# VCS®/VCSi™ User Guide

M-2017.03, March 2017

# Copyright Notice and Proprietary Information

© 2017 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®/VCSi™ and VCS® MX/VCS® MXi™ 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/VCS MX 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 http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

## 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.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

# Contents

1. Getting Started

Simulator Support with Technologies . . . . . . . . . . . . . . . . . . . . . . 1-2

Simulation Preemption Support. . . . . . . . . . . . . . . . . . . . . . . . . . 1-4

Setting Up the Simulator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1-4

Verifying Your System Configuration . . . . . . . . . . . . . . . . . . . 1-4

Obtaining a License . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1-5

Setting Up Your Environment. . . . . . . . . . . . . . . . . . . . . . . . 1-7

Setting Up Your C Compiler. . . . . . . . . . . . . . . . . . . . . . . . . 1-8

SeeisUsing the Simulator . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1-8

Basic Usage Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1-9

Default Time Unit and Time Precision . . . . . . . . . . . . . . . . . . . . . 1-9

Searching Identifiers in the Design Using UNIX Commands . . . 1-10

Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1-12

2. VCS Flow

Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-1

Using vcs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-2
Commonly Used Options . . . . . . . . . . . . . . . . . . . . . . . . . 2-3

Simulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-8
Interactive Mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-8
Batch Mode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2-9
Commonly Used Runtime Options. . . . . . . . . . . . . . . . . . . . . 2-9

3. Modeling Your Design

Avoiding Race Conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-2
Using and Setting a Value at the Same Time . . . . . . . . . . . . 3-3
Setting a Value Twice at the Same Time . . . . . . . . . . . . . . . . 3-3
Flip-Flop Race Condition . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-4
Continuous Assignment Evaluation . . . . . . . . . . . . . . . . . . . . 3-5
Counting Events. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-7
Time Zero Race Conditions . . . . . . . . . . . . . . . . . . . . . . . . . 3-7

Race Detection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-8
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 . . . . . . . . . . . . . . . . . . . . . . . 3-15
Debugging Simulation Mismatches . . . . . . . . . . . . . . . . . 3-17
The Static Race Detection Tool . . . . . . . . . . . . . . . . . . . . . . 3-20

Race Detection Tool to Identify Race between Clock and Data . 3-22
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-22
Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-23

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-25
Optimizing Testbenches for Debugging. . . . . . . . . . . . . . . . . . . . 3-25
Conditional Compilation. . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-26
Enabling Debugging Features at Runtime. . . . . . . . . . . . . . . 3-28
Combining the Techniques . . . . . . . . . . . . . . . . . . . . . . . . . 3-31
Creating Models That Simulate Faster . . . . . . . . . . . . . . . . . . . 3-32
Unaccelerated Data Types, Primitives, and Statements . . . . 3-33
Inferring Faster Simulating Sequential Devices. . . . . . . . . . . 3-34
Modeling Faster always Blocks . . . . . . . . . . . . . . . . . . . . . . 3-38
Using Verilog 2001 Constructs. . . . . . . . . . . . . . . . . . . . . . . 3-39
Case Statement Behavior . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-41
Precedence in Text Macro Definitions . . . . . . . . . . . . . . . . . . . . 3-42
Memory Size Limits in the Simulator . . . . . . . . . . . . . . . . . . . . . 3-42
Using Sparse Memory Models . . . . . . . . . . . . . . . . . . . . . . . . . 3-44
Obtaining Scope Information. . . . . . . . . . . . . . . . . . . . . . . . . . 3-46
Scope Format Specifications . . . . . . . . . . . . . . . . . . . . . . . . 3-46
Returning Information About the Scope. . . . . . . . . . . . . . . . . 3-49
Avoiding Circular Dependency . . . . . . . . . . . . . . . . . . . . . . . . . 3-52
Designing With $lsi_dumpports for Simulation and Test . . . . . . . 3-53
Dealing With Unassigned Nets . . . . . . . . . . . . . . . . . . . . . . 3-53
Code Values at Time 0. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3-55
Cross Module Forces and No Instance Instantiation . . . . . . . 3-55
Signal Value/Strength Codes . . . . . . . . . . . . . . . . . . . . . . . . 3-57

4. Compiling the Design


Compiling or Elaborating the Design in the Debug Mode . . . . . . 4-2

Compiling or Elaborating the Design in the Optimized Mode . . . 4-3

Optimizing Simulation Performance for Desired Debug Visibility With the -debug_access Option . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-4

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-5

Using -debug_access to Report Global Debug Capability Diagnostics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-7

Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-8

Specifying Design Regions for -debug_access Capabilities . 4-11

Enabling Additional Debug Capabilities . . . . . . . . . . . . . . . . . 4-16

Reduction in the Objects Being Dumped . . . . . . . . . . . . . . . 4-18

Testbench Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-18

Differences Between -debug_pp and -debug_access+pp . . . 4-19

Using -debug_access With Tab Files . . . . . . . . . . . . . . . . . . 4-20

Using -debug_access With -ucli/-gui at Compile Time . . . . . 4-20

Unused Tab File Calls . . . . . . . . . . . . . . . . . . . . . . . . . . 4-21

Including Tab Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-21

Interaction With Other Debug Options . . . . . . . . . . . . . . . . . 4-21

Dynamic Loading of DPI Libraries at Runtime . . . . . . . . . . . . . . 4-22

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-22

Dynamic Loading of PLI Libraries at Runtime . . . . . . . . . . . . . . . 4-24

Key Compilation or Elaboration Features . . . . . . . . . . . . . . . . . . 4-25

Initializing Verilog Variables, Registers, and Memories . . . . . 4-25

Initializing Verilog Variables, Registers, and Memories in an entire Design. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-26

Initializing Verilog Variables, Registers, and Memories in Selective Parts of a Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-27
Selections for Initialization of Registers or Memories. . . . 4-31
Reporting the Initialized Values of Variables, Registers, and Memories . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-32
Overriding Parameters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-32
Checking for x and z Values In Conditional Expressions. . . . 4-33
Enabling the Checking . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-35
Filtering Out False Negatives. . . . . . . . . . . . . . . . . . . . . . 4-35
Verilog Configurations and Libmaps . . . . . . . . . . . . . . . . . . 4-37
Library Mapping Files. . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-38
Configurations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-39
Hierarchical Configurations . . . . . . . . . . . . . . . . . . . . . . . 4-43
The -top Compile-Time Option . . . . . . . . . . . . . . . . . . . . . 4-44
Limitations of Configurations . . . . . . . . . . . . . . . . . . . . . . 4-44
Lint Warning Message for Missing ‘endcelldefine . . . . . . . . . 4-45
Error/Warning/Lint Message Control . . . . . . . . . . . . . . . . . . . 4-49
Controlling Error/Warning/Lint Messages Using Compile-Time Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-49
Controlling Error/Warning/Lint Messages Using a Configuration File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-66
Extracting the Files Used in Compilation . . . . . . . . . . . . . . . . 4-74
XML File Format. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4-75

5. Simulating the Design

Using DVE. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-2
Using UCLI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-3
ucli2Proc Command. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-5

Options for Debugging Using DVE and UCLI . . . . . . . . . . . . . . . . 5-6

Reporting Forces/Injections in a Simulation . . . . . . . . . . . . . . . . 5-9

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 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-17

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-19

Key Runtime Features. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-21

Passing Values from the Runtime Command Line . . . . . . . . . 5-22

Saving and Restarting the Simulation . . . . . . . . . . . . . . . . . . . 5-23

Save and Restart Example. . . . . . . . . . . . . . . . . . . . . . . . 5-24

Save and Restart File I/O . . . . . . . . . . . . . . . . . . . . . . . . 5-25

Save and Restart With Runtime Options . . . . . . . . . . . . . 5-26

Specifying Long Time Before Stopping the Simulation . . . . . 5-27

Preventing Time 0 Race Conditions . . . . . . . . . . . . . . . . . . . 5-29

Resolving RTL Simulation Races in Verilog Designs. . . . . . . 5-30

Recommended Approach to Resolve Race Conditions . . 5-30

Supporting Simulation Executable to Return Non-Zero Value on Error Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-35

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-36

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-37

Supporting Memory Load and Dump Task Verbosity. . . . . . . 5-37

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5-38

6. The Unified Simulation Profiler


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. . . . . 6-18

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. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-65

Hypertext Links to the Source Files . . . . . . . . . . . . . . . . . . . . 6-67

Single Text Format Report . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-70

Stack Trace Report Example . . . . . . . . . . . . . . . . . . . . . . . . . 6-71
SystemC Views . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-73
Constraint Profiling Integrated in the Unified Profiler . . . . . . . . . . 6-80
Changes to the Use Model for Constraint Profiling . . . . . . . . 6-81
The Time Constraint Solver View. . . . . . . . . . . . . . . . . . . . . . 6-82
The Memory Constraint Solver View . . . . . . . . . . . . . . . . . . . 6-91
Performance/Memory Profiling for Coverage Covergroups. . . . . 6-95
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-96
Example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-96
HTML Profiler Reports . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-96
Default Summary View. . . . . . . . . . . . . . . . . . . . . . . . . . 6-97
Time/Memory Summary View . . . . . . . . . . . . . . . . . . . . . 6-97
Time/Memory Module View . . . . . . . . . . . . . . . . . . . . . . . 6-98
Time/Memory Construct View . . . . . . . . . . . . . . . . . . . . . 6-99
Time/Memory Covergroup View. . . . . . . . . . . . . . . . . . . . 6-100
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-101
Reporting Debug Capabilities for Each Module. . . . . . . . . . . . . 6-102
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-103
HTML Reports . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-104
Text Reports. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-106
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-107
Supporting Line-Based CPU Time Profiler . . . . . . . . . . . . . . . . 6-107
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-108
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-109
Supporting Simulation Time Slice Based Profiler . . . . . . . . . . . . 6-110

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-110
Diagnostics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-113
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-114

Isolating the Cost of Garbage Collection . . . . . . . . . . . . . . . . . . 6-114
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6-114

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

7. Diagnostics

Using Diagnostics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-2
Using `-diag` Option . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-2
Using Smartlog . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-3

Compile-time Diagnostics . . . . . . . . . . . . . . . . . . . . . . . . . 7-5
Libconfig Diagnostics . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-5
Timescale Diagnostics . . . . . . . . . . . . . . . . . . . . . . . . . . 7-6
Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-6

Runtime Diagnostics . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-10
Diagnostics for VPI PLI Applications . . . . . . . . . . . . . . . . . . 7-10
Keeping the UCLI/DVE Prompt Active After a Runtime Error 7-14
UCLI Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-14
DVE Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-16
UCLI Usage Example . . . . . . . . . . . . . . . . . . . . . . . . . 7-18
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-20

Diagnosing Quickthread Issues . . . . . . . . . . . . . . . . . . . . . . . 7-20
Diagnosing Quickthread Issues in DPI. . . . . . . . . . . . . . . . 7-21
Diagnosing Quickthread Issues in SystemC . . . . . . . . . . 7-22
Post-Processing Diagnostics. . . . . . . . . . . . . . . . . . . . . . . . . . . 7-26
Using the vpdutil Utility to Generate Statistics . . . . . . . . . . . . 7-26
The vpdutil Utility Syntax . . . . . . . . . . . . . . . . . . . . . . . . 7-26
Options. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7-27

8. VCS Multicore Technology Application Level Parallelism

Enabling Multicore Technology Application Level Parallelism. . . 8-2
Multicore SAIF File Dumping . . . . . . . . . . . . . . . . . . . . . . . . 8-4
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8-5

9. VPD, VCD, and EVCD Utilities

Advantages of VPD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-2
Dumping a VPD File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-3
Using System Tasks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-3
Enable and Disable Dumping. . . . . . . . . . . . . . . . . . . . . . 9-4
Override the VPD Filename . . . . . . . . . . . . . . . . . . . . . . . 9-7
Dump Multi-Dimensional Arrays and Memories. . . . . . . . 9-8
Using $vcdplusmemorydump System Task . . . . . . . . . . . 9-11
Capture Delta Cycle Information . . . . . . . . . . . . . . . . . . . 9-11
Dumping an EVCD File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-12
Using $dumpports System Task . . . . . . . . . . . . . . . . . . . . . . 9-13
Dumping EVCD File for Mixed Designs Using UCLI dump Command 9-13

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-14
Use Model for Dumping CCN Driver Through INOUT . . . 9-15
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-16

Post-processing Utilities . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-19
The vcdpost Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-20
Scalarizing the Vector Signals . . . . . . . . . . . . . . . . . . . . . . 9-20
Uniquifying the Identifier Codes . . . . . . . . . . . . . . . . . . . . 9-21
The vcdpost Utility Syntax . . . . . . . . . . . . . . . . . . . . . . . . 9-22
The vcdiff Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-23
Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-23
The vcdiff Utility Output Example . . . . . . . . . . . . . . . . . . . 9-31
The vcat Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-33
The vcat Utility Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-34
Generating Source Files From VCD Files . . . . . . . . . . . . . 9-38
Writing the Configuration File . . . . . . . . . . . . . . . . . . . . . . 9-39
The vcsplit Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-43
The vcsplit Utility Syntax . . . . . . . . . . . . . . . . . . . . . . . . . 9-44
The vcd2vpd Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-47
Options for Specifying EVCD Options . . . . . . . . . . . . . . . 9-49
The vpd2vcd Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-49
The Command File Syntax . . . . . . . . . . . . . . . . . . . . . . . . 9-55
The vpdmerge Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-59
The vpdutil Utility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9-62

10. Performance Tuning

Compile-time Performance . . . . . . . . . . . . . . . . . . . . . . . . . . 10-2
Incremental Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . 10-3

Compile Once and Run Many Times . . . . . . . . . . . . . . . . . . . 10-4
Parallel Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-4

Runtime Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-5
Using Radiant Technology . . . . . . . . . . . . . . . . . . . . . . . . . . 10-5
Compiling With Radiant Technology. . . . . . . . . . . . . . . . . . 10-5
Applying Radiant Technology to Parts of the Design . . . . 10-6
Improving Performance When Using PLIs. . . . . . . . . . . . . . . 10-15
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-16
Enabling TAB File Capabilities in UCLI Using -debug_access 10-16
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-17
Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-17

Impact on Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-19

Obtaining VCS Consumption of CPU Resources . . . . . . . . . . . 10-20
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-20
Compile Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-20
Simulation Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10-21

11. Using X-Propagation

Introduction to X-Propagation . . . . . . . . . . . . . . . . . . . . . . . . . 11-2
Guidelines for Running X-Propagation Simulations. . . . . . . . 11-4

Using the X-Propagation Simulator . . . . . . . . . . . . . . . . . . . . . 11-6
Specifying X-Propagation Merge Mode . . . . . . . . . . . . . . . . . 11-9
Querying X-Propagation at Runtime . . . . . . . . . . . . . . . . . 11-13
X-Propagation Instrumentation Report. . . . . . . . . . . . . . . . 11-14
Automatic Hardware Inference of Flip-Flops Enabled by Default 11-15
X-Propagation Configuration File. . . . . . . . . . . . . . . . . . . . . . 11-17

X-Propagation Configuration File Syntax . . . . . . . . . . . . . . 11-18
Xprop Instrumentation Control . . . . . . . . . . . . . . . . . . . . . . . 11-22
Process Based X-Propagation Exclusion . . . . . . . . . . . . . . . 11-25
Support for XIndex Element Merging. . . . . . . . . . . . . . . . . . . 11-25
Index BSpace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-27
Addressing Models. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-28
Element Merging Methods . . . . . . . . . . . . . . . . . . . . . . . . 11-29
Disabling XIndex Merging for Read or Write Operations . 11-31
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-32
Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-33
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-37
Bounds Checking. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-37
Time Zero Initialization. . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-39
Handling Non-pure Functions Due to Static Lifetime . . . . . . . 11-39
Supporting UCLI Commands for X-Propagation Control Tasks 11-41
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-41
UCLI Command to Specify the Merge Mode . . . . . . . . . . 11-42
UCLI Command to Control Error Messages or Warning Messages 11-43
X-Propagation Code Examples . . . . . . . . . . . . . . . . . . . . . . . . 11-44
If Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-44
Verilog Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-44
Case Statement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-46
Verilog Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-46
Edge Sensitive Expression . . . . . . . . . . . . . . . . . . . . . . . . 11-47
Verilog Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-47
Latch . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-48
Verilog Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-48

Support for Active Drivers in X-Propagation . . . . . . . . . . . . . . . . 11-49
Combinational Logic. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-50
Latches. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-52
Flip-flops. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-54
Key points to Note . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-56

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11-57

12. Gate-Level Simulation

SDF Annotation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-2
Using the Unified SDF Feature . . . . . . . . . . . . . . . . . . . . . . 12-3
Using the $sdf_annotate System Task. . . . . . . . . . . . . . . . . . 12-4
Using the -xlrm Option for SDF Retain, Gate Pulse Propagation, and Gate Pulse Detection Warning . . . . . . . . . . . . . . . . . . . . . 12-5
Using the Optimistic Mode in SDF . . . . . . . . . . . . . . . . . 12-6
Using Gate Pulse Propagation . . . . . . . . . . . . . . . . . . . . . 12-8
Generating Warnings During Gate Pulses . . . . . . . . . . . . 12-9

Precompiling an SDF File . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-10
Creating the Precompiled Version of the SDF File. . . . . . 12-10

SDF Configuration File . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-12
Delay Objects and Constructs . . . . . . . . . . . . . . . . . . . . . 12-13
SDF Configuration File Commands . . . . . . . . . . . . . . . . . 12-14
The INTERCONNECT_MIPD Command. . . . . . . . . . . . . 12-14
The MTM Command . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-15
The SCALE Commands. . . . . . . . . . . . . . . . . . . . . . . . . 12-16
An SDF Example With Configuration File . . . . . . . . . . . 12-17

Delays and Timing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-20
Transport and Inertial Delays. . . . . . . . . . . . . . . . . . . . . . . 12-20

The Inertial Delay Implementation . . . . . . . . . . . . . . . . . . 12-22
Enabling Transport Delays . . . . . . . . . . . . . . . . . . . . . . . 12-23
Pulse Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-24
Pulse Control With Transport Delays . . . . . . . . . . . . . . . . . . . 12-26
Pulse Control With Inertial Delays . . . . . . . . . . . . . . . . . . 12-28
Specifying Pulse on Event or Detect Behavior . . . . . . . . . 12-32
Specifying the Delay Mode . . . . . . . . . . . . . . . . . . . . . . . . 12-37
Using the Configuration File to Disable Timing . . . . . . . . . . . . . . 12-39
Using the timopt Timing Optimizer . . . . . . . . . . . . . . . . . . . . . . 12-40
Editing the timopt.cfg File . . . . . . . . . . . . . . . . . . . . . . . . 12-42
Editing Potential Sequential Device Entries . . . . . . . . . . . 12-42
Editing Clock Signal Entries . . . . . . . . . . . . . . . . . . . . . . 12-43
Using Scan Simulation Optimizer . . . . . . . . . . . . . . . . . . . . . . 12-44
ScanOpt Configuration File Format . . . . . . . . . . . . . . . . . . . 12-45
ScanOpt Assumptions . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-46
Negative Timing Checks . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-47
The Need for Negative Value Timing Checks . . . . . . . . . . . . 12-48
The $setuphold Timing Check Extended Syntax . . . . . . . 12-53
Negative Timing Checks for Asynchronous Controls . . . . 12-56
The $recrem Timing Check Syntax . . . . . . . . . . . . . . . . . 12-57
Enabling Negative Timing Checks . . . . . . . . . . . . . . . . . . . . 12-59
Other Timing Checks Using the Delayed Signals . . . . . . . . . 12-60
Checking Conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12-64
Toggling the Notifier Register . . . . . . . . . . . . . . . . . . . . . . 12-65
SDF Back-Annotation to Negative Timing Checks . . . . . . . . . 12-66
How VCS Calculates Delays . . . . . . . . . . . . . . . . . . . . . . . . 12-67

13. Coverage

Code Coverage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13-2

Functional Coverage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13-3

Options For Coverage Metrics . . . . . . . . . . . . . . . . . . . . . . . . 13-3

14. Using OpenVera Native Testbench

Usage Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14-3

Example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14-3

Usage Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14-5

Using Template Generator . . . . . . . . . . . . . . . . . . . . . . 14-5

Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14-6

Key Features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14-18

Multiple Program Support . . . . . . . . . . . . . . . . . . . . . . . . 14-18

Configuration File Model . . . . . . . . . . . . . . . . . . . . . . 14-18

Configuration File . . . . . . . . . . . . . . . . . . . . . . . . . . . 14-19

Usage Model for Multiple Programs . . . . . . . . . . . . . . . . 14-20

NTB Options and the Configuration File. . . . . . . . . . . . . . 14-21

Class Dependency Source File Reordering. . . . . . . . . . . . . . 14-22

Circular Dependencies. . . . . . . . . . . . . . . . . . . . . . . . . 14-24

Dependency-based Ordering in Encrypted Files . . . . . . . 14-25

Using Encrypted Files . . . . . . . . . . . . . . . . . . . . . . . . . . 14-25

Functional Coverage . . . . . . . . . . . . . . . . . . . . . . . . . . . 14-26

Using Reference Verification Methodology . . . . . . . . . . . . . 14-26

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14-27

15. Using SystemVerilog


Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-2

Using UVM With VCS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-3

Update on UVM-1.2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-4

Natively Compiling and Elaborating UVM-1.1d . . . . . . . . . . . . 15-4

Natively Compiling and Elaborating UVM-1.2 . . . . . . . . . . . . . 15-4

Compiling the External UVM Library . . . . . . . . . . . . . . . . . . . 15-5

Using the -ntb_opts uvm Option. . . . . . . . . . . . . . . . . . . . 15-5

Explicitly Specifying UVM Files and Arguments. . . . . . . . 15-6

Accessing HDL Registers Through UVM Backdoor. . . . . . . . 15-6

Generating UVM Register Abstraction Layer Code . . . . . . . . 15-7

Recording UVM Transactions . . . . . . . . . . . . . . . . . . . . . . . 15-8

Debugging UVM Testbench Designs Using DVE. . . . . . . . . . 15-8

Recording UVM Phases. . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-9

UVM Template Generator . . . . . . . . . . . . . . . . . . . . . . . . . 15-10

Using Mixed VMM/UVM Libraries . . . . . . . . . . . . . . . . . . . . 15-11

Migrating from OVM to UVM . . . . . . . . . . . . . . . . . . . . . . . 15-13

Where to Find UVM Examples. . . . . . . . . . . . . . . . . . . . . . . 15-13

Where to Find UVM Documentation . . . . . . . . . . . . . . . . . . 15-14

UVM-1.1d Documentation . . . . . . . . . . . . . . . . . . . . . . . 15-14

UVM-VMM Interop Documentation . . . . . . . . . . . . . . . . . 15-14

Using VMM with VCS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-15

Using OVM with VCS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-15

Native Compilation and Elaboration of OVM 2.1.2 . . . . . . . . 15-15

Compiling the External OVM Library . . . . . . . . . . . . . . . . . . . 15-16

Using the -ntb_opts ovm Option. . . . . . . . . . . . . . . . . . . . 15-16

Explicitly Specifying OVM Files and Arguments. . . . . . . . 15-16
Recording OVM Transactions . . . . . . . . . . . . . . . . . . . . . . . 15-17
Debugging SystemVerilog Designs . . . . . . . . . . . . . . . . . . . . . . 15-18
Functional Coverage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-19
SystemVerilog Constructs . . . . . . . . . . . . . . . . . . . . . . . . . . 15-19
Extern Task and Function Calls through Virtual Interfaces . . 15-21
Modport Expressions in an Interface . . . . . . . . . . . . . . . . . . 15-24
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-25
Interface Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-26
Difference Between Extends and Implements . . . . . . . . . 15-29
Cast and Interface Class . . . . . . . . . . . . . . . . . . . . . . . . 15-31
Name Conflicts and Resolution . . . . . . . . . . . . . . . . . . . 15-32
Interface Class and Randomization . . . . . . . . . . . . . . . . 15-36
Package Exports . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-37
Severity System Tasks as Procedural Statements. . . . . . . . . 15-38
Width Casting Using Parameters. . . . . . . . . . . . . . . . . . . . . 15-40
The std::randomize() Function . . . . . . . . . . . . . . . . . . . . . . 15-42
SystemVerilog Bounded Queues . . . . . . . . . . . . . . . . . . . . 15-45
wait() Statement with a Static Class Member Variable. . . . . . 15-46
Support for Consistent Behavior of Class Static Properties. . 15-47
Parameters and Local Parameters in Classes. . . . . . . . . . . . 15-49
SystemVerilog Math Functions . . . . . . . . . . . . . . . . . . . . . 15-49
Streaming Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-50
Packing (Used on RHS) . . . . . . . . . . . . . . . . . . . . . . . . . 15-50
Unpacking (Used on LHS) . . . . . . . . . . . . . . . . . . . . . . . 15-51
Packing and Unpacking . . . . . . . . . . . . . . . . . . . . . . . . . 15-51

