with `A` in the `top.INST2` hierarchy. If assertions whose name starts with `A` exists in inner scopes under `top.INST2`, they are not disabled. This command has affect on assertions only in scope `top.INST2`.

```
+assert <assertion_name> or  
+assert <assertion_hierarchical_name>
```

If `<assertion_name>` is provided, VCS enables the assertions based on wildcard matching of the name in the full design. If `<assertion_hierarchical_name>` is provided, then VCS enables the assertions based on wildcard matching of the name in the given hierarchy.

## Examples

```
+assert my_assert
```

Enables all assertions with name `my_assert` in the full design.

```
+assert A*
```

Enables all assertions whose name starts with `A` in the full design.

```
+assert *
```

Enables all assertions in the full design.

```
+assert top.INST2.A
```

Enables assertion `A` in the hierarchy `top.INST2`.

```
+tree <module_instance_name> or  
+tree <assertion_hierarchical_name>
```

If `<module_instance_name>` is provided, VCS enables assertions in the specified module instance and all module instances hierarchically under that instance. If `<assertion_hierarchical_name>` is provided, VCS enables the specified SystemVerilog assertion. Wildcard characters can also be used for specifying the hierarchy.

## Examples

```
+tree top.inst1
```

Enables the assertions in module instance `top.inst1` and all the assertions in the module instances under this instance.

```
+tree top.inst1.a1
```

Enables SystemVerilog assertion with the hierarchical name `top.inst1.a1`.

```
+tree top.INST*.A1
```

Enables assertion `A1` from all the instances whose names start with `INST` under module `top`.

```
-tree <module_instance_name> or  
-tree <assertion_hierarchical_name>
```

If `<module_instance_name>` is provided, VCS disables the assertions in the specified module instance and all module instances hierarchically under that instance. If `<assertion_hierarchical_name>` is provided, VCS disables the specified SystemVerilog assertion. Wildcard characters can also be used for specifying the hierarchy.

## Examples

```
-tree top.inst1
```

Disables the assertions in module instance `top.inst1` and all the assertions in the module instances under this instance.

```
-tree top.inst1.a1
```

Disables the SystemVerilog assertion with the hierarchical name `top.inst1.a1`.

```
-tree top.INST*.A1
```

Disables assertion `A1` from all the instances whose names start with `INST` under module `top`.

```
+module module_identifier
```

VCS enables all the assertions in all instances of the specified module.

For example, `+module dev`. VCS enables the assertions in all instances of module `dev`.

```
-module module_identifier
```

VCS disables all the assertions in all instances of the specified module.

For example, `-module dev`. VCS disables the assertions in all instances of module `dev`.

```
-assert assertion_block_identifier
```

VCS disables the assertion with the specified block identifier. You can use wildcard characters in specifying the block identifier to specify more than one assertion.

You can enter more than one keyword using the plus (+) separator. For example,

```
-assert maxfail=10+maxsuccess=20+success+filter.
```

```
-cm assert
```

Specifies monitoring for SystemVerilog assertions coverage. When enabled, the `-cm assert` option does the following:

- Generates the number of attempts, pass, fail, and incomplete data.
- Generates vacuous and non-vacuous coverage.
- Irrespective of type of assert statement, reports coverage.
- Covers immediate and deferred assertions.
- Does not cover Expect statement.
- Affects SVA and OVA as well.

`-uniq_prior maxfail=integer`

Specifies the maximum number of unique or priority violations (see `-error=UNIQUE` and `-error=PRIORITY` in “[Options for SystemVerilog Assertions](#)” ) before VCS ends the simulation.

The types of error messages that this option controls are as follows:

RT Error: No condition matches in unique case statement  
“dev.v”, line 17, for top.dev, at time 0

RT Error: More than one conditions match in ‘unique case’ statement  
“dev.v”, line 18, for top.dev,  
Line 19 & 20 are overlapping at time 0.

This runtime option is enabled by the `-error=UNIQUE`, `-error=PRIORITY`, or `-error=UNIQUE, PRIORITY` compile time option and keyword arguments.

---

## Options to Control Termination of Simulation

`-ova_enable_case_maxfail`

Includes OVA case violations in computation of global failure count for the `-assert global_finish_maxfail=N` 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_pulse_msg`

Suppresses pulse error messages, however, not the generation of `StX` values at module path outputs when a pulse error condition occurs.

`+no_tchk_msg`

Disables the display of timing violations, however, 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.

Note:

The `+notimingcheck` option has higher precedence than any `tcheck` command in UCLI.

---

## 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 example, you can specify the following:

`+vcs+finish+9001us`

For both of these options, there is a special procedure (See “[Specifying Long Time Before Stopping the Simulation](#)” ) for specifying time values larger than  $2^{32}$ .

---

## Options for Recording Output

`-l filename`

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

---

## Options for Controlling Messages

`-error`

Revises the `+lint` and `+warn` options, to control error and warning messages. With them you can:

- Disable the display of any lint, warning, or error messages
- Disable the display of specific messages
- Limit the display of specific messages to a maximum number that you specify

Only the following feature is supported at runtime.

`-error=[no]message_ID[:max_number],...`

For more information on the option, see “[Error/Warning/Lint Message Control](#)” .

Note:

The `-error` option is also a compile time option.

`-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). Suppresses SystemC BMI warnings and notes at the start of simulation.

-V

Verbose mode; displays VCS version and extended summary information. Displays VCS compile and runtime version numbers, and copyright information, at the start of simulation.

+no\_pulse\_msg

Suppresses pulse error messages, however, not the generation of \$T\$E values at module path outputs when a pulse error condition occurs.

You can enter this runtime option in the `vcs` command line. You cannot enter this option in the file you use with the `-f` compile time option.

+vcs+nostdout

Disables all text output from VCS including messages and text from `$monitor` and `$display` and other system tasks for only the Verilog portion of the design. VCS still writes this output to the log file if you include the `-l` option.

---

## Options for VPD Files

`-vpd_bufsize 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, however, not less than 2 megabytes.

**Note:**

VCS automatically increases the buffer size as needed to comply with this limit.

`-vpd_file <file_name>`

Specifies the name of the output VPD file (default is `vcplus.vpd`). You must include the full file name with the `.vpd` extension.

`+vpdfysize+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 might 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.

`-vpd_fileswitchsize <size_in_MB>`

Specifies a size for the vpd file. When the vpd file reaches this size, VCS closes this file and opens a new file 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 -vpd_file test.vpd - vpd_fileswitchsize 10`. The first vpd file is named `test.vpd`. When its size reaches 10MB, VCS starts a new file `test_01.vpd`, the third vpd file is `test_02.vpd`, and so on.

#### +vpdignore

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

#### +vpdports

Causes VPD to store port information, which is then used by the Hierarchy Browser to show whether a signal is a port, and if so, its direction. This option to some extent affects simulation initialization time and memory usage for larger designs.

#### +vpdportsonly

Dumps only the port type information.

#### +vpdnoports

Dumps only the signal not the ports (input/output).

#### +vpddrivers

Stores data for changes on drivers of resolved nets.

`+vpdupdate`

Enables VPD file locking.

`+vpdnocompress`

Disables the default compression of data as it is written to the VPD file.

---

## Options for VCD Files

`+vcs+dumpfile+filename`

Sets the name of the `$dumpvars` output file to `filename`. The default file name is `verilog.dump`. A `$dumpfile` system task in the Verilog source code overrides this option.

`+vcs+dumpoff+t+ht`

Turns off value change dumping (`$dumpvars`) at time `t`. `ht` is the high 32 bits of a time value greater than 32 bits.

`+vcs+dumpon+t+ht`

Suppresses the `$dumpvars` system task until time `t`. `ht` is the high 32 bits of a time value greater than 32 bits.

`+vcs+dumparrays`

Enables recording memory and multi-dimensional array values in the VCD file. You must also have used the `+memcbk` compile-time option.

`+vcs+flush+dump`

Increases the frequency of dumping all VCD files.

---

## Options for Specifying Delays

`-novitaltiming`

Enables functional-only simulation of VITAL components. All timing information is discarded for VITAL models during simulation. Timing information includes wire delays, path delays and timing checks. Any SDF information supplied on the command line is ignored when this switch is present.

`+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 `+mindlays` option specifies using the compiled SDF file with the minimum delay.

Another use for this runtime option is to specify timing for SWIFT VMC and SmartModels when you also include the `+override_model_delays` runtime option.

`+typdelays`

Specifies using the typical delays in min:typ:max delay triplets in module path delays and timing checks, if you compiled your design with the `+allmtm` compile-time option. Also specifies using the typical timing delays in min:typ:max delay triplets in an uncompiled SDF file.

If you compiled the SDF file with the `+allmtm` compile-time option, the `+typdelays` option specifies using the compiled SDF file with the typical delays.

This is a default option. By default, VCS uses the typical delay in min:typ:max delay triplets in your source code and in uncompiled SDF files unless you specify otherwise with the `mtm_spec` argument to the `$sdf_annotate` system task. Also, by default, VCS uses the compiled SDF file with typical values.

Another use for this runtime option is to specify timing for SWIFT VMC and SmartModels when you also include the `+override_model_delays` runtime option.

---

## Options for Flushing Certain Output Text File Buffers

When VCS creates a log file, VCD file, or a text file specified with the `$fopen` system function. VCS writes the data for the file in a buffer and periodically dumps the data from the buffer to the file on disk. The frequency of these dumps varies depending on many factors including the amount of data that VCS has to write to the buffer as simulation or compilation progresses. If you need to see or use the latest information in these files more frequently than the rate at which VCS normally dumps this data, these options tell VCS to dump the data more frequently. The frequency also depends on many factors; however the increased frequency will always be significant.

`+vcs+flush+log`

Increases the frequency of dumping both the compilation and simulation log files.

`+vcs+flush+dump`

Increases the frequency of dumping all VCD files.

`+vcs+flush+fopen`

Increases the frequency of dumping all files opened by the `$fopen` system function.

`+vcs+flush+all`

Increases the frequency of dumping all log files, VCD files, and all files opened by the `$fopen` system function.

These options do not increase the frequency of dumping other text files including the VCDE files specified by the `$dumpports` system task or the simulation history file for LSI certification specified by the `$lsi_dumpports` system task.

You can also enter these options at compile time. There is no performance gain to entering them at compile time.

---

## Options for Licensing

`+vcs+lic+vcsi`

Checks out three VCSI licenses to run VCS.

`+vcsi+lic+vcs`

Checks out a VCS license to run VCSI when all VCSI licenses are in use.

`+vcs+lic+wait`

Waits for a network license if none is available when the job starts.

`-licwait timeout`

Enables license queuing, where *timeout* is the time in minutes that VCS waits for a license before finally exiting.

`-licqueue`

Tells VCS to wait for a network license if none is available.

---

## Option to Specify User-Defined Runtime Options in a File

`-f filename`

You can use the `-f` runtime option to specify user-defined plusargs in a file. The user-defined plusargs are the plus arguments on the `simv` command line defined using `$test$plusargs` or `$value$plusargs` system tasks in RTL code as per *IEEE Standard 1364-2001 17.10 Commandline input*. All other VCS runtime options should be specified on the `simv` command line.

---

## Option for the Support of Reading Gzipped Files

VCS supports an option to read files that are compressed using the `gzip`.

To enable this option, use the `-io_gz` runtime switch.

### Example

This example shows how the utility is used to read a gzipped file.

#### *Example D-1 Test.v*

```
module m1;

    integer mcd,code,code1,mcd1;
    string str;

initial
begin
    mcd = $fopen("test_file.txt", "w");
    $fwrite (mcd, "%s", "\nThis is VCS test file\n");
    $fclose (mcd);

    $system("gzip test_file.txt");
    mcd = $fopen("test_file.txt.gz", "r");
    do
begin
    code = $fgets(str,mcd);
```

```

        $display("%s",str);
    end
    while($feof(mcd)==0);

$fclose(mcd);

end
endmodule

```

You can use the following commands to run the test case.

```
vcs -sverilog test.v
simv -io_gz
```

## Output

This is VCS test file

## Supported APIs

Following are the APIs that are supported:

- \$fopen
- \$fgets
- \$fgetc
- \$fread
- \$fseek
- \$ftell
- \$frewind
- \$ferror
- \$feof
- \$fclose

- `$freadb`
- `$freadh`

## Limitations

- You cannot perform a write operation on gzipped files.
- The `$fscanf` API is not supported.

## Option for Initializing Verilog Variables, Registers and Memories at Runtime

`+vcs+initreg+0|1|random|seed_value`

Initializes all bits of the Verilog variables, registers defined in sequential UDPs, and memories including multi-dimensional arrays (MDAs) in your design to the specified values at time zero. The default seed is used when no random seed is specified. This option can only be used when the `+vcs+initreg+random` option is specified at compile time.

The supported data types are:

- reg
- bit
- integer
- int
- logic

The following table describes the initialization options at runtime:

| Syntax of Runtime Option | Description                                                                                                                                                                                              |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| +vcs+initreg+0           | Initializes all variables, registers and memories to value 0.                                                                                                                                            |
| +vcs+initreg+1           | Initializes all variables, registers and memories to value 1.                                                                                                                                            |
| +vcs+initreg+random      | Initializes all variables, registers and memories to random value 0 or 1, with the default seed.                                                                                                         |
| +vcs+initreg+100         | Initializes all variables, registers and memories to random value 0 or 1, with the user-defined seed 100.<br><br>Note: The <code>seed_value</code> cannot be 1 or 0. Those values have special meanings. |

The initialization options might cause potential race conditions due to the initialized values specified. For more information on race condition prevention, see “[Option for Initializing Verilog Variables, Registers and Memories with Random Values](#)” .

---

## Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design at Runtime

`+vcs+initreg+config+config_file`

Specifies a configuration file for initializing Verilog variables, registers defined in sequential UDPs, and memories including multi-dimensional arrays (MDAs) in your design at time zero. In the configuration file, you can define the parts of a design to apply the initialization and the initialization values of the variables.

This option can only be used at runtime when either the `+vcst+initreg+random` option or the `+vcst+initreg+config+config_file` option is specified at compile-time.

If the `+vcs+initreg+config+config_file` option is specified at both compile time and runtime, the configuration file specified at runtime overrides the configuration file at compile time.

The `+vcs+initreg+seed_value` option can be specified with the `+vcs+initreg+config+config_file` option at runtime to select a random seed for generating random initial values as defined in the configuration file.

If the `+vcs+initreg+0|1|random` and `+vcs+initreg+config+config_file` options are both specified at runtime, the `+vcs+initreg+0|1|random` option is ignored and a warning message is issued.

The following table describes the initialization options at runtime:

| Syntax of Runtime Options                                                             | Description                                                                                       |
|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <code>+vcs+initreg+config+config_file</code>                                          | Runtime configuration file overrides compile-time configuration file                              |
| <code>+vcs+initreg+config+config_file</code><br><code>+vcst+initreg+seed_value</code> | Uses specified seed for generating random initial values as defined in runtime configuration file |
| <code>+vcs+initreg+config+config_file</code><br><code>+vcs+initreg+random</code>      | Issues a warning message, ignores <code>+vcs+initreg+random</code>                                |
| <code>+vcs+initreg+config+config_file</code><br><code>+vcs+initreg+0</code>           | Issues a warning message, ignores <code>+vcs+initreg+0</code>                                     |
| <code>+vcs+initreg+config+config_file</code><br><code>+vcs+initreg+1</code>           | Issues a warning message, ignores <code>+vcs+initreg+1</code>                                     |

For more information on the configuration file, see “[Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design](#)”

---

## General Options

### Viewing the Compile Time Options

`-sig 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 -sig echo`

You cannot use any other runtime option with the `-sig` option.

### Recording Where ACC Capabilities are Used

`+vcs+learn+pli`

ACC capabilities enable debugging operations, however, they have a performance cost, so enable them where you need them. This option keeps track of where in your design 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

`+vcst+ignorestop`

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

## Enabling User-defined Plusarg Options

`+plus-options`

User-defined runtime options to perform some operation when the option is on the `simv` command line. The `$test$plusargs` system task can check for such options.

## Enabling Overriding the Timing of a SWIFT SmartModel

`+override_model_delays`

Instead of using the `DelayRange` parameter definition in the template file, this option enables the `+mindelays`, `+typdelays`, and `+maxdelays` runtime options to specify the timing used by SWIFT SmartModels.

## Enabling feature beyond VHDL LRM

`-x1rm`

Enables VHDL features beyond those described in *IEEE Standard VHDL 1076-2008 LRM*.

## Enabling Loop Detect

`+vcs+loopreport+number`

It is mandatory to include the `+vcs+loopreport+number` option at compile time, though you can override the threshold number at runtime.

When `+vcs+loopreport+number` is specified at compile time, VCS does the following based on the option specified at runtime:

- If `+vcs+loopreport` is specified at runtime, VCS checks if a simulation event loops for 2,000,000 times (by default) in the same simulation time tick, and issues a runtime warning message. VCS also terminates the simulation and generates a report when a zero delay loop is detected.
- If `+vcs+loopreport+N` is specified at runtime, VCS checks if the simulation event loops for 'N' times and issues a runtime warning message. VCS also terminates the simulation.

For information about using the `+vcs+loopreport+number` option during compile time, see "[Enabling Loop Detect](#)".

`+vcs+loopdetect+number`

When `+vcs+loopdetect+number` is not specified at compile time, VCS does the following based on the option specified at runtime:

- If `+vcs+loopdetect` is specified at runtime, VCS checks if a simulation event loops for 2,000,000 times (by default) in the same simulation time tick, and issues a runtime error message. VCS also terminates the simulation.
- If `+vcs+loopdetect+N` is specified at runtime, VCS checks if the simulation event loops for 'N' times and issues a runtime error message. VCS also terminates the simulation.

For information about using the `+vcs+loopdetect+number` option compile time, see Section "[Enabling Loop Detect](#)".

## **Specifying acc\_handle\_simulated\_net PLI Routine**

+vcs+mipd+noalias

For the `acc_handle_simulated_net` PLI routine, aliasing of a loconn net and a hiconn net across the port connection is disabled if MIPD delay annotation happens for the port. If you specify ACC capability: mip or mipb in the `pli.tab` file, such aliasing is disabled only when actual MIPD annotation happens.

If during a simulation run, `acc_handle_simulated_net` is called before MIPD annotation happens, VCS issues a warning message. When this happens you can use this option to disable such aliasing for all ports whenever mip, mipb capabilities have been specified. This option works for reading an ASCII SDF file during simulation and not for compiled SDF files.

## **Loading DPI Libraries Dynamically at Runtime**

```
-sv_lib library_path_name  
-sv_root library_path_name  
-sv_liblist library_path_name
```

The procedure for loading a DPI library at runtime is as follows:

1. Analyze and compile/elaborate the Verilog or SystemVerilog code, for example:

```
%vlogan -sverilog other_options test.v  
% vcs top_level  
%> vcs -sverilog other_options test.v
```

2. Compile the C code and create a shared object, for example:

```
%> gcc -fPIC -Wall ${CFLAGS} -I${VCS_HOME}/include \  
other_libraries -c test.c  
  
%> gcc -fPIC -shared ${CFLAGS} -o test.so test.o
```

3. Load the shared object at runtime using one of the following runtime options for this purpose:

```
-sv_lib -sv_root -sv_liblist
```

## Loading PLI Libraries Dynamically at Runtime

```
-load library_path_name
```

Loads a PLI library dynamically at runtime. Enter the `-load` option for each library you are dynamically loading. For example,

```
% simv -load ./pli1.so -load ./pli2.so
```

To use this runtime option, when you compile the design include the PLI table file for the PLI libraries with the `-P` compile-time option:

```
% vcs -P pli.tab design_source_files
```

## Independent Seeding Across Multiple Instances

```
-xlrn hier_inst_seed
```

Uses instance-specific initialization seeds for random number generators of modules, programs, interfaces, and packages. The default flow uses the same initialization seed across all instances. For example,

```
% simv -xlrn hier_inst_seed
```

# E

## Verilog 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 the compiler directives.

---

## Compiler Directives for Cell Definition

``celldefine`

Specifies that the modules under this compiler directive be tagged as “cell” for delay annotation. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.10.

Syntax: ``celldefine`

``endcelldefine`

Disables ``celldefine`. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.10.

Syntax: ``endcelldefine`

---

## Compiler Directives for Setting Defaults

``default_nettype`

Sets default net type for implicit nets. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.8.

**Syntax:** `'default_nettype wire | tri | tri0 | wand | triand | tril | wor | trior | trireg |none`

``resetall`

Resets all compiler directives. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.3.

**Syntax:** ``resetall`

---

## Compiler Directives for Macros

``define`

Defines a text macro. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.5.1.

**Syntax:** `'define text_macro_name macro_text`

``else`

Specifies an alternative group of source code lines that VCS compiles if the text macro specified with an ``ifdef` compiler directive is not defined. It is used with the ``ifdef` compiler directive. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.6.

**Syntax:** `'else second_group_of_lines`

``elseif`

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. It is used with the `'ifdef` compiler directive. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.6.

**Syntax:** `'elseif text_macro_name  
second_group_of_lines`

`'endif`

Specifies the end of a group of lines specified by the `'ifdef` or `'else` compiler directives. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.6.

**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.

**Syntax:** `'ifdef text_macro_name  
group_of_lines  
'endif`

```
'ifdef VCS
```

The character string `VCS` is a predefined text macro in VCS. The Verilog or SystemVerilog code that follows `'ifdef VCS` is the code that you want to compile by VCS. The code that follows a corresponding `'else` compiler directive is the source code that VCS ignores.

You can insert source code after the `'else` compiler directive that you intend for a third-party tool.

In the following source code, VCS compiles and executes the first block of code and ignores the second block, even when you do not include the ``define VCS` compiler directive or the `+define+VCS` compile-time option:

```
'ifdef VCS
    begin
        // Block of code for VCS           M
    end
`else
    begin
        // third party code
        :
    end
`endif
```

When you encrypt the source code, VCS inserts `'ifdef VCS` before all encrypted parts of the code.

Note:

You can use the option `-undef_vcs_macro` to cancel the VCS predefined text macro. As a result, the `'ifdef VCS ... `endif` block contents are skipped. You can use this option mainly for different VCS modes (other than simulation) where these block contents are not expected to pass through.

``ifndef`

Specifies compiling the source code that follows if the specified text macro is not defined. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.6.

Syntax: ``ifndef text_macro_name group_of_lines`

``undef`

Undefines a macro definition. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.5.2.

Syntax: ``undef text_macro_name`

---

## Compiler Directives for Delays

``delay_mode_path`

Ignores the delay specifications on all gates and switches in all those modules under this compiler directive that contain the 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 the 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 the 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 Back Annotating SDF Delay Values

``vcs_mipdexpand`

This compiler directive enables the runtime back-annotation of individual bits of a port declared in an ASCII text SDF file. This is done by entering the compiler directive over the port declarations for these ports. Similarly, entering this compiler directive over the port declarations enables a PLI application to pass delay values to the individual bits of a port.

As an alternative to using this compiler directive, you can use the `+vcst+mipdexpand` compile-time option, or you can enter the `mipb ACC` capability. For example:

```
$sdf_annotation call=sdf_annotation_call  
acc+=rw,mipb:top_level_mod
```

When you compile the SDF file, which Synopsys recommends, you do not need to use this compiler directive to back-annotate the delay values for individual bits of a port.

``vcs_mipdnoexpand`

Turns off the enabling of back-annotating delay values on individual bits of a port as specified by a previous ``vcs_mipdexpand` compiler directive.

---

## Compiler Directives for Source Protection

For information about compiler directives for source protection, see ["Encrypting Source Files"](#).

---

## General Compiler Directives

### Compiler Directive for Including a Source File

``include`

Includes (also compiles as part of the design) the specified source file. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.4.

Syntax:

``include "filename"`

Note:

If the included file is a different version of Verilog from the source file that contains the `include compiler directive, and you want VCS to compile the included file for the version specified by its filename extension, enter the -extinclude compile-time option, see “[Options for Different Versions of Verilog](#)”

## Compiler Directive for Setting the Time Scale

`timescale

Sets the timescale. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.4.

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 *file* | *directory*

Searches the specified library for unresolved modules. You can specify either a library file or a library directory.

Syntax:

`uselib *file* = *filename*

or

`uselib *dir* = *directory\_name libext+.ext* |  
*libext=.ext*

Enter path names if the library file or directory is not in the current directory. For example:

```
`uselib file = /sys/project/speclib.lib
```

If specifying a library directory, include the `libext+.ext` keyword and append to it the extensions of the source files in the library directory, similar to the `+libext+.ext` compile-time option. For example:

```
`uselib dir = /net/designlibs/project.lib  
libext+.v
```

To specify more than one search library, enter additional `dir` or `file` keywords, for example:

```
`uselib dir = /net/designlibs/library1.lib dir=/  
net/designlibs/library2.lib libext+.v
```

Here, the `libext+.ext` keyword applies to both the libraries.

## **Compiler Directive for File Names and Line Numbers**

```
`line line_number "filename" level
```

Maintains the file name and the line number. For more details, see IEEE SystemVerilog LRM Std 1800™-2012 Section 22.12.

---

## **Unimplemented Compiler Directives**

The following compiler directives are IEEE SystemVerilog LRM Std 1800™-2012 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 are described in the IEEE SystemVerilog LRM Std 1800™-2012 for more information.

---

### **System Tasks for SystemVerilog Assertions Severity**

`$fatal`

Generates a runtime fatal assertion error.

`$error`

Generates a runtime assertion error.

`$warning`

Generates a runtime warning message.

`$info`

Generates an information message.

---

## System Tasks for SystemVerilog Assertions Control

`$assertoff`

Tells VCS to stop monitoring any of the specified assertions that start at a subsequent simulation time.

`$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.

`$asserton`

Tells VCS to resume the monitoring of assertions that it stopped monitoring due to a previous `$assertoff` or `$assertkill` system task.

These system tasks provide file name and line number from where these system tasks are called, which might otherwise be difficult to track in the absence of this information.

Note:

- The runtime option `-assert old_ctrl_msg` reverts the messaging to the old style for backward compatibility.
- It is recommended to use the `$assertoff` system task with arguments, as shown to turn off reporting of assertions globally for the entire design:

```
$assertoff (0,"top_level_module")
```

You may not be able to enable assertions on the desired hierarchies, if you use `$assertoff` without arguments to turn off assertions.

---

## System Tasks for SystemVerilog Assertions

`$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.

`$countx (expression)`

Returns the number of expression bits set to X.

`$countz (expression)`

Returns the number of expression bits set to Z.

`$countunknown (expression)`

Returns the number of expression bits set to either X or Z.

`$onedriven`

Returns true if only one bit of the expression is not Z, and its value is defined (not X).

`$onedriven0`

Returns true if at most one bit of the expression is not Z, and if such a bit exists, its value is defined (not X).

---

## 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 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. They require the `$dumpvars` system task.

`$dumpall`

Creates a checkpoint in the VCD file. When VCS executes this system task, VCS records the current values of all specified nets and registers them into the VCD file, irrespective of whether there is a value change at this time or not.

`$dumpoff`

Stops recording value change information in the VCD file.

`$dumpon`

Starts recording value change information in the VCD file.

`$dumpfile`

Specifies the name of the VCD file that you want VCS to record.

Syntax: `$dumpfile ("filename") ;`

`$dumpflush`

Empties the VCD file buffer and writes all this data to the VCD file.

`$dumplimit`

Limits the size of a VCD file.

`$dumpvars`

Specifies the nets and variables whose transition times and values you want VCS to record in the VCD file.

**Syntax:** `$dumpvars (level_number, module_instance | net_or_var);`

You can specify individual nets or variables, or specify all the nets and variables in an instance.

The `$dumpvars` system task enables the other VCD system tasks, such as `$dumpon` and `$dumpfile`.

`$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 dump file buffer of the operating system. As simulation progresses, reads the date 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 specifies 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_dumports (top.middle1, "dumports.dmp");
```

If you would rather have the `$lси_dumports` system task generate an extended VCD (EVCD) file instead, include the `+dumports+ieee` runtime option.

```
$dumports
```

Creates an EVCD file as specified in IEEE Verilog LRM Std. 1364-2005, Pages 338-339. For example, you can input a EVCD file into TetraMAX for fault simulation. EVCD files are similar to the simulation history files generated by the `$lси_dumports` system task for LSI certification. But, there are differences in the internal statements in the file. Furthermore, the EVCD format is a proposed IEEE standard format, whereas the format of the LSI certification file is specified by LSI.

In the past, both the `$dumports` and `$lси_dumports` system tasks generated simulation history files for LSI certification and had identical syntax except for the name of the system task.

Syntax of the `$dumports` system task is now:

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

You can specify more than one module instance.

**Code example:** `$dumports (top.middle1, top.middle2,  
"dumports.evcd");`

If your source code contains a `$dumports` system task and you want it to generate the simulation history files for LSI certification, include the `+dumports+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. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 339-340.

**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. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 339-340.

**Syntax:** `$dumpportson ("filename") ;`

`$dumpportsall`

By default, VCS writes to files only when a signal changes value. The `$dumpportsall` system task records the values of the ports in the module instances, which are specified by the `$lси_dumpports` or `$dumpports` system task, irrespective of whether there is a value change on these ports or not. You can specify the file to which you want VCS to record the port values for the corresponding module instance or specify no particular file, in which case VCS writes port values in all files opened by the `$lси_dumpports` or `$dumpports` system task. For more details, see IEEE Verilog LRM Std 1364-2005, Page 340.

**Syntax:** `$dumportsall ("filename");`

`$dumportsflush`

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 `$dumportsflush` system task. You can specify the file to which you want VCS to write from the buffer or specify no particular file, in which case VCS writes all data from the buffer to all files opened by the `$lsi_dumports` or `$dumports` system task. For more details, see IEEE Verilog LRM Std 1364-2005, Page 341.

**Syntax:** `$dumportsflush ("filename");`

`$dumportslimit`

Specifies the maximum file size of the file specified by the `$lsi_dumports` or `$dumports` system task. You can 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 `$lsi_dumports` or `$dumports` system task. For more details, see IEEE Verilog LRM Std 1364-2005, Page 340.

**Syntax:** `$dumportslimit (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 GUI to view the simulation results that VCS recorded in a VPD file. You cannot actually load a VCD file directly into GUI. When you load a VCD file, GUI 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 GUI 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. After that you can review the simulation results using GUI.

There are system tasks that specify the information that VCS writes in the VPD file.

**Note:**

To use the system tasks for VPD files, you must compile your source code with the `-debug_access` option.

`$vcdplusautoflushoff`

Turns off the automatic flushing of simulation results to the VPD file whenever there is an interrupt, such as when VCS executes the `$stop` system task.

**Syntax:** `$vcdplusautoflushoff;`

`$vcplusautoflushon`

Tells VCS to flush or write all the simulation results in memory to the VPD file whenever there is an interrupt, such as when VCS executes a `$stop` system task or when you halt VCS using the UCLI `stop` command, or using the **Stop** button on the GUI.

**Syntax:** `$vcplusautoflushon;`

`$vcplusclose`

Tells VCS to mark the current VPD file as completed and close the file.

**Syntax:** `$vcplusclose;`

`$vcplusdeltacycleon`

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

Glitch detection is automatically turned on when VCS executes `$vcplusdeltacycleon` unless you have previously used `$vcplusglitchon/off`. Once you use `$vcplusglitchon/off`, GUI allows you to explicitly control the glitch detection.

**Syntax:** `$vcplusdeltacycleon;`

**Note:**

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

`$vcndlusevent`

The `$vcndlusevent` task allows you to record a unique event for a signal at the current simulation time unit.

## Syntax

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

A symbol is displayed in GUI on the signal's waveform and in the **Logic** browser. The `event_name` argument appears in the status bar when you click 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.

Do not enter space between the arguments `E | W | I` and `S | T | D`. Do not include angle brackets `< >`. There is a limit of 244 unique events.

`$vcndlusfile`

Specifies the next VPD file that GUI opens during simulation after it executes the `$vcndlusclose` system task and when it executes the next `$vcndluson` system task.

**Syntax:** `$vcplusfile("filename");`

`$vcplusglitchon`

Turns on the checking for zero delay glitches and other cases of multiple transitions for a signal at the same simulation time.

**Syntax:** `$vcplusglitchon;`

`$vcplusflush`

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 `$vcplusautoflushon` to enable automatic flushing of simulation results to the file when the simulation stops.

**Syntax:** `$vcplusflush;`

`$vcplusmemon`

Records value changes and times for memories and multidimensional arrays (MDAs).

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

Where,

Mda

Specifies the name of the 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.

dim1Lsb

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

`$vcdplusmemoff`

Stops recording value changes and times for memories and multidimensional arrays. Syntax is same as the `$vcdplusmenon` system task.

`$vcdplusmemorydump`

Records (dumps) a snapshot of the values in a memory or multidimensional array into the VPD file. Syntax is same as the `$vcdplusmenon` system task.

`$vcdplusoff`

Stops recording the transition times and values for the nets and registers in the specified module instance or individual nets or registers in the VPD file.

Syntax:

```
$vcdplusoff[ (level_number, module_instance |  
net_or_reg) ] ;
```

Where:

*level\_number*

Specifies the number of hierarchy scope levels for which to stop recording the signal value changes (a zero value records all scope instances to the end of the hierarchy, which is the default value).

*module\_instance*

Specifies the name of the scope for which to stop recording the signal value changes (default is all).

*net\_or\_reg*

Specifies the name of the signal for which to stop recording the signal value changes (default is all).

`$vcdpluson`

Starts recording the transition times and values for the nets and variables in the specified module instance or individual nets or variable in the VPD file. This system task does not enable recording memories or multidimensional arrays with an unpacked dimension.

Syntax:

```
$vcdpluson [ (level_number,module_instance |  
net_or_variable) ];
```

where:

*level\_number*

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

`module_instance`

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

`net_or_variable`

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

---

## 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 the specified assertions and displays what is happening with them (you can only have it display on the next clock of the assertion). The syntax is as follows:

`$assert_monitor([0|1,] assertion_identifier...);`

Where:

`0`

Specifies reporting on the assertions if they are active (VCS checks for its properties). If the assertion is not active, assertions are reported whenever they start.

1

Specifies reporting on the 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, you can 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 the operating system commands.

**Syntax:** `$system("command");`

**Code example:** `$system("mv -f savefile savefile.1");`

`$systemf`

Executes the 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. For more details, see IEEE Verilog LRM Std 1364-2005, Page 311.

`$itor[i]`

Converts integers to real numbers. For more details, see IEEE Verilog LRM Std 1364-2005, Page 310.

`$realtobits`

Passes bit patterns across module ports, converting a real number to a 64-bit representation. For more details, see IEEE Verilog LRM Std 1364-2005, Page 310.

`$rtoi`

Converts real numbers to integers. For more details, see IEEE std 1364-2001, page 310.

---

## System Tasks for Displaying Information

`$display [b|h|0] ;`

Displays the arguments. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 278-285.

`$monitor [b|h|0]`

Displays the data when the arguments change the value. For more details, see IEEE Verilog LRM Std 1364-2005, Page 286.

`$monitoroff`

Disables the `$monitor` system task. For more details, see IEEE Verilog LRM Std 1364-2005, Page 286.

`$monitoron`

Re-enables the `$monitor` system task after it was disabled with the `$monitoroff` system task. For more details, see IEEE Verilog LRM Std 1364-2005, Page 286.

`$strobe [b|h|0] ;`

Displays simulation data on the selected time. For more details, see IEEE Verilog LRM Std 1364-2005, Page 285.

`$write [b|h|0]`

Displays the text. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 278-285.

---

## System Tasks for File I/O

`$fclose`

**Closes a file.** For more details, see IEEE Verilog LRM Std 1364-2005, Pages 287-289.

`$fdisplay[b|h|0]`

**Writes to a file.** For more details, see IEEE Verilog LRM Std 1364-2005, Pages 288-289.

`$ferror`

**Returns additional information about an error condition in file I/O operations.** For more details, see IEEE Verilog LRM Std 1364-2005, Pages 294-295.

`$fflush`

**Writes buffered data to files.** For more details, see IEEE Verilog LRM Std 1364-2005, Page 295.

`$fgetc`

**Reads a character from a file.** For more details, see IEEE Verilog LRM Std 1364-2005, Page 290.

`$fgets`

**Reads a string from a file.** For more details, see IEEE Verilog LRM Std 1364-2005, Pages 290.

`$fmonitor[b|h|0]`

**Writes to a file when an argument value changes.** For more details, see IEEE Verilog LRM Std 1364-2005, Pages 288-289.

`$fopen`

**Opens the files.** For more details, see IEEE Verilog LRM Std 1364-2005, Pages 287-289.

`$fread`

Reads the binary data from a file. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 293.

`$fscanf`

Reads the characters in a file. For more details, see IEEE Verilog LRM Std 1364-2005, Page 291.

`$fseek`

Sets the position of the next read or write operation in a file. For more details, see IEEE Verilog LRM Std 1364-2005, Page 294.

`$fstrobe [b|h|0]`

Writes arguments to a file. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 288-289.

`$ftell`

Returns the offset of a file. For more details, see IEEE Verilog LRM Std 1364-2005, Page 294.

`$fwrite [b|h|0]`

Writes to a file. For more details, see IEEE Verilog LRM Std 1364-2005, Page 288.

`$rewind`

Sets the next read or write operation to the beginning of a file. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 294-295.

`$sformat`

Assigns a string value to a specified signal. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 289.

`$sscanf`

Reads characters from an input stream. For more details, see IEEE Verilog LRM Std 1364-2005, Page 291.

`$swrite`

Assigns a string value to a specified signal, similar to the `$sformat` system function. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 289.

`$ungetc`

Returns a character to the input stream. For more details, see IEEE Verilog LRM Std 1364-2005, Page 290.

---

## System Tasks for Loading Memories

`$readmemb`

Loads binary values from a specified file into a specified memory. For more details, see IEEE Verilog LRM Std 1364-2005, Page 296.

`$readmemh`

Loads hexadecimal values from a specified file into a specified memory. For more details, see IEEE Verilog LRM Std 1364-2005, Page 296.

`$sreadmemb`

Loads specified binary string values into memories. For more details, see IEEE Verilog LRM Std 1364-2005, Page 517.

`$sreadmemh`

Loads specified string hexadecimal values into memories. For more details, see IEEE Verilog LRM Std 1364-2005, Page 517.

`$writememb`

Writes binary data from a specified memory to a specified file.

**Syntax:** `$writememb ("filename", memory [, start_address] [, end_address]);`

**Code example:** `$writememb ("testfile.txt", mem, 0, 255);`

`$writememh`

Writes hexadecimal data from a specified memory to a specified 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. For more  
details, see IEEE Verilog LRM Std 1364-2005, Page 299.

\$timeformat

Specifies how the %t format specification reports time information.  
For more details, see IEEE Verilog LRM Std 1364-2005, Page  
300.

---

## System Tasks for Simulation Control

\$stop

Causes a simulation to be suspended. For more details, see IEEE  
Verilog LRM Std 1364-2005, Page 302.

\$finish

Causes a simulation to end. For more details, see IEEE Verilog  
LRM Std 1364-2005, Page 302.

---

## System Tasks for Timing Checks

`$disable_warnings`

Disables the display of timing violations and 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.

---

## Timing Checks for Clock and Control Signals

`$hold`

Reports a timing violation when a data event happens too soon after a reference event. For more details, see IEEE Verilog LRM Std 1364-2005, Page 242.

`$nochange`

Reports a timing violation if the data event occurs during the specified level of the control signal (the reference event). For more details, see IEEE Verilog LRM Std 1364-2005, Page 257.

`$period`

Reports a timing violation when an edge triggered event happens too soon after the previous matching edge triggered an event on a signal. For more details, see IEEE Verilog LRM Std 1364-2005, Page 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. For more details, see IEEE Verilog LRM Std 1364-2005, Page 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);
```

For more details, see IEEE Verilog LRM Std 1364-2005, Page 247.

`$removal`

Reports a timing violation if the reference event, typically an asynchronous control signal, happens too soon after the data event, the clock signal. For more details, see IEEE Verilog LRM Std 1364-2005, Page 245.

`$setup`

Reports a timing violation when the data event happens before and too close to the reference event. For more details, see IEEE Verilog LRM Std 1364-2005 Page 241. This timing check also has an extended syntax, such as the `$recrem` timing check. This extended syntax is not described in IEEE Verilog LRM Std 1364-2005.

`$setuphold`

Combines the `$setup` and `$hold` system tasks. For the official description, see IEEE Verilog LRM Std 1364-2005, Page 238.

The syntax is as follows: `$setuphold(reference_event,  
data_event, setup_limit, hold_limit, notifier,  
timestamp_cond, timecheck_cond, delay_reference,  
delay_data);`

`$skew`

Reports a timing violation when a reference event happens too long after a data event. For more information, see IEEE Verilog LRM Std 1364-2005, Page 249.

`$width`

Reports a timing violation when a pulse is narrower than the specified limit. For more information, see IEEE Verilog LRM Std 1364-2005, Page 255. VCS ignores the threshold argument.

---

## System Tasks for PLA Modeling

`$async$and$array to $sync$nor$plane`

For more details, see IEEE Verilog LRM Std 1364-2005, Page 303.

---

## System Tasks for Stochastic Analysis

`$q_add`

Places an entry on a queue in stochastic analysis. For more details, see IEEE Verilog LRM Std 1364-2005, Page 307.

`$q_exam`

Provides statistical information about the activity that is in the queue. For more details, see IEEE Verilog LRM Std 1364-2005, Page 307.

`$q_full`

VCS returns 0 if the queue is not full. It returns a 1 if the queue is full. For more details, see IEEE Verilog LRM Std 1364-2005 Page, 308.

`$q_initialize`

Creates a new queue. For more details, see IEEE Verilog LRM Std 1364-2005, Page 307.

`$q_remove`

Receives an entry from a queue. For more details, see IEEE Verilog LRM Std 1364-2005, Page 307.

---

## System Tasks for Simulation Time

`$realtime`

Returns a real number time. For more details, see IEEE Verilog LRM Std 1364-2005, Page 310.

`$stime`

Returns an unsigned integer that is a 32-bit time. For more details, see IEEE Verilog LRM Std 1364-2005, Page 309.

`$time`

Returns an integer that is a 64-bit time. For more details, see IEEE Verilog LRM Std 1364-2005, Page 309.

---

## System Tasks for Probabilistic Distribution

`$dist_exponential`

Returns random numbers where the distribution function is exponential. For more details, see IEEE Verilog LRM Std 1364-2005, Page 312.

`$dist_normal`

Returns random numbers with a specified mean and standard deviation. For more details, see IEEE Verilog LRM Std 1364-2005, Page 312.

`$dist_poisson`

Returns random numbers with a specified mean. For more details, see IEEE Verilog LRM Std 1364-2005, Page 312.

`$dist_uniform`

Returns random numbers uniformly distributed between parameters. For more details, see IEEE Verilog LRM Std 1364-2005, Page 312.

`$random`

Provides a random number. For more details, see IEEE Verilog LRM Std 1364-2005 Page 311. Using this system function in certain kind of statements might cause simulation failure.

`$get_initial_random_seed`

Returns the integer number used as the seed for a simulation run, if the seed was set by `+ntb_random_seed=value` or by `+ntb_random_seed_automatic` or returns the default random seed value if the seed was not set using one of those two options. The default random seed value is 1.

---

## System Tasks for Resetting VCS

`$reset`

Resets the simulation time to 0. For more details, see IEEE Verilog LRM Std 1364-2005, Page 514.

`$reset_count`

Keeps track of the number of times VCS executes the `$reset` system task in a simulation session. For more details, see IEEE Verilog LRM Std 1364-2005, Page 741-742.

```
$reset_value
```

System function that you can use to pass a value from, before or after VCS executes the \$reset system task, that is, you can enter a *reset\_value* integer argument to the \$reset system task. After VCS resets the simulation, the \$reset\_value system function returns this integer argument. For more details, see IEEE Verilog LRM Std 1364-2005, Page 514.

---

## General System Tasks and Functions

### Checks for a Plusarg

```
$test$plusargs
```

Checks for the existence of a given plusarg on the runtime executable command line.

Syntax: \$test\$plusargs ("plusarg\_without\_the\_+");.

### SDF Files

```
$sdf_annotation
```

Tells VCS to back-annotate delay values from an SDF file to your Verilog design.

### Counting the Drivers on a Net

```
$countdrivers
```

Counts the number of drivers on a net. For more details, see IEEE Verilog LRM Std 1364-2005, Pages 511-512.

## Depositing Values

`$deposit`

Deposits a value on a net, or variable, or cross-module references (XMRs). This deposited value overrides the value from any other driver of the net, or variable, or XMRs. The value propagates to all loads of the net, or variable, or XMRs. A subsequent simulation event can override the deposited value. You cannot use this system task to deposit values to part-selects.

Syntax: `$deposit (net_or_variable_or_xmr, value);`

The deposited value can be the value of another net, or variable, or XMRs. VCS also supports `$deposit` on array bit-select with non-constant index in behavioral context only.

## Fast Processing Stimulus Patterns

`$getpattern`

Provides for fast processing of stimulus patterns. For more details, see IEEE Verilog LRM Std 1364-2005, Page 512.

## Saving and Restarting The Simulation State

`$save`

Saves the current simulation state in a file. For more details, see IEEE Verilog LRM Std 1364-2005, Page 515.

`$restart`

Restores the simulation to the state that you had saved in the check file with the `$save` system task. For more details, see IEEE Verilog LRM Std 1364-2005, Page 515.

## Checking for X and Z Values in Conditional Expressions

`$xzcheckon`

Displays a warning message every time VCS evaluates a conditional expression to have an `X` or `Z` value.

Syntax: `$xzcheckon(level_number,hierarchical_name)`

*level\_number* (Optional)

Specifies the number of hierarchy scope levels from the specified module instance to check for `X` and `Z` values. If the number is 0 or not specified, VCS checks all scope instances to the end of the hierarchy.

*hierarchical\_name* (Optional)

Hierarchical name of the module instance, that is, the top-level instance of the subhierarchy for which you want to enable checking.

`$xzcheckoff`

Suppress the warning message every time VCS evaluates a conditional expression to have an `X` or `Z` value.

Syntax:

`$xzcheckoff(level_number,hierarchical_name)`

*level\_number* (Optional)

Specifies the number of hierarchy scope levels from the specified module instance, for which `X` and `Z` value check is disabled. If the number is 0 or not specified, VCS disables the check on all scope instances to the end of the hierarchy.

*hierarchical\_name* (Optional)

Hierarchical name of the module instance, that is, the top-level instance of the subhierarchy for which you want to disable checking.

## Calculating Bus Widths

`$clog2`

Use this system function to calculate the bus widths, such as from parameters. The following illustrates its use:

```
integer result;  
result = $clog2(n);
```

Note:

If the argument has `x` or `z` values, then that bit is considered as 1 or 0 respectively by VCS. The argument could be a vector with a few bits having `x` or `z` values.

For more information on this system function, see the IEEE SystemVerilog LRM Std 1800-2012 Section 17.11.1 “Integer math functions”.

## Displaying the Method Stack

```
$stack();
```

Displays stack information of lines in your code that trigger the execution of an entry of this system task. Multiple stacks are displayed for multiple entries of this system task. You can use this system task for debugging and back tracing.

You can enter this system task in modules and SystemVerilog programs, classes, packages, and interfaces. You can also enter this system task in user-defined tasks and functions and in initial, always, and final blocks (Synopsys recommends naming begin-end blocks in these initial, always, and final blocks).

The following code example illustrates an entry of this system task in the `test.sv` file:

```
program test;

    class C;
        static function f3();
            $stack(); // line 5
        endfunction
    endclass

    function f1();
        f2(); // line 10
    endfunction

    function f2();
        C::f3(); // line 14
    endfunction

    task t();
        f1(); // line 18
    endtask
```

```

task t1();
    t(); // line 22
endtask

initial begin :B0
    t1(); // line 26
end

endprogram

module top;
    test p();
endmodule

```

At runtime, VCS displays the following method stack information:

```

#0 in \C::f3  at test.sv:5
#1 in f2 at test.sv:14
#2 in f1 at test.sv:10
#3 in t at test.sv:18
#4 in t1 at test.sv:22
#5 in B0 at test.sv:26
#6 in top.p

```

In this method stack:

#0 is always the line containing the \$stack system task. In this example, it is in class C. The user-defined function f3, at line number 5 is test.sv.

#1 is a call of function f3 in the user-defined function f2 at line number 14. VCS executing function f2 causes VCS to execute the function f3.

#2 is a call of function f2 in user-defined function f1 at line number 10. VCS executing function f1 causes VCS to execute the function f2.

#3 is a call of function f1 in the user-defined task t at line number 18. VCS executing task t causes VCS to execute the function f1.