Propagation and force Statement . . . . . . . . . . . . . . . . . . . 15-52
Error Conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-52
Structures with Streaming Operators . . . . . . . . . . . . . . . . 15-52
Support for with Expression . . . . . . . . . . . . . . . . . . . . . . . 15-52
Constant Functions in Generate Blocks. . . . . . . . . . . . . . . . . 15-56
Support for Aggregate Methods in Constraints Using the "with" Construct . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-57
Debugging During Initialization SystemVerilog Static Functions and Tasks in Module Definitions . . . . . . . . . . . . . . . . . . . . . . . 15-58
Explicit External Constraint Blocks . . . . . . . . . . . . . . . . . . . 15-62
Generate Constructs in Program Blocks . . . . . . . . . . . . . . . . 15-65
Error Condition for Using a Genvar Variable Outside of its Generate Block. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-67
Randomizing Unpacked Structs. . . . . . . . . . . . . . . . . . . . . . 15-68
Using the Scope Randomize Method std::randomize() . . 15-68
Using the Class Randomize Method `randomize()` . 15-72
Disabling and Re-enabling Randomization . . . . . . . . . . . 15-74
Using In-Line Random Variable Control . . . . . . . . . . . . . . 15-78
Limitation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-82
Making wait fork Statements Compliant with the SV LRM . . . 15-83
Making disable fork Statements Compliant with the SV LRM 15-85
Using a Package in a SystemVerilog Module, Program, and Interface Header . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-87
Support for Overriding Parameter Values through Configuration 15-89
Example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-89
Precedence Override Rules. . . . . . . . . . . . . . . . . . . . . . . . . 15-90
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-90

Extensions to SystemVerilog . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-91
Unique/Priority Case/IF Final Semantic Enhancements . . . . 15-91
Using Unique/Priority Case/If with Always Block or Continuous Assign . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-92
Using Unique/Priority Inside a Function . . . . . . . . . . . . . . 15-96
System Tasks to Control Warning Messages . . . . . . . . . . 15-98
Controlling Runtime Warning Messages Generated Using Unique/ Priority If Constructs . . . . . . . . . . . . . . . . . . . . . . . . . 15-99
Single-Sized Packed Dimension Extension . . . . . . . . . . . . . . 15-101
Covariant Virtual Function Return Types . . . . . . . . . . . . . . . . 15-104
Self Instance of a Virtual Interface . . . . . . . . . . . . . . . . . . . . . 15-105
UVM Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15-107


16. Aspect Oriented Extensions


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


17. Using Constraints


Support for Array Slice in Unique Constraints . . . . . . . . . . . . . . . 17-2
Support for Object Handle Comparison in Constraint Guards. . . 17-4
Support for Pure Constraint Block . . . . . . . . . . . . . . . . . . . . . . . 17-8
Support for SystemVerilog Bit Vector Functions in Constraints. . 17-14
$countones Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-16

$onehot Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-17
$onehot0 Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-18
$countbits Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-20
$bits Function. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-21
Inconsistent Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-23
Constraint Debug . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-25
Partition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-26
Randomize Serial Number. . . . . . . . . . . . . . . . . . . . . . . . . 17-28
Solver Trace. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-28
Constraint Profiler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-33
Test Case Extraction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-34
Using multiple +ntb_solver_debug arguments . . . . . . . . . . . 17-36
Summary for the +ntb_solver_debug Option . . . . . . . . . . . . . 17-37
+ntb_solver_debug=serial . . . . . . . . . 17-37
+ntb_solver_debug=trace . . . . . . . . . 17-37
+ntb_solver_debug=profile . . . . . . . . 17-37
+ntb_solver_debug=extract . . . . . . . . 17-38
Support for Save and Restore Stimulus. . . . . . . . . . . . . . . . 17-38
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-39
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-39
Constraint Debug Using DVE . . . . . . . . . . . . . . . . . . . . . . . . 17-40
Constraint Guard Error Suppression. . . . . . . . . . . . . . . . . . . . . 17-42
Error Message Suppression Limitations . . . . . . . . . . . . . . . . 17-44
Flattening Nested Guard Expressions . . . . . . . . . . . . . . . 17-44
Pushing Guard Expressions into Foreach Loops . . . . . . . 17-45

Support for Array and Cross-Module References in std::randomize() 17-45
Error Conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-47
Support for Cross-Module References in Constraints . . . . . . . . . 17-48
XMR Function Calls in Constraints . . . . . . . . . . . . . . . . . . . 17-50
State Variable Index in Constraints . . . . . . . . . . . . . . . . . . . . . 17-50
Runtime Check for State Versus Random Variables . . . . . . . 17-51
Array Index . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-52
Using DPI Function Calls in Constraints . . . . . . . . . . . . . . . . . 17-52
Invoking Non-pure DPI Functions from Constraints. . . . . . . . 17-53
Using Foreach Loops Over Packed Dimensions in Constraints . 17-57
Memories with Packed Dimensions. . . . . . . . . . . . . . . . . . . 17-57
Single Packed Dimension . . . . . . . . . . . . . . . . . . . . . . 17-57
Multiple Packed Dimensions . . . . . . . . . . . . . . . . . . . . 17-58
MDAs with Packed Dimensions. . . . . . . . . . . . . . . . . . . . . . 17-58
Single Packed Dimension . . . . . . . . . . . . . . . . . . . . . . 17-58
Multiple Packed Dimensions . . . . . . . . . . . . . . . . . . . . 17-58
Just Packed Dimensions . . . . . . . . . . . . . . . . . . . . . . . 17-59
The foreach Iterative Constraint for Packed Arrays. . . . . . . . 17-59
Randomized Objects in a Structure. . . . . . . . . . . . . . . . . . . . . . 17-61
Support for Typecast in Constraints . . . . . . . . . . . . . . . . . . . . 17-63
Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-63
Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-63
Strings in Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-66
SystemVerilog LRM 1800™-2012 Update . . . . . . . . . . . . . . . . . 17-67

Using Soft Constraints in SystemVerilog . . . . . . . . . . . . . . . . 17-67
Using Soft Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . 17-68
Soft Constraint Prioritization. . . . . . . . . . . . . . . . . . . . . . . 17-69
Soft Constraints Defined in Classes Instantiated as rand Members in Another Class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-70
Soft Constraints Inheritance Between Classes . . . . . . . . 17-72
Soft Constraints in AOP Extensions to a Class . . . . . . . . 17-73
Soft Constraints in View Constraints Blocks. . . . . . . . . . . 17-76
Discarding Lower-Priority Soft Constraints. . . . . . . . . . . . 17-80
Unique Constraints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-82
Enhancement to the Randomization of Multidimensional Array Functionality. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-84
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-86
Supporting Random Array Index . . . . . . . . . . . . . . . . . . . . . . . 17-86
Limitation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-87
Supporting System Function Calls . . . . . . . . . . . . . . . . . . . . . 17-88
$size() System Function Call . . . . . . . . . . . . . . . . . . . . . . . 17-88
$clog2() System Function Call. . . . . . . . . . . . . . . . . . . . . . . 17-89
Usage Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17-89
Supporting Foreach Loop Iteration over Array Select . . . . . . . . . 17-90
18. Extensions for SystemVerilog Coverage
Support for Reference Arguments in get_coverage() and get_inst_coverage() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18-1
get_coverage() method . . . . . . . . . . . . . . . . . . . . . . . . . . . 18-2
get_inst_coverage() method . . . . . . . . . . . . . . . . . . . . . . . . 18-3

Functional Coverage Methodology Using the SystemVerilog C/C++ Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18-3
SystemVerilog Functional Coverage Flow . . . . . . . . . . . . . . . . 18-5
Covergroup Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18-6
SystemVerilog (Covergroup for C/C++): covg.sv . . . . . . . 18-7
C Testbench: test.c. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18-7
Approach #1: Passing Arguments by Reference . . . . . . . 18-8
Approach #2: Passing Arguments by Value . . . . . . . . . . . 18-8
Compile Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18-8
Runtime . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18-9
C/C++ Functional Coverage API Specification . . . . . . . . . . . 18-9

19. OpenVera-SystemVerilog Testbench Interoperability

Scope of Interoperability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-2
Importing OpenVera Types Into SystemVerilog. . . . . . . . . . . . . 19-3
Data Type Mapping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-6
Mailboxes and Semaphores . . . . . . . . . . . . . . . . . . . . . . . . 19-7
Events . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-9
Strings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-9
Enumerated Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-10
Integers and Bit-Vectors . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-12
Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-13
Structs and Unions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-15
Connecting to the Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-15
Mapping Modports to Virtual Ports. . . . . . . . . . . . . . . . . . . . 19-15
Virtual Modports . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-16
Importing Clocking Block Members Into a Modport . . . . . 19-16

Semantic Issues With Samples, Drives, and Expects . . . . . . 19-22

Notes to Remember . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-22

Blocking Functions in OpenVera . . . . . . . . . . . . . . . . . . . 19-22

Constraints and Randomization . . . . . . . . . . . . . . . . . . . 19-23

Functional Coverage . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-23

Usage Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-25

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19-25

20. Using SystemVerilog Assertions

Using SVAs in the HDL Design . . . . . . . . . . . . . . . . . . . . . . . . . 20-3

Using VCS Checker Library . . . . . . . . . . . . . . . . . . . . . . . . 20-3

Instantiating SVA Checkers in Verilog . . . . . . . . . . . . . . . 20-3

Binding SVA to a Design. . . . . . . . . . . . . . . . . . . . . . . . . . . 20-4

Inlining SVAs in the Verilog Design . . . . . . . . . . . . . . . . . . . 20-5

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-6

Number of SystemVerilog Assertions Supported in a Module 20-7

Controlling SystemVerilog Assertions . . . . . . . . . . . . . . . . . . . . 20-7

Compilation and Runtime Options . . . . . . . . . . . . . . . . . . . . . 20-8

Concatenating Assertion Options . . . . . . . . . . . . . . . . . . . . . 20-11

Assertion Monitoring System Tasks . . . . . . . . . . . . . . . . . . . . 20-11

Using Assertion Categories . . . . . . . . . . . . . . . . . . . . . . . . . 20-15

Using System Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-15

Using Attributes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-17

Starting and Stopping Assertions Using Assertion System Tasks 20-18

Viewing Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-24

Using a Report File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-24
Enhanced Reporting for SystemVerilog Assertions in Functions 20-25
Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-25
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-27
Name Conflict Resolution . . . . . . . . . . . . . . . . . . . . . . . . . 20-27
Checker and Generate Blocks . . . . . . . . . . . . . . . . . . . . . . . 20-27
Controlling Assertion Failure Messages . . . . . . . . . . . . . . . . . . 20-28
Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-28
Options for Controlling Default Assertion Failure Messages . 20-29
Options to Control Termination of Simulation. . . . . . . . . . . . . 20-30
Option to Enable Compilation of OVA Case Pragmas . . . . . . 20-33
Reporting Values of Variables in the Assertion Failure Messages 20-34
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-35
Reporting Messages When $uniq_prior_checkon/$uniq_prior_checkoff System Tasks are Called . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-36
Assertion and Unique/Priority Re-Trigger Feature . . . . . . . . . . . 20-38
Flushing Off the Assertion Re-Trigger Feature . . . . . . . . . . . 20-40
Enabling Lint Messages for Assertions . . . . . . . . . . . . . . . . . . . 20-41
Fail-Only Assertion Evaluation Mode . . . . . . . . . . . . . . . . . . . . 20-44
Key Points to Note . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-46
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-48
Using SystemVerilog Constructs Inside vunits . . . . . . . . . . . . . . 20-49
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-49
Calling $error Task When Else Block is Not Present. . . . . . . . . . 20-50

Disabling Default Assertion Success Dumping in -debug_pp Option 20-51

List of supported IEEE Std. 1800-2012 Compliant SVA Features 20-52

Support for $countbits System Function . . . . . . . . . . . . . . . . 20-55

Support for Real Data Type Variables . . . . . . . . . . . . . . . . . . 20-56

Support for $assertcontrol Assertion Control System Task . . 20-56

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-57

Enabling IEEE Std. 1800-2012 Compliant Features . . . . . . . 20-57

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20-57

SystemVerilog Assertions Limitations . . . . . . . . . . . . . . . . . . . . . 20-58

Debug Support for New Constructs . . . . . . . . . . . . . . . . . 20-58

Note on Cross Features . . . . . . . . . . . . . . . . . . . . . . . . . . 20-59

21. Using Property Specification Language

Including PSL in the Design . . . . . . . . . . . . . . . . . . . . . . . . . 21-1

Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21-2

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21-2

Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21-3

Using SVA Options, SVA System Tasks, and OV Classes . . . . . 21-4

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21-5

22. Using SystemC

23. C Language Interface

Using PLI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-2

Writing a PLI Application . . . . . . . . . . . . . . . . . . . . . . . . . 23-2

Functions in a PLI Application . . . . . . . . . . . . . . . . . . . . . . . . 23-4
Header Files for PLI Applications. . . . . . . . . . . . . . . . . . . . . . 23-5
PLI Table File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-6
Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-6
Using the PLI Table File . . . . . . . . . . . . . . . . . . . . . . . . 23-22
Enabling ACC Capabilities. . . . . . . . . . . . . . . . . . . . . . . . . . 23-23
Enabling ACC Capabilities Globally . . . . . . . . . . . . . . . . . 23-23
Using the Configuration File. . . . . . . . . . . . . . . . . . . . . . . 23-24
Selected ACC Capabilities . . . . . . . . . . . . . . . . . . . . . . . 23-28
PLI Access to Ports of Celldefine and Library Modules . . . . . 23-33
Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-34
Visualization in DVE . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-36
Using VPI Routines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-36
Support for VPI Callbacks . . . . . . . . . . . . . . . . . . . . . . . . . 23-37
Support for the vpi_register_systf Routine. . . . . . . . . . . . . . . 23-38
Integrating a VPI Application With VCS. . . . . . . . . . . . . . . . . 23-39
PLI Table File for VPI Routines . . . . . . . . . . . . . . . . . . . . . . 23-40
Virtual Interface Debug Support. . . . . . . . . . . . . . . . . . . . . . 23-41
Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-41
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-44
Unimplemented VPI Routines . . . . . . . . . . . . . . . . . . . . . . . 23-44
Modified VPI Features . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-46
Backwards Compatibility . . . . . . . . . . . . . . . . . . . . . . . . 23-50
Diagnostics for VPI PLI Applications . . . . . . . . . . . . . . . . . . . . 23-50
Using DirectC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-50
Using Direct C/C++ Function Calls . . . . . . . . . . . . . . . . . . . 23-52
Functioning of C/C++ Code in a Verilog Environment . . . 23-54

Declaring the C/C++ Function . . . . . . . . . . . . . . . . . . . . . . 23-55
Calling the C/C++ Function . . . . . . . . . . . . . . . . . . . . . . . 23-62
Storing Vector Values in Machine Memory . . . . . . . . . . . . 23-63
Converting Strings . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-66
Avoiding a Naming Problem . . . . . . . . . . . . . . . . . . . . . . . 23-69
Using Pass by Reference . . . . . . . . . . . . . . . . . . . . . . . . 23-69
Using Direct Access . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-70
Using the vc_hdrs.h File . . . . . . . . . . . . . . . . . . . . . . . . . 23-77
Access Routines for Multi-Dimensional Arrays . . . . . . . . 23-78
Using Abstract Access . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-80
Using vc_handle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23-80
Using Access Routines . . . . . . . . . . . . . . . . . . . . . . . . . . 23-82
Summary of Access Routines . . . . . . . . . . . . . . . . . . . . . 23-125
Enabling C/C++ Functions . . . . . . . . . . . . . . . . . . . . . . . . . 23-130
Mixing Direct And Abstract Access . . . . . . . . . . . . . . . . . 23-132
Specifying the DirectC.h File . . . . . . . . . . . . . . . . . . . . . 23-132
Extended BNF for External Function Declarations . . . . . . . . 23-133

24. SAIF Support

Using SAIF Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24-2
SAIF System Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24-2
The Flows to Generate a Backward SAIF File . . . . . . . . . . . . . 24-5
Generating an SDPD Backward SAIF File. . . . . . . . . . . . . . . 24-5
Generating a Non-SPDP Backward SAIF File . . . . . . . . . . . . 24-6
SAIF Support for Two-Dimensional Memories in v2k Designs . . 24-7
UCLI SAIF Dumping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24-7

Criteria for Choosing Signals for SAIF Dumping . . . . . . . . . . . . . 24-8

Improving Simulation Time by Reducing the Overhead due to SAIF File Dumping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24-9

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24-9

Example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24-10

Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24-11

25. Encrypting Source Files

IEEE Verilog Standard 1364-2005 Encryption. . . . . . . . . . . . . . . 25-2

The Protection Header File . . . . . . . . . . . . . . . . . . . . . . . . . 25-4

Unsupported Protection Pragma Expressions . . . . . . . . . 25-6

Other Options for IEEE Std. 1364-2005 Encryption Mode. . . 25-7

How Protection Envelopes Work . . . . . . . . . . . . . . . . . . . . . 25-9

The VCS Public Encryption Key . . . . . . . . . . . . . . . . . . . . . 25-10

Creating Interoperable Digital Envelopes Using VCS - Example 25-11

Discontinued -ipkey Option . . . . . . . . . . . . . . . . . . . . . . . . 25-16

128-bit Advanced Encryption Standard . . . . . . . . . . . . . . . . . . . 25-16

Compiler Directives for Source Protection. . . . . . . . . . . . . . . 25-17

Using Compiler Directives or Pragmas . . . . . . . . . . . . . . . . 25-17

Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25-18

Automatic Protection Options . . . . . . . . . . . . . . . . . . . . . . . 25-21

Using Automatic Protection Options . . . . . . . . . . . . . . . . . . 25-24

Protecting ‘include File Directive . . . . . . . . . . . . . . . . . . . . 25-32

+autoincludeprotect . . . . . . . . . . . . . . . . . . . . . . . . . . . 25-32

Enabling Debug Access to Ports and Instance Hierarchy . . . 25-33

+autobodyprotect . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25-33

Debugging Partially Encrypted Source Code. . . . . . . . . . . . . 25-33

Skipping Encrypted Source Code . . . . . . . . . . . . . . . . . . . . . . . 25-34

26. Integrating VC Formal With Coverage and Planner

Introduction to VC Formal . . . . . . . . . . . . . . . . . . . . . . . . . . . 26-2

VC Formal Coverage With Verdi Coverage and Planner . . . . . . 26-3

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26-3

Collecting VC Formal Results in the Coverage Database 26-3

Measuring VC Formal Assert Status in HVP . . . . . . . . . . 26-7

27. Integrating VCS With Certitude

Introduction to Certitude . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27-1

VCS and Certitude Integration . . . . . . . . . . . . . . . . . . . . . . . . 27-2

Loading Designs Automatically in Verdi with Native Certitude . . 27-4

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27-4

Points to Note . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27-5

Dumping and Comparing Waveforms in Verdi for SystemC Designs 27-6

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27-6

Point to Note . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27-8

Reducing Compilation Time in Native Certitude With VCS Partition Compile Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27-8

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27-8

Example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27-9

Limitation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27-11

28. Integrating VCS with Vera

Setting Up Vera and VCS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28-2

Using Vera with VCS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28-3

Usage Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28-4

29. Integrating VCS with Specman

Type Support . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29-2

Usage Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29-3

Setting Up The Environment . . . . . . . . . . . . . . . . . . . . . . . . 29-3

Specman e Code Accessing Verilog . . . . . . . . . . . . . . . . . . . 29-3

Using specrun and specview. . . . . . . . . . . . . . . . . . . . . . . . . . . 29-5

Adding Specman Objects To DVE. . . . . . . . . . . . . . . . . . . . . . . 29-8

Version Checker for Specman. . . . . . . . . . . . . . . . . . . . . . . . . . 29-10

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29-10

30. Integrating VCS with Denali

Setting Up Denali Environment for VCS . . . . . . . . . . . . . . . . . . . 30-1

Integrating Denali with VCS . . . . . . . . . . . . . . . . . . . . . . . . . . . 30-2

Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30-2

Use Model for Verilog Memory Models . . . . . . . . . . . . . . . . . 30-2

Execute Denali Commands at UCLI Prompt . . . . . . . . . . . . . 30-3

31. VCS and CustomSim Cosimulation

Integrating VCS with CustomSim . . . . . . . . . . . . . . . . . . . . . . . . 31-1

Setting up the Environment . . . . . . . . . . . . . . . . . . . . . . . . . 31-2
Licenses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31-3
Required UNIX Paths and Variable Settings . . . . . . . . . . 31-3
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31-4
Scheduling Analog-to-Digital Events in the NBA Region. . . . . . . 31-4
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31-4

32. Integrating VCS with Native Low Power (NLP)

33. Unified UVM Library for VCS and Verdi

Transaction/Message Recording in Verdi/DVE with VCS . . . . . . 33-4
Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33-4
Enabling FSDB or DVE Transaction Recording . . . . . . . . 33-4
Simulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33-6
Dumping Transactions or Messages in Verdi Flow . . . . . 33-6
Dumping Transactions or Messages in DVE Flow . . . . . . 33-7

34. Integrating VCS with Verdi

Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-2
Unified Compile Front End . . . . . . . . . . . . . . . . . . . . . . . . . 34-3
Generating Verdi KDB with Unified Compile Front End. . . . . 34-3
Reading Compiled Design with Verdi. . . . . . . . . . . . . . . . 34-4
Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-7
Key Points to Note . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-8
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-9
Dumping FSDB File for Various Flows . . . . . . . . . . . . . . . . . 34-9
Setting Up Verdi. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-10

Use Model for FSDB Dumping. . . . . . . . . . . . . . . . . . . . . . . . 34-10
Using Verilog System Tasks . . . . . . . . . . . . . . . . . . . . . . . 34-11
Using UCLI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-11
Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-12

Interactive and Post-Processing Debug . . . . . . . . . . . . . . . . . . . 34-13
Prerequisites . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-14
Interactive Simulation Debug Flow . . . . . . . . . . . . . . . . . . . 34-15
Key Points to Note . . . . . . . . . . . . . . . . . . . . . . . . . . 34-16
Post-Processing Debug Flow. . . . . . . . . . . . . . . . . . . . . . . . 34-16
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-17

Unified UCLI Dump Command . . . . . . . . . . . . . . . . . . . . . . . . 34-18
Default Dump File. . . . . . . . . . . . . . . . . . . . . . . . . . . 34-18
Default Dump Type. . . . . . . . . . . . . . . . . . . . . . . . . . . 34-18
Use Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34-19
Use Model for FSDB Dumping . . . . . . . . . . . . . . . . . . . . . 34-19
Key Points to Note . . . . . . . . . . . . . . . . . . . . . . . . . . 34-20
UCLI FSDB Dump Commands . . . . . . . . . . . . . . . . . . . . . . 34-21