#4 is a task enabling statement for task t in the user-defined task t1 at line number 22. VCS executing task t1 causes VCS to execute the task t.

#5 is a task enabling statement for task t1 in the begin-end block named B0. VCS executing block B0 causes VCS to execute the task t1.

#6 is the instance of program test. VCS does not include the line number because this instantiation is in the top-level module.

If debug mode is enabled, you can call the \$stack system task from GUI.

For example:

```
ucli% step
in program p 3 1 4
mda_stack.v, 17 : $stack();

ucli% stack
0 : -line 14 -file mda_stack.v -scope
{test.P1.unnamed$$_4}
1 : -line 14 -file mda_stack.v -scope
{test.P1.unnamed$$_4.unnamed$$_3}
2 : -line 17 -file mda_stack.v -scope
{test.P1.unnamed$$_4.unnamed$$_3.unnamed$$_1}
```

If the \$stack system task is called inside a function that is exported to DPI, the “DPI function” name is displayed. The line number and details of the C code are not displayed. For example:

```
#0 in int_from_sv at dpi_test.v:14
#1 in DPI function
#2 in int_test
```

In mixed-language simulations, the `$stack()` system task call displays only the information about the hierarchy. For example:

```
#0 in \top.r1.U1.d1.XOR2_INST at xor.v:6
#0 in top.xor_i at xor.v:6
#0 in \top.r1.U1 at t_ff_using_xor.v:11
```

In simulations with System-C at the top level, the `$stack()` system task call displays hierarchical information similar to the following:

```
#0 in \c::f at adder.v:5
#1 in \c::t at adder.v:9
#2 in unnamed$$_1 at adder.v:18
#3 in sYsTeMcToP.sc_top.adder_inst.p1
#0 in \c::f at adder.v:5
#1 in unnamed$$_1 at adder.v:19
#2 in sYsTeMcToP.sc_top.adder_inst.p1
```

In the OpenVera-SystemVerilog interoperability flow, the `$stack()` system task call displays both SystemVerilog and OpenVera information similar to the following:

```
#0 in f1 at fn_rt_sv.vr:14
#1 in \C::vera_method2 at fn_rt_sv.vr:7
#2 in unnamed$$_2 at fn_rt_sv.v:36
#3 in p
```

```
$psstack();
```

Returns a SystemVerilog string. The string provides hierarchical information of the scopes from where the system function is being called.

For example:

```
program test;

    function f();
        $display("psstack = %s", $psstack());
    endfunction

    task t2();
        f();
    endtask

    initial begin:psstack
        t2();
    end

endprogram
```

At runtime, VCS displays the following hierarchical information:

```
psstack = test.psstack.t2.f
```

```
$sstack();
```

Writes the output of \$stack in the form of multi-line string to the variable argument of \$sstack.

The following is an example for the \$sstack usage:

```

class C;
    string str;
    function foo1;
        $sstack(str);
        $display(str);
    endfunction
    function foo2;
        foo1();
    endfunction
    task run;
        foo2();
    endtask
endclass

program test;
    C obj;
    initial begin
        obj = new();
        obj.run();
    end
endprogram

```

**The output is as follows:**

```

#0 in \C::foo1  at test.v:4
#1 in \C::foo2  at test.v:8
#2 in \C::run   at test.v:11
#3 in test      at test.v:19
#4 in test

```

---

## **IEEE Standard System Tasks Not Yet Implemented**

The following Verilog system tasks are included in the IEEE Verilog LRM Std 1364-2005, but are not yet implemented in VCS:

- `$dist_chi_square`
- `$dist_erlang`
- `$dist_t`

# F

## PLI Access Routines

---

VCS includes a number of access routines. This appendix describes these access routines in the following sections:

- “Access Routines for Reading and Writing to Memories”
- “Access Routines for Multidimensional Arrays”
- “Access Routines for Probabilistic Distribution”
- “Access Routines for Returning a Pointer to a Parameter Value”
- “Access Routines for Extended VCD Files”
- “Access Routines for Line Callbacks”
- “Access Routines for Source Protection”
- “Access Routine for Signal in a Generate Block”
- “VCS API Routines”

---

## Access Routines for Reading and Writing to Memories

VCS includes a number of access routines for reading and writing to a memory.

These access routines are as follows:

`acc_setmem_int`

Writes an integer value to specific bits in a Verilog memory word. See “[“acc\\_setmem\\_int”](#) for details.

`acc_getmem_int`

Reads an integer value from specific bits in a Verilog memory word. See “[“acc\\_getmem\\_int”](#) for details.

`acc_clearmem_int`

Clears a memory, that is, writes zeros to all bits. See “[“acc\\_clearmem\\_int”](#) for details.

`acc_setmem_hexstr`

Writes a hexadecimal string value to specific bits in a Verilog memory word. See “[“acc\\_setmem\\_hexstr”](#) for details.

`acc_getmem_hexstr`

Reads a hexadecimal string value from specific bits in a Verilog memory word. See “[“acc\\_getmem\\_hexstr”](#) for details.

`acc_setmem_bitstr`

Writes a string of binary bits (including x and z) to a Verilog memory word. See “[“acc\\_setmem\\_bitstr”](#) for details.

`acc_getmem_bitstr`

Reads a bit string from specific bits in a Verilog memory word.  
See “[“acc\\_getmem\\_bitstr”](#) for details.

`acc_handle_mem_by_fullname`

Returns the handle used by `acc_readmem`. See  
“[“acc\\_handle\\_mem\\_by\\_fullname”](#) for details.

`acc_readmem`

Reads a data file and writes the contents to a memory. See  
“[“acc\\_readmem”](#) for details.

`acc_getmem_range`

Returns the upper and lower limits of a memory. See  
“[“acc\\_getmem\\_range”](#) for details.

`acc_getmem_size`

Returns the number of elements (or words or addresses) in a  
memory. See “[“acc\\_getmem\\_size”](#) for details.

`acc_getmem_word_int`

Returns the integer of a memory element. See  
“[“acc\\_getmem\\_word\\_int”](#) 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”](#) 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.

**Table 0-1.**

| 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> |             |                                                                                                |
| Returns:                                                                                                                                                                                                                                                   | Type                                                             | Description |                                                                                                |
| Arguments:                                                                                                                                                                                                                                                 | Type                                                             | Name        | Description                                                                                    |
| Related routines:                                                                                                                                                                                                                                          | handle                                                           | memhand     | Handle to memory                                                                               |
|                                                                                                                                                                                                                                                            | int                                                              | value       | The integer value written in binary format to the bits in the word.                            |
|                                                                                                                                                                                                                                                            | int                                                              | row         | The memory array index.                                                                        |
|                                                                                                                                                                                                                                                            | int                                                              | start       | Bit number of the leftmost bit in the memory word where this routine starts writing the value. |
|                                                                                                                                                                                                                                                            | int                                                              | length      | Starting with the start bit, specifies the total number of bits this routine writes to.        |
| acc_getmem_int<br>acc_setmem_hexstr<br>acc_getmem_hexstr<br>acc_setmem_bitstr<br>acc_getmem_bitstr<br>acc_clearmem_int<br>acc_handle_mem_by_fullname<br>acc_readmem<br>acc_getmem_range<br>acc_getmem_size<br>acc_getmem_word_int<br>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.

**Table 0-1.**

| 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 leftmost bit in the memory word where this routine starts reading the value. |
|                   | int                                                                                                                                                                                                                                                                                                                                                                                                                    | length                                        | Specifies the total number of bits this routine reads starting with the start bit.             |
| Related routines: | <code>acc_setmem_int</code><br><code>acc_setmem_hexstr</code><br><code>acc_getmem_hexstr</code><br><code>acc_setmem_bitstr</code><br><code>acc_getmem_bitstr</code><br><code>acc_clearmem_int</code><br><code>acc_handle_mem_by_fullname</code><br><code>acc_readmem</code><br><code>acc_getmem_range</code><br><code>acc_getmem_size</code><br><code>acc_getmem_word_int</code><br><code>acc_getmem_word_range</code> |                                               |                                                                                                |

---

## **acc\_clearmem\_int**

You use the `acc_clearmem_int` access routine to write zeros to all bits in a memory.

**Table 0-1.**

| acc_clearmem_int    |                                                                                                                                                                                                                                                                                                                                                                                                                      |                  |      |             |                     |                      |                  |  |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|------|-------------|---------------------|----------------------|------------------|--|
| Synopsis:           | Clears a memory word.                                                                                                                                                                                                                                                                                                                                                                                                |                  |      |             |                     |                      |                  |  |
| Syntax:             | <code>acc_clearmem_int (memhand)</code>                                                                                                                                                                                                                                                                                                                                                                              |                  |      |             |                     |                      |                  |  |
| Returns:            | <table><thead><tr><th>Type</th><th>Description</th></tr></thead><tbody><tr><td><code>void</code></td><td></td></tr></tbody></table>                                                                                                                                                                                                                                                                                  |                  | Type | Description | <code>void</code>   |                      |                  |  |
| Type                | Description                                                                                                                                                                                                                                                                                                                                                                                                          |                  |      |             |                     |                      |                  |  |
| <code>void</code>   |                                                                                                                                                                                                                                                                                                                                                                                                                      |                  |      |             |                     |                      |                  |  |
| Arguments:          | <table><thead><tr><th>Type</th><th>Name</th><th>Description</th></tr></thead><tbody><tr><td><code>handle</code></td><td><code>memhand</code></td><td>Handle to memory</td></tr></tbody></table>                                                                                                                                                                                                                      | Type             | Name | Description | <code>handle</code> | <code>memhand</code> | Handle to memory |  |
| Type                | Name                                                                                                                                                                                                                                                                                                                                                                                                                 | Description      |      |             |                     |                      |                  |  |
| <code>handle</code> | <code>memhand</code>                                                                                                                                                                                                                                                                                                                                                                                                 | Handle to memory |      |             |                     |                      |                  |  |
| Related routines:   | <code>acc_setmem_int</code><br><code>acc_getmem_int</code><br><code>acc_setmem_hexstr</code><br><code>acc_getmem_hexstr</code><br><code>acc_setmem_bitstr</code><br><code>acc_getmem_bitstr</code><br><code>acc_handle_mem_by_fullname</code><br><code>acc_readmem</code><br><code>acc_getmem_range</code><br><code>acc_getmem_size</code><br><code>acc_getmem_word_int</code><br><code>acc_getmem_word_range</code> |                  |      |             |                     |                      |                  |  |

## **Examples**

The following code examples illustrate how to use `acc_getmem_int`, `acc_setmem_int`, and `acc_clearmem_int`:

- [Example F-1](#) shows C code that includes a number of functions to be associated with user-defined system tasks.

- [Example F-2](#) shows the PLI table for associating these functions with these system tasks.
- [Example F-3](#) shows the Verilog source code containing these system tasks.

*Example F-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 F-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 F-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.

**Table 0-1.**

| 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:          | 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 leftmost bit in the memory word where this routine starts writing the string. |
| Related routines: | <code>acc_setmem_int</code><br><code>acc_getmem_int</code><br><code>acc_getmem_hexstr</code><br><code>acc_setmem_bitstr</code><br><code>acc_getmem_bitstr</code><br><code>acc_clearmem_int</code><br><code>acc_handle_mem_by_fullname</code><br><code>acc_readmem</code><br><code>acc_getmem_range</code><br><code>acc_getmem_size</code><br><code>acc_getmem_word_int</code><br><code>acc_getmem_word_range</code> |             |                                                                                                 |

This routine takes a value argument which is a hexadecimal string of any size and puts its corresponding binary representation into the memory word indexed by `row`, starting at the bit number `start`.

## Examples

The following code examples illustrates the use of `acc_setmem_hexstr`:

- [Example F-4](#) shows the C source code for an application that calls `acc_setmem_hexstr`.
- [Example F-5](#) shows the contents of a data file read by the application.
- [Example F-6](#) shows the PLI table file that associates the user-defined system task in the Verilog code with the application.
- [Example F-7](#) shows the Verilog source that calls the application.

### *Example F-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 F-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 F-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 F-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 F-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 F-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.

**Table 0-1.**

| acc_getmem_hexstr |                                                                                                                                                                                                                                                                                                                                                                                                                     |             |                                                                                                 |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|-------------------------------------------------------------------------------------------------|
| Synopsis:         | Returns a hexadecimal string from a Verilog memory.                                                                                                                                                                                                                                                                                                                                                                 |             |                                                                                                 |
| Syntax:           | <code>acc_getmem_hexstr (memhand, hexStrValue, row, start, len)</code>                                                                                                                                                                                                                                                                                                                                              |             |                                                                                                 |
| Returns:          | Type                                                                                                                                                                                                                                                                                                                                                                                                                | Description |                                                                                                 |
|                   | void                                                                                                                                                                                                                                                                                                                                                                                                                |             |                                                                                                 |
| Arguments:        | Type                                                                                                                                                                                                                                                                                                                                                                                                                | Name        | Description                                                                                     |
|                   | handle                                                                                                                                                                                                                                                                                                                                                                                                              | memhand     | Handle to memory                                                                                |
|                   | char *                                                                                                                                                                                                                                                                                                                                                                                                              | hexStrValue | Pointer to a character array into which the string is written                                   |
|                   | int                                                                                                                                                                                                                                                                                                                                                                                                                 | row         | The memory array index                                                                          |
|                   | int                                                                                                                                                                                                                                                                                                                                                                                                                 | start       | Bit number of the leftmost bit in the memory word where this routine starts reading the string. |
|                   | int                                                                                                                                                                                                                                                                                                                                                                                                                 | length      | Specifies the total number of bits this routine reads starting with the start bit.              |
| Related routines: | <code>acc_setmem_int</code><br><code>acc_getmem_int</code><br><code>acc_setmem_hexstr</code><br><code>acc_setmem_bitstr</code><br><code>acc_getmem_bitstr</code><br><code>acc_clearmem_int</code><br><code>acc_handle_mem_by_fullname</code><br><code>acc_readmem</code><br><code>acc_getmem_range</code><br><code>acc_getmem_size</code><br><code>acc_getmem_word_int</code><br><code>acc_getmem_word_range</code> |             |                                                                                                 |

---

## acc\_setmem\_bitstr

You use the `acc_setmem_bitstr` access routine for writing a string of binary bits (including x and z) to a Verilog memory.

**Table 0-1.**

| 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 |                                                                                                 |
| Arguments:        | void                                                                                                                                                                                                                                                                                                                                                                                                                |             |                                                                                                 |
|                   | Type                                                                                                                                                                                                                                                                                                                                                                                                                | Name        | Description                                                                                     |
| Related routines: | handle                                                                                                                                                                                                                                                                                                                                                                                                              | memhand     | Handle to memory                                                                                |
|                   | char *                                                                                                                                                                                                                                                                                                                                                                                                              | bitStrValue | Bit string                                                                                      |
|                   | int                                                                                                                                                                                                                                                                                                                                                                                                                 | row         | The memory array index                                                                          |
|                   | int                                                                                                                                                                                                                                                                                                                                                                                                                 | start       | Bit number of the leftmost bit in the memory word where this routine starts writing the string. |
| Related routines: | <code>acc_setmem_int</code><br><code>acc_getmem_int</code><br><code>acc_setmem_hexstr</code><br><code>acc_getmem_hexstr</code><br><code>acc_getmem_bitstr</code><br><code>acc_clearmem_int</code><br><code>acc_handle_mem_by_fullname</code><br><code>acc_readmem</code><br><code>acc_getmem_range</code><br><code>acc_getmem_size</code><br><code>acc_getmem_word_int</code><br><code>acc_getmem_word_range</code> |             |                                                                                                 |

This routine takes a value argument that is a bit string of any size, which can include the x and z values, and puts its corresponding binary representation into the memory word indexed by `row`, starting at the bit number `start`.

---

## **acc\_getmem\_bitstr**

You use the `acc_getmem_bitstr` access routine to get a bit string, including x and z values, from a Verilog memory.

**Table 0-1.**

| acc_getmem_bitstr |                                                                                                                                                                                                                                                                                                                                                                                                                     |             |                                                                                                 |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|-------------------------------------------------------------------------------------------------|
| Synopsis:         | Returns a hexadecimal string from a Verilog memory.                                                                                                                                                                                                                                                                                                                                                                 |             |                                                                                                 |
| Syntax:           | <code>acc_getmem_bitstr (memhand,bitStrValue,row,start,len)</code>                                                                                                                                                                                                                                                                                                                                                  |             |                                                                                                 |
| Returns:          | Type                                                                                                                                                                                                                                                                                                                                                                                                                | Description |                                                                                                 |
|                   | void                                                                                                                                                                                                                                                                                                                                                                                                                |             |                                                                                                 |
| Arguments:        | Type                                                                                                                                                                                                                                                                                                                                                                                                                | Name        | Description                                                                                     |
|                   | handle                                                                                                                                                                                                                                                                                                                                                                                                              | memhand     | Handle to memory                                                                                |
|                   | char *                                                                                                                                                                                                                                                                                                                                                                                                              | hexStrValue | Pointer to a character array into which the string is written                                   |
|                   | int                                                                                                                                                                                                                                                                                                                                                                                                                 | row         | The memory array index                                                                          |
|                   | int                                                                                                                                                                                                                                                                                                                                                                                                                 | start       | Bit number of the leftmost bit in the memory word where this routine starts reading the string. |
|                   | int                                                                                                                                                                                                                                                                                                                                                                                                                 | length      | Specifies the total number of bits this routine reads starting with the start bit.              |
| Related routines: | <code>acc_setmem_int</code><br><code>acc_getmem_int</code><br><code>acc_setmem_hexstr</code><br><code>acc_getmem_hexstr</code><br><code>acc_setmem_bitstr</code><br><code>acc_clearmem_int</code><br><code>acc_handle_mem_by_fullname</code><br><code>acc_readmem</code><br><code>acc_getmem_range</code><br><code>acc_getmem_size</code><br><code>acc_getmem_word_int</code><br><code>acc_getmem_word_range</code> |             |                                                                                                 |

---

## **acc\_handle\_mem\_by\_fullname**

Returns a handle to a memory that can only be used as a parameter to acc\_readmem.

**Table 0-1.**

| acc_handle_mem_by_fullname |                                                                                                                                                                                                                                                          |                        |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|
| Synopsis:                  | Returns a handle to be used as a parameter to acc_readmem only                                                                                                                                                                                           |                        |
| Syntax:                    | acc_handle_mem_by_fullname (fullMemInstName)                                                                                                                                                                                                             |                        |
| Type                       | Description                                                                                                                                                                                                                                              |                        |
| Returns:                   | handle                                                                                                                                                                                                                                                   | Handle to the instance |
| Type                       | Name                                                                                                                                                                                                                                                     | Description            |
| Arguments:                 | char*                                                                                                                                                                                                                                                    | fullMemInstName        |
| Related routines:          | acc_setmem_int<br>acc_getmem_int<br>acc_setmem_hexstr<br>acc_getmem_hexstr<br>acc_setmem_bitstr<br>acc_getmem_bitstr<br>acc_handle_mem_by_fullname<br>acc_readmem<br>acc_getmem_range<br>acc_getmem_size<br>acc_getmem_word_int<br>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.**

**Table 0-1.**

| acc_readmem       |                                                                                                                                                                                                                                                          |                                                                                                    |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| Synopsis:         | Reads a data file into a memory                                                                                                                                                                                                                          |                                                                                                    |
| Syntax:           | acc_readmem (memhandle, data_file, format)                                                                                                                                                                                                               |                                                                                                    |
| >Returns:         | Type Description                                                                                                                                                                                                                                         |                                                                                                    |
| Arguments:        | Type                                                                                                                                                                                                                                                     | Name Description                                                                                   |
|                   | handle                                                                                                                                                                                                                                                   | memhandle Handle returned by acc_handle_mem_fullname                                               |
|                   | 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: | acc_setmem_int<br>acc_getmem_int<br>acc_setmem_hexstr<br>acc_getmem_hexstr<br>acc_setmem_bitstr<br>acc_getmem_bitstr<br>acc_handle_mem_by_fullname<br>acc_readmem<br>acc_getmem_range<br>acc_getmem_size<br>acc_getmem_word_int<br>acc_getmem_word_range |                                                                                                    |

## Examples

The following code examples illustrate the use of acc\_readmem and acc\_handle\_mem\_by\_fullname.

**Example F-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 F-9 The PLI Table File**

```
$test_acc_readmem call=test_acc_readmem
```

**Example F-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.

**Table 0-1.**

| 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>                                                                                                                                                                                                                                                                                                               |                                           |      |             |                   |  |
| Returns:          | <table><thead><tr><th>Type</th><th>Description</th></tr></thead><tbody><tr><td><code>void</code></td><td></td></tr></tbody></table>                                                                                                                                                                                                                                                 |                                           | Type | Description | <code>void</code> |  |
| Type              | Description                                                                                                                                                                                                                                                                                                                                                                         |                                           |      |             |                   |  |
| <code>void</code> |                                                                                                                                                                                                                                                                                                                                                                                     |                                           |      |             |                   |  |
| Arguments:        | Type                                                                                                                                                                                                                                                                                                                                                                                | Name Description                          |      |             |                   |  |
|                   | <code>handle</code>                                                                                                                                                                                                                                                                                                                                                                 | <code>memhandle</code> Handle to a memory |      |             |                   |  |
|                   | <code>int*</code>                                                                                                                                                                                                                                                                                                                                                                   | <code>p_left_index</code> Pointer to int  |      |             |                   |  |
|                   | <code>int</code>                                                                                                                                                                                                                                                                                                                                                                    | <code>p_right_index</code> Pointer to int |      |             |                   |  |
| Related routines: | <code>acc_setmem_int</code><br><code>acc_getmem_int</code><br><code>acc_setmem_hexstr</code><br><code>acc_getmem_hexstr</code><br><code>acc_setmem_bitstr</code><br><code>acc_getmem_bitstr</code><br><code>acc_handle_mem_by_fullname</code><br><code>acc_readmem</code><br><code>acc_getmem_size</code><br><code>acc_getmem_word_int</code><br><code>acc_getmem_word_range</code> |                                           |      |             |                   |  |

---

## **acc\_getmem\_size**

You use the `acc_getmem_size` access routine to access the number of elements in a memory.

**Table 0-1.**

| acc_getmem_size   |                                            |                                    |
|-------------------|--------------------------------------------|------------------------------------|
| Synopsis:         | Returns the number of elements in a memory |                                    |
| Syntax:           | <code>acc_getmem_size (memhandle)</code>   |                                    |
|                   | Type                                       | Description                        |
| Returns:          | int                                        | The number of elements in a memory |
|                   | Type                                       | Name                               |
| Arguments:        | handle                                     | memhandle                          |
| Related routines: | 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).

**Table 0-1.**

| acc_getmem_word_int |                                                                                                                                                                                                                                                                                                                                                                                  |                            |                                                  |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------|--------------------------------------------------|
| Synopsis:           | Returns the integer value of an element                                                                                                                                                                                                                                                                                                                                          |                            |                                                  |
| Syntax:             | <code>acc_getmem_word_int (memhandle, row)</code>                                                                                                                                                                                                                                                                                                                                |                            |                                                  |
| Returns:            | Type                                                                                                                                                                                                                                                                                                                                                                             | Description                |                                                  |
|                     | int                                                                                                                                                                                                                                                                                                                                                                              | The integer value of a row |                                                  |
| Arguments:          | Type                                                                                                                                                                                                                                                                                                                                                                             | Name                       | Description                                      |
|                     | handle                                                                                                                                                                                                                                                                                                                                                                           | memhandle                  | Handle to a memory                               |
|                     | int                                                                                                                                                                                                                                                                                                                                                                              | row                        | The element (word address, or row) in the memory |
| Related routines:   | <code>acc_setmem_int</code><br><code>acc_getmem_int</code><br><code>acc_setmem_hexstr</code><br><code>acc_getmem_hexstr</code><br><code>acc_setmem_bitstr</code><br><code>acc_getmem_bitstr</code><br><code>acc_handle_mem_by_fullname</code><br><code>acc_readmem</code><br><code>acc_getmem_range</code><br><code>acc_getmem_size</code><br><code>acc_getmem_word_range</code> |                            |                                                  |

---

## **acc\_getmem\_word\_range**

You use the `acc_getmem_word_range` access routine to access the least significant bit of an element (or word, address, or row) and the length of the element.