Appendix A. VCS Environment Variables

Simulation Environment Variables. . . . . . . . . . . . . . . . . . . . . . . . A-1

Optional Environment Variables . . . . . . . . . . . . . . . . . . . . . . . A-3

Using Environment Variables in Verilog Source Code. . . . . . . . . A-5

Appendix B. Compile-Time Options

Option for Code Generation. . . . . . . . . . . . . . . . . . . . . . . . . . B-4
Options for Accessing Verilog Libraries . . . . . . . . . . . . . . . . . B-4

Options for Incremental Compilation . . . . . . . . B-7
Options for Help . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-9
Option for SystemVerilog . . . . . . . . . . . . . . . . . . . . . . . . . . B-9
Options for SystemVerilog Assertions . . . . . . . . . . . . . . . . . B-9
Options to Enable Compilation of OVA Case Pragmas . . . . . B-20
Options for Native Testbench. . . . . . . . . . . . . . . . . . . . . . . . B-20
Options for Different Versions of Verilog . . . . . . . . . . . . . . . B-28
Option for Initializing Verilog Variables, Registers and Memories with Random Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-30
Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design . . . . . . . . . . . . . . . . . . . . . . B-33
Options for Selecting Register or Memory Initialization . . . . . B-37
Options for Using Radiant Technology. . . . . . . . . . . . . . . . . B-38
Options for Starting Simulation Right After Compilation . . . . B-38
Options for Specifying Delays and SDF Files . . . . . . . . . . . B-38
Options for Compiling an SDF File . . . . . . . . . . . . . . . . . . . B-46
Options for Specify Blocks and Timing Checks . . . . . . . . . . B-46
Options for Pulse Filtering . . . . . . . . . . . . . . . . . . . . . . . . . . B-48
Options for Negative Timing Checks . . . . . . . . . . . . . . . . . . B-49
Options for Profiling Your Design. . . . . . . . . . . . . . . . . . . . . B-50
Options to Specify Source Files and Compile-time Options in a File B-50
Options for Compiling Runtime Options Into the Executable. B-53
Options for PLI Applications . . . . . . . . . . . . . . . . . . . . . . . . B-53
Options to Enable the VCS DirectC Interface . . . . . . . . . . . B-56
Options for Flushing Certain Output Text File Buffers . . . . . . B-57
Options for Simulating SWIFT VMC Models and SmartModels B-58

Options for Controlling Messages . . . . . . . . . . . . . . . . . . . . . B-58
Option to Run VCS in Syntax Checking Mode. . . . . . . . . . . . B-64
Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-65
Options for Cell Definition . . . . . . . . . . . . . . . . . . . . . . . . . . B-66
Options for Licensing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-67
Options for Controlling the Linker . . . . . . . . . . . . . . . . . . . . B-68
Options for Controlling the C Compiler . . . . . . . . . . . . . . . . . B-71
Options for Source Protection . . . . . . . . . . . . . . . . . . . . . . . B-73
Options for Mixed Analog/Digital Simulation . . . . . . . . . . . . . B-73
Options for Changing Parameter Values . . . . . . . . . . . . . . . . B-75
Checking for x and z Values in Conditional Expressions . . . . B-75
Options for Detecting Race Conditions . . . . . . . . . . . . . . . . . B-76
Options to Specify the Time Scale . . . . . . . . . . . . . . . . . . . . B-77
Option to Exclude Environment Variables During Timestamp Checks B-78
Options for Overriding Parameters . . . . . . . . . . . . . . . . . . . B-79
Option to Enable Bounds Check at Compile-Time . . . . . . . . B-82
Option to Enable Bounds Check at Runtime . . . . . . . . . . . . . B-83
Error-[DT-OBAE] Out of Bounds Access for Queues . . . . B-84
Error-[DT-OBAE] Out of Bounds Access for Dynamic Arrays B-84
Warning-[AOOBAW] Array Out of Bounds Access for Fixed Size Unpacked Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-85
Warning-[AOOBAW] Array Out of Bounds Access for Fixed Size Packed Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-86
Error-[DT-OBAE] Intermediate Access for Dynamic Arrays B-87
Warning-[AAIIW] Array Access with Intermediate Index . B-88
Warning-[AAIIW] Array Access with Intermediate Index for Fixed Size Packed Arrays . . . . . . . . . . . . . . . . . . . . . . . . . B-89

General Options. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-89
Specifying Directories for ‘include Searches . . . . . . . . . . B-89
Enable the VCS/SystemC Cosimulation Interface . . . . . . B-90
TetraMAX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-91
Suppressing Port Coersion to inout . . . . . . . . . . . . . . . . . B-91
Allow Inout Port Connection Width Mismatches. . . . . . . . B-91
Allow Zero or Negative Multiconcat Multiplier . . . . . . . . . B-92
Specifying a VCD File. . . . . . . . . . . . . . . . . . . . . . . . . . . B-92
Enabling Dumping . . . . . . . . . . . . . . . . . . . . . . . . . . . . B-92
Enabling Identifier Search . . . . . . . . . . . . . . . . . . . . . . . B-93
Memories and Multi-Dimensional Arrays (MDAs) . . . . . . B-94
Specifying a Log File . . . . . . . . . . . . . . . . . . . . . . . . . . B-94
Changing Source File Identifiers to Upper Case . . . . . . . B-95
Defining a Text Macro. . . . . . . . . . . . . . . . . . . . . . . . . . B-95
Option for Macro Expansion. . . . . . . . . . . . . . . . . . . . . . B-96
Specifying the Name of the Executable File. . . . . . . . . . . B-97
Returning The Platform Directory Name . . . . . . . . . . . . . B-97
Enabling Loop Detect. . . . . . . . . . . . . . . . . . . . . . . . . . . B-98
Changing the Time Slot of Sequential UDP Output Evaluation B-99
Gate-Level Performance . . . . . . . . . . . . . . . . . . . . . . . . B-99
Option to Omit Compilation of Code Between Pragmas . B-99
Generating a List of Source Files. . . . . . . . . . . . . . . . . . . B-101
Option for Dumping Environment Variables . . . . . . . . . . B-102

Appendix C. Simulation Options

Options for Simulating Native Testbenches . . . . . . . . . . . . . . C-2
Options for SystemVerilog Assertions . . . . . . . . . . . . . . . . . C-10
Options to Control Termination of Simulation. . . . . . . . . . . . C-20

Options for Enabling and Disabling Specify Blocks . . . . . . . . C-20
Options for Specifying When Simulation Stops . . . . . . . . . . . C-21
Options for Recording Output . . . . . . . . . . . . . . . . . . . . . . . . C-22
Options for Controlling Messages . . . . . . . . . . . . . . . . . . . . . C-22
Options for VPD Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-24
Options for VCD Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-26
Options for Specifying Delays . . . . . . . . . . . . . . . . . . . . . . . C-27
Options for Flushing Certain Output Text File Buffers . . . . . . C-29
Options for Licensing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-30
Option to Specify User-Defined Runtime Options in a File . . C-30
Option for Initializing Verilog Variables, Registers and Memories at Runtime . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-31
Option for Initializing Verilog Variables, Registers and Memories in Selective Parts of a Design at Runtime . . . . . . . . . . . . . . . C-32
General Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C-34
Viewing the Compile Time Options . . . . . . . . . . . . . . . . . C-34
Recording Where ACC Capabilities are Used . . . . . . . . . C-34
Suppressing the $stop System Task . . . . . . . . . . . . . . . . C-34
Enabling User-defined Plusarg Options . . . . . . . . . . . . . . C-35
Enabling Overriding the Timing of a SWIFT SmartModel. C-35
Enabling Loop Detect . . . . . . . . . . . . . . . . . . . . . . . . . . . C-35
Specifying acc_handle_simulated_net PLI Routine . . . . . C-36
Loading DPI Libraries Dynamically at Rutime . . . . . . . . . C-37
Loading PLI Libraries Dynamically at Runtime. . . . . . . . . C-37

Appendix D. Compiler Directives and System Tasks

Compiler Directives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . D-1
Compiler Directives for Cell Definition . . . . . . . . . . . . . . . . . . D-2

Compiler Directives for Setting Defaults . . . . . . . . . . . . . . . . . D-2
Compiler Directives for Macros . . . . . . . . . . . . . . . . . . . . . . . D-3
Compiler Directives for Delays. . . . . . . . . . . . . . . . . . . . . . . . D-6
Compiler Directives for Back Annotating SDF Delay Values . D-8
Compiler Directives for Source Protection . . . . . . . . . . . . . . . D-8
General Compiler Directives . . . . . . . . . . . . . . . . . . . . . . . . D-9
Compiler Directive for Including a Source File . . . . . . . . . D-9
Compiler Directive for Setting the Time Scale . . . . . . . . . D-9
Compiler Directive for Specifying a Library . . . . . . . . . . . D-10
Compiler Directive for File Names and Line Numbers . . . D-11
Unimplemented Compiler Directives . . . . . . . . . . . . . . . . . . D-11

System Tasks and Functions. . . . . . . . . . . . . . . . . . . . . . . . . . D-11
System Tasks for SystemVerilog Assertions Severity . . . . . . D-11
System Tasks for SystemVerilog Assertions Control . . . . . . . D-12
System Tasks for SystemVerilog Assertions . . . . . . . . . . . . . D-13
System Tasks for VCD Files . . . . . . . . . . . . . . . . . . . . . . . . . D-14
System Tasks for LSI Certification VCD and EVCD Files . . . D-16
System Tasks for VPD Files. . . . . . . . . . . . . . . . . . . . . . . . . . D-20
System Tasks for SystemVerilog Assertions . . . . . . . . . . . . . D-27
System Tasks for Executing Operating System Commands . D-29
System Tasks for Log Files . . . . . . . . . . . . . . . . . . . . . . . . . . D-29
System Tasks for Data Type Conversions . . . . . . . . . . . . . . . D-30
System Tasks for Displaying Information. . . . . . . . . . . . . . . . D-31
System Tasks for File I/O . . . . . . . . . . . . . . . . . . . . . . . . . . . D-31
System Tasks for Loading Memories. . . . . . . . . . . . . . . . . . . D-34
System Tasks for Time Scale. . . . . . . . . . . . . . . . . . . . . . . . . D-35

System Tasks for Simulation Control . . . . . . . . . . . . . . . . . . . . D-36
System Tasks for Timing Checks. . . . . . . . . . . . . . . . . . . . . . D-36
Timing Checks for Clock and Control Signals . . . . . . . . . . . . D-37
System Tasks for PLA Modeling . . . . . . . . . . . . . . . . . . . . . . D-40
System Tasks for Stochastic Analysis . . . . . . . . . . . . . . . . . . D-40
System Tasks for Simulation Time. . . . . . . . . . . . . . . . . . . . . D-41
System Tasks for Probabilistic Distribution . . . . . . . . . . . . . . D-41
System Tasks for Resetting VCS. . . . . . . . . . . . . . . . . . . . . . D-42
General System Tasks and Functions . . . . . . . . . . . . . . . . . . D-43
Checks for a Plusarg . . . . . . . . . . . . . . . . . . . . . . . . . . D-43
SDF Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . D-43
Counting the Drivers on a Net . . . . . . . . . . . . . . . . . . . . D-43
Depositing Values. . . . . . . . . . . . . . . . . . . . . . . . . . . . . D-44
Fast Processing Stimulus Patterns. . . . . . . . . . . . . . . . . . D-44
Saving and Restarting The Simulation State . . . . . . . . . . D-44
Checking for X and Z Values in Conditional Expressions D-45
Calculating Bus Widths . . . . . . . . . . . . . . . . . . . . . . . . D-46
Displaying the Method Stack . . . . . . . . . . . . . . . . . . . . . D-46
IEEE Standard System Tasks Not Yet Implemented . . . . . . . D-51

Appendix E. PLI Access Routines

Access Routines for Reading and Writing to Memories . . . . . . . E-2
acc_setmem_int. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-4
acc_getmem_int . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-5
acc_clearmem_int . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-6
Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-6
acc_setmem_hexstr. . . . . . . . . . . . . . . . . . . . . . . . . . . . E-11

Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-12
acc_getmem_hexstr . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-15
acc_setmem_bitstr. . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-16
acc_getmem_bitstr. . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-17
acc_handle_mem_by_fullname . . . . . . . . . . . . . . . . . . . . . . . E-18
acc_readmem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-18
Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-19
acc_getmem_range . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-21
acc_getmem_size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-22
acc_getmem_word_int. . . . . . . . . . . . . . . . . . . . . . . . . . . E-23
acc_getmem_word_range . . . . . . . . . . . . . . . . . . . . . . . . . E-24
Access Routines for Multidimensional Arrays . . . . . . . . . . . . . . . E-24
tf_mdanodeinfo and tf_imdanodeinfo. . . . . . . . . . . . . . . . . . . E-26
acc_get_mda_range . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-27
acc_get_mda_word_range() . . . . . . . . . . . . . . . . . . . . . . . . E-29
acc_getmda_bitstr() . . . . . . . . . . . . . . . . . . . . . . . . . . . E-30
acc_setmda_bitstr() . . . . . . . . . . . . . . . . . . . . . . . . . . . E-31
Access Routines for Probabilistic Distribution . . . . . . . . . . . . . . . E-32
vcs_random . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-33
vcs_random_const_seed. . . . . . . . . . . . . . . . . . . . . . . . . . E-34
vcs_random_seed . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-34
vcs_dist_uniform . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-35
vcs_dist_normal. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-35
vcs_dist_exponential . . . . . . . . . . . . . . . . . . . . . . . . . . . E-36
vcs_dist_poisson . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-37

Access Routines for Returning a Pointer to a Parameter Value . E-37
acc_fetch_paramval_str. . . . . . . . . . . . . . . . . . . . . . . . . . E-38
Access Routines for Extended VCD Files . . . . . . . . . . . . . . . . . . E-38
acc_lsi_dumpports_all . . . . . . . . . . . . . . . . . . . . . . . . . . E-40
acc_lsi_dumpports_call . . . . . . . . . . . . . . . . . . . . . . . . . E-41
acc_lsi_dumpports_close. . . . . . . . . . . . . . . . . . . . . . . . . E-43
acc_lsi_dumpports_flush . . . . . . . . . . . . . . . . . . . . . . . . E-44
acc_lsi_dumpports_limit. . . . . . . . . . . . . . . . . . . . . . . . . E-45
acc_lsi_dumpports_misc . . . . . . . . . . . . . . . . . . . . . . . . E-46
acc_lsi_dumpports_off . . . . . . . . . . . . . . . . . . . . . . . . . . E-47
acc_lsi_dumpports_on . . . . . . . . . . . . . . . . . . . . . . . . . . E-48
acc_lsi_dumpports_setformat . . . . . . . . . . . . . . . . . . . . . . E-50
acc_lsi_dumpports_vhdl_enable . . . . . . . . . . . . . . . . . . . . E-51
Access Routines for Line Callbacks . . . . . . . . . . . . . . . . . . . . E-52
acc_mod_lcb_add . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-53
acc_mod_lcb_del. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-55
acc_mod_lcb_enabled. . . . . . . . . . . . . . . . . . . . . . . . . . . E-57
acc_mod_lcb_fetch . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-57
acc_mod_lcb_fetch2 . . . . . . . . . . . . . . . . . . . . . . . . . . . E-59
acc_mod_sfi_fetch. . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-61
Access Routines for Source Protection. . . . . . . . . . . . . . . . . . . E-62
vcsSpClose . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-66
vcsSpEncodeOff . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-67
vcsSpEncodeOn . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-68
vcsSpEncoding . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-70

vcsSpGetFilePtr . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-71
vcsSpInitialize . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-72
vcsSpOvaDecodeLine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-73
vcsSpOvaDisable. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-74
vcsSpOvaEnable . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-75
vcsSpSetDisplayMsgFlag . . . . . . . . . . . . . . . . . . . . . . . . . . E-77
vcsSpSetFilePtr . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-77
vcsSpSetLibLicenseCode . . . . . . . . . . . . . . . . . . . . . . . . . . E-78
vcsSpSetPliProtectionFlag. . . . . . . . . . . . . . . . . . . . . . . . . E-79
vcsSpWriteChar . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-80
vcsSpWriteString . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-81

Access Routine for Signal in a Generate Block. . . . . . . . . . . . . . E-83
acc_object_of_type . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-83

VCS API Routines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-83
Vcsinit() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-84
VcsSimUntil() . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . E-84

# 1

# Getting Started

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

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

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

This chapter includes the following sections:


- “Simulator Support with Technologies”
- “Simulation Preemption Support”
- “Setting Up the Simulator”
- “SeeisUsing 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 Languag*e (IEEE Std 1364).
- 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 and SystemVerilog compilation and simulation capabilities, VCS includes the following integrated set of features and tools:

- SystemC - VCS / SystemC Co-simulation Interface enables VCS and the SystemC modeling environment to work together when simulating a system described in the Verilog and SystemC languages. For more information, refer to “Using SystemC” .

- Discovery Visualization Environment (DVE) — For more information, refer to “Using DVE” .
- 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, Debussy, Denali, and other acceleration and emulation systems.

## Simulation Preemption Support

VCS supports simulation preemption. If one suspends 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"

### 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.

## 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 Appendix A, "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`).

RHEL32, RHEL64 and IBM RS/6000 AIX platforms all include a C compiler, and VCS assumes the compiler is located in its default location (`/usr/bin`).

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

# SeeisUsing the Simulator

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

- Compiling the Design
- Simulating the Design

### Compiling the Design

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

### Simulating the Design

Simulate your design by executing the binary simulation executable, `simv`. For more information, see Chapter - "VCS Flow".

## Basic Usage Model

### Compilation

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

### Simulation

```
% simv [run_options]
```

# Default Time Unit and Time Precision

The default time unit is 1 s.

The default time precision is 1 s.

## 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 DVE GUI settings has no effect on the `vcsfind` command.

*Table 1-1    Supported Search Options*

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

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

## 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

Simulating a design using VCS involves two basic steps:

- “Compilation”
- “Simulation”

This flow is supported only for Verilog HDL and SystemVerilog designs. For information on supported technologies, refer to “Simulator Support with Technologies” .

## 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 "Compiling or Elaborating the Design in the Optimized Mode" .

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 or Elaborating the Design in the Debug Mode" .

## 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. For a complete list of options, see Compilation 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.

`+incdir+`*`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 Discovery Visualization Environment (DVE) and UCLI

`-gui`

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

For information on DVE, see the *DVE User Guide*. For information on UCLI, see the *UCLI User Guide*.

### 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 the Discovery Visualization Environment (DVE), and to debug through the command-line interface, you can use the Unified Command-line Interface (UCLI).

Note:

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

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 `-debug`, `-debug_all`, or `-debug_access(+<option>)`. Use these options only when you require runtime debug abilities.

The following command line simulates the design in batch mode:

```
% simv
```

## Commonly Used Runtime Options

Use the following command line to simulate the design:

```
% executable [runtime_options]
```

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

For a complete list of 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”
- “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 active time slot of a simulation time. 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

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.

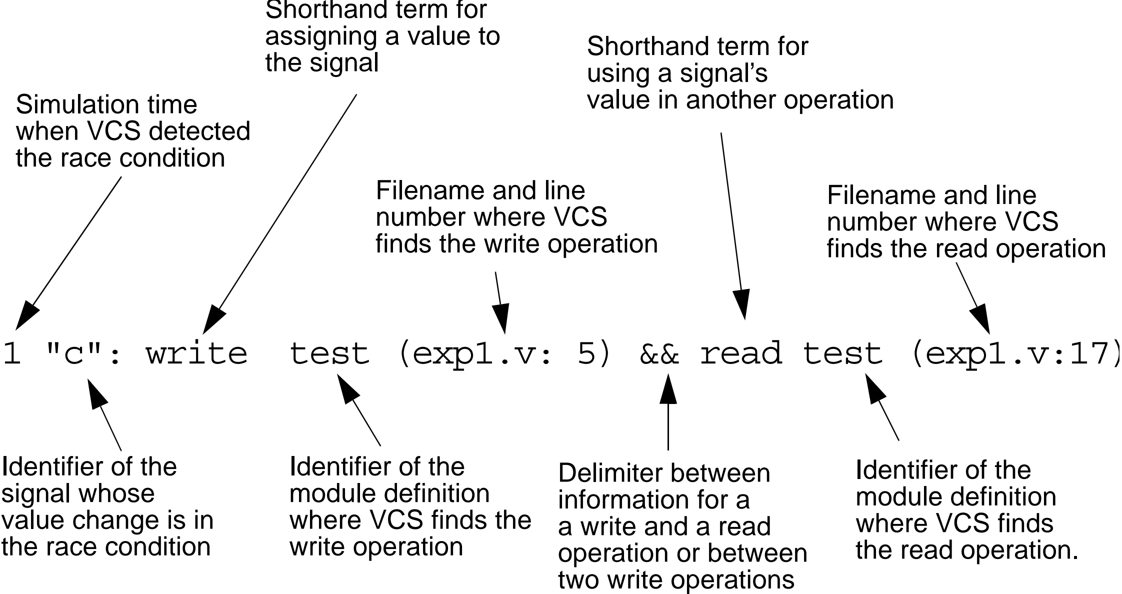
Note:

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.

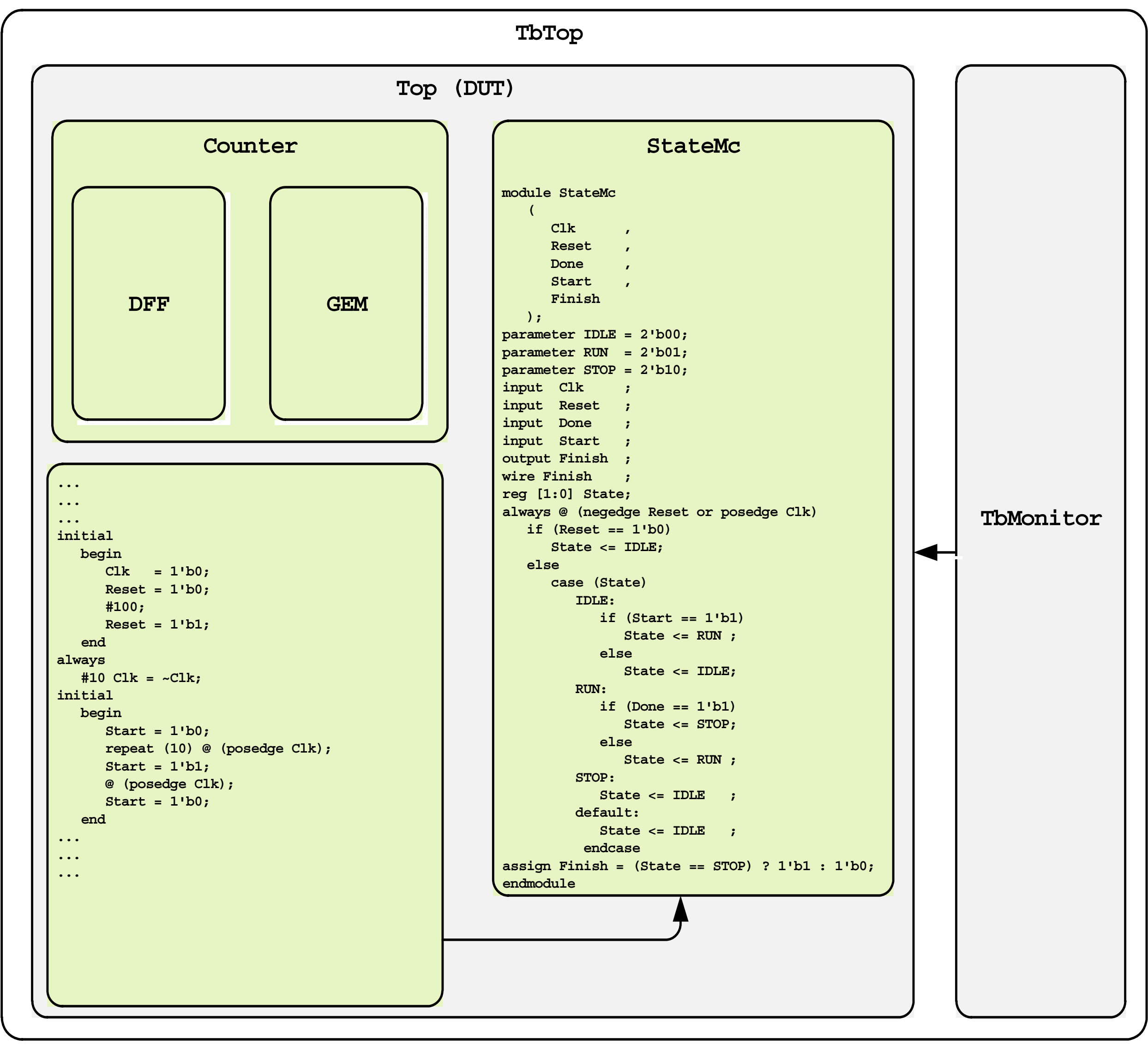
If no clock-data race is detected during simulation, VCS reports the following information:

```
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:



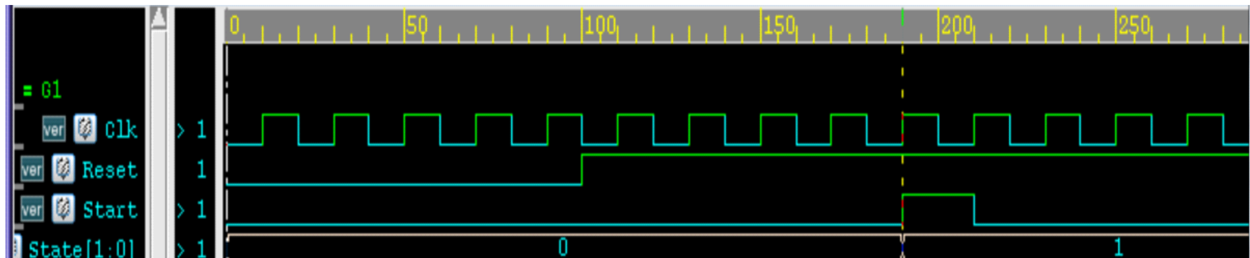
Compile the test cases as follows:

```
% 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
  - MX design
  - FGP flow
  - `-dbsflagsbytearray` 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 VPD files that you can input into DVE.

The `$dumpvars` system task specifies writing a VCD file, and the `$vcdpluson` system task specifies writing a VPD file. You can also input a VCD file to DVE, which translates the VCD file to a VPD file and then displays the results from the new VPD file. For details on using DVE, see the *Discovery Visualization Environment User Guide.*

Debugging features significantly slow down the simulation performance of any logic simulator including VCS. This is particularly true for operations that make VCS display text on the screen and even more so for operations that make VCS write information to a file. For this reason, you 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 VPD file. You can use the VPD file with DVE to post-process the design. In this particular case, the information is for all the signals in the design, so the performance cost is extensive. You would only want to do this early in the development cycle of the design when finding bugs is more important than simulation speed.

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

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

### Advantages and Disadvantages

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

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

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

## Enabling Debugging Features at Runtime

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

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

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

```
initial
if ($test$plusargs("postprocess"))
begin
$vcdpluson(0,design_1);
$vcdplusdeltacycleon;
$vcdplusglitchon;
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
    $vcdpluson(0,design_1);
    $vcsplusdeltacycleon;
    $vcdplusglitchon;
   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:

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

### Avoid Unaccelerated Primitives

VCS cannot accelerate tranif1, tranif0, rtranif1, rtranif0, tran, and rtran switches. They are defined in 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_indentifier(y,rb);
endmodule
```

### Avoid Strength Specifications in Continuous Assignment Statements

Omit strength specifications in continuous assignment statements. For example:

```
assign net1 = flag1;
```

Simulates faster than:

```
assign (strong1, pull0) net1= flag1;
```

Continuous assignment statements are described 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 |
|---|---|
| `force` and `release` procedural statements | Page 126-127 |
| `repeat` statements | Page 134-135, see the other looping statements on these pages and consider them as an alternative. |
| `wait` statements, also called as level-sensitive event controls | Page 141 |
| `disable` statements | Page 162-164 |
| `fork-join` 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:<br>`always @ (r1,r2,r3)` | yes |
| name-based parameter passing:<br>`modname #(.param_name(value)) inst_name(sig1,...);` | yes |
| ANSI-style port and argument lists:<br>`module dev(output reg [7:0] out1, input wire [7:0] w1);` | yes |
| initialize a reg in its declaration:<br>`reg [15:0] r2 = 0;` | yes |
| conditional compiler directives:<br>`‘ifndef` and `‘elseif` | yes |
| disabling the default net data type:<br>`‘default_nettype` | yes |
| signed arithmetic extensions:<br>`reg signed [7:0] r1;` | yes |
| file I/O system tasks:<br>`$fopen $fsanf $scanf` and more | yes |
| passing values from the runtime command line:<br>`$value$plusarg` system function | yes |
| indexed part-selects:<br>`reg1[8+:5]=5’b11111;` | yes |
| multi-dimensional arrays:<br>`reg [7:0] r1 [3:0] [3:0];` | yes |
| maintaining file name and line number:<br>`‘line` | yes |

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

## Case Statement Behavior

The IEEE Std 1364-2001 standards for the Verilog language state that you can enter the question mark character (`?`) in place of the `z` character in the `casex` 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:

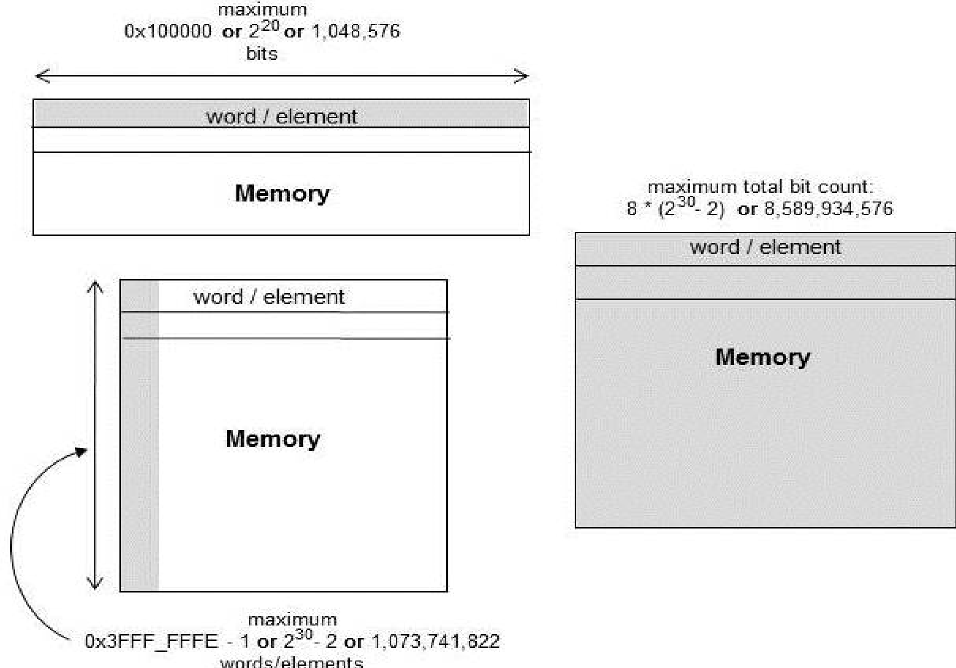
1. The `` `undef `` compiler directive has a higher precedence than the `+define` compile-time option.
2. The `+define` compile-time option has a higher precedence than the `` `define `` and `` `undefineall `` compiler directives.

## 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 bits (1G Bytes)

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

2 * 2**26 * 128 = 17,179,869,184 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 will 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 metacomment in the memory declaration to specify a sparse memory model. For example:

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

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

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

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
```

## Returning Information About the Scope

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

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

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

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

The following example contains these system functions:

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

module dev1(in);
input in;
always @ (posedge in)
```

```
begin:named
if ($activeinst("top.d0","top.d1"))
    $display("%i");
if ($activescope("top.d0.block","top.d1.named"))
    $display("%-i");
end
endmodule
```

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

```
extern void showInst(input bit[31:0]);

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

declaration of C function named showInst

$activeinst system function without arguments passed to the C function

```
module child;
initial discriminator.t;
endmodule

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

In task `t`, the following occurs:

1. The `$activeinst` system function returns a pointer to the current scope, which is passed to the C function `showInst`. It is a pointer to a volatile or temporary char buffer containing the name of the instance.

2. A nested begin block executes only if the current scope is either `top.c1` or `top.c3`.

3. VCS displays whether `$activeinst` points to a zero or non-zero value.

The C code is as follows:

```
#include <stdio.h>

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

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

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

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

## Avoiding Circular Dependency

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

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

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

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

This circular dependency doesn't usually occur, but it can occur, so VCS displays a warning message when you use a seeded argument with these kinds of statements. This 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_dumpports for Simulation and Test

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

Note:

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

### Dealing With Unassigned Nets

Consider the following example:

```
module test(A);
```

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

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

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

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

The designer actually meant for a code of `Z` to be returned, input tristated. To achieve this code, the input `A` needs to be assigned a value of `z`. This is achieved by removing the comment from the line, `// assign A = 1'bz;`, in the above code. Now, when the code is executed, VCS is able to identify that the wire `A` going into `DUT_1` is being driven to a `z`. With the wire driven from the outside and not the inside, `$lsi_dumpports` returns a code of `Z`.

## Code Values at Time 0

Another issue can occur at time `0`, before values have been assigned to ports as you intended. As a result, `$lsi_dumpports` makes an evaluation for drivers when all of the users intended assignments have not been made. To correct this situation, you need to advance simulation time just enough to have your assignments take place. This can be accomplished by adding a `#1` before `$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. Since `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 |
| l | low (2 or more DUT drivers active) |
| Testbench Level (only z drivers from the DUT) | |
| h | high (2 or more DUT drivers active |
| Drivers Active on Both Levels | |
| 0 | low (both input and output are active with 0 values) |

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

# 4

# Compiling the Design

This chapter describes the following sections:


- “Compiling or Elaborating the Design in the Debug Mode”
- “Compiling or Elaborating the Design in the Optimized 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 or 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 DVE or UCLI. For information on DVE or UCLI, see the *DVE User Guide* and *UCLI User Guide* respectively.
- 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_pp`, `-debug`, `-debug_access(+<option>)`,`-debug_all`, `-debug_region=(<option>)(+<option>)`

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

### Compiling the Design in the Partial Debug Mode

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

### Compiling the Design in the Full Debug Mode

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

### Compiling 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 “Optimizing Simulation Performance for Desired Debug Visibility With the -debug_access Option”section.

## Compiling or Elaborating the Design in the Optimized Mode

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

For more information on performance, see Chapter “Performance Tuning”.

Note:

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

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

You can use the `-debug_access` option at compile time to have more granular control over debug capabilities in a simulation.

The `-debug_access` option enables the dumping of the VPD and FSDB files for post-processing debug, and reduces debug capabilities when compared to the `-debug_pp` option.

You can specify additional options with the `-debug_access` option to selectively enable the required debug capabilities. You can optimize simulation performance by enabling only the required debug capabilities.

This section consists of the following subsections:


- "Use Model"
- "Specifying Design Regions for -debug_access Capabilities"
- "Enabling Additional Debug Capabilities"
- "Reduction in the Objects Being Dumped"
- "Testbench Definition"
- "Differences Between -debug_pp and -debug_access+pp"
- "Using -debug_access With Tab Files"
- "Using -debug_access With -ucli/-gui at Compile Time"
- "Unused Tab File Calls"
- "Including Tab Files"

- “Interaction With Other Debug Options”

## Use Model

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

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

Table 4-1 describes the supported options of `-debug_access`:

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

| Option | Description |
|---|---|
| `drivers` | The `-debug_access+drivers` option enables driver debugging capability. |
| `r` | The `-debug_access+r` option enables the read capability for the entire design. |
| `w` | The `-debug_access+w` option applies write (deposit) capability to the registers and variables for the entire design. |
| `wn` | The `-debug_access+wn` option applies write (deposit) capability to the nets for the entire design. |
| `f` | The `-debug_access+f` option enables the following:<br>• Write (deposit) capability on registers and variables<br>• Force capability on registers, variables, and nets<br>This option is equivalent to `-debug_access+w+fn` |
| `fn` | The `-debug_access+fn` option applies force capability to the nets for the entire design. |
| `fwn` | The `-debug_access+fwn` option applies write (deposit) and force capability to all nets in the design. |
| `line` | The `-debug_access+line` option enables line debugging. It allows you to use the commands for step/next and line breakpoints.<br>This option is equivalent to `-debug_access+pp -line` |

| Option | Description |
|---|---|
| cbk | The -debug_access+cbk option enables PLI-based callbacks on static nets, registers, and variables. |
| cbkd | The -debug_access+cbkd option enables both dumping and PLI-based callbacks on dynamic nets, registers, and variables. |
| thread | The -debug_access+thread option enables the debugging of the SystemVerilog threads. |
| class | The -debug_access+class option is equivalent to the following command:<br>-debug_access+r+w+thread+class+line+cbk+cbkd<br><br>The -debug_access+class option enables testbench debug capabilities. |
| nomemcbk | The -debug_access+nomemcbk option disables callbacks for memories and multidimensional arrays (MDAs). By default, -debug_access enables callbacks for memories and MDAs. |
| dmptf | The -debug_access+dmptf option enables dumping of task/function ports and internal nodes/memories for the entire design. |
| pp | The -debug_access+pp option is equivalent to the following command:<br>-debug_access+w+cbk+drivers<br><br>The -debug_access+pp option enables debug capabilities equal to -debug_pp (except for no thread debugging and dumping of task/function signals, and does not apply capability inside cells and encrypted modules). |
| all | The -debug_access+all option is equivalent to the following commands:<br><br>-debug_access+line+class+wn+driver+r+w+cbk+f+fn+thread +cbkd<br><br>The -debug_access+all option enables debug capabilities equal to -debug_all (except it does not apply capability inside cells and encrypted modules). |
| report | The -debug_access+report option enables the reporting of the global debug capability diagnostics. For more information, see the "Using -debug_access to Report Global Debug Capability Diagnostics"section. |

Example: `-debug_access+r+line+class+drivers`

**Key Points to Note**

- In the GENIP flow, the `-debug_access` option does not affect the recompilation of IPs in the shared libraries.
- To enable assertions debug in DVE, use `-debug_access+f` or `-debug_access+fwn`.
- 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.

## Using -debug_access to Report Global Debug Capability Diagnostics

You can use the `-debug_access+report` option at compile time to enable the reporting of the global debug capability diagnostics.

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.

Table 4-2 lists the types of debug capabilities reported by the `-debug_access+report` option.

*Table 4-2 Debug Capabilities Reported by -debug_access+report*

| Capability | Types |
| --- | --- |
| Read (`r`) | Read, Read (DKI) |
| Write (`w`) | Write, Write-net |
| Callback (`cbk`) | Callback, Callback (`$dumpvars`), Callback (DKI), Callback (memories, arrays), Callback (class member) |
| Force (`frc`) | Force, Force-net |
| Static | Static (traversal) |

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:*
```

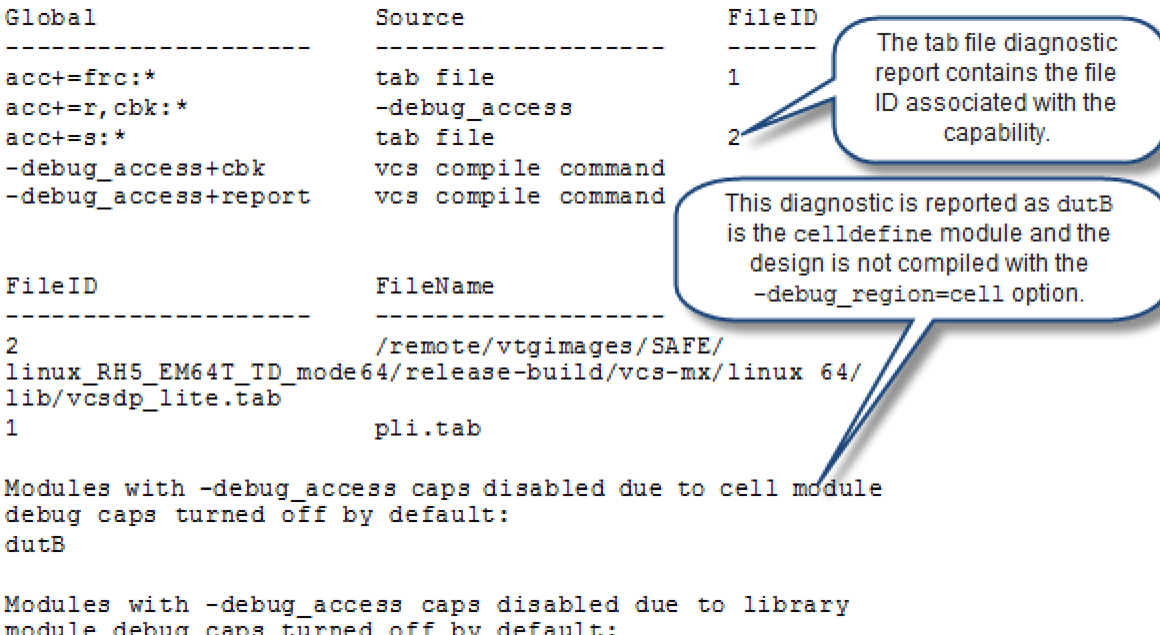
To compile the `top.sv` file, execute the following command:

```
% vcs -P pli.tab -debug_access+cbk -debug_access+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:



```
Global                  Source                FileID
--------------------    -------------------   ------
acc+=frc:*              tab file              1
acc+=r,cbk:*            -debug_access
acc+=s:*                tab file              2
-debug_access+cbk       vcs compile command
-debug_access+report    vcs compile command


FileID                  FileName
--------------------    -------------------
2                       /remote/vtgimages/SAFE/
linux_RH5_EM64T_TD_mode64/release-build/vcs-mx/linux 64/
lib/vcsdp_lite.tab
1                       pli.tab

Modules with -debug_access caps disabled due to cell module
debug caps turned off by default:
dutB

Modules with -debug_access caps disabled due to library
module debug caps turned off by default:
```



### 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 [DUT, cell, testbench, and standard package (OVM, UVM, VMM, and RAL), or encrypted instances (modules, programs, packages, and interfaces)].

You must use the `-debug_region` option along with the `-debug_access` option at compile time. Following is the syntax of `-debug_region`:

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

Table 4-3 describes the options supported by `-debug_region`.

*Table 4-3 -debug_region Options*

| Option | Description | Default Functionality if -debug_region is not specified |
| --- | --- | --- |
| `lib` | Applies debug capabilities to the cells inside libraries. | Debug capability is not applied to the libraries. |
| `cell` | Applies debug capabilities to the cells. | Debug capability is not applied to the cells. |
| `encrypt` | Applies debug capabilities to the fully-encrypted instances (modules, programs, packages, and interfaces). | Debug capability is not applied to the fully-encrypted instances. |

| Option | Description | Default Functionality if -debug_region is not specified |
|---|---|---|
| tb | Applies debug capabilities only to the testbench, but does not apply debug capabilities to the standard packages. For more information on what defines testbench, see "Testbench Definition". It does not apply debug capability to the standard packages. The VPD/FSDB dumping of the DUT is not affected by this option. | Debug capability is applied to testbench and DUT. |
| dut | Applies debug capabilities only to the non-testbench objects. For more information on what defines testbench, see "Testbench Definition". | Debug capability is applied to testbench and DUT. |
| stdpkg | Applies debug capabilities to the standard packages.<br><br>You must use the `stdpkg` option in combination with the `tb` option. VCS issues a warning message if you use `-debug_region=stdpkg` only.<br><br>The `-debug_region=tb+stdpkg` option applies debug capabilities to both testbench and standard packages. | Debug capability is applied to the standard packages. |

## Examples

- `-debug_access+class -debug_region=tb`

  Applies class debug capability only to the testbench. Debug capability is not applied to the standard packages.

- `-debug_access+force -debug_region=dut`

  Applies force debug capability to the DUT.

- `-debug_access+class -debug_region=tb+stdpkg`

Applies testbench debug capability to the TB and standard packages.

- `-debug_access+drivers -debug_region=cell+lib`

  Applies `-debug_access` and driver debug capability to the cells both inside and outside libraries.

- `-debug_access+class -debug_region=tb`
  `-debug_access+drivers -debug_region=dut`

  This option is equivalent to not specifying the `-debug_region` option, so the following warning message, is issued in this case:

```
Warning-[DBGACC_REG_CMB2] Illegal '-debug_region' usage
The combination of options 'tb+dut' is not valid. -
debug_access capability will be enabled for the entire
design
Please recompile using the '-debug_access<+options>'
switch and incremental options as required. Recommended
options are '-debug_access' for post-process debug, '-
debug_access+class' for testbench debug, and
'-debug_access+all' for all debug capabilities. Refer the
VCS user guide for more granular options for debug control
under the switch '-debug_access' and refer to '-
debug_region' for region control.
```

- `-debug_access+drivers -debug_region=dut+stdpkg`

  Debug capability is applied outside the testbench, but the standard packages are considered as testbench, so the following warning message, is issued in this case:

```
Warning-[DBGACC_REG_CMB2] Illegal '-debug_region' usage
The combination of options 'dut+stdpkg' is not valid. -
debug_access capability will be enabled for the entire
design.
Please recompile using the '-debug_access<+options>'
switch and incremental options as required. Recommended
options are '-debug_access' for post-process debug, '-
```

```
debug_access+class' for testbench debug, and
'-debug_access+all' for all debug capabilities. Refer the
VCS user guide for more granular options for debug control
under the switch '-debug_access'and refer to '-
debug_region' for region control.
```

- `-debug_access+class -debug_region=tb`
  `-debug_access -debug_region=tb`

  Applies testbench debug capability only to the testbench.

**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 (`cell`, `tb`, `dut`, `encrypt`, `lib`, or `stdpkg`).
- An error message is issued if you use `-debug_region` without `-debug_access`.
- A warning message is issued if you use `-debug_region=tb/dut/stdpkg` with other debug options, such as `-debug_pp`, `-debug`, or `-debug_all`. Following is the sample warning message:

```
Warning-[DBGACC_CMB] Illegal '-debug_access' usage
The combination of options '(-debug_all, -debug_obj, -
debug or -debug_pp) and -debug_region=dut' is not valid.
Ignoring the option '-debug_region=dut'
Please recompile using the '-debug_access<+options>'
switch and incremental options as required. Recommended
options are '-debug_access' for post-process debug, '-
debug_access+class' for testbench debug, and
'-debug_access+all' for all debug capabilities. Refer the
VCS user guide for more granular options for debug control
under the switch '-debug_access' and refer to '-
debug_region' for region control.
```

- Use of `-debug_region=tb` (without the `stdpkg` option) may effectively enable debugging in the code that belongs to `stdpkg`. If some `stdpkg` code belongs to a parameterized class, the code is instrumented in the “parameterized class repository”. The “parameterized class repository” is part of `tb`, and is not part of `stdpkg`.

  The `-debug_region` option is applicable only to the SystemVerilog portion of a design. It is not applicable to pure VHDL or SystemC designs.

  For mixed designs, if you specify the `-debug_region=tb` option, then the `-debug_access` option applies to the testbench portion of SystemVerilog and all the VHDL/SystemC portions of the design.

  Note:

  If the standard packages cannot be debugged, then the UCLI command, `config stepintotblib (on|off)`, returns a warning message.

- The `+vcsd` and `+memcbk` global debug options are 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=tb` option is specified.
  - Inside the testbench when the `-debug_region=dut` option is specified.

- Dynamic callbacks are enabled if any of the following `-debug_access` options are specified:

  `class`, `all`

  Dynamic callbacks are turned off for the following cases:

  - Inside the testbench if the `-debug_region=dut` option is specified.
  - Inside the fully-encrypted modules if the `-debug_region=encrypt` option is not specified.

## Enabling Additional Debug Capabilities

This section consists of the following subsections:

- "Driver/Load Debug Capability"
- "Statement Debug Capability"
- "Value Change 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
- DVE and UCLI show driver/load

- `$countdrivers`
- `$dumpports`, `$lsi_dumpports`

VCS generates an error message, if you use any 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.

### Value Change 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 DVE. It also allows for constraint debugging and thread debugging in DVE/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.

## Testbench Definition

This section describes the objects that are considered as part of the testbench.

- SystemVerilog program/package instances are part of the testbench. All objects declared inside the program/package block are considered as part of the testbench.
- SystemVerilog module (including all objects declared inside the module block) is considered as part of the testbench, if the module contains any of the following elements:
  - Class definitions
  - Declarations of dynamic or associate arrays
  - Declarations of smart queues
  - Declarations of class variables
  - Imports of all or part of a package

- SystemVerilog interface instances that contain clocking blocks or the elements (class definitions, dynamic arrays, and so on) mentioned in the above point are part of the testbench. All objects declared inside the interface block are considered as part of the testbench.

Note:

`$unit` is considered as part of the testbench if it contains the elements (class definitions, dynamic arrays, and so on) mentioned in the above point.

## Differences Between -debug_pp and -debug_access+pp

Table 4-4 describes some of the important functionality differences between `-debug_pp` and `-debug_access+pp`.

*Table 4-4 Comparing -debug_pp and -debug_access+pp*

| -debug_pp | -debug_access+pp |
|---|---|
| Read, write, and callback capability is supported. | Read, write, and callback capability is supported. |
| Debug capability is enabled for the entire cell by default. | By default, this option disables debugging in cells and libraries. You must use the `-debug_region=cell+lib` option along with the `-debug_access+pp` option to enable debug capability. |
| Debug capability inside an encrypted region is enabled by default. | By default, this option disables debug capability inside an encrypted region. You must use the `-debug_region=encrypt` option along with the `-debug_access+pp` option to enable the encrypted object debug capability. |
| Driver debugging is enabled. | You must use the `-debug_access+pp+drivers` option to enable driver/load debugging. |

| -debug_pp | -debug_access+pp |
|---|---|
| Signals in tasks/functions are dumped. | You must use the -debug_access+pp+dmptf option to enable dumping of task/function ports and internal signals. |
| Thread debugging is supported. | Thread debugging is not supported by default. You must use the -debug_access+pp+thread option to enable thread debugging. |

## 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, UCLI and DVE are enabled.

## Using -debug_access With -ucli/-gui at Compile Time

If you use the -ucli or -gui compile-time option without an additional -debug_pp, -debug, -debug_all, or -debug_access* option, it is treated as compiling with the -debug option.

## 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, then you can use -debug_access to dump FSDB. You must ensure that your source code contains the $fsdbDumpvars task. There is no need to specify Verdi PLI and tab files on the VCS command line to dump FSDB.

## Interaction With Other Debug Options

The following points describe the interaction of -debug_access 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.

- If you use `-debug_access` with `-debug_pp`, `-debug`, `-debug_all`, or `-debug_obj`, then all debug options are unioned.

## 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. 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 \
-I 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:

```
%> simv -sv_liblist bootstrap_file

%> simv -sv_root path_relative_or_absolute_to_shared_object \
-sv_lib test

%> simv -sv_lib test
```



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 Parameters"
- "Checking for x and z Values In Conditional Expressions"
- "Lint Warning Message for Missing 'endcelldefine"
- "Error/Warning/Lint Message Control"
- "Extracting the Files Used in 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:

- `+vcs+initreg+random`

  This option enables initialization for an entire design.

- `+vcs+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*

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

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

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

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

For more information on the `+vcs+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.

```
% vcs +vcs+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 4-4*

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

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

For more information on the `+vcs+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:

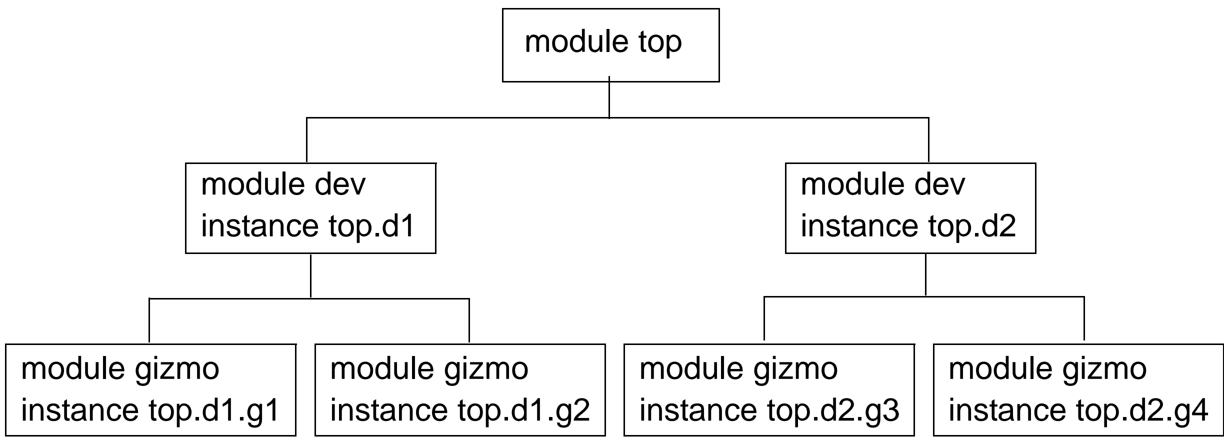
```
defaultvalue x|z|0|1|random|random seed_value

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

- `+vcs+initreg+random+nomem`
- `+vcs+initreg+random+noreg`

The `+vcs+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 `+vcs+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 to a file named `vcs_initreg_random_value.txt`.

For example:

```
% setenv VCS_PRINT_INITREG_INITIALIZATION 1
```

## Overriding Parameters

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

- `-pvalue`
- `-parameters`

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

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

For example:

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

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

```
assign value path_to_the_parameter
```

Here:

`assign`

Keyword that starts a line in the file.

`value`

New value of the parameter.

`path_to_the_parameter`

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

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

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

Note:

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

## Checking for x and z Values In Conditional Expressions

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

A conditional expression is of the following types or statements:

- A conditional or `if` statement:

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

- A `case` statement:

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

- A statement using the conditional operator:

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

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

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

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

VCS does not display a warning message when a sub-expression has the value `x` or `z`, but the conditional expression evaluates to `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 4-5 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 4-6 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  -xzcheck nofalseneg example.v
```

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

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

 r1 false at 1

 r1 true at 1
```

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

## 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 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.

### 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 .

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.

## 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 . 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 4-7 Source Code With Missing ‘endcelldefine*

```
`celldefine
module mod;
endmodule
```

In this example, there is no corresponding `‘endcelldefine` compiler directive.

In VCS, 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.
```

The source code in Example4-8 does not display this warning message when you include the +lint=CDUB.

*Example 4-8 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 Example4-9.

*Example 4-9 Source Code With ‘celldefine and ‘resetall*

```
`celldefine
 module mod;
 endmodule
`resetall
```

The source code in both Example4-8 and Example4-9 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 4-10 ‘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 4-11 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 Example4-11, 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 4-12 Recursive 'celldefine/'endcelldefine Compiler Directives*

```
'celldefine
'celldefine
module dev (...,...);
'celldefine
'celldefine
module dev (...,...);
  ...
endmodule
'endcelldefine
'endcelldefine
```

Example4-12 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 49.
- 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.

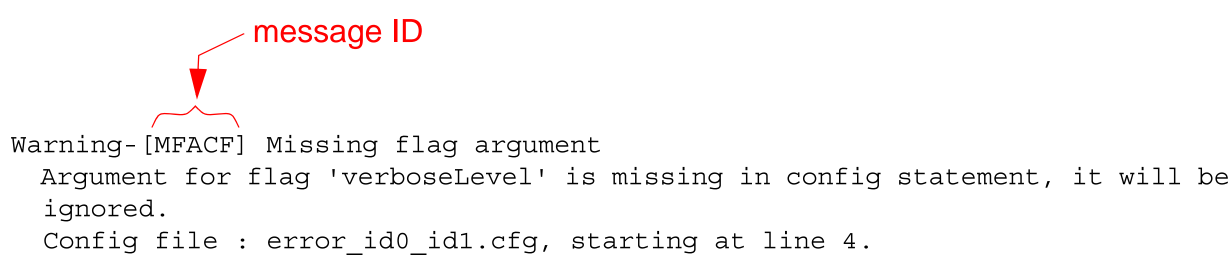
- Limit the display of specific messages to a maximum number that you specify.

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 4-2, the message ID is MFACF.

Note:

The `-error` option is also a runtime option.

*Figure 4-2 Message IDs*




The new compile-time options for controlling messages and their syntax are as follows:

-error=[no]*message_ID*[:*max_number*],...|none|all

-error=all,no*Warn_ID*|no*Lint_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 `TFATFC` is the ID for the following error message:

```
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.
```

### 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 that 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:

```
-suppress=messageID
-error=messageID:max            -error=all
 +warn=messageID:max            +warn=all
 +lint=messageID:max            +lint=all
```

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,no`*`ID1`*`,no`*`ID2`*`...`

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=no`*`ID1`*`,no`*`ID2`*`...`

The comma separated list is a list of error message IDs, preceded by the keyword `no`, for example:

```
-error=noURMI,noETMFCB
```

Not all error messages are downgradable. If you enter an error message ID for an 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 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:

```
% 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. This file 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.

# 5

# Simulating the Design

This chapter describes the following:

- “Using DVE”
- “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 must use DVE or UCLI. To simulate your design in batch mode, refer to the section entitled, “Batch Mode”.

## Using DVE

DVE provides you with a graphical user interface to debug your design. Using DVE, you can debug the design in interactive mode or in post-processing mode. You must use the same version of VCS and DVE to ensure problem-free debugging of your simulation.

In the interactive mode, apart from running the simulation, DVE 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, a VPD/VCD/EVCD file is created during simulation, and you use DVE to:

- View waveforms
- Compare waveforms
- Trace drivers and loads
- View schematics and path schematics

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

```
% simv -gui
```

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

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

Note:
The interactive mode of DVE is not supported when you are running VCS slave mode simulation.

For information on generating a VPD/EVCD dump file, see "VPD, VCD, and EVCD Utilities".

For more information on using DVE, see *Discovery Visualization Environment User Guide* under VCS 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 a VPD file
- 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_pp, -debug, -debug_all options during compilation. 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 `-debug` switches during compilation. This is called a “mini UCLI” feature, where full power of Tcl is provided with just `run` and `quit` UCLI commands.

Note the following behavioral changes when UCLI is the default command-line interface:

- The `-s` option is not allowed in `simv`
- 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`) expects a UCLI script by default

```
%> simv -ucli -i ucli_script.inc
```

- The `-R` feature in VCS continues to take you to the old CLI/MX UI, unless you explicitly add `-ucli` to VCS command line.

## ucli2Proc Command

There are a few scenarios after UCLI became the default command line interface, which may require using of the `-ucli2Proc` command:

- 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 DVE and UCLI

`-debug_pp`

Gives best performance with the ability to generate the VPD/VCD file for post-process debug. It is the recommended option for post-process debug.

It enables read/write access and callbacks to design nets, memory callback, assertion debug, VCS DKI, and VPI routine usage. You can also run interactive simulation when the design is compiled with this option, but certain capabilities are not enabled. It does not provide force net and reg capabilities. Set value and time breakpoints are permissible, but line breakpoints cannot be set.

`-debug`

Gives average performance and debug visibility/control, that is, more visibility/control than `-debug_pp` and better performance than `-debug_all`. It provides force net and reg capabilities in addition to all capabilities of the `-debug_pp` option. Similar to the `-debug_pp` option, you can use the `-debug` option to set value and time breakpoints, but not line breakpoints.

`-debug_all`

Gives the most visibility/control. You can use this option typically for debugging with interactive simulation. This option provides the same capabilities as the `-debug` option, in addition it adds simulation line stepping and allows you to track the simulation line-by-line and setting breakpoints within the source code. With this option, you can set all types of breakpoints (line, time, value, event, and so on).

`-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, and enables reduced debug capabilities when compared to `-debug_pp`.

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 `-debug_access`, 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 (DUT, cell, testbench (TB), standard package (OVM, UVM, VMM, and RAL), or encrypted instances (modules, programs, packages, interfaces)).

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.

`-ucli`

During compile time, this option enables `-debug` capabilities if no debug option (`-debug`, `-debug_access`, `-debug_pp`, or `-debug_all`) is specified. Also, during runtime, this option starts `simv` in UCLI mode.

`-gui`

When used at compile time, starts DVE at runtime.

`+vpdfile+`*`filename`*

Specifies the name of the generated VPD file. You can also use this option for post-processing where it specifies the name of the VPD file.

`+vpdfileswitchsize+`*`number_in_MB`*

Specifies a size for the VPD file. When the VPD file reaches this size, VCS closes this file and opens a new one with the same 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.

## 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>`

Debug option (`-debug_pp`, `-debug`). For PLI force, you must use the `-debug` option. For language force, the minimum debug option required is `-debug_pp`.

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

- Table 5-1 describes the usage of the `-force_list` option in detail.

*Table 5-1 Usage of the -force_list Option*

| -force_list at compile time | -force_list at runtime | Language forces logged | External forces logged |
|---|---|---|---|
| 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.<br><br>Example: `top.test.child2.a` | Name of the module.<br><br>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.<br><br>Binary value is prefixed with `'b`.<br><br>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.<br><br>Binary value is prefixed with `'b`.<br><br>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 Example

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 -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
```

## Limitations

- The `-R` option is not supported.
- Language forces from encrypted code are not reported.
- VCS does not determine `$deposit()` drivers for every value change. Therefore, all value changes are recorded on the nodes that are target or 1 or more deposit statements.
- 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:

- "Passing Values from the Runtime Command Line"
- "Saving and Restarting the Simulation"
- "Specifying Long Time Before Stopping the Simulation"
- "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”

## 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
```

## 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 5-3 illustrates the basic functionality of save and restart.

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

*Example 5-3 Save and Restart Example*

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

Now compile the example source file:

```
% vcs test.v
```

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");
     #0 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 X $2^{32}$ 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 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 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 Verilog design. The following section illustrates how to resolve race conditions.

### Recommended Approach to Resolve Race Conditions

It is recommended to use the following methodology to resolve the race conditions:

- 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.

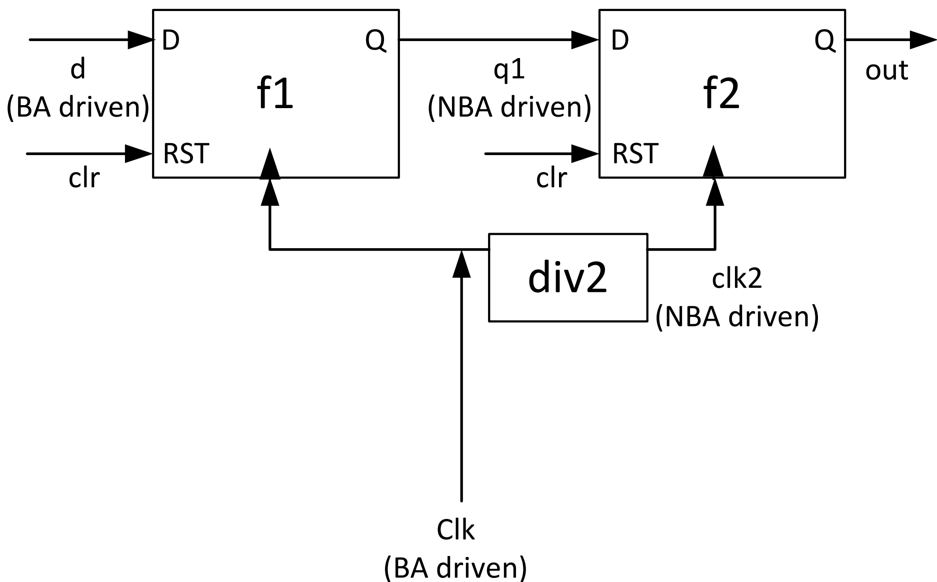
### Example

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
```

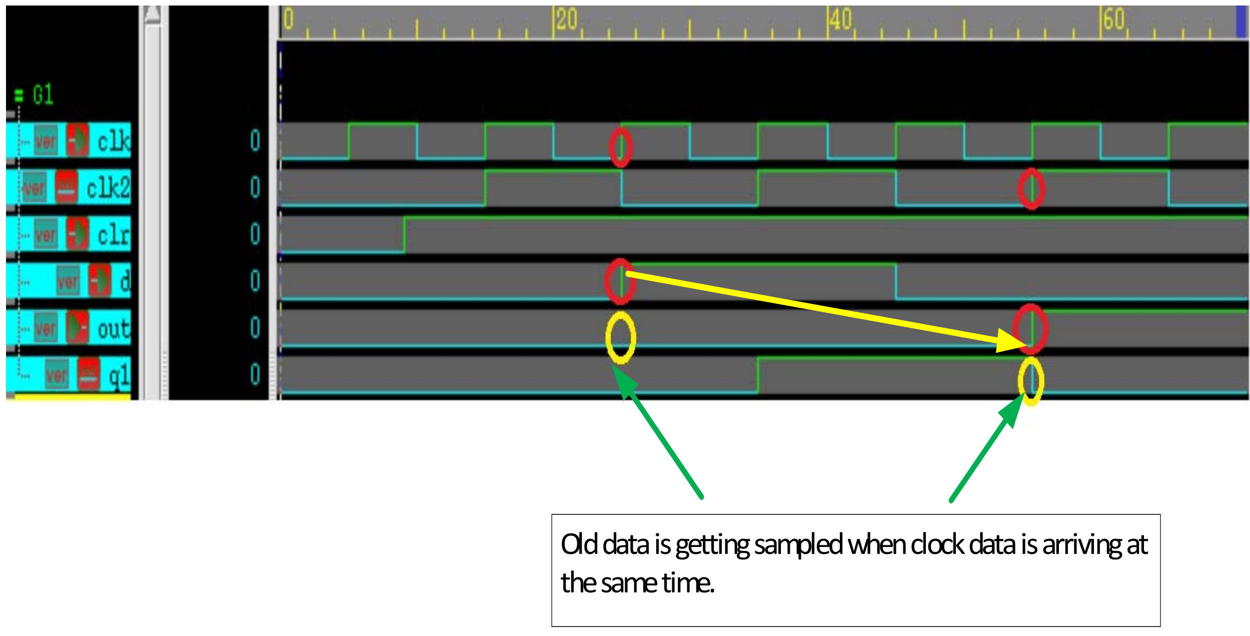
To compile the test case with FSDB dumping enabled, use the following command line:

```
% vcs -sverilog -deraceclockdata test.v  -debug_access
-debug_region=cell+lib +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 5-4 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 |

*Table 5-4 List of Scenarios and Return Error Values*

| Scenario | Return Error Value |
|---|---|
| $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 5-4 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 5-5 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 or SystemVerilog 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_rprt_default_constraints
```

This example creates the following profile report and report directory named `memory_rprt_default_constraints.html` file and `memory_rprt_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.

`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.

`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:

- 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:

```
% vcs -simprofile -P pli.tab pli.v 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:

```
% profrpt -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 | 2.97 ms | 1.56 % |
| $foo_200M | 2.97 ms | 1.56 % |
| $foo_200M /remote/vtghome11/chandrab/SIMPROFILE/Simprofile_EPL/1.v:4 | 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.

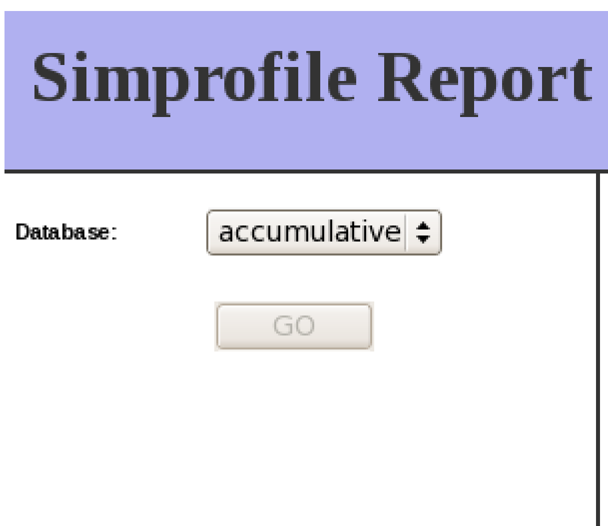
*Figure 6-3 The Right Pane of the Accumulated CPU Time View in accum_time.html*

**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% |
| **total** | **1.5 s** | **100%** |

This view shows the accumulated CPU times.

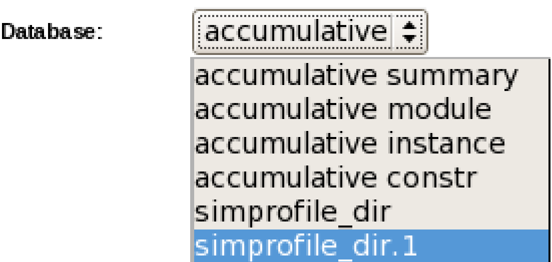
*Figure 6-4 The Left Pane of the Accumulated CPU Time Views in accum_time.html*




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.