**Table 0-1.**

| 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>                                                                                                                                                                                                                                                                                                                       |                        |                                      |
|                       | Type                                                                                                                                                                                                                                                                                                                                                                           | Description            |                                      |
| Returns:              | <code>void</code>                                                                                                                                                                                                                                                                                                                                                              |                        |                                      |
|                       | Type                                                                                                                                                                                                                                                                                                                                                                           | Name                   | Description                          |
| Arguments:            | <code>handle</code>                                                                                                                                                                                                                                                                                                                                                            | <code>memhandle</code> | Handle to a memory                   |
|                       | <code>int*</code>                                                                                                                                                                                                                                                                                                                                                              | <code>lsb</code>       | Pointer to the least significant bit |
|                       | <code>int*</code>                                                                                                                                                                                                                                                                                                                                                              | <code>len</code>       | Pointer to the length of the element |
| Related routines:     | <code>acc_setmem_int</code><br><code>acc_getmem_int</code><br><code>acc_setmem_hexstr</code><br><code>acc_getmem_hexstr</code><br><code>acc_setmem_bitstr</code><br><code>acc_getmem_bitstr</code><br><code>acc_handle_mem_by_fullname</code><br><code>acc_readmem</code><br><code>acc_getmem_range</code><br><code>acc_getmem_size</code><br><code>acc_getmem_word_int</code> |                        |                                      |

---

## **Access Routines for Multidimensional Arrays**

The type for multi-dimensional arrays is defined in the `vcs_acc_user.h` file. Its name is `accMda`.

You also have the following tf and access routines for accessing data in a multi-dimensional array:

`tf_mdanodeinfo` and `tf_imdanodeinfo`

Returns access parameter node information from a multi-dimensional array. See “[tf\\_mdanodeinfo and tf\\_imdanodeinfo](#)” for details.

`acc_get_mda_range`

Returns all the ranges of the multi-dimensional array. See “[acc\\_get\\_mda\\_range](#)” for details.

`acc_get_mda_word_range`

Returns the range of an element in a multi-dimensional array. See “[acc\\_get\\_mda\\_word\\_range\(\)](#)” for details.

`acc_getmda_bitstr`

Reads a bit string, including X and Z values, from an element in a multi-dimensional array. See “[acc\\_getmda\\_bitstr\(\)](#)” for details.

`acc_setmda_bitstr`

Writes a bit string, including X and Z values, from an element in a multi-dimensional array. See “[acc\\_setmda\\_bitstr\(\)](#)” for details.

---

## **tf\_mdanodeinfo and tf\_imdanodeinfo**

You use these routines to access parameter node information from a multi-dimensional array.

**Table 0-1.**

| tf_mdanodeinfo(), tf_imdanodeinfo()                                  |                                                                                                                       |                                                                                                    |
|----------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| Synopsis:                                                            | Returns access parameter node information from a multi-dimensional array.                                             |                                                                                                    |
| Syntax:                                                              | <code>tf_mdanodeinfo(nparam, mdanodeinfo_p)</code><br><code>tf_imdanodeinfo(nparam, mdanodeinfo_p, instance_p)</code> |                                                                                                    |
| >Returns:                                                            | Type                                                                                                                  | Description                                                                                        |
| mdanodeinfo_p *                                                      |                                                                                                                       |                                                                                                    |
| The value of the second argument if successful; 0 if an error occurs |                                                                                                                       |                                                                                                    |
| Arguments:                                                           |                                                                                                                       |                                                                                                    |
| Arguments:                                                           | int                                                                                                                   | nparam<br>Index number of the multi-dimensional array parameter                                    |
|                                                                      | struct t_tfmdanodeinfo *                                                                                              | mdanodeinfo_p<br>Pointer to a variable declared as the <code>t_tfmdanodeinfo</code> structure type |
|                                                                      | char *                                                                                                                | instance_p<br>Pointer to a specific instance of a multi-dimensional array                          |
| Related routines:                                                    | acc_get_mda_range<br>acc_get_mda_word_range<br>acc_getmda_bitstr<br>acc_setmda_bitstr                                 |                                                                                                    |

Structure `t_tfmdanodeinfo` is defined in the `vcsuser.h` file as follows:

```
typedef struct t_tfmdanodeinfo
{
    short node_type;
    short node_fulltype;
    char *memoryval_p;
    char *node_symbol;
    int   node_ngroups;
```

```

int    node_vec_size;
int    node_sign;
int    node_ms_index;
int    node_ls_index;
int    node_mem_size;
int    *node_lhs_element;
int    *node_rhs_element;
int    node_dimension;
int    *node_handle;
int    node_vec_type;
} s_tfmdanodeinfo, *p_tfmdanodeinfo;

```

---

## **acc\_get\_mda\_range**

The `acc_get_mda_range` routine returns the ranges of a multi-dimensional array.

**Table 0-1.**

| <b>acc_get_mda_range()</b> |                                                                                |                        |                                                    |
|----------------------------|--------------------------------------------------------------------------------|------------------------|----------------------------------------------------|
| Synopsis:                  | Gets all the ranges of the multi-dimensional array.                            |                        |                                                    |
| Syntax:                    | <code>acc_get_mda_range(mdaHandle, size, msb, lsb, dim, plndx, prindex)</code> |                        |                                                    |
|                            | Type                                                                           | Description            |                                                    |
| Returns:                   | <code>void</code>                                                              |                        |                                                    |
| Arguments:                 | Type                                                                           | Name                   | Description                                        |
|                            | <code>handle</code>                                                            | <code>mdaHandle</code> | Handle to the multi-dimensional array              |
|                            | <code>int *</code>                                                             | <code>size</code>      | Pointer to the size of the multi-dimensional array |
|                            | <code>int *</code>                                                             | <code>msb</code>       | Pointer to the most significant bit of a range     |

**Table 0-1.**

|                   |                                                                                                        |       |                                                                    |
|-------------------|--------------------------------------------------------------------------------------------------------|-------|--------------------------------------------------------------------|
|                   | int *                                                                                                  | lsb   | Pointer to the least significant bit of a range                    |
|                   | int *                                                                                                  | dim   | Pointer to the number of dimensions in the multi-dimensional 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<br>acc_get_mda_word_range<br>acc_getmda_bitstr<br>acc_setmda_bitstr |       |                                                                    |

If you have a multi-dimensional array such as the following:

```
reg [7:0] my_mem[255:0][255:0][31:0];
```

And you call a routine, such as the following:

```
handle hN = acc_handle_by_name(my_mem);  
acc_get_mda_range(hN, &size, &msb, &lsb, &dim, &plndx,  
&prndx);
```

It yields the following result:

```
size = 8;  
msb = 7, lsb = 0;  
dim = 4;  
plndx[] = {255, 255, 31}  
prndx[] = {0, 0, 0}
```

---

## **acc\_get\_mda\_word\_range()**

The `acc_get_mda_word_range` routine returns the range of an element in a multi-dimensional array.

**Table 0-1.**

| acc_get_mda_word_range() |                                                                                                                                                                    |                        |                                                 |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|-------------------------------------------------|
| Synopsis:                | Gets the range of an element in a multi-dimensional array.                                                                                                         |                        |                                                 |
| Syntax:                  | <code>acc_get_mda_range (mdaHandle, msb, lsb)</code>                                                                                                               |                        |                                                 |
|                          | Type                                                                                                                                                               | Description            |                                                 |
| Returns:                 | <code>void</code>                                                                                                                                                  |                        |                                                 |
|                          | Type                                                                                                                                                               | Name                   | Description                                     |
| Arguments:               | <code>handle</code>                                                                                                                                                | <code>mdaHandle</code> | Handle to the multi-dimensional array           |
|                          | <code>int *</code>                                                                                                                                                 | <code>msb</code>       | Pointer to the most significant bit of a range  |
|                          | <code>int *</code>                                                                                                                                                 | <code>lsb</code>       | Pointer to the least significant bit of a range |
| Related routines:        | <code>tf_mdanodeinfo</code> and <code>tf_imdanodeinfo</code><br><code>acc_get_mda_range</code><br><code>acc_getmda_bitstr</code><br><code>acc_setmda_bitstr</code> |                        |                                                 |

If you have a multi-dimensional array such as the following:

```
reg [7:0] my_mem[255:0][255:0][31:0];
```

And you call a routine, such as the following:

```
handle hN = acc_handle_by_name(my_mem);  
acc_get_mda_word_range(hN, &left, &right);
```

It yields the following result:

```

left = 7;
right = 0;

```

## **acc\_getmda\_bitstr()**

You use the `acc_getmda_bitstr` access routine to read a bit string, including x and z values, from a multi-dimensional array.

**Table 0-1.**

| acc_getmda_bitstr() |                                                                                                                                                                         |           |                                                         |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|---------------------------------------------------------|
| Synopsis:           | Gets a bit string from a multi-dimensional array.                                                                                                                       |           |                                                         |
| Syntax:             | <code>acc_getmda_bitstr(mdaHandle, bitStr, dim, start, len)</code>                                                                                                      |           |                                                         |
|                     |                                                                                                                                                                         | Type      | Description                                             |
| Returns:            | void                                                                                                                                                                    |           |                                                         |
| Arguments:          | Type                                                                                                                                                                    | Name      | Description                                             |
|                     | handle                                                                                                                                                                  | mdaHandle | Handle to the multi-dimensional array                   |
|                     | char *                                                                                                                                                                  | bitStr    | Pointer to the bit string                               |
|                     | int *                                                                                                                                                                   | dim       | Pointer to the dimension in the multi-dimensional array |
|                     | int *                                                                                                                                                                   | start     | Pointer to the start element in the dimension           |
| Related routines:   | int *                                                                                                                                                                   | len       | Pointer to the length of the string                     |
|                     | <code>tf_mdanodeinfo</code> and <code>tf_imdanodeinfo</code><br><code>acc_get_mda_range</code><br><code>acc_get_mda_word_range</code><br><code>acc_setmda_bitstr</code> |           |                                                         |

If you have a multi-dimensional array such as the following:

```
reg [7:0] my_mem[255:0][255:0][31:0];
```

And you call a routine, such as the following:

```

dim[]={5, 5, 10};
handle hN = acc_handle_by_name(my_mem);
acc_getmda_bitstr(hN, &bitStr, dim, 3, 3);

```

It yields the following string from my\_mem[5][5][10][3:5].

### **acc\_setmda\_bitstr()**

You use the `acc_setmda_bitstr` access routine to write a bit string, including x and z values, into a multi-dimensional array.

**Table 0-1.**

| acc_setmda_bitstr() |                                                                                                                                                                                        |             |                                                         |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|---------------------------------------------------------|
| Synopsis:           | Sets a bit string in a multi-dimensional array.                                                                                                                                        |             |                                                         |
| Syntax:             | <code>acc_setmda_bitstr(mdaHandle, bitStr, dim, start, len)</code>                                                                                                                     |             |                                                         |
|                     | Type                                                                                                                                                                                   | Description |                                                         |
| Returns:            | void                                                                                                                                                                                   |             |                                                         |
| Arguments:          | Type                                                                                                                                                                                   | Name        | Description                                             |
|                     | handle                                                                                                                                                                                 | mdaHandle   | Handle to the multi-dimensional array                   |
|                     | char *                                                                                                                                                                                 | bitStr      | Pointer to the bit string                               |
|                     | int *                                                                                                                                                                                  | dim         | Pointer to the dimension in the multi-dimensional array |
|                     | int *                                                                                                                                                                                  | start       | Pointer to the start element in the dimension           |
|                     | int *                                                                                                                                                                                  | len         | Pointer to the length of the string                     |
| Related routines:   | <a href="#">tf_mdanodeinfo</a> and <a href="#">tf_imdanodeinfo</a><br><a href="#">acc_get_mda_range</a><br><a href="#">acc_get_mda_word_range</a><br><a href="#">acc_getmda_bitstr</a> |             |                                                         |

If you have a multi-dimensional array such as the following:

```
reg [7:0] my_mem[255:0][255:0][31:0];
```

And you call a routine, such as the following:

```
dim[]={5, 5, 10};  
bitstr="111";  
handle hN = acc_handle_by_name(my_mem);  
acc_setmda_bitstr(hN, &bitStr, dim, 3, 3);
```

It writes 111 in my\_mem[5] [5] [10] [3:5].

---

## Access Routines for Probabilistic Distribution

VCS includes the following API routines that duplicate the behavior of the Verilog system functions for probabilistic distribution:

vcs\_random

Returns a random number and takes no argument. See “[vcs\\_random](#)” for details.

vcs\_random\_const\_seed

Returns a random number and takes an integer argument. See “[vcs\\_random\\_const\\_seed](#)” for details.

vcs\_random\_seed

Returns a random number and takes a pointer to integer argument. See “[vcs\\_random\\_seed](#)” for details.

vcs\_dist\_uniform

Returns random numbers uniformly distributed between parameters. See “[vcs\\_dist\\_uniform](#)” for details.

`vcs_dist_normal`

Returns random numbers with a specified mean and standard deviation. See “[“vcs\\_dist\\_normal”](#) for details.

`vcs_dist_exponential`

Returns random numbers where the distribution function is exponential. See “[“vcs\\_dist\\_exponential”](#) for details.

`vcs_dist_poisson`

Returns random numbers with a specified mean. See “[“vcs\\_random”](#) for details.

These routines are declared in the `vcs_acc_user.h` file in the `$VCS_HOME/lib` directory.

---

## **vcs\_random**

You use this routine to obtain a random number.

**Table 0-1.**

| vcs_random()      |                                                                                                                                                                                               |               |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|
| Synopsis:         | Returns a random number.                                                                                                                                                                      |               |
| Syntax:           | <code>vcs_random()</code>                                                                                                                                                                     |               |
|                   | Type                                                                                                                                                                                          | Description   |
| Returns:          | <code>int</code>                                                                                                                                                                              | Random number |
|                   | Type                                                                                                                                                                                          | Name          |
| Arguments:        | <code>None</code>                                                                                                                                                                             |               |
| Related routines: | <code>vcs_random_const_seed</code> <code>vcs_random_seed</code> <code>vcs_dist_uniform</code><br><code>vcs_dist_normal</code> <code>vcs_dist_exponential</code> <code>vcs_dist_poisson</code> |               |

---

## **vcs\_random\_const\_seed**

You use this routine to return a random number and you supply an integer constant argument as the seed for the random number.

**Table 0-1.**

| vcs_random_const_seed |                                                                                                      |               |                      |
|-----------------------|------------------------------------------------------------------------------------------------------|---------------|----------------------|
| Synopsis:             | Returns a random number.                                                                             |               |                      |
| Syntax:               | vcs_random_const_seed(integer)                                                                       |               |                      |
|                       | Type                                                                                                 | Description   |                      |
| Returns:              | int                                                                                                  | Random number |                      |
|                       | Type                                                                                                 | Name          | Description          |
| Arguments:            | int                                                                                                  | integer       | An integer constant. |
| Related routines:     | vcs_random vcs_random_seed vcs_dist_uniform vcs_dist_normal<br>vcs_dist_exponential vcs_dist_poisson |               |                      |

---

## **vcs\_random\_seed**

You use this routine to return a random number and you supply a pointer argument.

**Table 0-1.**

| vcs_random_seed() |                                                                                                            |               |                         |
|-------------------|------------------------------------------------------------------------------------------------------------|---------------|-------------------------|
| Synopsis:         | Returns a random number.                                                                                   |               |                         |
| Syntax:           | vcs_random_seed(seed)                                                                                      |               |                         |
|                   | Type                                                                                                       | Description   |                         |
| Returns:          | int                                                                                                        | Random number |                         |
|                   | Type                                                                                                       | Name          | Description             |
| Arguments:        | int *                                                                                                      | seed          | Pointer to an int type. |
| Related routines: | vcs_random vcs_random_const_seed vcs_dist_uniform vcs_dist_normal<br>vcs_dist_exponential vcs_dist_poisson |               |                         |

---

## vcs\_dist\_uniform

You use this routine to return a random number uniformly distributed between parameters.

**Table 0-1.**

| vcs_dist_uniform  |                                                                                                           |               |                                            |
|-------------------|-----------------------------------------------------------------------------------------------------------|---------------|--------------------------------------------|
| Synopsis:         | Returns random numbers uniformly distributed between parameters.                                          |               |                                            |
| Syntax:           | vcs_dist_uniform(seed, start, end)                                                                        |               |                                            |
|                   | 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: | vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_normal<br>vcs_dist_exponential vcs_dist_poisson |               |                                            |

---

## vcs\_dist\_normal

You use this routine to return a random number with a specified mean and standard deviation.

**Table 0-1.**

| vcs_dist_normal |                                                                      |               |
|-----------------|----------------------------------------------------------------------|---------------|
| Synopsis:       | Returns random numbers with a specified mean and standard deviation. |               |
| Syntax:         | vcs_dist_normal(seed, mean, standard_deviation)                      |               |
|                 | Type                                                                 | Description   |
| Returns:        | int                                                                  | Random number |

**Table 0-1.**

|                   | 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: | vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform<br>vcs_dist_exponential vcs_dist_poisson |                    |                                                                                      |

## **vcs\_dist\_exponential**

You use this routine to return a random number where the distribution function is exponential.

**Table 0-1.**

| vcs_dist_exponential |                                                                                                       |               |                                                                               |
|----------------------|-------------------------------------------------------------------------------------------------------|---------------|-------------------------------------------------------------------------------|
| Synopsis:            | Returns random numbers where the distribution function is exponential.                                |               |                                                                               |
| Syntax:              | vcs_dist_exponential(seed, mean)                                                                      |               |                                                                               |
| Returns:             | Type                                                                                                  | Description   |                                                                               |
| Arguments:           | int                                                                                                   | Random number |                                                                               |
|                      | Type                                                                                                  | Name          | Description                                                                   |
|                      | int *                                                                                                 | seed          | Pointer to a seed integer value.                                              |
| Related routines:    | int                                                                                                   | mean          | An integer that is the average value of the possible returned random numbers. |
|                      | vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform<br>vcs_dist_normal vcs_dist_poisson |               |                                                                               |
|                      |                                                                                                       |               |                                                                               |

---

## vcs\_dist\_poisson

You use this routine to return a random number with a specified mean.

**Table 0-1.**

| vcs_dist_poisson  |                                                                                                           |               |                                                                               |
|-------------------|-----------------------------------------------------------------------------------------------------------|---------------|-------------------------------------------------------------------------------|
| Synopsis:         | Returns random numbers with a specified mean.                                                             |               |                                                                               |
| Syntax:           | vcs_dist_poisson(seed, mean)                                                                              |               |                                                                               |
| >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: | vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform<br>vcs_dist_normal vcs_dist_exponential |               |                                                                               |

---

## Access Routines for Returning a Pointer to a Parameter Value

The 1364 Verilog standard states that for access routine `acc_fetch_paramval`, you can cast the return value to a character pointer using the C language cast operators `(char*) (int)`. For example:

```
str_ptr=(char*)(int)acc_fetch_paramval(...);
```

In 64-bit simulation, you should use `long` instead of `int`:

```
str_ptr=(char*)(long)acc_fetch_paramval(...);
```

For your convenience, VCS provides the `acc_fetch_paramval_str` routine to directly return a string pointer.

---

## **acc\_fetch\_paramval\_str**

Returns the value of a string parameter directly as `char*`.

**Table 0-1.**

| 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”](#)” for details.

`acc_lsi_dumpports_call`

Monitors instance ports. See “[acc\\_lsi\\_dumports\\_call](#)” for details.

`acc_lsi_dumports_close`

Closes specified VCDE files. See “[acc\\_lsi\\_dumports\\_close](#)” for details.

`acc_lsi_dumports_flush`

Flushes cached data to the VCDE file on disk. See “[acc\\_lsi\\_dumports\\_flush](#)” for details.

`acc_lsi_dumports_limit`

Sets the maximum VCDE file size. See “[acc\\_lsi\\_dumports\\_limit](#)” for details.

`acc_lsi_dumports_misc`

Processes miscellaneous events. See “[acc\\_lsi\\_dumports\\_misc](#)” for details.

`acc_lsi_dumports_off`

Suspends VCDE file dumping. See “[acc\\_lsi\\_dumports\\_off](#)” for details.

`acc_lsi_dumports_on`

Resumes VCDE file dumping. See “[acc\\_lsi\\_dumports\\_on](#)” for details.

`acc_lsi_dumports_setformat`

Specifies the format of the VCDE file. See “[acc\\_lsi\\_dumports\\_setformat](#)” for details.

`acc_lsi_dumports_vhdl_enable`

Enables or disables the inclusion of VHDL drivers in the determination of driver values. See “[acc\\_lsi\\_dumpports\\_vhdl\\_enable](#)” 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 F-11 Example of acc\_lsi\_dumpports\_all*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
/* rut-roh, error ... */
}
acc_lsi_dumpports_limit(100000, outfile);
...
```

```

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
    ...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
    ...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
    ...

```

## **Caution**

This routine may affect files opened by the \$dumpports and \$lsi\_dumpports system tasks.

## **acc\_lsi\_dumpports\_call**

### **Syntax**

```
int acc_lsi_dumpports_call(handle instance, char *filename)
```

### **Synopsis**

Monitors instance ports.

This is a PLI interface to the \$lsi\_dumpports task. The default file format is the original LSI format, but you can select the IEEE format by calling the routine acc\_lsi\_dumpports\_setformat() prior to calling this routine. Your tab file will need the following acc permissions:

```
acc=cbka,cbk,cbkv:[<instance_name>|*].
```

## Returns

Zero on success, non-zero otherwise. VCS displays error messages through `tf_error()`. A common error is specifying a file name also being used by a `$dumpports` or `$lsi_dumpports` system task.

### *Example F-12 Example of acc\_lsi\_dumpports\_all*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);

if (acc_lsi_dumpports_call(instance, outfile)) {
    /* error */
}
```

## Caution

Multiple calls to this routine are allowed, but the output file name must be unique for each call.

For proper dumpports operation, your task's miscellaneous function must call `acc_lsi_dumpports_misc()` with every call it gets. This ensures that the dumpports routines sees all of the simulation events needed for proper update and closure of the 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 F-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_DUMPPORTS_FORMAT_LSI);

acc_lsi_dumpports_call(instance, outfile1);
acc_lsi_dumpports_call(instance, outfile2);
...
acc_lsi_dumpports_close(NULL, outfile1);
...
acc_lsi_dumpports_close(NULL, outfile2);
```

## **Caution**

A call to this function can also close files opened by the \$lsi\_dumpports or \$dumpports system tasks.

---

## **acc\_lsi\_dumpports\_flush**

### **Syntax**

```
int acc_lsi_dumpports_flush(char *filename)
```

### **Synopsis**

Flushes cached data to the VCDE file on disk.

This is a PLI interface to the \$dumpportsflush system task. If the filename is NULL all open files are flushed.

### **Returns**

The number of files matched.

#### *Example F-14 Example of acc\_lsi\_dumpports\_flush*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...
```

```

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...
if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...
if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
...

```

---

## **acc\_lsi\_dumpports\_limit**

### **Syntax**

```
int acc_lsi_dumpports_limit(unsigned long filesize, char *filename)
```

### **Synopsis**

Sets the maximum VCDE file size.

This is a PLI interface to the \$dumpportslimit task. If the filename is NULL, the file size is applied to all files.

### **Returns**

The number of files matched.

#### *Example F-15 Example of acc\_lsi\_dumpports\_limit*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";
```

```

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
...

```

## **Caution**

This routine may affect files opened by the \$dumpports and \$lsi\_dumpports system tasks.

## **acc\_lsi\_dumpports\_misc**

### **Syntax**

```
void acc_lsi_dumpports_misc(int data, int reason)
```

### **Synopsis**

Processes miscellaneous events.

This is a companion routine for `acc_lsi_dumpports_call()`.

For proper dumports operation, your task's miscellaneous function must call this routine for each call it gets.

### Returns

No return value.

#### *Example F-16 Example of acc\_lsi\_dumports\_misc*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

void my_task_misc(int data, int reason)
{
    acc_lsi_dumports_misc(data, reason);
    ...
}
```

---

## **acc\_lsi\_dumports\_off**

### Syntax

```
int acc_lsi_dumports_off(char *filename)
```

### Synopsis

Suspends VCDE file dumping.

This is a PLI interface to the \$dumports off system task. If the file name is NULL, dumping is suspended on all open files.

### Returns

The number of files that matched.

#### *Example F-17 of acc\_lsi\_dumports\_off Example*

```
#include "acc_user.h"
#include "vcs_acc_user.h"
```

```

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
...

```

## **Caution**

This routine may suspend dumping on files opened by the \$dumpports and \$lsi\_dumpports system tasks.