**constr 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 -debug_pp

% simv -simprofile time -ucli -i run1

% simv -simprofile time -ucli -i run2

% profrpt -view time_all -f file
```

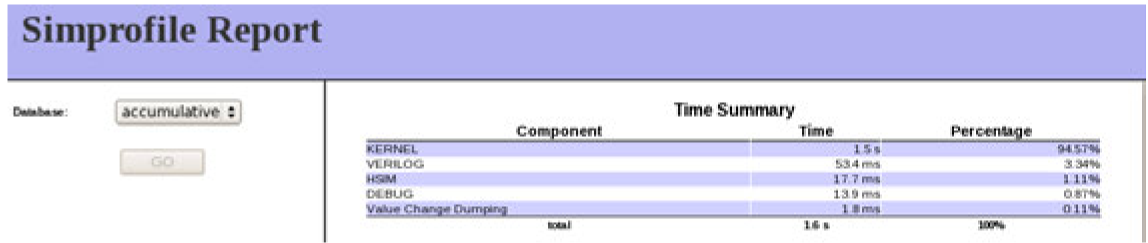
VCS writes the first profile database

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*




| Component | Time | Percentage |
|---|---|---|
| KERNEL | 1.5 s | 94.57% |
| VERILOG | 53.4 ms | 3.34% |
| HSIM | 17.7 ms | 1.11% |
| DEBUG | 13.9 ms | 0.87% |
| Value Change Dumping | 1.8 ms | 0.11% |
| total | 1.6 s | 100% |

*Figure 6-7 The Accumulative Module View*

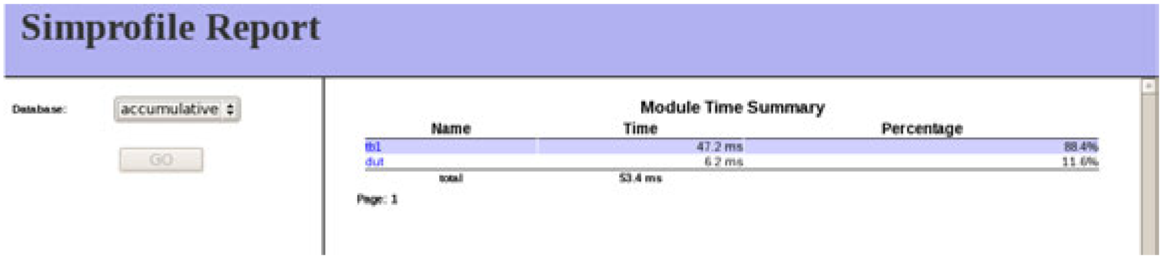



*Figure 6-8 The Accumulative Instance View*

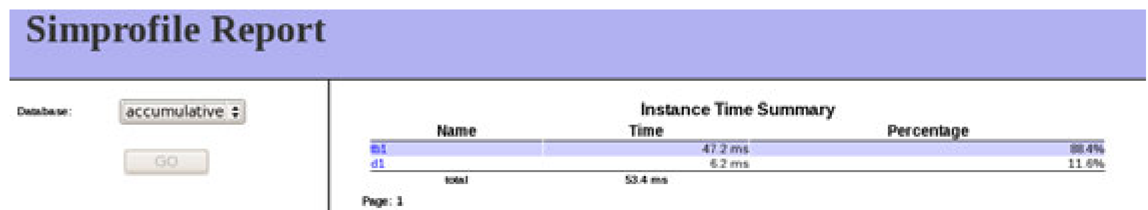
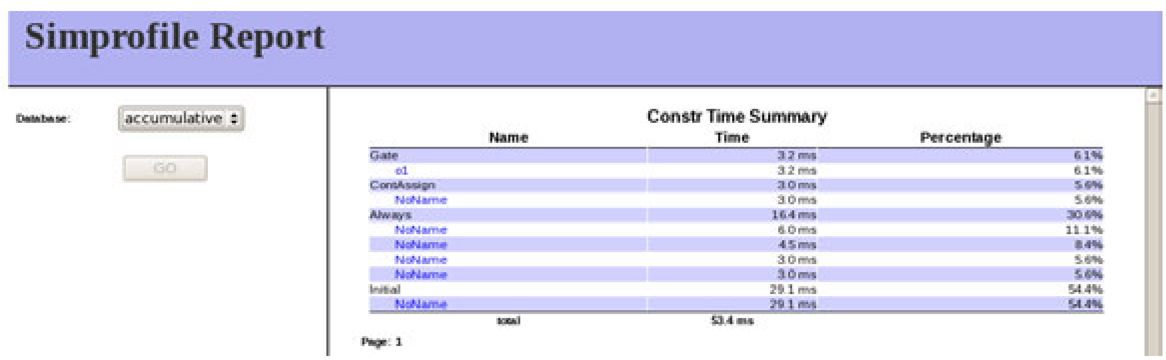



*Figure 6-9 The Accumulative Construct View*




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 |

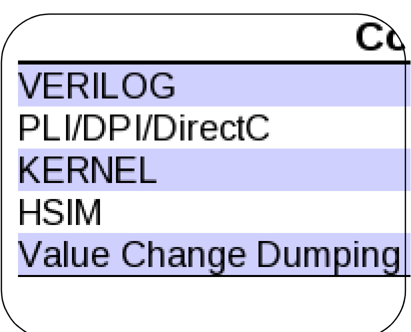
**Dynamic Summary**

| 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*




| 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 |
| **114.4 s** | **117.1 s** | **-2.7 s** |

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.

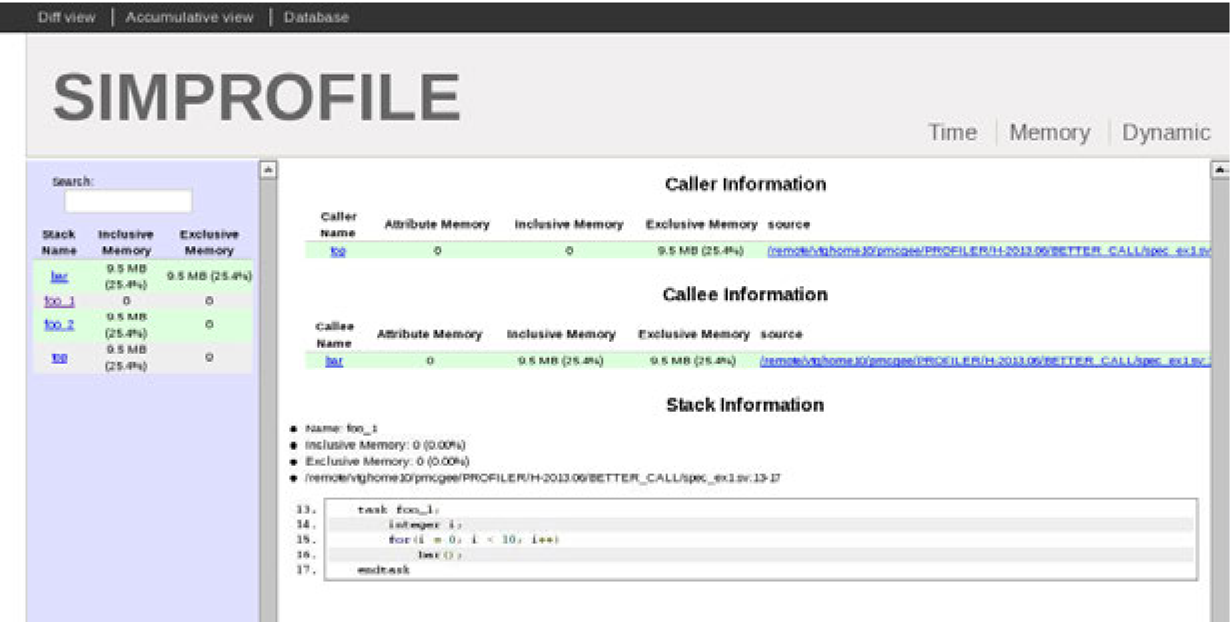
## The Caller-Callee Views

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 | 22.11 MB | 34.01 % |
| bar | 21.83 MB | 33.59 % |
| **total** | **22.11 MB** | **34.01 %** |

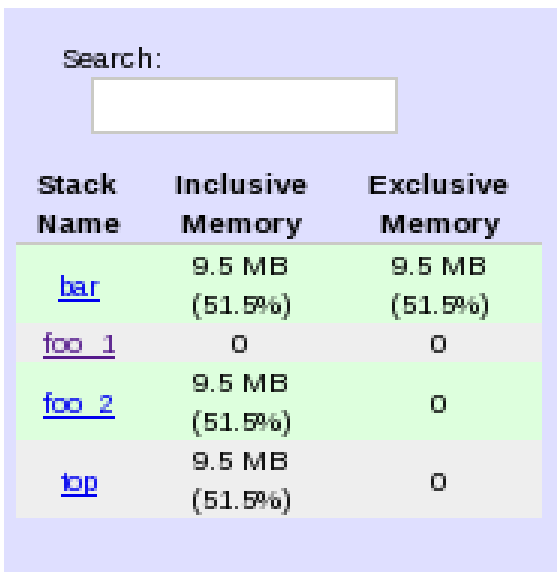
Page: 1

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*




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.

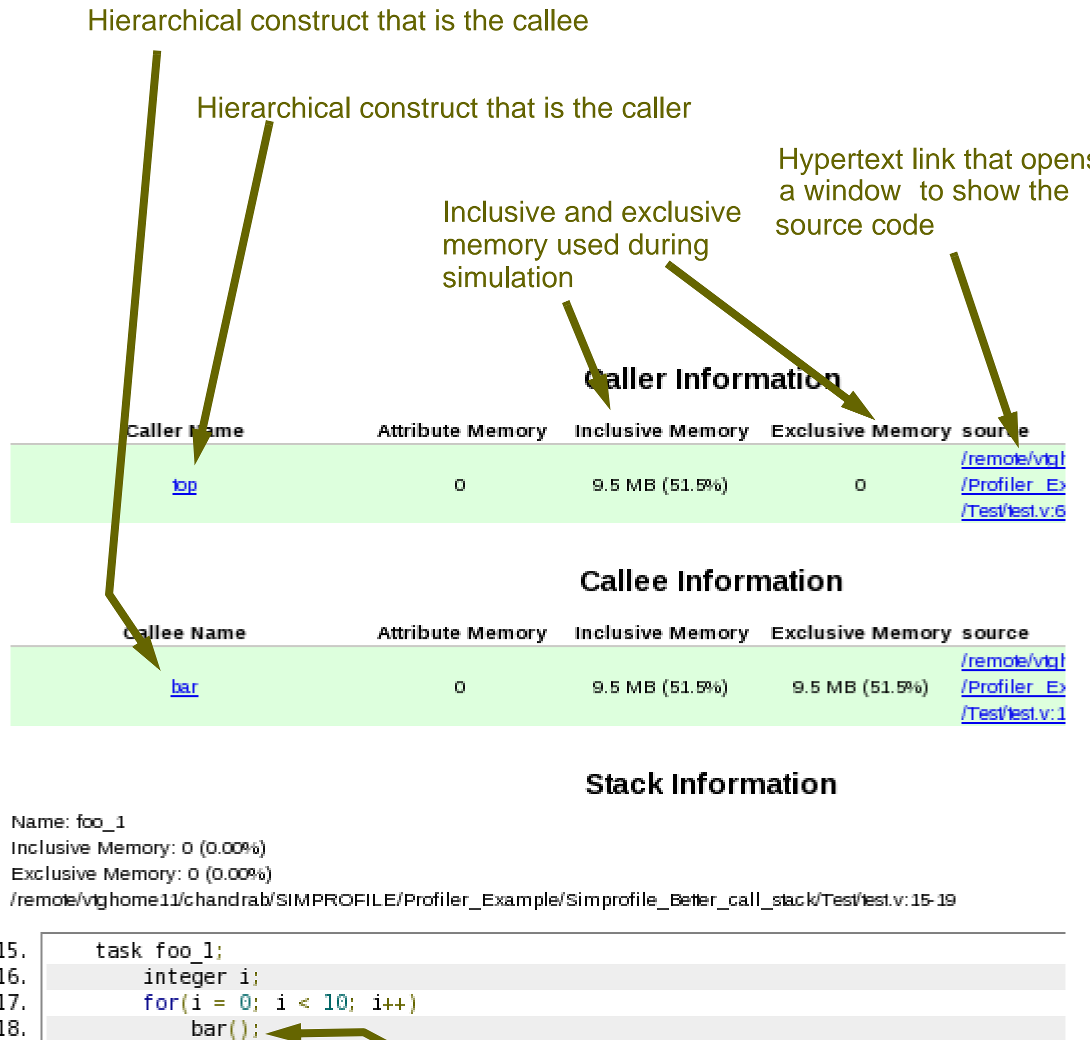
Task bar is a callee, tasks `foo_1`, `foo_2`, and module top are callers.

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:

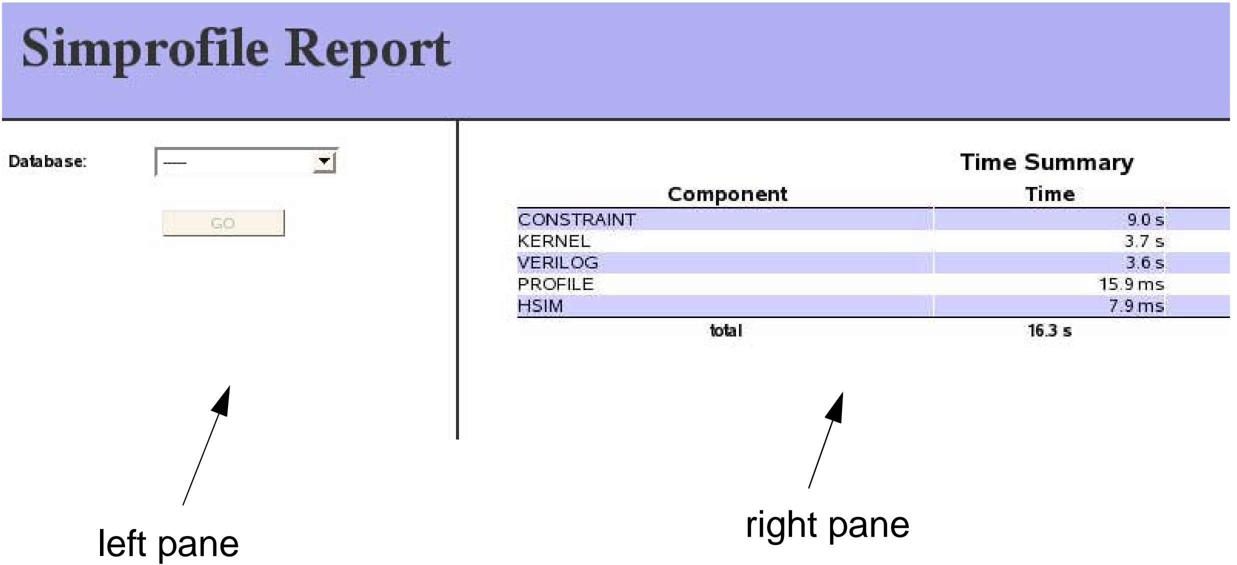
```
vcs smart_queue.v -simprofile -sverilog
```

```
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*

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*

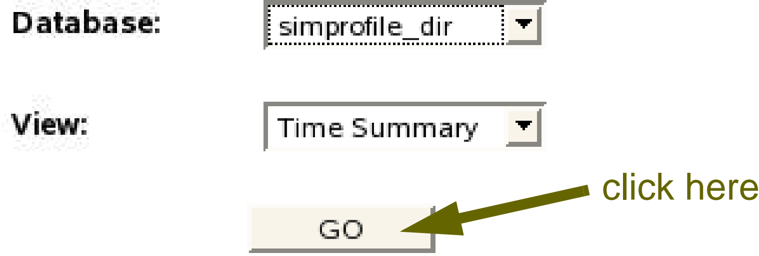


*Figure 6-19 The Right Pane of the simprofileReport.html file for CPU Time Summary Information*

**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 % |
| **total** | **16.30 s** | **100%** |

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, 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 DVE and the UCLI or to write a simulation history VCD or VPD 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.

PLI/DPI/DirectC

The CPU time needed by VCS to simulate the C/C++ in a PLI, DPI, or DirectC application.

HSIM

This is about the CPU time used by HSOPT optimizations.

COVERAGE

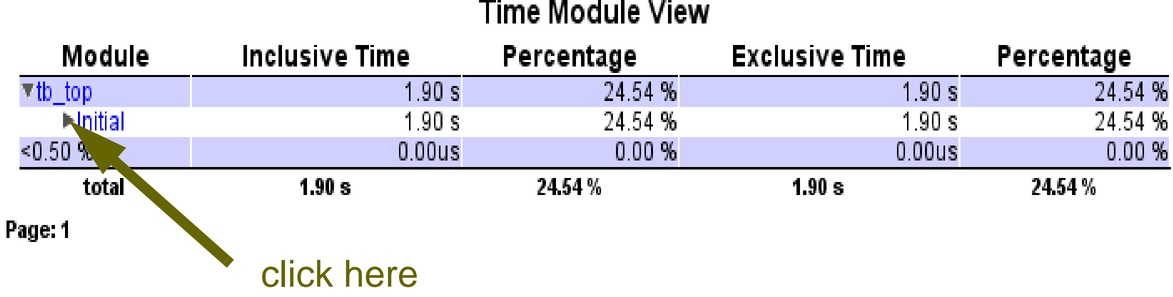
The CPU time needed for functional coverage (testbench and assertion coverage). Code coverage is not part of this component.

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

As explained earlier, modules not only include Verilog and SystemVerilog modules, but can also include SystemVerilog programs and interfaces,.

*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.

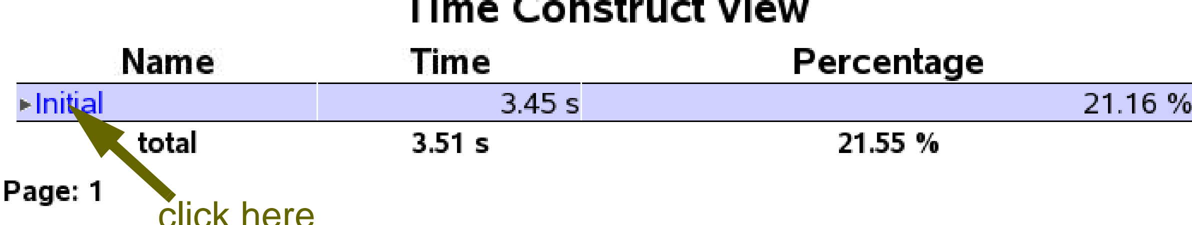
The program name is a hypertext link to expand the display in this view. If you click the hypertext link, you can see the scopes inside the program block.

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*




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.

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.

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.

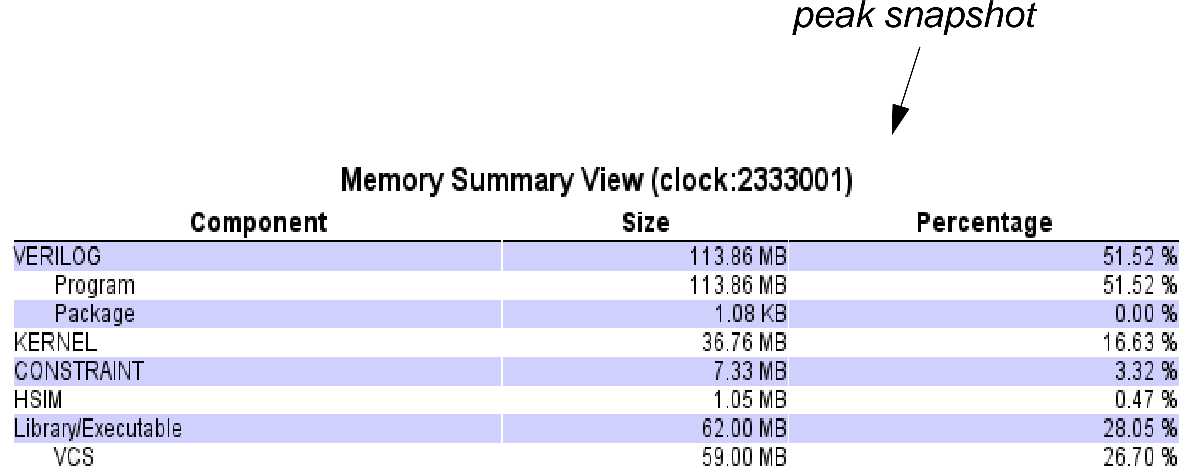
In the left pane, in the **Database** field, select the simprofile_dir profile database directory again. This adds the following fields to the left pane:

- 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** | **221.00 MB** | **100%** |

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 HSOPT optimizations.

### 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 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 DVE and the UCLI or to write a simulation history VCD or VPD 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.

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*

Database: simprofile_dir

View: Memory Sumr

Snapshot: #10 (clock:193

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*

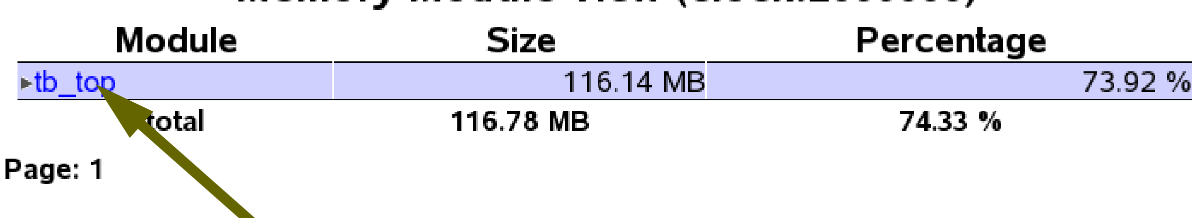
**Memory Summary View (clock:193131)**

| 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*




As explained earlier, modules not only include Verilog and SystemVerilog modules, but can also include SystemVerilog programs and interfaces.

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 % |
| **total** | **116.78 MB** | **74.33 %** |

**Page: 1**

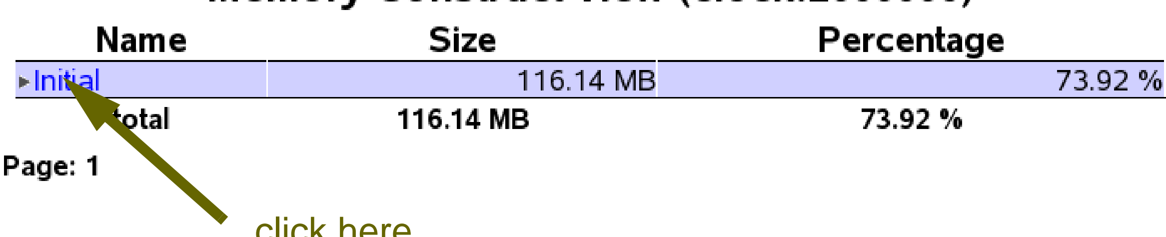
In this example view, the scope inside the program block is a begin-end block that is not named. Therefore, `profrpt` calls it NoName. This begin-end block uses most of the machine memory used by the program block. In this example view, NoName is not a hypertext link.

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*




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.

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 % |
| **total** | **116.14 MB** | **73.92 %** |

**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 % |
| **total** | **116.16 MB** | **73.93 %** | **116.16 MB** | **73.93 %** |

**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.

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.

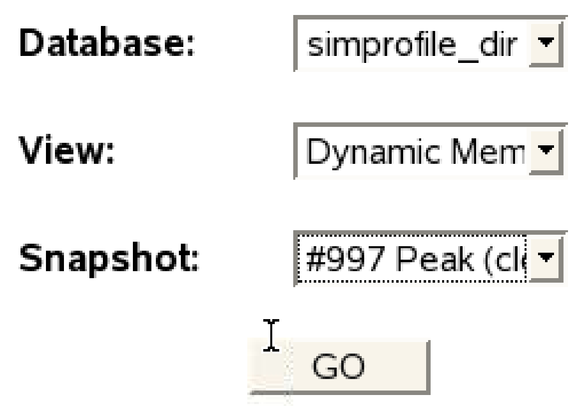
The instance name `tb_top` is not a hypertext link.

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 the this queue.

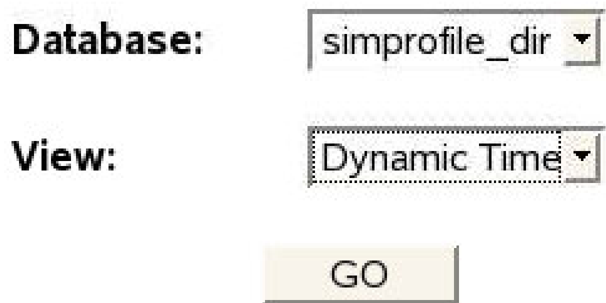
The `profrpt` profile report generator cannot report the number of instances of this queue.