## **acc\_lsi\_dumpports\_on**

### **Syntax**

```
int acc_lsi_dumpports_on(char *filename)
```

## Synopsis

Resumes VCDE file dumping.

This is a PLI interface to the \$dumpportson system task. If the filename is NULL, dumping is resumed on all open files.

## Returns

The number of files that matched.

### *Example F-18 Example of acc\_lsi\_dumpports\_on*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
```

...

## **Caution**

This routine may resume dumping on files opened by the \$dumpports and \$lsi\_dumpports system tasks.

---

## **acc\_lsi\_dumpports\_setformat**

### **Syntax**

```
int acc_lsi_dumpports_setformat(lsi_dumpports_format_type  
format)
```

Where the valid `lsi_dumpports_format_types` are as follows:

`USE_DUMPPORTS_FORMAT_IEEE`

`USE_DUMPPORTS_FORMAT_LSI`

### **Synopsis**

Specifies the format of the VCDE file.

Use this routine to specify which output format (IEEE or the original LSI) should be used. This routine must be called before `acc_lsi_dumpports_call()`.

### **Returns**

Zero if success, non-zero if error. Errors are reported through `tf_error()`.

### *Example F-19 Example of acc\_lsi\_dumpports\_setformat*

```
#include "acc_user.h"  
#include "vcs_acc_user.h"
```

```

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile1 = "device.evcd1";
char *outfile2 = "device.evcd2";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile1)) {
    /* error */
}

/* use LSI format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_LSI);
if (acc_lsi_dumpports_call(instance, outfile2)) {
    /* error */
}
...

```

## **Caution**

The runtime plusargs +dumpports+ieee and +dumpports+lsi have priority over this routine.

The format of files created by calls to the \$dumpports and \$lsi\_dumpports tasks are not affected by this routine.

## **acc\_lsi\_dumpports\_vhdl\_enable**

### **Syntax**

```
void acc_lsi_dumpports_vhdl_enable(int enable)
```

The valid enable integer parameters are as follows:

1 enables VHDL drivers

0 disables VHDL drivers

## Synopsis

Use this routine to enable or disable the inclusion of VHDL drivers in the determination of driver values.

## Returns

No return value.

### *Example F-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 includes a number of access routines to monitor code execution. These access routines are as follows:

`acc_mod_lcb_add`

Registers a line callback routine with a module so that VCS calls the routine whenever VCS executes the specified module. See “[“acc\\_mod\\_lcb\\_add”](#) 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”](#) for details.

`acc_mod_lcb_enabled`

Tests to see if line callbacks is enabled. See “[“acc\\_mod\\_lcb\\_enabled”](#) for details.

`acc_mod_lcb_fetch`

Returns an array of breakable lines. See “[“acc\\_mod\\_lcb\\_fetch”](#) for details.

`acc_mod_lcb_fetch2`

Returns an array of breakable lines. See “[“acc\\_mod\\_lcb\\_fetch2”](#) for details.

`acc_mod_sfi_fetch`

Returns the source file composition for a module. See “[“acc\\_mod\\_sfi\\_fetch”](#) 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 F-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 F-22 Example of acc\_mod\_lcb\_del*

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"
```

```

/* VCS 4.x callback rtn */
void line_call_back(filename, lineno, userdata, tag)
char *filename;
int lineno;
char *userdata;
int tag;
{
    handle handle_mod = (handle)userdata;

    io_printf("Tag %2d, file %s, line %2d, module %s\n",
              tag, filename, lineno,
              acc_fetch_fullname(handle_mod));
}

/* unregister all line callbacks (recursive) */
void unregister_lcb (parent_mod)
handle parent_mod;
{
    handle child = NULL;

    if (! acc_object_of_type(parent_mod, accModule)) return;

    io_printf("Unregistering %s\n",
              acc_fetch_fullname (parent_mod));

    acc_mod_lcb_del (parent_mod, line_call_back, parent_mod);

    while ((child = acc_next_child (parent_mod, child))) {
        register_lcb (child);
    }
}

```

## **Caution**

The module handle, consumer routine, and user data arguments must match those supplied to the `acc_mod_lcb_add()` routine for a successful delete.

For example, using the result of a call such as `acc_fetch_name()` as the user data will fail, because that routine returns a different pointer each time it is called.

---

## **acc\_mod\_lcb\_enabled**

### **Syntax**

```
int acc_mod_lcb_enabled()
```

### **Synopsis**

Test to see if line callbacks is enabled.

By default, the extra code required to support line callbacks is not added to a simulation executable. You can use this routine to determine if line callbacks have been enabled.

### **Returns**

Non-zero if line callbacks are enabled; 0 if not enabled.

#### *Example F-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 F-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 F-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 F-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](#)” 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"](#)" for details.

#### `vcsSpEncodeOn`

This routine inserts a trailer section containing the '`' protected`' compiler directive into the output file. It also toggles the encryption flag to false so that subsequent calls to `vcsSpWriteString()` and `vcsSpWriteChar()` will have their data encrypted. See "["vcsSpEncodeOn"](#)" for details.

#### `vcsSpEncoding`

This routine gets the current state of encoding. See "["vcsSpEncoding"](#)" for details.

#### `vcsSpGetFilePtr`

This routine just returns the value previously passed to the `vcsSpSetFilePtr()` routine. See "["vcsSpGetFilePtr"](#)" for details.

#### `vcsSpInitialize`

Allocates a source protect object. See "["vcsSpInitialize"](#)" for details.

#### `vcsSpOvaDecodeLine`

Decrypts one line. See "["vcsSpOvaDecodeLine"](#)" for details.

#### `vcsSpOvaDisable`

Switches to regular encryption. See “[vcsSpOvaDisable](#)” for details.

#### vcsSpOvaEnable

Enables the OpenVera assertions (OVA) encryption algorithm. Tells VCS’s encrypter to use the OVA IP algorithm. See “[vcsSpOvaEnable](#)” for details.

#### vcsSpSetDisplayMsgFlag

Sets the DisplayMsg flag. See “[vcsSpSetDisplayMsgFlag](#)” for details.

#### vcsSpSetFilePtr

Specifies the output file stream. See “[vcsSpSetFilePtr](#)” for details.

#### vcsSpSetLibLicenseCode

Sets the OEM license code. See “[vcsSpSetLibLicenseCode](#)” for details.

#### vcsSpSetPliProtectionFlag

Sets the PLI protection flag. See “[vcsSpSetPliProtectionFlag](#)” for details.

#### vcsSpWriteChar

Writes one character to the protected file. See “[vcsSpWriteChar](#)” for details.

#### vcsSpWriteString

Writes a character string to the protected file. See “[vcsSpWriteString](#)” for details.

**Example F-27** outlines the basic use of the source protection routines.

*Example F-27 Using the Source Protection Routines*

```
#include <stdio.h>
#include "enclib.h"
void demo_routine()
{
    char *filename = "protected.vp";
    int write_error = 0;
    vcsspStateID esp;
    FILE *fp;

    /* Initialization */

    if ((fp = fopen(filename, "w")) == NULL) {
        printf("Error: opening file %s\n", filename);
        exit(1);
    }

    if ((esp = vcsspInitialize()) == NULL) {
        printf("Error: Initializing src protection
routines.\n");
        printf("          Out Of Memory.\n");
        fclose(fp);
        exit(1);
    }

    vcsspSetFilePtr(esp, fp); /* tell rtns where to write */

    /* Write output */

    write_error += vcsspWriteString(esp,
                                   "This text will *not* be encrypted.\n");

    write_error += vcsspEncodeOn(esp);
    write_error += vcsspWriteString(esp,
                                   "This text *will* be encrypted.");
    write_error += vcsspWriteChar(esp, '\n');

    write_error += vcsspEncodeOff(esp);
```

```

write_error += vcsSpWriteString(esp,
                               "This text will *not* be encrypted.\n");

/* Clean up */

write_error += fclose(fp);
vcsSpClose(esp);

if (write_error) {
    printf("Error while writing to '%s'\n", filename);
}
}

```

## **Caution**

If you are encrypting SDF or Verilog code that contains include directives, you must switch off encryption (`vcsSpEncodeOff`), output the include directive and then switch encryption back on. This ensures that when the parser begins reading the included file, it is in a known (non-decode) state.

If the file being included has proprietary data it can be encrypted separately. (Don't forget to change the '`include`' compiler directive to point to the new encrypted name.)

## **vcsSpClose**

### **Syntax**

```
void vcsSpClose(vcsSpStateID esp)
```

### **Synopsis**

This routine frees the memory allocated by `vcsSpInitialize()`. Call it when source encryption is finished on the specified stream.

## Returns

No return value.

### *Example F-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 F-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.

- 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 F-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 F-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 F-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 F-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 F-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 F-35 Example of vcsSpOvaDisable*

```
#include "enclib.h"
#include "encint.h"

int write_error = 0;
vcsSpStateID esp;

if ((esp = vcsSpInitialize()) == NULL)
    printf("Out Of Memory");

vcsSpSetFilePtr(esp, fp); /* previously opened FILE* pointer */

/* Configure for OVA IP encryption */
vcsSpOvaEnable(esp, "synopsys");
```

```

if (vcsspWriteString(esp, "This text will NOT be
encrypted.\n"))
    ++write_error;

if (vcsspEncodeOn(esp)) ++write_error;

if (vcsspWriteString(esp, "This text WILL be encrypted.\n"))
    ++write_error;

if (vcsspEncodeOff(esp)) ++write_error;

if (vcsspWriteString(esp, "This text will NOT be
encrypted.\n"))
    ++write_error;
/* Switch back to regular encryption */
vcsspOvaDisable(esp);

if (vcsspEncodeOn(esp)) ++write_error;

if (vcsspWriteString(esp, "This text WILL be encrypted.\n"))
    ++write_error;

if (vcsspEncodeOff(esp)) ++write_error;

vcsspClose(esp);

```

## **vcsspOvaEnable**

### **Syntax**

```
void vcsspOvaEnable(vcsspStateID esp, char *vendor_id)
```

### **Synopsis**

Enables the OpenVera assertions (OVA) encryption algorithm. Tells VCS's encrypter to use the OVA IP algorithm.

### **Returns**

No return value.

### *Example F-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 F-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 F-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 F-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 F-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 F-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](#)” ) the specified character is encrypted as it is written to the output file.

If encoding is disabled (see “[vcsSpEncodeOff](#)” ) 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](#)” ) or if there was an error writing to the output file (out-of-disk-space, etc.)

Returns 0 if the write was successful.

#### *Example F-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](#)” ) the specified string is encrypted as it is written to the output file.

If encoding is disabled (see “[vcsSpEncodeOff](#)” ) 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](#)” ) or if there was an error writing to the output file (out-of-disk-space, etc.)

Returns 0 if the write was successful.

### *Example F-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

[C-120](#), [D-8](#)

[-a filename](#) [C-115](#)

[-ams\\_discipline](#) [C-78](#)

[-ams\\_iereport](#) [C-78](#)

[-assert](#) [14-44](#), [C-11](#)

svaext [14-44](#)

[-bfl](#) and [-bom](#) [C-123](#)

[-bom](#) and [-bfl](#) [C-123](#)

[-C](#) [C-77](#)

[-c](#) [C-73](#)

[-CC](#) [C-75](#)

[-cc](#) [C-75](#)

[-CFLAGS](#) [C-75](#)

[-cm assert](#) [C-21](#)

[-cpp](#) [C-76](#)

[-debug](#) [22-33](#)

[-e name\\_for\\_main](#) [C-57](#)

[-E program runtime option](#) [D-40](#)

[-error](#) [4-111](#), [4-112](#), [4-114](#), [4-117](#), [C-11](#), [C-62](#),

[D-26](#)

[-error=PRIORITY](#) [D-23](#)

[-error=UNIQUE](#) [D-23](#)

[-extinclude](#) [B-27](#), [C-31](#), [E-9](#)

[-f filename](#) [C-54](#)

[-gui](#) [2-26](#)

[-h](#) [2-17](#), [2-23](#), [C-10](#)

[-help](#) [2-17](#), [2-23](#), [C-10](#)

[-ID](#) [2-17](#), [2-23](#), [C-72](#)

[-jnumber\\_of CPUs](#) [C-76](#)

[-I](#) [D-27](#)

[-I filename](#) [2-19](#), [2-27](#), [C-115](#), [D-26](#)

[-ld linker](#) [C-73](#)

[-LDFLAGS](#) [C-73](#)

[-lmc-swift](#) [C-61](#)

[-lmc-swift-template](#) [C-61](#)

[-lname](#) [C-73](#)

[-load](#) [4-22](#), [22-36](#), [C-59](#), [D-44](#)

[-location](#)

vlogan option [B-14](#)

[-Makep](#) [C-9](#)

[-Mdir](#) [C-8](#)

[-Mdirectory](#) [C-8](#)

[-Mlib=dir](#) [C-9](#)

[-msg\\_config](#) [4-112](#), [4-129](#)

[-Mupdate](#) [C-9](#)

[-nc](#) [C-62](#)

[-negdelay](#) [C-52](#)

[-nolncComp](#) [C-10](#)

[-ntb](#) [2-9](#), [B-14](#), [C-22](#)

[-ntb\\_define](#) [C-23](#)

[-ntb\\_filext](#) [C-23](#)

[-ntb\\_incdir](#) [C-23](#)

[-ntb\\_opts](#) [B-14](#), [C-24](#)

[-ntb\\_sfname](#) [C-30](#)

-ntb\_vipext C-30  
 -o name C-117  
 -O number C-77  
 -O0 C-77  
 -ova\_enable\_case C-22  
 -ova\_file 2-10  
 -override\_timescale B-18, C-82  
 -P pli.tab C-58  
 -parameters 2-26, 4-31, C-80  
 -platform C-118  
 -pvalue 2-26, 4-31, C-80  
 -q 2-18, 2-27, C-65, D-26  
 -R 2-18, 2-26, C-40, C-115  
 -sig program runtime option D-40  
 -simprofile C-53  
 -simprofile\_dir\_path 6-7  
 -suppress 4-111, C-62  
 -sv\_pragma 2-10, B-17  
 -sysc C-112  
 -timescale B-18  
 -u C-115  
 -uniq\_prior maxfail=integer D-23  
 -V 2-19, 2-27, C-65, D-27  
 -v 2-23, C-5  
 -Vt C-66  
 -work  
     vlogan option B-23  
 -Xman 25-38  
 -Xmangle 25-38  
 -Xnoman 25-39  
 -Xnomangle 25-39  
 -Xova C-22  
 -y 2-24, C-6  
 assert hier=file.txt C-20  
 'celldefine C-70, C-71, E-2, E-3  
 'default\_nettpe E-3  
 'define E-3  
 'delay\_mode\_distributed E-6  
 'delay\_mode\_path E-6  
 'delay\_mode\_unit E-7  
 'delay\_mode\_zero E-7  
 'else E-3  
 'elseif E-4  
 'endcelldefine E-2  
 'endif E-4  
 'endprotect 25-32  
 'endprotected 25-33  
 'endprotected128 25-33  
 'endrace C-81  
 'ifdef E-4  
 'ifndef VCS E-5  
 'ifndef E-6  
 'include B-27, C-31, E-8  
     specifying the search directories C-111  
     with a different verion of Verilog B-27, C-31  
 'line E-10  
 'nounconnected\_drive E-11  
 'protect 25-32, 25-38  
 'protect128 25-33  
 'protected 25-32  
 'race C-81  
 'resetall E-3  
 'timescale C-83, E-9  
     overriding with -override\_timescale B-18  
 'unconnected\_drive E-11  
 'undef E-6  
 'uselib E-9  
 'vcs\_mipdexpand E-7  
 "A" specifier of abstract access 22-53  
 "C" specifier of direct access 22-53  
 \*\*NC 3-15  
 /\*synopsys translate\_off\*/ pragma C-120  
 /\*synopsys translate\_on\*/ pragma C-121  
 //synopsys translate\_off pragma C-120  
 //synopsys translate\_on pragma C-121  
 %CELL 22-15, 22-21  
 %for 20-9, 20-14  
 %if 20-9, 20-14  
 %TASK 22-15  
 + 5-24  
 +gc+high\_threshold D-10  
 +abstract 22-128

+acc+3 C-57  
 +acc+4 C-57  
 +acc+level\_number C-57  
 +allhdrs 22-128  
 +allmtm C-41, D-31, D-32  
 +applylearn 22-29-??, D-40  
 +applylearn+filename C-57  
 +auto2protect 25-37  
 +auto3protect 25-37  
 +autoprotect 25-37  
 +charge\_decay C-41  
 +define+macro=value 2-27, C-116  
 +delay\_mode\_distributed 11-41, B-22, C-42  
 +delay\_mode\_path 11-41, B-21, C-41  
 +delay\_mode\_unit 11-41, B-22, C-41  
 +delay\_mode\_zero 11-41, B-22, C-41  
 +deleteprotected 25-37  
 +error+count C-67  
 +inmdir 2-25, C-111  
 +iopath+edge C-48  
 +libext 2-12, 2-14, 2-25, B-23, C-7  
 +liborder 2-13, 2-25, C-7  
 +librescan 2-13, C-7  
 +libverbose C-8, C-63  
 +lint 2-12, 4-111, B-23, C-63  
 +lint=PWLNT  
     C-22  
 +lint=sva 19-47  
 +list 22-128  
 +maxdelays C-41, C-44, D-31, D-41  
 +memcbk C-114  
 +mindelays C-41, C-45, D-31, D-41  
 +module module\_identifier D-22  
 +multisource\_int\_delays C-46  
 +nbaopt C-46  
 +neg\_tchk 11-66, 11-73, C-53  
 +no\_notifier 11-66, C-50, D-24  
 +no\_pulse\_msg D-24, D-27  
 +no\_tchk\_msg 11-66, C-50, D-24  
 +nocelldefinepli+0 C-70  
 +nocelldefinepli+1 C-70  
 +nocelldefinepli+2 C-71  
 +noerrorIOPCWM C-113  
 +nolibcell C-70  
 +noportcoerce C-112  
 +nospecify 11-67, C-50  
 +notimingcheck 11-67, C-50, D-24  
 +ntb\_cache\_dir D-3  
 +ntb\_delete\_disk\_cache D-3  
 +ntb\_disable\_cnst\_null\_object\_warning D-3  
 +ntb\_enable\_checker\_trace D-4  
 +ntb\_enable\_checker\_trace\_on\_failure D-4  
 +ntb\_enable\_solver\_trace\_on\_failure D-5  
 +ntb\_enable\_solver\_trace\_on\_failure=value D-5  
 +ntb\_exit\_on\_error D-5  
 +ntb\_random\_seed D-6  
 +ntb\_random\_seed\_automatic D-6  
 +ntb\_solver\_array\_size\_warn D-7  
 +ntb\_solver\_debug 16-29, D-7  
     extract 16-35, 16-38  
     profile 16-34, 16-38  
     serial 16-37  
     trace 16-30, 16-33, 16-37  
 +ntb\_solver\_debug\_dir D-8  
 +ntb\_solver\_debug\_filter 16-30, 16-33, 16-35, D-8  
 +ntb\_solver\_mode D-9, D-10  
 +ntb\_solver\_mode=value D-9  
 +NTC2 11-72, C-53  
 +object\_protect 25-50  
 +old\_ntc C-53  
 +optconfigfile 9-8, C-40  
 +override\_model\_delays D-31, D-32, D-41  
 +pathpulse C-49  
 +pli\_unprotected 25-38  
 +plusarg\_ignore C-57  
 +plusarg\_save C-56  
 +plus-options D-41  
 +protect\_file\_suffix 25-38  
 +pulse\_e/number 11-27, 11-29, 11-31, 11-36, 11-37, C-51

```

+pulse_int_e 11-26, 11-27, 11-29, 11-31, C-51
+pulse_int_r 11-26, 11-27, 11-29, 11-31, C-51
+pulse_on_detect 11-37, C-52
+pulse_on_event 11-36, C-51
+pulse_r/number 11-27, 11-29, 11-31, 11-36,
11-37, C-51
+putprotect+target_dir 25-38
+race=all 3-21, C-81
+rad 9-8, C-40
+sdf_nocheck_celltype C-47
+sdfprotect file_suffix 25-38
+sdfverbose C-62
+systemverilogext 2-13, B-26, C-30
+tetramax C-112
+timopt 11-44
+transport_int_delays 11-26, 11-29, 11-31,
C-47
+transport_path_delays 11-26, 11-29, 11-31,
C-47
+typdelays C-41, C-45, D-32, D-41
+UVM_LOG_RECORD 34-6
+UVM_TR_RECORD 34-6
+vc 22-127, C-59
+vcs+dumpfile+filename D-30
+vcs+dumpoff+t+ht D-30
+vcs+dumpon+t+ht D-30
+vcs+finish 5-52, D-25
+vcs+flush+all C-61, D-33
+vcs+flush+dump C-61, D-30, D-33
+vcs+flush+fopen C-61, D-33
+vcs+flush+log C-60, D-33
+vcs+ignorestop D-40
+vcs+initreg
    restricting initialization to either registers or
    memories" 4-30
+vcs+initreg+0|1|random| D-37
+vcs+initreg+0|1|random|seed_value D-37
+vcs+initreg+config 4-26, C-35, D-38
+vcs+initreg+config+config_file D-39
+vcs+initreg+random| D-39
+vcs+learn+pli ??-22-33, D-40
+vcs+lic+vcsi D-34
+vcs+lic+wait D-34
+vcs+loopdetect+number C-119, D-42
+vcs+loopreport+number C-118, D-41
+vcs+mipd+noalias D-43
+vcs+mipdexpand E-7
+vcs+nostdout D-27
+vcs+stop 5-52, D-25
+vcs+vcdpluson C-113
+vcsi+lic+vcs D-34
+verilog1995ext 2-13, B-26, C-31
+verilog2001ext 2-13, B-26, C-31
+vpddrivers D-29
+vpdnoports D-29
+vpdportsonly D-29
+vpdupdate D-30
+vpi C-58
+vpi+1 C-58
+vpi+1+assertion C-59
+warn 3-55, 4-111, B-27, C-66
$assert_category_start 19-25, 19-29
$assert_category_stop 19-24
$assert_monitor 19-17, E-27
$assert_monitor_off 19-17, E-28
$assert_monitor_on 19-17, E-28
$assert_set_category 19-22, 19-25
$assert_set_severity 19-22
$assert_severity_stop 19-24
$assertcontrol 19-63
$assertkill E-12
$assertoff E-12
$asserton E-12
$async$and$array E-40
$bitstoreal E-30
$countdrivers E-44
$countunknown(expression) E-13
$countx(expression) E-13
$countz(expression) E-13
$deposit E-45
$disable_warnings E-37
$display E-31

```

**\$dist\_exponential** E-42  
**\$dist\_normal** E-42  
**\$dist\_poisson** E-42  
**\$dist\_uniform** E-43  
**\$dumpall** E-14  
**\$dumpfile** E-14  
**\$dumpflush** E-14  
**\$dumplimit** E-15  
**\$dumpoff** E-14  
**\$dumpon** E-14  
**\$dumports** E-17  
**\$dumports system task** D-33  
**\$dumportsall** E-18  
**\$dumportsflush** E-19  
**\$dumportslimit** E-19  
**\$dumpportsoff** E-18  
**\$dumpportson** E-18  
**\$dumpvars** E-15  
**\$enable\_warnings** E-37  
**\$error** 19-58, E-11  
**\$fatal** 19-37, E-11  
**\$fclose** E-31  
**\$fdisplay** E-32  
**\$ferror** E-32  
**\$fflush** E-15, E-32  
**\$fflushall** E-16  
**\$fgetc** E-32  
**\$fgets** E-32  
**\$finish** E-36  
**\$fmonitor** E-32  
**\$fopen** C-60, E-32  
    increasing the frequency of flushing C-61  
**\$fopen system function** D-33  
    increasing the frequency of \$fopen file, log  
        file, and VCD file dumping D-33  
    increasing the frequency of dumping to files  
        opened by \$fopen D-33  
**\$fread** E-33  
**\$fscanf** E-33  
**\$fseek** E-33  
**\$fstobe** E-33  
**\$ftell** E-33  
**\$fwrite** E-33  
**\$get\_initial\_random\_seed** E-43  
**\$getpattern** E-45  
**\$gr\_waves** E-16  
**\$hold** E-38  
**\$info** E-12  
**\$isunknown** E-13  
**\$itor** E-30  
**\$log** E-29  
**\$lsi\_dumports** 3-55–3-60, E-16  
**\$lsi\_dumports system task** D-33  
**\$monitor** E-31  
**\$monitoroff** E-31  
**\$monitoron** E-31  
**\$nolog** E-29  
**\$onedriven** E-13  
**\$onedriven0** E-14  
**\$onehot0** E-13  
**\$past**  
    ignoring C-20  
**\$period** E-38  
**\$printtimescale** E-36  
**\$q\_add** E-41  
**\$q\_exam** E-41  
**\$q\_full** E-41  
**\$q\_initialize** E-41  
**\$q\_remove** E-41  
**\$random** 3-54, E-43  
**\$read\_lib\_saif** 24-6  
**\$readmemb** 5-65, E-35  
**\$readmemh** 5-65, E-35  
**\$realtime** E-42  
**\$realtobits** E-30  
**\$recovery** E-38  
**\$recrem** E-39  
    checking timestamp and timecheck  
        conditions C-53  
    disabling delayed versions of signals in other  
        timing checks C-53  
**\$removal** E-39

**\$reset** E-43  
**\$reset\_count** E-43  
**\$reset\_value** E-44  
**\$restart** E-45  
**\$rtoi** E-30  
**\$save** E-45  
**\$sdf\_annotation** E-44  
**\$set\_gate\_level\_monitoring** 24-4  
**\$set\_toggle\_region** 24-6  
**\$setup** E-39  
**\$setuphold** E-40  
 checking timestamp and timecheck  
 conditions C-53  
 disabling delayed versions of signals in other  
 timing checks C-53  
**\$skew** E-40  
**\$sreadmemb** E-35  
**\$sreadmemh** E-35  
**\$stime** E-42  
**\$stop** E-36  
 ignoring D-40  
**\$strobe** E-31  
**\$sync\$nor\$plane** E-40  
**\$system** E-28  
**\$systemf** E-29  
**\$test\$plusargs** D-41, E-44  
**\$time** E-42  
**\$timeformat** E-36  
**\$ungetc** E-34  
**\$uniq\_prior\_checkoff** system task 14-123,  
 14-124  
**\$uniq\_prior\_checkon** system task 14-123  
**\$value\$plusargs** 5-46  
**\$vcdplusautoflushoff** E-20  
**\$vcdplusautoflushon** E-21  
**\$vcdplusclose** E-21  
**\$vcdplusdeltacycleoff** 8-11  
**\$vcdplusdeltacycleon** 8-11, E-21  
**\$vcdplusevent** E-22  
**\$vcdplusfile** E-22  
**\$vcdplusflush** E-23  
**\$vcdplusglitchon** E-23  
**\$vcdplusmemoff** 8-8, E-25  
**\$vcdplusmemon** 8-8, E-23  
**\$vcdplusmemorydump** 8-8, E-25  
**\$vcdplusoff** E-25  
**\$vcdpluson** C-113, E-26  
**\$vcdplusxx** system tasks  
 ignoring D-29  
**\$warning** E-11  
**\$width** E-40  
**\$write** E-31  
**\$writememb** 5-65, E-35  
**\$writememh** 5-65, E-35

## A

-a filename C-115  
 "A" specifier of abstract access 22-53  
**+abstract** 22-128  
 abstract access for C/C++ functions  
 access routines for 22-78–22-122  
 enabling with a compile-time option 22-128  
 using 22-76–22-122  
**+acc+level\_number** C-57  
**ACC** capabilities 22-32, C-57  
 applying in the design only where they are  
 needed D-40  
**cbk** 22-12, 22-22  
**cbka** 22-12  
 enabling debugging C-57  
**frc** 22-12, 22-22  
**gate** 22-13  
**mip** 22-13, D-43  
**mipp** 22-13  
**mipd** D-43  
**mp** 22-13  
**prx** 22-12  
**r** 22-11, 22-21  
 recording where in the design they are  
 needed D-40  
**rw** 22-12, 22-22  
**s** 22-12  
 specifying 22-10–22-23

tchk 22-12  
 acc\_handle\_simulated\_net D-43  
 access routines for abstract access of C/C++ functions 22-78–22-122  
 Active time slot  
     changing UDP output evaluation to the NBA time slot 3-5  
 -ad C-78  
 Addressing Models 10-31  
     Signed indexing 10-31  
     Unsigned Indexing 10-31  
 -adopt wildcard 30-18  
 AICMs  
     information messages C-78  
 all keyword 23-13  
 +allhdrs 22-128  
 +allmtm C-41, D-31, D-32  
 alt\_retain 11-7  
 always@\* 14-97  
 -ams\_discipline C-78  
 -ams\_iereport C-78  
 analysis  
     setup variables A-4  
 ansi argument to -ntb\_opts C-24  
 ANSI mode  
     in OpenVera files C-24  
 aop  
     advice  
         before/after/around 15-16  
     dominates 15-7  
     extends directive 15-3  
     placement element  
         after 15-11  
         around 15-11  
     D-40  
     +applylearn 22-29–??  
 arb.v 13-11, 13-12  
 args PLI Specification 22-8  
 array  
     output and inout argument type 22-69  
 array index 16-50  
 array members 16-70

assembler  
     passing options to C-75  
 -assert 14-44, 19-58, C-11, D-11  
     svaext 14-44  
 assert 27-5  
 -assert assertion\_block\_identifier D-22  
 -assert failonly 19-50  
 -assert funchier 19-32  
 -assert hier=file.txt C-20  
 -assert no\_default\_msg 19-35  
 –assert no\_default\_msg 19-34  
 –assert no\_fatal\_action 19-36  
 -assert psl\_in\_block 20-7  
 -assert quiet 19-35  
 -assert report 19-35  
 -assert success 19-17  
 -assert svvunit 19-56  
 assert\_ignore setup variable A-11  
 assert\_ignore\_optimized\_libs setup variable A-13  
 \$assert\_monitor 19-17, E-27  
 \$assert\_monitor\_off 19-17, E-28  
 \$assert\_monitor\_on 19-17, E-28  
 assert\_stop setup variable A-14  
 assertion failure messages  
     controlling 19-33  
 assertion warning messages  
     suppressing C-10  
 Assertions  
     SystemVerilog  
         enabling or disabling a module or a hierarchy C-14  
 assertions  
     fatal error generating E-11  
 OpenVera C-22  
     blind signals C-29  
     bounds check in dynamic arrays C-24  
     bounds check in fixed-size arrays C-24  
     circular dependency check C-27  
         display on screen C-28  
     disabling default failure messages D-15, D-16  
     encrypted IP mode

filename extension [C-30](#)  
**encryption**  
 tokens file [C-29](#)  
**file name extension** [C-23](#)  
**file-by-file preprocessing**  
 disabling [C-27](#)  
**include directory path** [C-23](#)  
 including case violations in the global  
     failure count [D-24](#)  
 interface ports named ifc\_signal [C-29](#)  
 left padding in strings [C-28](#)  
 RVM enabling [C-28](#)  
 signal property access functions  
     enabling [C-29](#)  
**testbench shell**  
 filename specifying [C-30](#)  
 generating only [C-29](#)  
 not generating [C-24](#)  
**testbench shell and shared object files**  
 specifying the directory [C-30](#)  
**timescale** [C-28](#)  
 VMM enabling [C-28](#)  
**Openvera**  
 ANSI mode [C-24](#)  
**PSL**  
 disabling default failure messages [D-15](#),  
     [D-16](#)  
**resume monitoring** [E-12](#)  
 returning true if one bit is true [E-13](#)  
 returning true if one bit is X [E-13](#)  
 returning true if only one bit is true or no bits  
     are true [E-13](#)  
 runtime error generating [E-11](#)  
 runtime warning generating [E-11](#)  
**SystemVerilog**  
 cover statements  
     disabling [C-21](#)  
**disabling** [C-21](#)  
 disabling assertion failure messages [D-16](#)  
     but enabling summary information [D-16](#)  
 disabling default failure messages [D-15](#),  
     [D-16](#)  
 disabling from a file  
     specifying assertion block [D-22](#)  
     specifying module definitions [D-22](#)  
 disabling information in a VPD file [C-21](#)  
 dumping SVA in VPD file  
     disabling [D-12](#)  
 enabling and disabling from a file [D-18](#)  
 enabling assertion match (success)  
     messages [D-18](#)  
 enabling from a file  
     specifying module definitions [D-22](#)  
 enabling runtime options [C-14](#)  
 enabling the -assert hier=file.txt runtime  
     option for turning assertions off [C-20](#)  
 enabling vacuous success messages [D-18](#)  
 enhancing reporting for assertions in  
     functions [C-14](#)  
 excluding assertion failures with fail action  
     blocks [D-15](#)  
 failure simulation time in Debussy [C-22](#)  
 generating a report file [D-16](#)  
     adding more information [D-18](#)  
 ignoring \$past [C-20](#)  
 maximum number of cover statement  
     specifying the total number of cover  
     statements in the assertion coverage  
     information [D-13](#)  
 monitoring for assertion coverage [D-22](#)  
 no match simulation time in Debussy [C-22](#)  
 not displaying the assert or cover  
     statement summary [D-15](#)  
 not writing the program\_name.db database  
     file [D-13](#)  
 specifying configuration file [C-14](#)  
 specifying the maximum number of failures  
     for each assertion [D-13](#)  
 specifying the maximum number of  
     successes for each assertion [D-13](#)  
 specifying the number of failures for an  
     assertion [D-12](#)  
 specifying the total number of assertion  
     failures [D-13](#)  
 starting simulation time in Debussy [C-22](#)  
 turning off monitoring [E-12](#)  
**\$assertkill** [E-12](#)  
**\$assertoff** [E-12](#)  
**\$asserton** [E-12](#)  
**assert.report** file [D-16](#)  
     adding more information [D-18](#)

assume 19-41, 27-5  
\$async\$and\$array E-40  
+auto2protect 25-37  
+auto3protect 25-37  
auto-inserted connect modules (AICMs)  
    displaying information about C-78  
+autoprotect 25-37  
avoid using +verilogamsext+vams C-79

## B

Backward SAIF File 24-6  
base time for simulation C-83  
-bfl and -bom C-123  
bidirectional registered mixed-signal net  
    displaying a list of C-78  
    finishing compilation at C-78  
bit  
    C/C++ function argument type 22-56  
    C/C++ function return type 22-55  
    input argument type 22-68  
    output and inout argument type 22-69  
    reg data type in two-state simulation 22-52  
bit string literal. 23-8  
\$bitstoreal E-30  
-bom and -bfl C-123  
BoM file  
    list of source files C-123  
bounds check  
    in OpenVera dynamic arrays C-24  
    in OpenVera fixed-size arrays C-24  
buffer  
    emptying into VCD files E-14

## C

-C C-77  
C 16-50  
-c 13-12, C-73  
C code generating  
    halt before compiling the generated C code  
        C-77

passing options to the compiler C-75  
specifying another compiler C-75  
specifying the optimization level C-77  
suppressing optimization for faster  
    compilation C-77  
C compilation setup variables A-18  
C compiler  
    not passing default options C-77  
    optimization levels C-75  
    passing options to C-75  
    specifying C-75  
C compiler, environment variable specifying  
the A-31  
C pre-processing 20-14  
"C" specifier of direct access 22-53  
C/C++ functions  
    argument direction 22-54, 22-55  
    argument type 22-54, 22-56  
    calling 22-58–22-60  
    declaring 22-52–22-58  
    extern declaration 22-53  
    in a Verilog environment 22-51–22-52  
    return range 22-54  
    return type 22-53, 22-55  
    using abstract access 22-76–22-122  
        access routines for 22-78–22-122  
    using direct access 22-67–22-76  
        examples 22-69–22-74  
C++ compiler  
    specifying C-76  
call PLI specification 22-7  
callbacks for memories and multi-dimensional  
arrays  
    enabling C-114  
calling C/C++ functions in your Verilog code  
22-58–22-60  
Case Expression 23-98  
case pragmas  
    enabling C-22  
cbk ACC capability 22-12, 22-22  
cbka ACC capability 22-12  
-CC C-75  
-cc C-75

cell  
   for delay annotation  
     disabling [E-2](#)  
     specifying [E-2](#)  
 cell modules  
   excluding from compilation [C-70](#)  
 'celldefine [C-70](#), [C-71](#), [E-2](#), [E-3](#)  
 CELLTYPE entries in SDF files  
   disabling [C-47](#)  
 -CFLAGS [C-75](#)  
 -cg\_coverage\_control [D-2](#)  
 char\*  
   direct access for C/C++ functions  
     formal parameter type [22-67](#)  
 char\*\*  
   direct access for C/C++ functions  
     formal parameter type [22-67](#)  
 charge decay  
   enabling [C-41](#)  
 +charge\_decay [C-41](#)  
 check argument to -ntb\_opts [B-15](#), [C-24](#)  
 check PLI specification [22-7](#)  
 check=all [C-24](#)  
 check=fixed [C-24](#)  
 checkers  
   clocking [19-62](#)  
   instantiating [19-62](#)  
 checkpoint  
   in VCD files  
     recording current values [E-14](#)  
     start recording current values [E-14](#)  
     stop recording current values [E-14](#)  
 circular dependency check check  
   in OpenVera [C-27](#)  
     display on screen [C-28](#)  
 class [16-68](#)  
 classes  
   inheritance between [16-70](#)  
 clock signals [11-44](#)–[11-48](#)  
 -cm [12-3](#), [D-22](#)  
 -cm assert [C-21](#)  
 code generation  
   mixed code generation [C-5](#)  
     command line options [13-11](#)  
     compiler directives [E-1](#)–[E-11](#)  
       resetting [E-3](#)  
     compile-time options [C-1](#)–[??](#)  
       displaying at runtime [D-40](#)  
       specifying in a file [C-54](#)  
     compiling  
       incremental compilation  
         triggering ??–[9-4](#)  
       omitting compilation between pragmas  
         [C-120](#)  
       verbose messages [2-19](#), [2-27](#), [C-66](#)  
       with 'include and -extinclude [B-27](#), [C-31](#)  
 compression  
   disasbling for VPD files [D-30](#)  
 conditional expressions  
   warning when evaluate to X or Z [C-80](#)  
     filtering out false negatives [C-80](#)  
 configuration file  
   for Radiant technology [C-40](#)  
 Connecting Nettypes  
   through Tranif Gates [14-106](#)  
 consistent behavior of class static properties  
[14-52](#)  
 constraint solver  
   array size warning [D-7](#)  
   OpenVera  
     trace information [D-5](#)  
 constraints  
   conflicts [16-65](#)  
   constraint profiling [16-34](#), [16-38](#)  
   debugging [D-7](#), [D-8](#)  
   partitions [16-26](#)  
   strings in [16-64](#)  
   test case extraction [16-35](#), [16-38](#)  
   test case verbose [16-38](#)  
   unique [16-80](#)  
 copyright information  
   displaying [D-27](#)  
 copyright message  
   suppressing [C-62](#)  
 \$countdrivers [E-44](#)  
 cover [27-5](#)  
 Coverage Database [27-3](#)

coverage groups  
     OpenVera  
         enabling [D-2](#)  
 -cpp [C-76](#)  
 cs\_assert\_stop\_next\_wait setup variable [A-14](#)  
 cs\_ccflags setup variable [A-18](#)  
 cs\_ccpath setup variable [A-19](#)  
 cs\_nocheck setup variable [A-5](#)  
 CustomSim [30-1](#)

+delay\_mode\_path [11-41, B-21, C-41](#)  
 'delay\_mode\_unit [E-7](#)  
 +delay\_mode\_unit [11-41, B-22, C-41](#)  
 'delay\_mode\_zero [E-7](#)  
 +delay\_mode\_zero [11-41, B-22, C-41](#)  
 delays [D-31, D-32](#)  
     changing all delays to zero [E-7](#)  
     changing all to 0 [C-41](#)  
     ignoring all delays except gate, switch, and  
         continuous assignment delays [E-6](#)  
     ignoring all delays except module path  
         delays [E-6](#)  
     ignoring all module path delays and using for  
         all other delay specifications the shortest  
         time precision argument [E-7](#)  
     ignoring module path delays and changing  
         gate, switch, and continuous assignment  
         delays to time precision [C-41](#)  
     ignoring module path delays and using gate,  
         switch, and continuous assignment  
         delays [C-42](#)  
     module path delays  
         X value [C-51](#)  
             with error message [C-52](#)  
     on gates and switches  
         ignoring [C-41](#)  
     specifies using max of min|typ|max delays  
[C-44](#)  
     specifies using min of min|typ|max delays  
[C-45](#)  
     specifies using typ of min|typ|max delays  
[C-45](#)  
         transport delays [C-47](#)  
 +deleteprotected [25-37](#)  
 delta cycle information [E-21](#)  
 Denali [32-1](#)  
 dep\_check argument to -ntb\_opts [B-15, C-27](#)  
 \$deposit [E-45](#)  
 -deraceclockdata option [5-58](#)  
 Design Description [13-12](#)  
 -diag libconfig [4-87](#)  
 -diag libdepends [7-10](#)  
 -diag pkgusage [7-12, 7-14](#)  
 -diag run\_env [C-124](#)

-diag sys\_task\_mem [5-64](#), [5-66](#)  
 diagnostics [6-113](#)  
 direct access for C/C++ functions  
     examples [22-69](#)–[22-74](#)  
     formal parameters  
         types [22-67](#)  
     rules for parameter types [22-67](#)–[22-69](#)  
     using [22-67](#)–[22-127](#)  
 DirectC  
     abstract access  
         specifying [C-60](#)  
     enabling [C-59](#)  
     listing the C/C++ functions [C-60](#)  
     using pass by reference [22-66](#)  
     vc\_hdrs.h file [C-60](#)  
 direction of a C/C++ function argument [22-55](#)  
 directory for constraint solver profiles and  
 testcases [D-8](#)  
 disable [C-21](#)  
 disable soft [16-78](#)  
 disable\_cover [C-21](#)  
 \$disable\_warnings [E-37](#)  
 Disabling XIndex Merging [10-35](#)  
 \$display [E-31](#)  
 DISPLAY\_VCS\_HOME [A-30](#)  
 displaying your environment setup [1-13](#), [1-14](#)  
 \$dist\_exponential [E-42](#)  
 \$dist\_normal [E-42](#)  
 \$dist\_poisson [E-42](#)  
 \$dist\_uniform [E-43](#)  
 dmptf [4-5](#)  
 donut layers  
     specifying the maximum number of [C-118](#)  
 double\*  
     direct access for C/C++ functions  
         formal parameter type [22-67](#)  
 DPI [16-50](#)  
 \$dumpall [E-14](#)  
 \$dumpfile [E-14](#)  
 \$dumpflush [E-14](#)  
 \$dumplimit [E-15](#)  
 \$dumpoff [E-14](#)  
 dumpoff [C-21](#), [D-12](#)  
 \$dumpon [E-14](#)  
 \$dumpports [E-17](#)  
 \$dumpportsall [E-18](#)  
 \$dumpportsflush [E-19](#)  
 \$dumpportslimit [E-19](#)  
 \$dumpportsoff [E-18](#)  
 \$dumpportson [E-18](#)  
 \$dumpvars [E-15](#)  
 dynamic race detection [C-81](#)

## E

-e name\_for\_main [C-57](#)  
 -E program [D-40](#)  
 echo [D-40](#)  
 edge operator [19-62](#)  
 edge sensitivity  
     in SDF file IOPATH entries [C-48](#)  
 elaboration system tasks [14-44](#)  
 Element Merging Methods  
     Dimensional [10-33](#)  
     Index Resolution [10-34](#)  
     Randomized [10-34](#)  
 'else [E-3](#)  
 'elseif [E-4](#)  
 enable\_diag [19-17](#), [C-14](#)  
 enable\_hier [C-20](#)  
 \$enable\_warnings [E-37](#)  
 enabling [D-2](#)  
     only where used in the last simulation [22-32](#)  
 enabling writing initialized values to a file [4-31](#)  
 encoding [25-19](#)  
 encryption  
     all modules [25-37](#)  
         but not the module header [25-37](#)  
         but not the module header and parameter  
             declarations [25-37](#)  
 enabling overwriting of existing files [25-37](#)  
 enabling PLI and UCLI access [25-38](#)  
 OpenVera  
     tokens file [C-29](#)

SDF files [25-38](#)  
 specifying the directory for encrypted files [25-38](#)  
 specifying with ‘protect’ [endprotect 25-38](#)  
 ‘endcelldefine [E-2](#)  
 ‘endif [E-4](#)  
 ending simulation at a specified time [D-25](#)  
 ‘endprotect [25-32](#)  
 ‘endprotected [25-33](#)  
 ‘endprotected128 [25-33](#)  
 Environment variables [1-7-1-8, A-1-??](#)  
 -error [4-111, 4-112, 4-114, 4-117, C-11, C-62, D-26](#)  
 \$error [E-11](#)  
 ERROR message [A-11, A-14](#)  
 error messages  
     changing to warning [C-65](#)  
 error\_default\_action\_block [19-58](#)  
 -error=PRIORITY [D-23](#)  
 -error=UNIQUE [D-23](#)  
 EVCD files [E-17](#)  
     flushing the buffer [E-19](#)  
     recording all port values [E-18](#)  
     resume recording [E-18](#)  
     specifying the file size [E-19](#)  
     suspending [E-18](#)  
 -exclude [30-18](#)  
 executable  
     specifying the name of [C-117](#)  
 -exitstatus [5-64](#)  
 exporting Vera tasks [13-10](#)  
 extended summary information  
     displaying [D-27](#)  
 extends directive  
     advice [15-4](#)  
     introduction [15-4](#)  
 extern declaration [22-53](#)  
 extern declarations [22-74](#)  
 -extinclude [B-27, C-31, E-9](#)

F  
 -F [C-55](#)  
 -f filename [C-54](#)  
 fail action blocks [D-15](#)  
 fail-only [19-50](#)  
 FAILURE message [A-11, A-14](#)  
 \$fatal [E-11](#)  
 fatal assertion error generating [E-11](#)  
 \$fclose [E-31](#)  
 \$fdisplay [E-32](#)  
 \$ferror [E-32](#)  
 \$fflush [E-15, E-32](#)  
 \$fflushall [E-16](#)  
 \$fgetc [E-32](#)  
 \$fgets [E-32](#)  
 -file [2-18, 2-26, C-54](#)  
 file  
     for runtime options [D-34](#)  
 file name extension in Verilog library directories  
     specifying [C-7](#)  
 files  
     grw.dump file [E-16](#)  
     tokens.v [25-38](#)  
 VCD files  
     specifying the filename [E-14](#)  
 VPD  
     specifying at compile-time [C-113](#)  
     start recording [E-26](#)  
 filter\_past [C-20](#)  
 \$finish [E-36](#)  
 finish\_maxfail=N [D-12](#)  
 \$fmonitor [E-32](#)  
 \$fopen [C-60, E-32](#)  
     increasing the frequency of flushing [C-61](#)  
 force [23-27](#)  
 -force\_list [5-9](#)  
 -force\_list\_hier [5-24](#)  
 -force\_list=uniqueID [5-29, 5-31, 5-33](#)  
 foreach loops [16-55](#)  
 four state Verilog data  
     stored in vec32 [22-61](#)

frc ACC capability [22-12](#), [22-22](#)

\$fread [E-33](#)

\$fscanf [E-33](#)

FSDB files [C-114](#)

\$fseek [E-33](#)

\$fstobe [E-33](#)

\$ftell [E-33](#)

function calls

    context [16-52](#)

    DPI [16-50](#)

    non-pure [16-51](#)

    pure [16-51](#)

\$fwrite [E-33](#)

## G

-gl-generics cmdfile [C-92](#)

gate ACC capability [22-13](#)

gate-level

    improving runtime performance [C-120](#)

gd\_pulsewarn [11-9](#)

generic classes [23-18](#)

generic constants [23-22](#)

generic list [23-22](#)

generic type [23-22](#)

generics

    constants [23-18](#)

    overriding [C-92](#)

        from a file [C-92](#)

    overriding with the -gfile elaboration option  
        [C-79](#), [C-87](#)

    packages [23-18](#)

    subprograms [23-18](#)

    types [23-18](#)

-genid\_db [C-113](#)

\$get\_initial\_random\_seed [E-43](#)

\$getpattern [E-45](#)

-gfile [C-87](#)

-gfile cmdfile [C-79](#)

global\_finish\_maxfail=N [D-13](#)

globalDirective [19-29](#)

gmake [A-2](#)

\$gr\_waves [E-16](#)

grw.dump file [E-16](#)

-gui [2-21](#), [2-26](#)

-gv|gvalue generic=value [C-92](#)