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 4**

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();
```

```
        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 |
|---|---|---|---|---|
| ▼_global_ | 1.23 s | 2.65 % | 1.23 s | 2.65 % |
| ▼Function | 1.23 s | 2.65 % | 1.23 s | 2.65 % |
| ▼disp_count | 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 | disp_count |
| Time | 1.23 s |
| Constuct Type | Function |
| Parent Module | _global_ |
| Source Information | /TESTS/vcs1412/key_acct/8439_simprofile_task_fn/test.v:14-20 |

## 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.

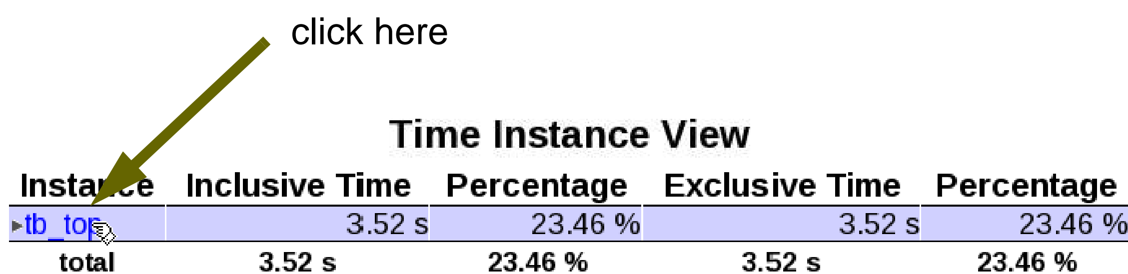
Note:

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 `profrpt` 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*




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 | 23.46 % |
| Inclusive Time | 3.52 s |
| Inclusive Percentage | 23.46 % |
| Master Module | tb_top |
| Child Instance Number | 0 |
| Source Information | /file_system/big_design/VCS_user_files/smart_queue.v:2 |

click here

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*

```
about:blank - /remote/vtghome10/pmcgee/2012.09/KEY-ACCOUNTS/1760_hyp_link_to_source_files/smart_queue.v
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
13 initial
14   begin
15
16     for(int y = 0 ; y < 1000 ; y++)
17       begin
18         PACKET packet_inst;
19
20         packet_inst = new();
21         packet_inst.randomize();
22          #1;
23
24         Squeue_data_info.push_back(packet_inst.packet_val);
25          #1;
26
27       end
28
29 repeat(10)
30   $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: be1
          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 % |
| **total** | **362.90 s** | **100%** |

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 that 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 % |
| **total** | **282.53 s** | **77.85 %** |

**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 % |
| **total** | **282.53 s** | **77.85 %** | **282.53 s** | **77.85 %** |

**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 |
|---|---|---|---|---|
| ▾sv_top | 282.53 s | 77.85 % | 2.00 ms | 0.00 % |
| ▸sv_mod_inst | 282.53 s | 77.85 % | 282.53 s | 77.85 % |
| ▸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 % |
| **total** | **282.53 s** | **77.85 %** | **282.53 s** | **77.85 %** |

**Page: 1**

*Figure 6-49 The PLI/DPI/DirectC View*

**Time PLI/DPI/DirectC View**

| Name | Time | Percentage |
|---|---|---|
| PLI | 999.89us | 0.00 % |
| $sc_mod_init | 999.89us | 0.00 % |
| $vcdplusfilter | 0.00us | 0.00 % |
| $msglog | 0.00us | 0.00 % |
| $lsi_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 % |
| **total** | **0.00us** | **0.00 %** |

**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. 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 orSystemVerilog, 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:

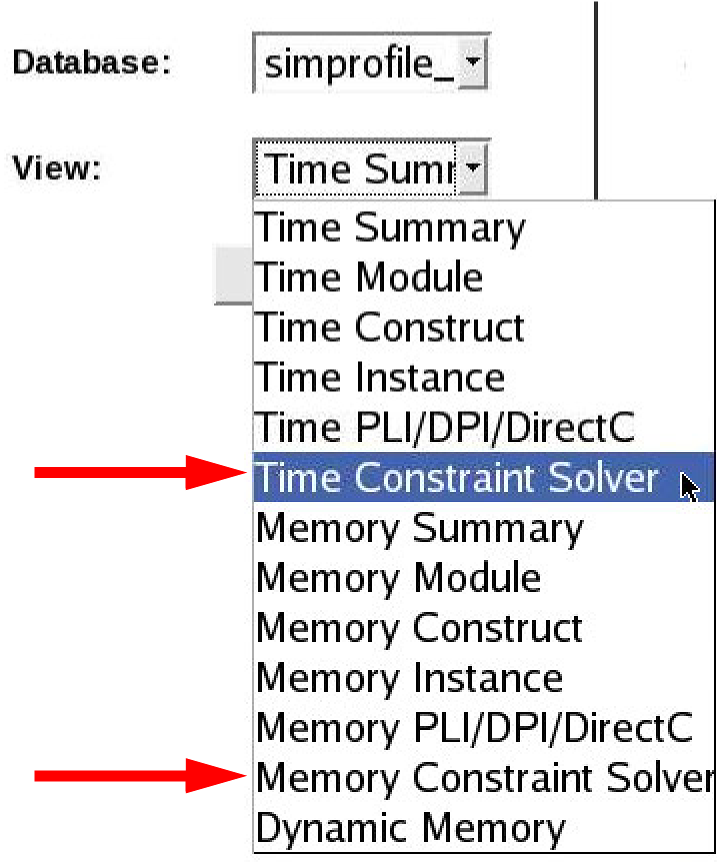
- `time_solver` to specify generating the Time Constraint Solver view
- `mem_solver` to specify generating the Memory Constraint Solver view.

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 |
|---|---|---|---|---|---|
| ./env/nvs_atapi_env.sv:118@1 | 1 | 0.030 | 27 | 37 | 9 |
| ./env/nvs_atapi_env.sv:120@1 | 2 | 0.000 | 3 | 3 | 1 |

### Top randomize calls based on cumulative cpu runtime

| File:line | calls | time (sec) |
|---|---|---|
| ./env/nvs_atapi_env.sv:118 | 1 | 0.030 |
| ./env/nvs_atapi_env.sv:120 | 1 | 0.000 |

### Top partitions based on cpu time

| File:line@visit | Rand.Partition | cpu time (sec) | variables | constraints | cnst blocks |
|---|---|---|---|---|---|
| ./env/nvs_atapi_env.sv:118@1 | 1.1 | 0.03 | 2 | 4 | 2 |
| ./env/nvs_atapi_env.sv:118@1 | 1.2 | 0.00 | 1 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.3 | 0.00 | 7 | 12 | 3 |
| ./env/nvs_atapi_env.sv:118@1 | 1.4 | 0.00 | 5 | 10 | 3 |
| ./env/nvs_atapi_env.sv:118@1 | 1.5 | 0.00 | 2 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.6 | 0.00 | 1 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.7 | 0.00 | 1 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.8 | 0.00 | 2 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.9 | 0.00 | 2 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 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 |
|---|---|---|---|---|---|
| ./env/nvs_atapi_env.sv:118@1 | 1 | 0.030 | 27 | 37 | 9 |
| ./env/nvs_atapi_env.sv:120@1 | 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) |
|---|---|---|
| ./env/nvs_atapi_env.sv:118 | 1 | 0.030 |
| ./env/nvs_atapi_env.sv:120 | 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 (sec) | variables | constraints | cnst blocks |
|---|---|---|---|---|---|
| ./env/nvs_atapi_env.sv:118@1 | 1.1 | 0.03 | 2 | 4 | 2 |
| ./env/nvs_atapi_env.sv:118@1 | 1.2 | 0.00 | 1 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.3 | 0.00 | 7 | 12 | 3 |
| ./env/nvs_atapi_env.sv:118@1 | 1.4 | 0.00 | 5 | 10 | 3 |
| ./env/nvs_atapi_env.sv:118@1 | 1.5 | 0.00 | 2 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.6 | 0.00 | 1 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.7 | 0.00 | 1 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.8 | 0.00 | 2 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 1.9 | 0.00 | 2 | 1 | 1 |
| ./env/nvs_atapi_env.sv:118@1 | 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 `randmomize()` 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**

hypertext link

| 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 |

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) |
|---|---|---|
| ./env/nvs_atapi_env.sv:118 | 1 | 640 |
| ./env/nvs_atapi_env.sv:120 | 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 in 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. In Figure 6-63, 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*

Time Module View

| Module | 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 | 8.68 s | 62.14 % |
| cg | 8.68 s | 62.14 % |
| ▼CoverPoint | 3.16 s | 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 | 832.51 ms | 5.96 % |
| cg_cc_0 | 428.23 ms | 3.07 % |
| cg_cc | 404.28 ms | 2.89 % |
| ▼Initial | 233.58 ms | 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 % | /remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example/Simprofile_Fcov_Support/Test/test.v:4-17 |
| cpt1_cp(CoverPoint) | 965.72 ms | 6.91 % | /remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example/Simprofile_Fcov_Support/Test/test.v:5-5 |
| cpt4_cp(CoverPoint) | 746.79 ms | 5.35 % | /remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example/Simprofile_Fcov_Support/Test/test.v:11-11 |
| cpt3_cp(CoverPoint) | 731.79 ms | 5.24 % | /remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example/Simprofile_Fcov_Support/Test/test.v:10-10 |
| cpt2_cp(CoverPoint) | 716.79 ms | 5.13 % | /remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example/Simprofile_Fcov_Support/Test/test.v:9-9 |
| cg_cc_0(CoverCross) | 428.88 ms | 3.07 % | /remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example/Simprofile_Fcov_Support/Test/test.v:16-16 |
| cg_cc(CoverCross) | 404.88 ms | 2.90 % | /remote/vtghome11/chandrab/SIMPROFILE/Profiler_Example/Simprofile_Fcov_Support/Test/test.v:15-15 |
| **total** | **8.69 s** | **62.24 %** | |

**Page: 1**

# Limitations

The following technologies are not supported in the unified profiler:

- Multicore — Both for Application Level Parallelism (ALP) and Design Level Parallelism (DLP is an LCA feature).
- 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 following are the methods that can enable the debug capabilities:

- `.tab` files
- `-debug` flags
- `+acc` flags

The ACC 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>
  ```

- `Profpt`

  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.

**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 | 7.17 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**

**show full module list**

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 capabilities 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 capabilities 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 indicates 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*

Full Module View

| Module | Exclusive Time | read | read_write | callback | callback_all | force | static |
|---|---|---|---|---|---|---|---|
| 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*

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,

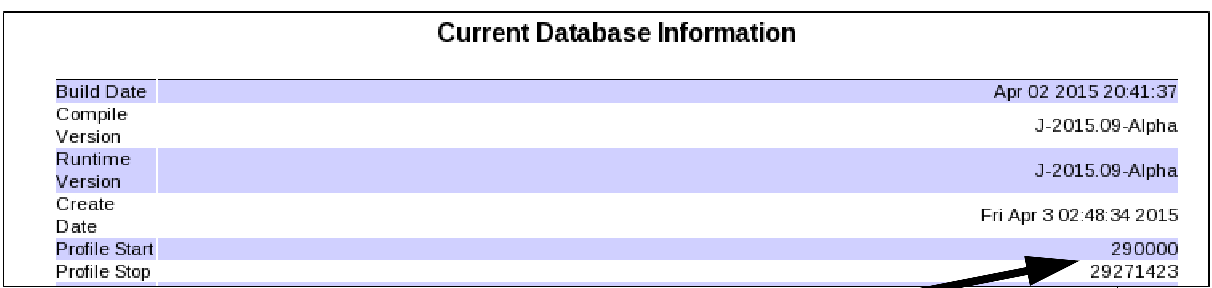
```
-start <time>
```

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 Version | J-2015.09-Alpha |
| Runtime Version | J-2015.09-Alpha |
| Create Date | Fri Apr 3 02:48:34 2015 |
| Profile Start | 290000 |
| Profile Stop | 29271423 |

All individual profile report contains the profile start and profile 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:                              Apr 02 2015 20:41:37
ompiler version:                              J-2015.09-Alpha
Runtime version:                              J-2015.09-Alpha
   Machine Name:                              vgintsb56
 Profile runner:                              meenar
   Profile data:                              /remote/vtghome12/meenar/SIMPROFILE/time_slice_based
                                              ofiler/BM_sel/WB_DMA_SVD/simprofile_dir
  Creation date:                              Fri Apr  3 03:07:05 2015
  Profile start:                              291000
   Profile stop:                              29270000
```

Profile start time and stop time is profrpt time.
All individual reports is 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

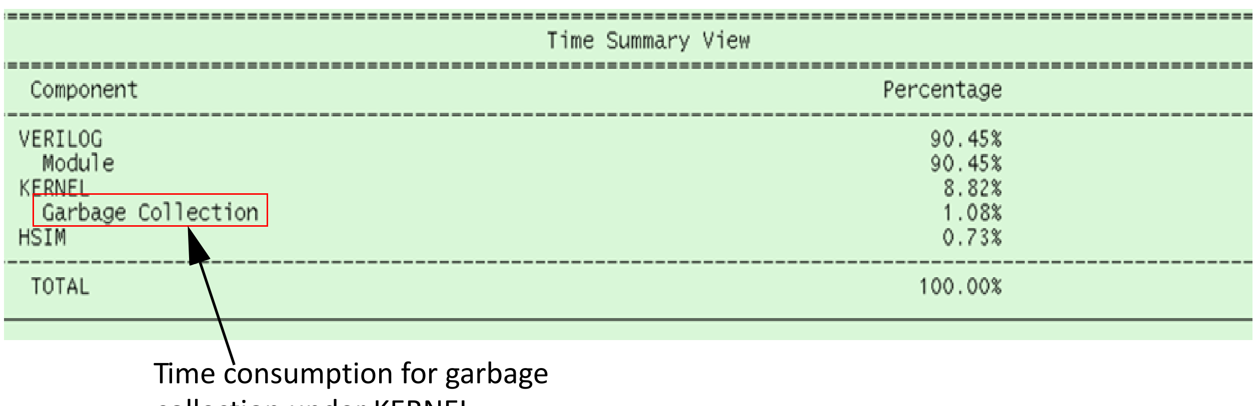
VCS isolates the CPU time consumed by the garbage collection.

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% |



# 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.

# 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”

# Using Diagnostics

This section describes the following topics:

- "Using `-diag` Option"
- "Using Smartlog"

## 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 "Libconfig Diagnostics" . |
| `timescale` | `vcs -diag timescale` | Enables timescale diagnostics. For more information, see "Timescale Diagnostics" . |
| `vpi` | `simv -diag vpi` | Enables VPI diagnostics. For more information, see "Diagnostics for VPI PLI Applications" . |
| `all` | `vcs -diag all` | Enables the `libconfig` and `timescale` diagnostics. |
| | `simv -diag all` | Enables the `vpi` diagnostics. |
| `help` | `vcs -diag help`<br><br>`simv -diag help` | Displays the following help message:<br>`Usage for -diag flag:` **`-diag <option>,<option>,...`**<br>`Options:`<br>**`all`** `Enable all diagnostics`<br>**`help`** `Display this message`<br>**`libconfig`** `Library binding diagnostics (compile time)`<br>**`timescale`** `Timescale diagnostics (compile time)`<br>**`vpi`** `VPI diagnostics (simulation time)`<br>**`vhpi`** `VHPI diagnostics (simulation time)` |

## Using Smartlog

DVE Smartlog provides log analysis (diagnostic information) for each line in the log file. It takes the compile log and simulation log created by VCS and summarizes the data into reports. Smartlog provides the diagnostic information in a separate log file called smartlog file. Following are the main features of Smartlog:

- Hyperlink the log occurrences to the Source View
- Highlight the words, namely, Error, Warning, and so on, in different colors
- Display the selected message within a blue rectangle

For more information, refer to the *Using Smartlog* section of the *Discovery Visualization Environment User Guide* category in the VCS Online Documentation.

## Compile-time Diagnostics

This section describes the following topics:

- "Libconfig Diagnostics"
- "Timescale Diagnostics"

### 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 Verilog instance, this option generates the instance name, location, binding rule, and entity-architecture pair/module to which it is bound.

Note:

- If VCS option `-l` is specified, the output is dumped into the corresponding text log file.
- If VCS option `-sml` is also specified, smart log output is also dumped into the corresponding smart log file. For more information, refer to the *Using Smartlog* section in the *Discovery Visualization Environment User Guide* category in the VCS Online Documentation.

## 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 VCS option `-l` is specified, the output is dumped into the corresponding text log file.
- If VCS option `-sml` is also specified, smart log information is dumped into the corresponding smart log file. For more information, refer to the *Using Smartlog* section in the *Discovery Visualization Environment User Guide* category in the VCS Online Documentation.

## 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/ll_svb/
Source/test.v', 1
module 'test' gets time precision '1ns' from source code '/
remote/vgscratch7/timescale_diag/tests/cft/sva_bind/
ll_svb/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/ll_svb/
Source/test.v', 1
module 'test' gets time precision '1ns' from source code '/
remote/vgscratch7/timescale_diag/tests/cft/sva_bind/
ll_svb/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
```

## Runtime Diagnostics

This section describes the following topics:

- "Diagnostics for VPI PLI Applications"
- "Keeping the UCLI/DVE Prompt Active After a Runtime Error"
- "Diagnosing Quickthread Issues"

### Diagnostics for VPI PLI Applications

As per LRM, VPI 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 diagnostic option to make the PLI application to report errors without code modification:

- `-diag vpi`

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 VCS option `-l` is specified, the output is dumped into the corresponding text log file.

- If the `-sml` option is also specified, smart log information is dumped into the corresponding smart log file. For more information, refer to the *Using Smartlog* section in the *Discovery Visualization Environment User Guide* category in the VCS Online Documentation.

For example, consider the following test case `tokens.v` and files `value.tab` and `value.c`.

*Example 7-1 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 7-2   value.tab*

```
$putValue  call=put_value  acc=rw:top
```

*Example 7-3   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 the `tokens.v` code shown in Example 7-1, as follows:

```
% vcs -sverilog +vpi -P value.tab value.c tokens.v
```

Run the `tokens.v` code, as follows:

```
% 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/[illegible]/12-09/VPI_EM/tokens.v, 14
        Delay Propagation Method - 1
```

## Keeping the UCLI/DVE Prompt Active After a Runtime Error

VCS allows you to debug an unexpected error condition by not exiting and keeping the UCLI or DVE command prompt active for debugging commands.

DVE 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 7-5) 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 7-5).

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.

## DVE Use Model

By default, DVE enables the onfail configuration on all types of failures.

DVE systematically enables the onfail configuration on all error types. In previous versions, if there is error or failure, simv stops, and many DVE functionalities like expand hierarchy, show data for a given module (if not already loaded before the simv crash), create schematic, do not work, especially when DVE is running with the preference option “Use simulation as design debug library in interactive”.

From this version, if you enable the onfail configuration, simv stays active and continue to respond to DVE queries. Also, DVE shows the location of the error with the simulation pointer (yellow 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.

*Figure 7-1 The DVE Prompt After a Runtime Error*

## UCLI Usage Example

Consider the following test case `test.v`. This code causes simv to exit during simulation:

*Example 7-4    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:

```
% vcs -sverilog -debug_all test.v
```

If you run the above test case using the `simv -ucli` command, VCS generates the following NOA error message:

*Figure 7-2 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 7-5 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 7-3:

```
% simv -ucli -i test.tcl
```

*Figure 7-3    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 DVE.

## 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
'0x800a00000' 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 SC_CTOR. 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.

# 8

# VCS Multicore Technology Application Level Parallelism

VCS Multicore Technology takes advantage of the computing power of multiple processors or cores in one machine to improve simulation turnaround time. These cores in Multicore technology are sometimes described as consumers, processes, or threads.

With Multicore application level parallelism, you can use different cores to compile and simulate in parallel the following applications:

- SystemVerilog assertions
- Toggle coverage
- SAIF file dumping

## Enabling Multicore Technology Application Level Parallelism

You use the VCS `-parallel` compile time option to enable parallel compilation and simulation. The syntax is:

```
vcs source_files -parallel[+mulitcore_keyword_arguments]
[-o multicore_executable_name]
[other_compile-time_options]
```

These options and arguments are as follows:

`-parallel`

Enables parallel compilation and simulation for various applications.

If you omit the keyword arguments, you enable parallel compilation and simulation for all the types of the Multicore applications. If you include keyword arguments, you enable parallel compilation and simulation for only the Multicore applications that they specify.

In some Multicore applications you can specify the number of cores that VCS uses to compile for and simulate that application. You can do so if the Multicore application's keyword argument has an optional *=NCORES* argument to specify the number of cores.

If you do not enter the optional *=NCORES* argument to a keyword argument, VCS uses one core for the application.

The keyword arguments are as follows:

`+saif`

Specifies one or more cores for SAIF file dumping, see "Multicore SAIF File Dumping" .

`+tgl [=NCORES]`

Specifies one or more cores for toggle coverage.

`+show_features`

This keyword argument is not meant for enabling a Multicore application.However, it is used to instruct VCS to display the applications you enabled during compilation. The following is an example of the Multicore information displayed:

```
PVCS features:
- DLP: disabled
- Parallel SVA: disabled
- Parallel TGL: disabled
- Parallel SAIF: disabled
```

In this example DLP is Design Level Parallelism, which is an LCA feature.

You can enter more than one keyword argument using the + delimiter.For example,

```
vcs example.sv -parallel+tgl+sva -sverilog \
-debug_pp -assert hier=svafile -cm tgl+assert
```

This command line specifies parallel compilation and simulation with:

- One core for toggle coverage
- One core for SystemVerilog assertions

The `-o` compile time option is for naming the VCS executable file. Its default name is `simv`. The following vcs command line shows its use:

```
vcs tb0.sv dut.sv -parallel+tgl -o psimv0
vcs tb1.sv dut.sv -parallel+tgl -o psimv1
vcs tb2.sv dut.sv -parallel+tgl -o psimv2
```

Assigning different names to the executable in Multicore ALP enables you to run multiple simultaneous Multicore simulations. VCS MX stores Multicore-specific information in the *executable_name*`.daidir` directory.

## Multicore SAIF File Dumping

SAIF is Switching Activity Interchange Format, a file format for Power Compiler. VCS writes or dumps SAIF files for it.

If you enabled Parallel SAIF at compile-time and want to disable it at runtime, you can do so with the `-parallel+saif=0` runtime option and keyword argument.

Parallel SAIF has the following limitations:

- Parallel SAIF is not implemented for VCS Multicore Design Level Parallelism (DLP).
- Parallel SAIF only works with one core, so for example specifying more results in an error condition.
- SAIF file read mode is not implemented for Multicore SAIF file dumping.

- Multiple $toggle_start system tasks are not supported in Multicore SAIF file dumping. Only full dump mode is supported, which is one $toggle_start and $toggle_stop system task. Entering multiple $toggle_start system tasks in Multicore SAIF file dumping is an error condition.

## Limitations

Multicore ALP has limitations and it does not work with the following technologies:

- Partition Compile (an LCA feature).
- Hierarchical Cross Coverage (an LCA feature).

# 9

# 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 propriety 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 `$vcdpluson` 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. This allows instant replay of source execution in the DVE Source View.

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 DVE - See the *Discovery Visualization Environment User Guide*.

## Using System Tasks

VCS provides Verilog system tasks to:

- "Enable and Disable Dumping"
- "Override the VPD Filename"
- "Dump Multi-Dimensional Arrays and Memories"
- "Capture Delta Cycle Information"

## Enable and Disable Dumping

You can use the `$vcdpluson` and `$vcdplusoff` 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
$vcdpluson;

...
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
$vcdpluson(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`.

### $vcdplusoff

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

The following is the syntax of the `vcdplusoff` system task:

```
$vcdplusoff (level|”LVL=integer”,scope*,signal*);
```

### Example 1: Turn recording off

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

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
  $vcdpluson; // Enable Dumping
  $vcdplusoff(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 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 3x3x3 (27) locations; each location is 8 bits in length, as shown in Figure 9-1.