## H

-h [2-17](#), [2-23](#), [C-10](#)

hard constraint [16-66](#)

header and summary

    suppressing [D-26](#)

header\_file [25-16](#)

-help [2-17](#), [2-23](#), [C-10](#)

help with compile-time options, runtime options, and environment variables [C-10](#)

hier=file\_name [D-18](#)

\$hold [E-38](#)

-hsopt=gates [C-120](#)

-hsopt=racedetect [3-23](#)

HVP [27-7](#)

## I

-ID [2-17](#), [2-23](#), [C-72](#)

IEEE-1850-2010 [20-9](#)

ifc\_signal

    OpenVera interface ports named [C-29](#)

'ifdef [E-4](#)

'ifndef [E-6](#)

-ignore [2-8](#), [B-12](#), [C-10](#)

Importing VHDL procedures [13-8](#)

importing VHDL procedures [13-8](#)

+inccdir [2-25](#), [C-111](#)

'include [E-8](#)

    specifying the search directories [C-111](#)

including one source file in another [E-8](#)

Inclusion of Dynamic Types in Sensitivity List  
[14-97](#)

incremental compilation [C-8–C-10](#)

    central place for descriptor information and  
    object files [C-9](#)

disabling [C-10](#)

specifying the make path C-9  
updating the makefile C-9  
incremental compile directory  
  specifying C-8  
Index BSpace 10-29  
Index Selection Methods 10-33  
\$info E-12  
information messages  
  about finding module definitions in a library  
    C-63  
  lint C-63  
-ignore 2-8, B-12, C-10  
initializing Verilog memories and variables  
  in only parts of the design 4-24, 4-26  
inout  
  C/C++ function argument direction 22-55  
input  
  C/C++ function argument direction 22-55  
int  
  C/C++ function argument type 22-56  
  C/C++ function return type 22-55  
  direct access for C/C++ functions  
    formal parameter type 22-67  
  input argument type 22-68  
  output and inout argument type 22-69  
int\*  
  direct access for C/C++ functions  
    formal parameter type 22-67  
INTERCONNECT delays  
  rejecting C-51  
  SDF files C-46  
    changing to transport delays C-47  
    negative values enabling C-52  
interface 13-14  
  self() 14-138  
Interface Description 13-20  
internal disk cache for randomization  
  delete before simulation D-3  
  location D-3  
intra-assigment delays  
  removing C-46  
-io\_gz D-35

IOPATH delays  
  SDF files  
    negative values enabling C-52  
+iopath+edge C-48  
-ipopt=ext 25-20  
-ipopt=outdir 25-20  
-ipopt=overwrite 25-21  
-ipopt=partialprotect 25-17  
-ipout 25-21  
-ipprotect 25-16  
Isolate  
  Cost of Garbage Collection 6-114  
\$isunknown E-13  
\$itor E-30

## J

-jnumber\_of CPUs C-76

## K

-kdb 35-3  
key word conflicts C-79  
key\_keyname 25-18  
key\_keyowner 25-18  
key\_method 25-19  
key\_public\_key 25-19  
keywords  
  after 15-11  
  around 15-11  
  before 15-11  
  extends 15-3  
  virtuals 15-31

## L

-I D-27  
-I filename 2-19, 2-27, C-115, D-26  
-ld linker C-73  
-LDFLAGS options C-73  
let operator 19-62  
+libext 2-12, 2-14, 2-25, B-23, C-7

**-liblist** [4-70](#)  
**-liblist lib1+lib2+lib3+...libn** [B-13](#)  
**-liblist\_nocelldiff** [4-73, 4-74](#)  
**-liblist\_work** [4-73](#)  
**-libmap** [4-58, B-14](#)  
**-libmap\_verbose** [4-87](#)  
**+libborder** [2-13, 2-25, C-7](#)  
**library**  
  name mapping [1-11](#)  
**+librescan** [2-13, C-7](#)  
**+libverbose** [C-8, C-63](#)  
**licenses**  
  enabling license queuing [D-34](#)  
  running VCS with three VCSI licenses [D-34](#)  
  running VCSI with a VCS license [D-34](#)  
  waiting for a license [D-34](#)  
  waiting for a network license [D-34](#)  
**'line** [E-10](#)  
**Line-Based CPU Time Profiler** [6-108](#)  
**linker**  
  linking a library to the executable [C-73](#)  
  linking by hand [C-73](#)  
  passing flags to [C-73](#)  
  specifying [C-73](#)  
  temporary object files [C-73](#)  
**linking**  
  linking a specified library to the executable  
    [C-73](#)  
  linking by hand [C-73](#)  
  passing options to the linker [C-73](#)  
  specifying another linker [C-73](#)  
**+lint** [2-12, 4-111, B-23, C-63](#)  
**lint messages** [C-63](#)  
**+list** [22-128](#)  
**list file** [B-4](#)  
**LMC SWIFT interface**  
  including [C-61](#)  
**-lmc-swift** [C-61](#)  
**-lmc-swift-template** [C-61](#)  
**-lname** [C-73](#)  
**-load** [4-22, 22-36, C-59, D-44](#)  
**-location**  
  vlog option [B-14](#)  
  
**\$log** [E-29](#)  
**log file**  
  appending to [C-115](#)  
  simulation  
    specifying [D-26](#)  
**log file buffers**  
  increasing the frequency of flushing [C-60](#)  
**log files**  
  increasing the frequency of log file dumping  
    [D-33](#)  
  increasing the frequency of log file, VCD file,  
    and \$open file dumping [D-33](#)  
  specifying compilation log file [2-19, 2-27, C-115](#)  
  specifying with a system task [E-29](#)  
**loops**  
  specifying the maximum number of loops  
    [C-118, C-119, D-41, D-42](#)  
**LSI certification** [E-16](#)  
**EVCD files** [E-17](#)  
**flushing the buffer** [E-19](#)  
**including strength levels in the VCD file** [E-16](#)  
**recording all port values** [E-18](#)  
**resume recording** [E-18](#)  
**specifying the file size** [E-19](#)  
**suspends recording** [E-18](#)  
**\$lsl\_dumpports** [3-55–3-60, E-16](#)

## M

**macros**  
  text macro defining [C-116](#)  
**text macros**  
  defining [E-3](#)  
  else defining [E-3](#)  
  else if end [E-4](#)  
  elseif defining [E-4](#)  
  if defining [E-4](#)  
  if not defined [E-6](#)  
  undefining [E-6](#)  
**main() routine**  
  specifying for PLI [C-57](#)  
**maintaining filename and line number** [E-10](#)  
**make** [A-2](#)

makefile  
    updating [C-9](#)  
mangled file source protection [25-38](#)  
    except module and port identifiers [25-39](#)  
mapping, library name [1-11](#)  
-Marchive [C-8, C-73](#)  
MATCH\_ENUM\_VECTOR [A-30](#)  
matching relational operators [23-15](#)  
maxargs PLI specification [22-8](#)  
maxcover=N [D-13](#)  
    [D-41](#)  
    +maxdelays [C-41, C-44, D-31](#)  
maxfail=N [D-13](#)  
-maxLayers value [C-118](#)  
maxsuccess=N [D-13](#)  
-mcg [C-5](#)  
MDAs [16-56](#)  
-Mdir [C-8](#)  
-Mdirectory [C-8](#)  
mem\_pli [6-17](#)  
mem\_solver  
    argument to the profrpt -view option [6-13, 6-82](#)  
+memcbk [C-114](#)  
Memory Constraint Solver view  
    in profiler reports [6-13, 6-82, 6-92](#)  
Memory Modeler - Advanced Verification (MMAV) [32-1](#)  
memory size limits [3-43](#)  
message control  
    by module definition [4-135](#)  
    by source file [4-135](#)  
    by sub-hierarchy [4-136](#)  
    downgrading error messages [4-133](#)  
    error messages [4-132](#)  
    lint messages [4-130](#)  
    suppressing messages [4-134](#)  
    upgrading lint and warning messages [4-133](#)  
    warning messages [4-132](#)  
messages  
    about finding module definitions in a library [C-63](#)  
    changing error to warning [C-65](#)  
copyright message  
    suppressing [C-62](#)  
lint [C-63](#)  
quiet mode [C-65](#)  
verbose mode [C-65](#)  
    including CPU time information [C-66](#)  
warning  
    disabling [B-27, C-66](#)  
-metadump [4-137](#)  
-metadump\_txt [4-144](#)  
minargs PLI specification [22-8](#)  
    [D-41](#)  
+mindelays [C-41, C-45, D-31](#)  
mip ACC capability [22-13](#)  
mipb ACC capability [22-13, E-7](#)  
MIPDs [D-43](#)  
    disabling connection upon MIPD delay annotation [D-43](#)  
misc PLI specification [22-7](#)  
mixed analog/digital simulation  
    specifying [C-78](#)  
mixed code generation [C-5](#)  
mixed signal simulation  
    specifying [C-78](#)  
-Mlib=dir [C-9](#)  
-Mmakep [C-9](#)  
module description , Verilog [13-21](#)  
-module module\_identifier [D-22](#)  
module path delays  
    allowing  
        in specific module instances [11-43](#)  
    changing to tranport delays [C-47](#)  
    disabling [C-50](#)  
    disabling for an instance [11-43](#)  
    suppressing  
        in specific module instances [11-43](#)  
    X value [C-51](#)  
    X value with error message [C-52](#)  
\$monitor [E-31](#)  
\$monitoroff [E-31](#)  
\$monitoron [E-31](#)  
-monsigs option [C-83](#)  
mp ACC capability [22-13](#)

-msg\_config [4-112](#), [4-129](#)  
 multiple packed dimensions [16-56](#)  
 +multisource\_int\_delays [C-46](#)  
 -Mupdate [C-9](#)

**N**

+nbaopt [C-46](#)  
 -nbaudpsched [C-120](#)  
 \*\*NC [3-15](#)  
 -nc [C-62](#)  
 +neg\_tchk [11-66](#), [11-73](#), [C-53](#)  
 negative timing checks [C-52](#)  
 -negdelay [C-52](#)  
 nets  
     specifung defult data type [E-3](#)  
 no\_default\_msg [D-15](#)  
 -no\_error ID+ID [C-65](#)  
 no\_fatal\_action [D-15](#)  
 no\_file\_by\_file\_pp argument to -ntb\_opts [B-15](#), [C-27](#)  
 +no\_identifier [D-24](#)  
 +no\_notifier [11-66](#), [C-50](#)  
 +no\_pulse\_msg [D-27](#)  
 +no\_tchk\_msg [11-66](#), [C-50](#), [D-24](#)  
 +nocelldefinepli+1 [C-70](#)  
 nocelldefinepli PLI specification [22-8](#)  
 +nocelldefinepli+0 [C-70](#)  
 +nocelldefinepli+2 [C-71](#)  
 nocovdb [D-13](#)  
 -noerror UPIMI+IOPCWM [C-65](#)  
 -xzcheck [C-80](#)  
 -nogenid\_db [C-113](#)  
 NOIGNORE message [A-11](#)  
 -nolncrComp [C-10](#)  
 -noinherit\_timescale option [B-19](#), [C-83](#)  
 +nolibcell [C-70](#)  
 \$nolog [E-29](#)  
 nonblocking assignments  
     removing intra-assignment delays [C-46](#)  
 non-static expressions

port map [23-16](#)  
 +noportcoerce [C-112](#)  
 nopostproc [D-15](#)  
 +nospecify [11-67](#), [C-50](#)  
 NOSTOP message [A-14](#)  
 NOTE message [A-11](#), [A-14](#)  
 -notice [2-18](#), [2-27](#)  
 notifier register  
     in timincheck system tasks  
         disabling [C-50](#)  
 notifier registers, suppressing the toggling of [D-24](#)  
 +notimingcheck [11-67](#), [C-50](#), [D-24](#)  
 'nounconnected\_drive [E-11](#)  
 -novitaltiming [D-31](#)  
 -ntb [2-9](#), [B-14](#), [C-22](#)  
 +ntb\_cache\_dir [D-3](#)  
 -ntb\_define [2-9](#), [2-14](#), [B-14](#), [C-23](#)  
 +ntb\_delete\_disk\_cache [D-3](#)  
 +ntb\_enable\_solver\_trace\_on\_failure [D-5](#)  
 +ntb\_exit\_on\_error [D-5](#)  
 -ntb\_filext [2-10](#), [B-14](#), [C-23](#)  
 -ntb\_incdir [2-10](#), [B-14](#), [C-23](#)  
 -ntb\_noshell [C-24](#)  
 -ntb\_opts [B-14](#), [C-24](#)  
     print\_deps [B-15](#), [C-28](#)  
     rvm [C-28](#)  
     sv\_fmt [C-28](#)  
 -ntb\_opts no\_cb\_edge\_override [14-144](#)  
 -ntb\_opts no\_file\_by\_file\_pp [13-33](#)  
 +ntb\_random\_seed [D-6](#)  
 +ntb\_random\_seed\_automatic [D-6](#)  
 -ntb\_sfname [C-30](#)  
 -ntb\_shell\_only [C-29](#)  
 -ntb\_sname [C-30](#)  
 +ntb\_solver\_array\_size\_warn [D-7](#)  
 +ntb\_solver\_debug [16-29](#), [D-7](#)  
     extract [16-35](#), [16-38](#)  
     profile [16-38](#)  
     serial [16-37](#)  
     trace [16-30](#), [16-37](#)  
     verbose+ntb\_solver\_debug

verbose 16-38  
+ntb\_solver\_debug\_dir D-8  
+ntb\_solver\_debug\_filter 16-30, 16-33, 16-35,  
D-8  
+ntb\_solver\_mode D-9, D-10  
-ntb\_spath C-30  
-ntb\_vipext 13-33, C-30  
+NTC2 11-72, C-53

## O

-o name C-117  
-O number C-77  
-O0 C-77  
object files  
    enabling position independent code C-74  
    specifying temporary C-73  
+object\_protect 25-50  
+old\_ntc C-53  
\$onehot  
    \$onehot E-13  
\$onehot0 E-13  
OpenVera  
    constraint solver mode D-9  
    coverage groups D-2  
    diagnostics  
        when randomize() method called D-4  
    enabling debugging  
        when randomize() method called D-4  
    exit on error D-5  
    internal disk cache D-3  
        delete before simulation D-3  
    on null object handle of object randomized  
        D-3  
trace information  
    when randomize() returns 0 D-4  
trace information when constraint solver fails  
    D-5  
operating system commands, executing E-28  
+optconfigfile 9-8, C-40  
optimization  
    suppresssing for faster compilation C-77  
options for macro expansion B-17, C-116

options, command line 13-11  
output  
    C/C++ function argument direction 22-55  
OVA 19-33  
-ova\_enable\_case 19-39, C-22  
-ova\_enable\_case\_maxfail D-24  
-ova\_enable\_case\_maxfail 19-37, D-24  
-ova\_file 2-10  
-ova\_inline C-22  
-ova\_inline 19-39, C-22  
+override\_model\_delays D-31, D-32, D-41  
-override\_timescale B-18, C-82  
-override-cflags C-77  
Overriding Parameter Values  
    Through Configuration File 14-95  
OVERWRITE\_PREDECLARATION\_IN\_LIB  
A-33

## P

-P pli.tab 22-23, C-58  
-p188\_macro\_expansion B-17, C-116  
packed constraints 16-55  
packed dimensions 16-55  
parallel compilation C-10, C-77  
    disabling C-10  
    specifying the number of forks C-76  
parallel\_compile setup variable 11-80, A-5, A-7  
-parallel\_compile\_off C-10  
-parameters 2-26, 4-31, C-80  
parameters  
    overriding C-80, C-89  
    overriding with the -gfile elaboration option  
        C-79, C-87  
partitions  
    in constraints 16-26  
pass by reference in DirectC 22-66  
+pathpulse C-49  
PATHPULSE\$ specparam, enabling C-49  
performance  
    improving for gate-level designs C-120  
\$period E-38

persistent 22-9  
**PERSISTENT\_FLAG** A-31  
**-picarchive** C-74  
 placement element  
     after 15-11  
     around 15-11  
**-platform** C-118  
 platform directory in the VCS installation  
     returning C-118  
**PLI**  
     ACC capabilities C-57  
         enabling debugging C-57  
     allowing access to ports and parameters C-71  
     disabling capabilities for 'celldesign and library modules' C-71  
     disabling capabilities for 'celldesign modules' C-71  
     enabling in encrypted files 25-38  
     specifying the name of your main() routine C-57  
**PLI library**  
     loading dynamically at runtime 4-22, D-44  
**PLI specifications**  
     args 22-8  
     call 22-7  
     check 22-7  
     data 22-7  
     maxargs 22-8  
     minargs 22-8  
     misc 22-7  
     nocelldesignpli 22-8  
     size 22-8  
**PLI table file** 22-6–22-24, D-40  
     specifying C-58  
**pli\_learn.tab** D-40  
**+pli\_unprotected** 25-38  
**pli.tab file** 22-6–22-24, D-43  
     specifying C-58  
**+plusarg\_ignore** C-57  
**+plusarg\_save** C-56  
 plusargs, checking for on the simv command line E-44  
**+plus-options** D-41  
**pointer**  
     C/C++ function argument type 22-56  
     C/C++ function return type 22-55  
     input argument type 22-68  
     output and inout argument type 22-69  
**port coercion**  
     disabling C-112  
**position independent code**  
     enabling C-74  
**print\_deps** argument to -ntb\_opts B-15, C-28  
**\$printtimescale** E-36  
**priority keyword** 14-116  
**PRIORITY violations**  
     limiting the number of D-23  
**procedure\_prototype**  
     example 15-29, 15-30  
**procedures, importing** 13-8  
**profile database**  
     specifying the pathname 6-7  
**profiler**  
     simulation 6-1  
**Profiler Report**  
     Third-Party Shared Library 6-116  
**profiling** C-53  
**profprt** 6-19  
**profprt** 6-20  
**program\_name.db** database file  
     not writing D-13  
**proprietary message**  
     suppressing D-26  
**'protect** 25-32, 25-38  
**+protect\_file\_suffix** 25-38  
**'protect128** 25-33  
**'protected** 25-32  
**prx ACC capability** 22-12  
**PSL** 19-33, 19-56, 20-6  
**PSL macros** 20-9  
**pulse error messages**  
     suppressing D-27  
**pulse error messages**  
     suppressing D-24

**+pulse\_e/number** 11-27, 11-29, 11-31, 11-36, 11-37, C-51  
**+pulse\_int\_e** 11-26, 11-27, 11-29, 11-31, C-51  
**+pulse\_int\_r** 11-26, 11-27, 11-29, 11-31, C-51  
**+pulse\_on\_detect** 11-37, C-52  
**+pulse\_on\_event** 11-36, C-51  
**+pulse\_r/number** 11-27, 11-29, 11-31, 11-36, 11-37, C-51  
**pulses**  
    filtering out narrow pulses C-51  
    and flag as error C-51  
    on INTERCONNECT delays  
        INTERCONNECT delays  
            filtering out  
                SDF files  
                    INTERCONNECT  
                        delays  
                        filtering out  
                            C-51  
        rejecting narrow pulses C-51  
        on SDF INTERCONNECT delays C-51  
    X value C-51, C-52  
**+putprotect+target\_dir** 25-38  
**-pvalue** 2-26, 4-31, C-80, C-89

## Q

**-q** 2-18, 2-27, C-65, D-26  
**\$q\_add** E-41  
**\$q\_exam** E-41  
**\$q\_full** E-41  
**\$q\_initialize** E-41  
**\$q\_remove** E-41  
queue=blocking|nonblocking 30-8  
quiet mode - suppressing  
    header and summary information D-26  
    proprietary message D-26  
    simulation report at the end of simulation D-26

## R

**-R** 2-18, 2-26, C-40, C-115  
**rACC capability** 22-11, 22-13, 22-15, 22-21  
**-race** C-80  
**race conditions**  
    avoiding 3-2–3-8  
    continuous assignment evaluations 3-5  
    generating a report of C-80, C-81  
    in counting events 3-7  
    in flip-flops 3-4  
    setting a value twice at the same time 3-3  
    time zero 3-7  
    using and setting a value at the same time 3-3  
**+race=all** 3-21, C-81  
**-racecd** C-81  
**race.out file** C-81  
**+rad** 9-8, C-40  
**Radiant technology**  
    configuration file C-40  
    enabling C-40  
**rand members** 16-68  
**rand\_mode() method** 14-73  
**\$random** 3-54, E-43  
**random number generator**  
    re-seeding D-6  
**random values**  
    setting the seed D-6  
    after restore D-6  
**randomize() method** 14-73  
**randomize() serial number** 16-37  
**randomize()solver trace** 16-29  
**randomized objects in a structure** 16-59  
**\$readmemb** E-35  
**\$readmemh** E-35  
**real**  
    C/C++ function argument type 22-56  
    input argument type 22-68  
    output and inout argument type 22-69  
**\$realtime** E-42  
**\$realtobits** E-30  
**\$recovery** E-38

**\$recmem** E-39  
 checking timestamp and timecheck  
     conditions C-53  
 disabling delayed versions of signals in other  
     timing checks C-53  
**reg**  
     C/C++ function argument type 22-56  
     C/C++ function return type 22-55  
     input argument type 22-68  
     output and inout argument type 22-69  
**release** 23-27  
 reporting debug capabilities for each module  
**6-103**  
**\$reset** E-43  
**\$reset\_count** E-43  
**\$reset\_value** E-44  
**'resetall** E-3  
 resetting  
     keeping track of the number of resets E-43  
     passing a value from before to after a reset  
         E-44  
     resetting VCS to simulation time 0 E-43  
 Resolving message upon instance resolution  
**C-8**  
 resolving module instances E-9  
**\$restart** E-45  
**restrict** 19-41  
 RETAIN entries  
     SDF files  
         enabling C-47, C-48  
 return range of a C/C++ function 22-54  
 return type of a C/C++ function 22-53, 22-55  
 RTL Verilog example 13-13  
**\$rtoi** E-30  
 runtime assertion error generating E-11  
 runtime assertion warning generating E-11  
 runtime options  
     compiling into the executable C-56  
     prevent compiling into the executable C-57  
     specifying in as file D-34  
**RVM** C-28  
**rvm** C-28  
 rw ACC capability 22-12, 22-14, 22-22

## S

**s** ACC capability 22-12  
**-saif\_opt** 24-11  
**\$save** E-45  
**scalar**  
     direct access for C/C++ functions  
         formal parameter type 22-67  
**scalar\***  
     direct access for C/C++ functions  
         formal parameter type 22-67  
**scope randomize method** 14-73  
**SDF** 11-7  
     optimistic mode 11-7  
**SDF backannotating**  
     enabling more than 10 warning and error  
         messages C-62  
**SDF delay back-annotation**  
     disabling back-annotation to individual bits of  
         an input port E-8  
     to individual bits of an input port E-7  
**SDF files**  
     compiling separate files for min|typ|max  
         delays C-41  
     disabling CELLCYPE entries C-47  
     encryption 25-38  
     INTERCONNECT delays C-46  
         changing to transport delays C-47  
         negative values enabling C-52  
         rejecting C-51  
     INTERCONNECT entries  
         negative values enabling C-52  
     IOPATH delays  
         negative values enabling C-52  
     IOPATH entries  
         edge sensitiviry C-48  
         negavive values enabling C-52  
     min|typ|max delays  
         specified in a file C-40  
     RETAIN entries  
         enabling C-47, C-48  
**-sdf** min|typ|max  
     instance\_name  
         file.sdf C-40

**\$sdf\_annotate** [E-44](#)  
**+sdf\_nocheck\_celltype** [C-47](#)  
**+sdfprotect file\_suffix** [25-38](#)  
**-sdfretain** [11-7, C-47](#)  
**-sdfretain=warning** [C-48](#)  
**SDFRT\_IRV**  
  **wanring** [C-48](#)  
**+ sdfverbose** [C-62](#)  
**search order of Verilog library directories** [2-13, C-7](#)  
  **rescan** [2-13, C-7](#)  
**sequential devices**  
  **inferring** [3-35–3-39, 11-44–11-48](#)  
**sequential UDPs**  
  **changing output evaluation to the active region** [C-120](#)  
  **changing output evaluation to the NBA tile slot** [3-5](#)  
**serial2trace.txt file** [16-34](#)  
**\$setup** [E-39](#)  
**setup files**  
  **synopsys\_sim.setup** [1-9](#)  
**setup variables** [A-3](#)  
  **assert\_ignore** [A-11](#)  
  **assert\_ignore\_optimized\_libs** [A-13](#)  
  **assert\_stop** [A-14](#)  
  **assigning values to** [A-3](#)  
  **cs\_assert\_stop\_next\_wait** [A-14](#)  
  **cs\_ccflags** [A-18](#)  
  **cs\_ccpath** [A-19](#)  
  **cs\_nocheck** [A-5](#)  
  **parallel\_compile** [11-80, A-5, A-7](#)  
  **spc** [A-5](#)  
  **timebase** [A-7](#)  
  **use** [A-16](#)  
**\$setuphold** [E-40](#)  
  **checking timestamp and timecheck conditions** [C-53](#)  
  **disabling delayed versions of signals in other timing checks** [C-53](#)  
**show\_setup command** [1-13](#)  
**-sig program** [D-40](#)  
**signal property access functions**  
  **OpenVera**  
            **enabling** [C-29](#)  
            **-simBin** [35-23](#)  
            **-simdir** [35-6, 35-23](#)  
            **-simflow** [35-5, 35-23](#)  
            **-simprofile** [6-18, C-53](#)  
            **-simprofile compile-time option** [6-5](#)  
            **-simprofile runtime option** [6-5](#)  
            **-simprofile compile-time option** [6-5](#)  
            **-simprofile mem** [6-20](#)  
            **-simprofile runtime option** [6-5](#)  
            **-simprofile time** [6-19](#)  
            **simprofile\_dir** [6-19](#)  
            **-simprofile\_dir\_path** [6-7](#)  
**simulation**  
  **immediately after compilation** [C-40](#)  
  **setup variables** [A-11](#)  
**simulation report at the end of simulation**  
  **suppressing** [D-26](#)  
**simulation state**  
  **saving** [E-45](#)  
**simulation time slice based profiler** [6-110](#)  
**simv executable**  
  **specifying a deifferent name** [C-117](#)  
**single class** [16-67](#)  
**single packed dimension** [16-55](#)  
**size PLI specification** [22-8](#)  
**\$skew** [E-40](#)  
**skip\_celldefine\_scopes** [24-12](#)  
**-skip\_translate\_body** [C-120](#)  
**Slices in Array Aggregates** [23-90](#)  
**-sn=+nbavhd** [5-56](#)  
**-sn=+rdr** [5-56](#)  
**-sn=+rfx** [5-56](#)  
**soft constraint** [16-66](#)  
**soft constraints** [16-65, 16-74](#)  
  **disabling** [16-65](#)  
  **prioritization** [16-67](#)  
**soft keyword** [16-66](#)  
**solver trace reporting**  
  **for the specified randomize() calls** [16-37](#)  
**SOMA** [32-2](#)

souce protection  
 enabling overwriting of existing files 25-37  
 enabling PLI and UCLI access 25-38  
 encrypting all modules 25-37  
     but not the module headers 25-37  
     but not the module headers and parameter declarations 25-37  
 specifying the directory for protected files 25-38  
 specifying with 'protect' 'endprotect' 25-38

source file  
 specifying in a file C-54

source files  
 generating a list of C-123

source protection  
 mangling 25-38  
     except module and port identifiers 25-39

SDF files 25-38  
 specifying the end of the code to be protected 25-32  
 specifying the end of the protected code 25-33  
 specifying the start of the code to be protected 25-32, 25-33  
 specifying the start of the protected code 25-32

spc setup variable A-5

specify blocks  
 allowing  
     in specific module instances 11-43  
 disabling for an instance 11-43  
 suppressing C-50  
     in specific module instances 11-43

random(seed) system function D-6, D-7

\$readmemb E-35  
 \$readmemh E-35

state variables 16-49

Static Race Detection Tool C-81

std  
     randomize() method 14-73

\$stimen E-42  
 D-40  
 \$stop E-36

stopping simulation at a specified time D-25

string  
 C/C++ function argument type 22-56  
 C/C++ function return type 22-55  
 input argument type 22-68  
 output and inout argument type 22-69

strings  
 in constraints 16-64

\$strobe E-31

-suppress 4-111, C-62

SV and RT assertions  
 browse, enable, and disable C-59  
 sv\_fmt argument to -ntb\_opts C-28  
 -sv\_opts B-7  
 -sv\_pragma 2-10, B-17  
 SVA 19-33, 19-56  
 -sva B-7  
 -sverilog C-10

SWIFT SmartModels D-41  
 generating a template C-61  
 replaying DelayRange parameter definition with +mindelay, +typdelay, or +maxdelay D-41

\$sync\$nor\$plane E-40  
 /\*synopsys translate\_off\*/ pragma C-120  
 //synopsys translate\_off pragma C-120  
 /\*synopsys translate\_on\*/ pragma C-121  
 //synopsys translate\_on pragma C-121  
 synthesis policy checking A-5  
 -sysc C-112  
 -sysc=ams 30-13  
 \$system E-28  
 System Function Call  
     \$clog2() 16-87  
     \$size() 16-86

system tasks E-11–E-54, ??–E-54  
 disabling text output from D-27  
 IEEE standard system tasks not implemented E-54

SystemC  
 cosimulating with Verilog 1-3, 21-1

SystemC cosimulation C-114  
 enabling C-112

time resolution [C-112](#)  
**SYSTEMC\_OVERRIDE** [A-31](#)  
**\$systemf** [E-29](#)  
**SystemVerilog** [16-65, 19-56](#)  
 enabling [C-10](#)  
 randomized objects in a structure [16-59](#)  
 specifying the filename extension [C-30](#)  
 unpacked dimensions [C-113, E-26](#)  
**SystemVerilog assertions** [19-1-??](#)  
**SystemVerilog LRM** [16-50](#)  
**+systemverilogext** [2-13, B-26, C-30](#)

**T**

**-t** [13-11](#)  
**tasks, exporting** [13-10](#)  
**tb\_timescale** argument to **-ntb\_opts** [B-16, C-28](#)  
**tchk ACC capability** [22-12](#)  
**temporary object files** [C-73](#)  
**\$test\$plusargs** [E-44](#)  
**testbench**  
 OpenVera  
 enabling [C-22](#)  
 macro on command line [C-23](#)  
 timescale [C-28](#)  
**testbench template** [13-14](#)  
**+tetramax** [C-112](#)  
**TetraMAX testbench simulation in zero delay mode** [C-112](#)  
**text macros**  
 defining [C-116, E-3](#)  
 else defining [E-3](#)  
 else if end [E-4](#)  
 elseif defining [E-4](#)  
 if defining [E-4](#)  
 if not defined [E-6](#)  
 undefining [E-6](#)  
**text output display from system tasks**  
 disabling [D-27](#)  
**The %if Construct** [20-12](#)  
**\$time** [E-42](#)  
**time base** [C-83](#)  
**Time Constraint Solver view**

in profiler reports [6-12, 6-82, 6-83](#)  
**time precision**  
 as delay specification [E-7](#)  
**time resolution** [C-83](#)  
**time scale**  
 for the compilation-unit scope [C-82](#)  
 overrideing the 'timescale compiler directive  
 f4rom the vcs command line [C-82](#)  
 specifying on the vcs command line [C-81](#)  
**time scale for time units and time precision** [E-9](#)  
**time\_pli** [6-17, 6-19](#)  
**time\_solver**  
 argument to the proprt -view option [6-82](#)  
**timebase setup variable** [A-7](#)  
**timebase variable** [C-83](#)  
**\$timeformat** [E-36](#)  
**-timescale** [B-18, C-81](#)  
**'timescale** [E-9](#)  
**timescale**  
 OpenVera testbench [C-28](#)  
 overriding [B-18](#)  
 specifying with **-timescale** [B-18](#)  
**timing check system tasks**  
 checking timestamp and timecheck  
 conditions [C-53](#)  
 disabling [C-50](#)  
 in specific module instances [11-43](#)  
 disabling delayed versions of signals [C-53](#)  
 disabling display of timing violations [C-50](#)  
 disabling toggling the notifier register [C-50](#)  
 enabling  
 in specific module instances [11-43](#)  
 negative values enabling [C-53](#)  
**timing check system tasks, disabling** [C-50](#)  
**timing checks**  
 disabling [D-24](#)  
 disabling for an instance [11-43](#)  
 suppressing the toggling of notifier registers  
[D-24](#)  
**timing violations**  
 disabling [C-50](#)  
 disabling the display of [D-24](#)  
**timming checks**

disabling the display of timing violations [D-24](#)  
**Timopt**  
 the timing optimizer [11-44–11-48](#)  
**+timopt** [11-44](#)  
**TMPDIR** [A-31](#)  
**TO\_HSTRING** [23-11](#)  
**TO\_OSTRING** [23-11](#)  
**TO\_STRING** [23-10](#)  
**toggle\_start\_at\_set\_region** [24-11](#)  
**toggle\_stop\_at\_toggle\_report** [24-12](#)  
**tokens argument to -ntb\_opts** [C-29](#)  
**tokens.v file** [19-24](#), [25-38](#)  
**-top** [4-64](#)  
**top-level Verilog Module** [13-14](#)  
**transport delays** [C-47](#)  
**+transport\_int\_delays** [11-26](#), [11-29](#), [11-31](#),  
[C-47](#)  
**+transport\_path\_delays** [11-26](#), [11-29](#), [11-31](#),  
[C-47](#)  
**.triggered property** [14-114](#)  
[D-41](#)  
**+typdelays** [C-41](#), [C-45](#), [D-32](#)  
**Type Conversion** [23-95](#)

**U**

**U**  
 direct access for C/C++ functions  
 formal parameter type [22-67](#)  
**-u** [C-115](#)  
**U\***  
 direct access for C/C++ functions  
 formal parameter type [22-67](#)  
**UB\***  
 direct access for C/C++ functions  
 formal parameter type [22-67](#)  
**UCLI**  
 enabling in encrypted files [25-38](#)  
**-ucli** [2-21](#)  
**UCLI Commands for X-Propagation Control Tasks** [10-45](#)  
**UDPs**  
 sequential UDPs  
 changing output evaluation to the active region [C-120](#)  
 changing output evaluation to the NBA time slot [3-5](#)  
**unaccelerated**  
 definitions and declarations [3-34–3-35](#)  
 structural instance declarations [3-34](#)  
**unary logical operators** [23-14](#)  
**'unconnected\_drive** [E-11](#)  
**'undef** [E-6](#)  
**\$ungetc** [E-34](#)  
**unified profiler**  
**ALL**  
 argument to the profrpt -view option [6-13](#)  
**caller-callee views** [6-31](#)  
**dynamic\_mem**  
 argument to the profrpt -view option [6-13](#)  
**dynamic\_mem+stack**  
 argument to the profrpt -view option [6-13](#)  
**mem\_all**  
 argument to the profrpt -view option [6-13](#)  
**mem\_callercallee**  
 argument to the profrpt -view option [6-13](#)  
**mem\_const**  
 argument to the profrpt -view option [6-12](#)  
**mem\_inst**  
 argument to the profrpt -view option [6-12](#)  
**mem\_mod**  
 argument to the profrpt -view option [6-12](#)  
**mem\_summary**  
 argument to the profrpt -view option [6-12](#)  
**time\_all**  
 argument to the profrpt -view option [6-12](#)  
**time\_callercallee**  
 argument to the profrpt -view option [6-12](#)  
**time\_const**  
 argument to the profrpt -view option [6-12](#)  
**time\_inst**  
 argument to the profrpt -view option [6-11](#)  
**time\_mod**  
 argument to the profrpt -view option [6-11](#)  
**time\_solver**  
 argument to the profrpt -view option [6-12](#)  
**time\_summary**

argument to the profrpt -view option 6-11  
 unified simulation profiler 6-1  
 uninstantiated package 23-21  
 -uniq\_prior maxfail=integer D-23  
 uniq\_prior\_final compiler switch 14-116  
 unique constraints 16-80  
 unique keyword 14-116  
 UNIQUE violations  
     limiting the number of D-23  
 uniquifying identifier codes in VCD files 8-19  
 -unit\_timescale C-82  
 unpacked E-26  
 unpacked dimensions C-113, E-26  
 unused 27-5  
 uppercase  
     changing Verilog identifiers to C-115  
 use setup variable A-16  
 use\_sigprop B-16, C-29  
 use\_sigprop argument to -ntb\_opts B-16, C-29  
 'uselib E-9  
 user-defined plusarg enabling D-41  
 utility, vcsplit 8-45

## V

-V 2-19, 2-27, C-65, D-27  
 -v 2-23, C-5, C-70  
 vacuous success message enabling D-18  
 \$value\$plusargs 5-46  
 +vc 22-127, C-59  
 VC Formal 27-2  
 VC Formal Assert 27-7  
 VC Formal Results 27-3  
 vc\_2stVectorRef() 22-98  
 vc\_4stVectorRef() 22-96  
 vc\_argInfo() 22-120  
 vc\_arraySize() 22-86  
 vc\_FillWithScalar() 22-117  
 vc\_get2stMemoryVector() 22-113  
 vc\_get2stVector() 22-102  
 vc\_get4stMemoryVector() 22-111  
 vc\_get4stVector() 22-100  
 vc\_getInteger() 22-96  
 vc\_getMemoryInteger() 22-108  
 vc\_getMemoryScalar() 22-107  
 vc\_getPointer() 22-94  
 vc\_getReal() 22-91  
 vc\_getScalar() 22-86  
 vc\_handle  
     definition 22-76  
     using 22-77–22-78  
 vc\_hdrs.h file 22-74–22-75  
     in DirectC C-60  
 vc\_Index() 22-121  
 vc\_Index2() 22-122  
 vc\_Index3() 22-122  
 vc\_is2state() 22-83  
 vc\_is2stVector() 22-85  
 vc\_is4state() 22-82  
 vc\_is4stVector() 22-84  
 vc\_isMemory() 22-81  
 vc\_isScalar() 22-79  
 vc\_isVector() 22-80, 22-123  
 vc\_mdaSize() 22-122  
 vc\_MemoryElemRef() 22-105  
 vc\_MemoryRef() 22-102  
 vc\_MemoryString() 22-115  
 vc\_MemoryStringF() 22-116  
 vc\_put2stMemoryVector() 22-113  
 vc\_put2stVector() 22-102  
 vc\_put4stMemoryVector() 22-113  
 vc\_put4stVector() 22-100  
 vc\_putInteger() 22-96  
 vc\_putMemoryInteger() 22-110  
 vc\_putMemoryScalar() 22-108  
 vc\_putMemoryValue() 22-114  
 vc\_putMemoryValueF() 22-114  
 vc\_putPointer() 22-94  
 vc\_putReal() 22-91  
 vc\_putScalar() 22-87  
 vc\_putValue() 22-91  
 vc\_putValueF() 22-92

**VC\_STATIC\_HOME** 27-3  
**vc\_StringToVector()** 22-95  
**vc\_toChar()** 22-87  
**vc\_toInteger()** 22-87  
**vc\_toString()** 22-89  
**vc\_toStringF()** 22-90  
**vc\_VectorToString()** 22-96  
**vc\_width()** 22-86  
**vcat utility** 8-31  
**VCD file**  
 specifying on the vcs command line C-113  
**VCD files**  
 checkpoint  
     recording current values E-14  
     start recording current values E-14  
     stop recording current values E-14  
 emptying or flushing the buffer E-14  
 enabling VCD dumping for memories and  
     multi-dimensional arrays D-30  
 flushing the latest data to all open VCD files  
     E-16  
 flushing the latest data to the VCD file E-15  
 for LSI certification E-16  
**grw.dump file** E-16  
 including strength levels E-16  
 increasing the frequency of flushing C-61  
 increasing the frequency of VCD file dumping  
     D-30, D-33  
**LSI certification**  
     flushing the buffer E-19  
     recording all port values E-18  
     resume recording E-18  
     specifying the file size E-19  
     suspending E-18  
 recording in another VCD file E-15  
 specifying the time to turn on VCD dumping  
     D-30  
 specifying a limit to the VCD file size E-15  
 specifying the filename E-14  
 specifying the name of the VCD file D-30  
 specifying the nets and variables recorded in  
     the file E-15  
 specifying the time to turn off VCD dumping  
     D-30

**VCD+** 8-2  
**Advantages** 8-2  
**System Tasks**  
     \$vcplusdeltacycleoff 8-11  
     \$vcplusdeltacycleon 8-11  
     \$vcplusmemoff 8-8  
     \$vcplusmemon 8-8  
     \$vcplusmemorydump 8-8  
**vcdiff utility** 8-21  
     syntax 7-32, 8-21  
**\$vcplusautoflushoff** E-20  
**\$vcplusautoflushon** E-21  
**\$vcplusclose** E-21  
**\$vcplusdeltacycleon** E-21  
**\$vcplusdevent** E-22  
**\$vcplusfile** E-22  
**\$vcplusflush** E-23  
**\$vcplusglitchon** E-23  
**\$vcplusmemoff** E-25  
**\$vcplusmemon** E-23  
**\$vcplusmemorydump** E-25  
**\$vcplusoff** E-25  
**\$vcpluson** E-26  
**vcpost utility** 8-18  
     syntax 8-20  
**vcf** 27-2, 27-3  
**VCS**  
     predefined text macro E-5  
**VCS\_CC** A-31  
**VCS\_COM** A-31  
**VCS\_HOME** C-73  
**VCS\_LIC\_EXPIRE\_WARNING** A-32  
**VCS\_LOG** A-32  
     'vcs\_mipdexpand E-7  
**VCS\_NO\_RT\_STACK\_TRACE** A-32  
**VCS\_PRINT\_INITREG\_INITIALIZATION**  
 environment variable 4-31  
**VCS\_SWIFT\_NOTES** A-32, A-33  
**VCS\_TARGET\_ARCH** A-32  
     +vcs+dumpoff+t+ht D-30  
     +vcs+dumpfile+filename D-30  
     +vcs+dumpon+t+ht D-30

+vcs+finish 5-52, D-25  
 +vcs+flush+all D-33  
 +vcs+flush+dump D-30, D-33  
 +vcs+flush+fopen D-33  
 +vcs+flush+log D-33  
 +vcs+ignorestop D-40  
 +vcs+initreg+config 4-26, C-35, D-38  
 +vcs+learn+pli ??–22-33, D-40  
 +vcs+lic+vcsi D-34  
 +vcs+lic+wait D-34  
 +vcs+mipd+noalias D-43  
 +vcs+mipdexpand E-7  
 +vcs+nostdout D-27  
 +vcs+stop 5-52, D-25  
 +vcs+vcdblusion C-113  
 vcsfind 1-21  
 +vcsi+lic+vcs D-34  
 vcsplit utility 8-45  
 \$vcdblusion C-113  
 vec32  
     storing four state Verilog data 22-61  
 vec32\*  
     direct access for C/C++ functions  
         formal parameter type 22-67  
 vera\_portname argument to -ntb\_opts B-17, C-29  
 Vera, exporting tasks 13-10  
 verbose mode - displaying  
     compile-time and runtime numbers D-27  
     copyright information D-27  
     version and extended summary information D-27  
 -verdi 27-5, 35-18  
 verdi -cov 27-6  
 -verdi\_opts 35-22  
 Verilog 4-57  
     different versions  
         filename extension C-30, C-31  
     different versions'include  
         filename extension C-31  
 Verilog 1995  
     specifying the filename extension C-31  
 Verilog 2001  
     specifying the filename extension C-31  
 Verilog identifiers  
     changing to uppercase C-115  
 Verilog library  
     resolving module instances E-9  
 Verilog library directories  
     displaying a message upon instance resolution C-8  
     file name extensions  
         specifying C-7  
     specifying C-6  
     specifying the search order 2-13, C-7  
         rescan 2-13, C-7  
 Verilog library files  
     specifying C-5  
 Verilog model, example 13-13  
 Verilog module 13-14  
 Verilog module description 13-21  
 Verilog parameters  
     overriding C-80, C-89  
     overriding with the -gfile elaboration option C-79, C-87  
 +verilog1995ext 2-13, B-26, C-31  
 +verilog2001ext 2-13, B-26, C-31  
 VerilogAMS  
     default discrete discipline C-78  
 version number  
     returning C-72  
 VHDL  
     block statements 20-6  
 VHDL 2008  
     bit string literal 23-7  
     TO\_STRING (x) 23-10  
 VHDL generics  
     overriding C-92  
         from a file C-92  
     overriding with the -gfile elaboration option C-79, C-87  
 VHDL procedures, importing 13-8  
 VHDL two state objects in Xprop 10-48  
 VHDL-93 2-5, B-3  
 vhdl analyzer B-2

virtual interface  
     self instance [14-138](#)  
 VITAL models  
     error messages [11-80](#)  
     ignoring timing [D-31](#)  
 VITAL netlist [11-82](#)  
     negative constraints calculation [11-86](#)  
 VMM [C-28](#)  
 void  
     C/C++ function return type [22-55](#)  
 void\*  
     direct access for C/C++ functions  
         formal parameter type [22-67](#)  
 void\*\*  
     direct access for C/C++ functions  
         formal parameter type [22-67](#)  
 VPD file  
     specifying on the vcs command line [C-113](#)  
 VPD files [E-20](#)  
     buffer for  
         specifying the size of [D-27](#)  
     disable recording values for memories and  
         MDAs [E-25](#)  
     disabling file compression [D-30](#)  
     disabling recording in transition times an  
         values defined under 'celldefine' [C-70](#)  
     disabling recording in transition times an  
         values defined under 'celldefine or in a  
             library' [C-71](#)  
     enabling recording in transition times an  
         values defined under 'celldefine' [C-70](#)  
     enabling VPD file locking [D-30](#)  
     ignoring \$vcplusxx system tasks [D-29](#)  
     marking as completed and closing [E-21](#)  
     record a unique event for a signal [E-22](#)  
     recording changes on the drivers of resolved  
         nets [D-29](#)  
     recording delta cycle information [E-21](#)  
     recording only ports and their direction [D-29](#)  
     recording ports and their direction [D-29](#)  
     recording signals but not ports [D-29](#)  
     recording values for memories and MDAs  
         [E-23](#)  
     records a snapshot of memories and MDAs  
         [E-25](#)  
     specifying the next VPD file [E-22](#)  
     specifying the size of [D-28](#)  
     start recording [E-26](#)  
     stop recording [E-25](#)  
     turning off automatic flushing [E-20](#)  
     turning on automatic flushing [E-21](#)  
     turning on zero delay glitches [E-23](#)  
     write simulation results to the VPD file [E-23](#)  
     -vpd\_file [5-7](#)  
     -vpd\_fileswitchsize [5-8](#)  
 VPI [16-53](#)  
     specifying the registration routine in a shared  
         library [C-59](#)  
     SV and RT assertions  
         browse, enable, and disable [C-59](#)  
 +vpi [C-58](#)  
 VPI PLI access routines  
     enabling [C-58](#)  
 vpiSeqBeginTime [C-22](#)  
 vpiSeqFail [C-22](#)  
 -Vt [C-66](#)  
 -vts\_ignore\_env=ENV1,ENV2,... [C-86](#)  
 vunit [19-56, 20-6](#)

## W

WAIT statement [A-14](#)  
 +warn [3-55, 4-111, B-27, C-66](#)  
 \$warning [E-11](#)  
 WARNING message [A-11, A-14](#)  
 warning messages  
     disabling [B-27, C-66](#)  
     sover array size warning [D-7](#)  
 \$width [E-40](#)  
 with expression in streaming [14-57](#)  
 wn ACC capability [22-12](#)  
 -work  
     vlogan option [B-23](#)  
 WORK library [2-5, 2-12, B-3](#)  
 \$write [E-31](#)  
 \$writememb [E-35](#)  
 \$writememh [E-35](#)

## X

XIndex Element Merging [10-28](#)  
-xlm [11-6](#), [C-118](#), [D-41](#)  
-xlm alt\_retain [11-7](#)  
-xlm env\_expand [A-35](#)  
-xlm gd\_pulseprop [11-8](#)  
-xlm gd\_pulsegarn [11-9](#)  
-xlm sv\_string\_slice [14-146](#)  
-xlm uniq\_prior\_final compile switch [14-116](#)  
-xlm\_uniq\_prior\_observed [14-124](#)  
-Xman [25-38](#)  
-Xmangle [25-38](#)

## XMR

[16-48](#)  
-Xnoman [25-39](#)  
-Xnomangle [25-39](#)  
-Xova [19-39](#)  
-Xova [C-22](#)  
Xprop configuration file  
  merge\_mode [10-36](#)  
  select\_method [10-36](#)

## Y

-y [2-24](#), [C-6](#), [C-70](#)

[Feedback](#)

IN-32