### Example: To dump all elements to the VPD File

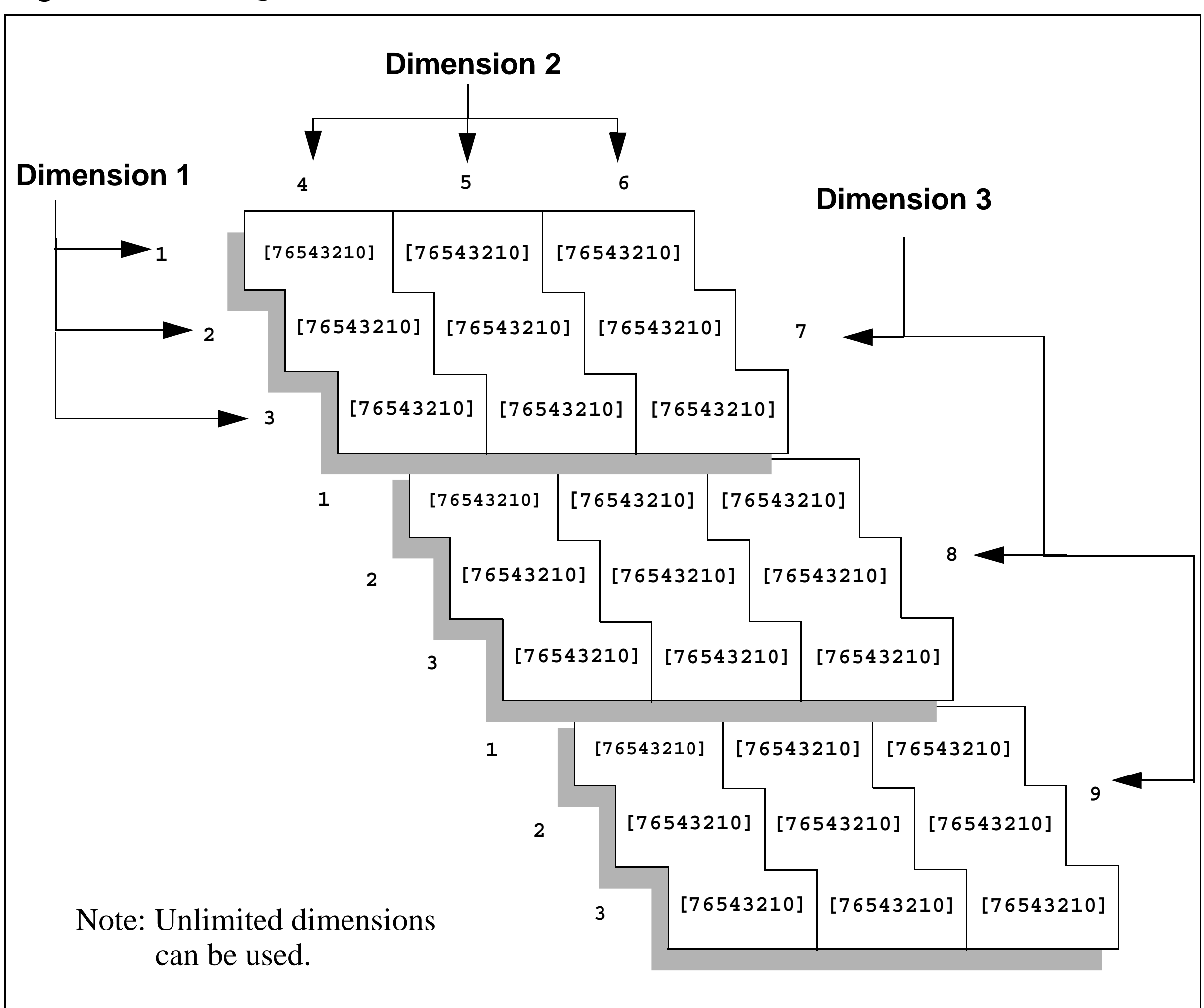
Consider the following example code:

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



## Using $vcdplusmemorydump System Task

The `$vcdplusmemorydump` 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 `$vcdplusmemorydump` commands, but they must occur in the same simulation time. In subsequent simulation times, `$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`, DVE allows explicit control of glitch detection.

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`, DVE allows explicit control of glitch detection.

**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

## 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,]"fil
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+pp 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 will 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.

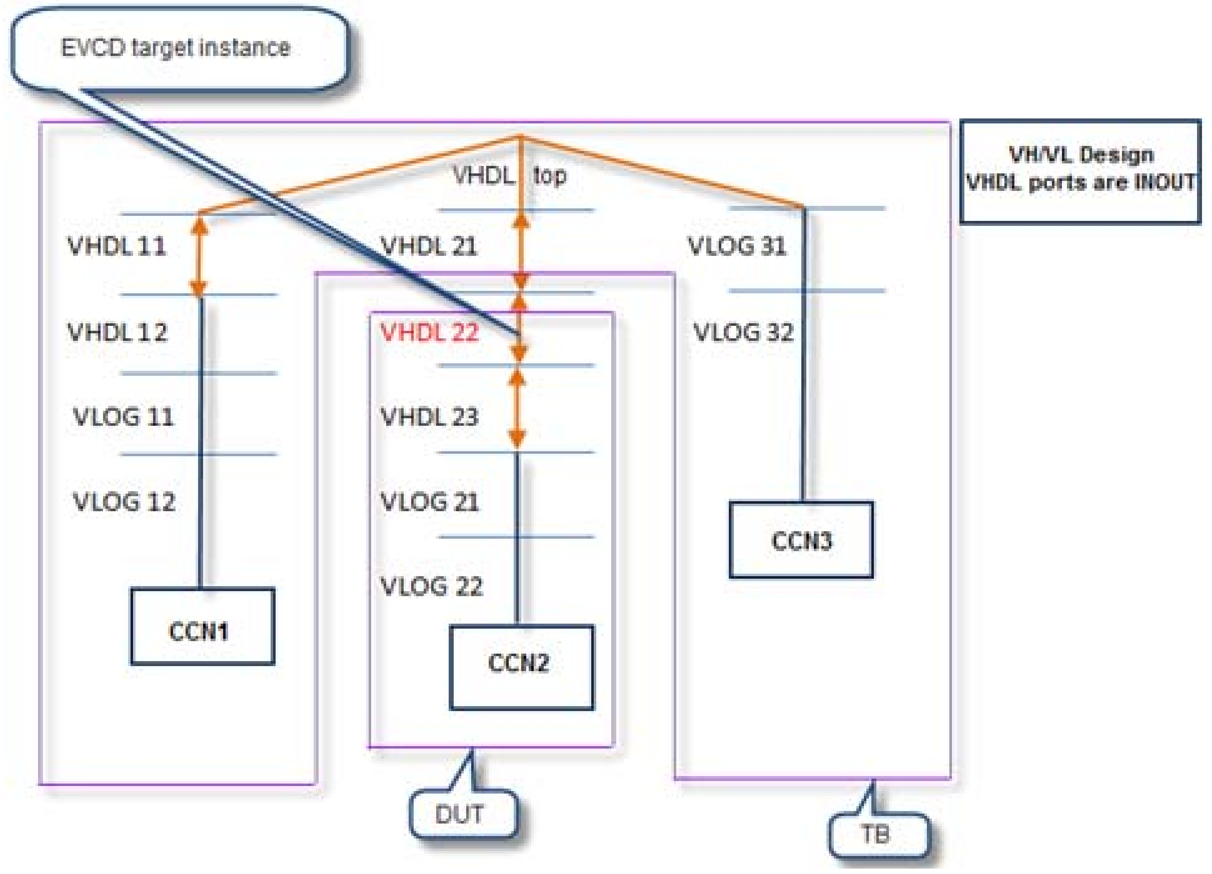
## Use Model for Dumping CCN Driver Through INOUT

EVCD file contains the CCN driver when the CCN is connected through INPUT or OUTPUT ports in Verilog-VHDL or VHDL-Verilog MX 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_pp
  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.

**Unsupported Driver Types**

- Since tran gates divide a net into different segments, the EVCD behavior might be different in presence of XMR drivers.
- `$deposit` and `force -deposit` (UCLI command) associated with EVCD port are not supported. They are not true drivers, and LRM is silent about the intended behavior.
- If drivers of port are in encrypted region, they are ignored.

- Drivers through virtual interface/nested interface and so on, are not supported.
- High-conn logical expressions are not supported.
- For Verilog-VHDL-Verilog and VHDL-Verilog-VHDL topology designs, tran gate drivers are not supported.

### 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.

  Note:

  - All forces are considered as TB regardless of where the force is applied from (TB, DUT, or UCLI).
  - EVCD port associated with SDF timing may not be properly handled. LRM does not specify how the delay has to be handled for various scenarios (whether to add delay on driver side for EVCD and so on).

    In case of SDF, value is not same for different net segments of the same net (there is a delay) and whether they should be treated as the same net or different net for EVCD purpose. Current behavior is all net segments are treated as part of the same net, all drivers are reported, and driver value change is reported as it occurs in core simulation.

## 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 uniquifies the identifier codes.

`+unique`

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

`input_VCD_file`

The name of the VCD file created by VCS.

`output_VCD_file`

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

## The vcdiff Utility

The vcdiff utility compares two dump files and reports any differences it finds. The dump file can be of type VCD, EVCD, or 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] [-ignoreregs]
[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.

`-ignoreregs`

Does not report value change differences on signals that are of type register.

`-ignorewires`

Does not report value change differences on signals that are of type wire.

`-ignorereals`

Does not report value change differences on signals that are of type real.

-ignorefunctaskvars

Does not report value change differences on signals that are function or task variables.

**-ignorestrength** (EVCD only)

EVCD files contain a richer set of signal strength and directional information than VCD or even VPD files. This option ignores the strength portion of a signal value when checking for differences.

**-compare01xz** (EVCD only)

Converts all signal state information to equivalent 4-state values (0, 1, x, z) before 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 <)

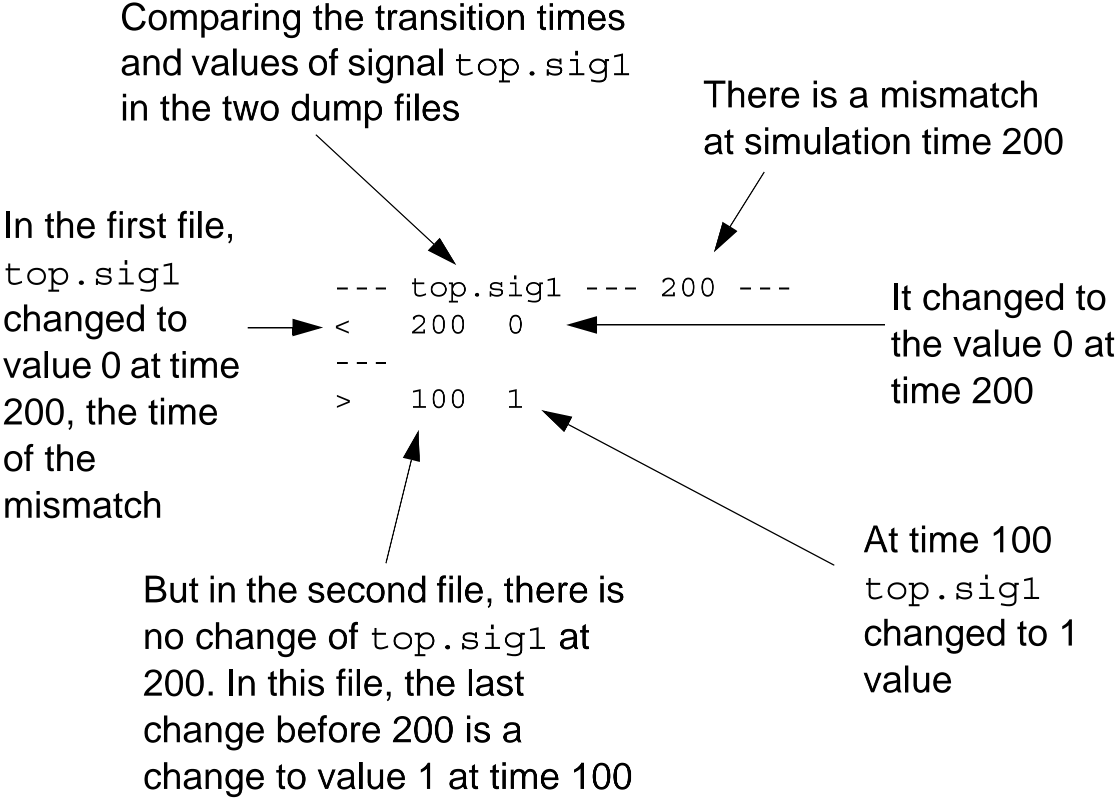
Contains the time of the last value change of the signal (at or before the time of the mismatch) in the first dump file on the `vcdiff` command line.

> (line beginning with >)

Contains the corresponding information in the second dump file on the `vcdiff` command line.

Figure 9-2 shows the Callout notes on the different names and values in an example difference.

*Figure 9-2    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 `vcdiff` to compare the results.

Module generation enables you to use much faster simulating "canned" modules for a part of the design to enable the faster simulation of other parts of the design that need investigation.

The name of the generated source file from testbench generation begins with testbench followed by the module and instance names in the hierarchical name of the module instance, separated by underscores. For example `testbench_top_ad1.v`.

Similarly, the name of the generated source file from module generation begins with `moduleGeneration` followed by the module and instance names in the hierarchical name of the module instance, separated by underscores. For example `moduleGeneration_top_ad1.v`.

You enable `vcat` to generate these files by doing the following:

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

## Writing the Configuration File

The configuration file is named `vgen.cfg` by default, and `vcat` looks for it in the current directory. This file needs three types of information specified in the following order:

1. The hierarchical name of the module instance.
2. Specification of testbench generation with the keyword `testbench` or specification of module generation with the keyword `moduleGeneration`.
3. The module header and the port declarations from the module definition of the module instance.

You can use Verilog comments in the configuration file.

The following is an example of a configuration file:

*Example 9-1    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 9-2    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 9-1 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 file are assigned directly to the output port.

## The vcsplit Utility

The `vcsplit` utility generates a VCD, EVCD, or VPD file that contains a selected subset of value changes found in a given input VCD, EVCD, or VPD file (the output file has the same type as the

input file). You can select the scopes/signals to be included in the generated file either via a command-line argument, or a separate "`include`" file.

## The vcsplit Utility Syntax

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 `$vcdpluson` or UCLI dump commands to a VCD or EVCD file.

The syntax is as shown below:

```
vpd2vcd [-h] [-q] [-s] [-x] [-xlrm] [+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.

`-xlrm`

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.

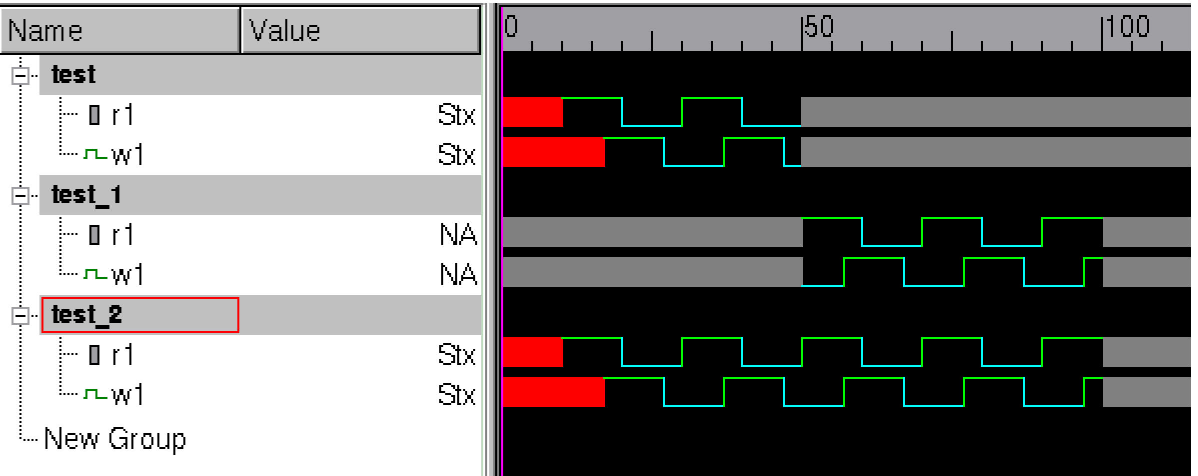
**Limitations**

- `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. For example, in the DVE Wave View in Figure 9-3, there are three signal groups for the same signals in different VPD files.

*Figure 9-3 DVE Wave View with Signal Groups from Different VPD Files*




Signal group `test` is from a VPD file from the first half of a simulation, signal group `test_1` is from a VPD file for the second half of a simulation, and signal group `test_2` is from the merged VPD file.

The syntax is as shown below:

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

Usage:

`-h`

Displays a list of the valid options and their purpose.

`-o` *`merged_VPD_filenames`*

Specifies the name of the output merged VPD file. This option is required.

`-q`

Specifies quiet mode. Disables the display of most output to the terminal.

`-hier`

Specifies 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:

- To read the merged VPD file, DVE must have the same or later version than that of the `vpdmerge` utility.
- VCS must have written the input VPD files on the same platform as the `vpdmerge` utility.

- The input VPD files cannot contain delta cycle data (different values for a signal during the same time step).
- The input VPD files cannot contain named events.
- The merged line stepping data does not always accurately replay scope changes within a time step.
- If you are merging VPD files from different parts of the design using the `-hier` option, the VPD files must be used for distinctly different parts of the design, they cannot contain information for the same scope.
- 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” .

# 10

# Performance Tuning

VCS delivers the best performance during both compile-time and runtime by reducing the size of the simulation executable, and the amount of memory consumed for compilation and simulation. By default, it is optimized for the following types of designs:

- Designs with many layers of hierarchy
- Gate-level designs
- Structural RTL-level designs - Using libraries where the cells are RTL-level code
- Designs with extensive use of timing such as delays, timing checks, and SDF back annotation, particularly to INTERCONNECT delays

However, depending on the phase of your design cycle, you can fine-tune VCS for a better compile-time and runtime performance.

This chapter describes the following sections:

- Compile-time Performance

  Compile-time performance plays a very important role when you are in the initial phase of your design development cycle. In this phase, you may want to modify and recompile the design to observe the behavior. Since, this phase involves lot many recompiling cycles, achieving a faster compilation is important. For additional information, see the section entitled, "Compile-time Performance" .

- Runtime Performance

  Runtime performance is important in regression phase or in the final phase of the design development cycle. For additional information, see the section entitled, "Runtime Performance" .

- 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"

## Compile-time Performance

You can improve compile-time performance in the following ways:

- "Incremental Compilation"
- "Compile Once and Run Many Times"
- "Parallel Compilation"

## Incremental Compilation

During compilation, VCS builds the design hierarchy. By default, when you recompile the design, VCS compiles only those design units that have changed since the last compilation. This is called incremental compilation.

The incremental compilation feature is the default in VCS. It triggers recompilation of design units under the following conditions:

- Changes in the command-line options
- Change in the target of a hierarchical reference
- Change in the ports of a design unit
- Change in the functional behavior of the design
- Change in a compile-time constant such as a parameter

The following conditions do not cause VCS to recompile a module:

- Change of time stamp of any source file
- Change in file name or grouping of modules in any source file
- Unrelated change in the same source file
- 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 “SeeisUsing 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. 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. You should specify this using the compile-time option `-j[no_of_processes]`, as shown below:

```
% vcs -j[no_of_processes] [options] top_entity/module/
config
```

For example, the following command line forks off two parallel processes to generate a binary executable:

```
% vcs -j2 top
```

## Runtime Performance

VCS runtime performance is based on the following:

- Coding Style (see Chapter - "Modeling Your Design" )
- Access to the internals of your design at runtime, using PLIs, UCLI, debugging using GUI, dumping waveforms, and so on

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`.

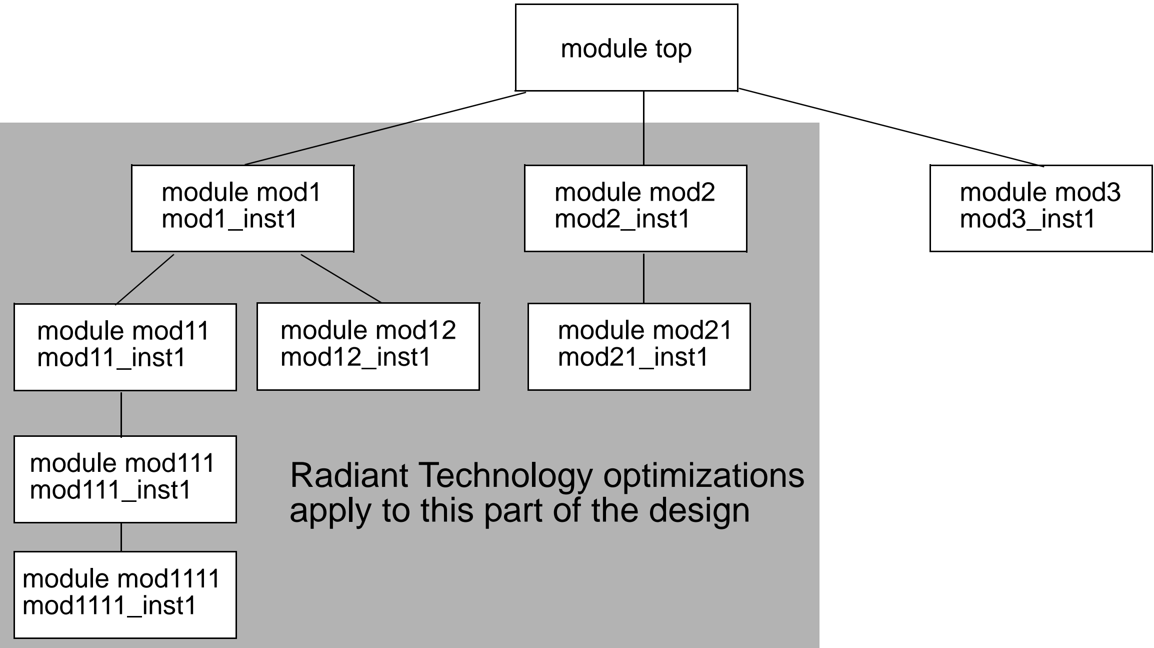
The instance statement disables Radiant optimizations for the following:

- 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:

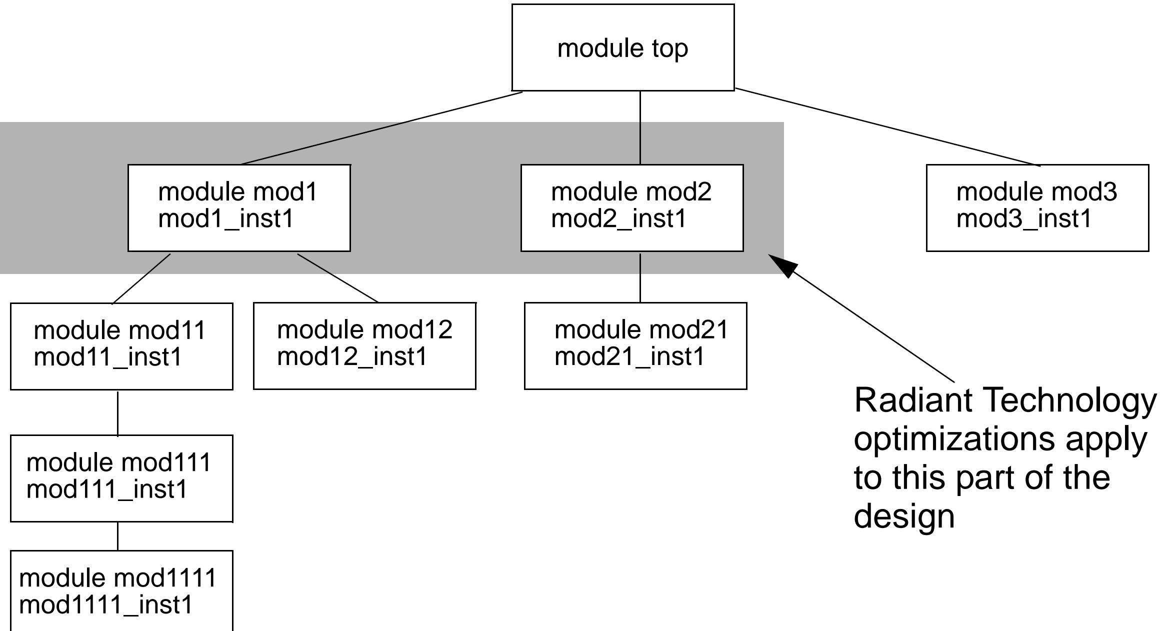



The statement enables Radiant Technology optimizations for the instances of modules `mod1` and `mod2` and for all the module instances hierarchically under these instances.

### 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`.



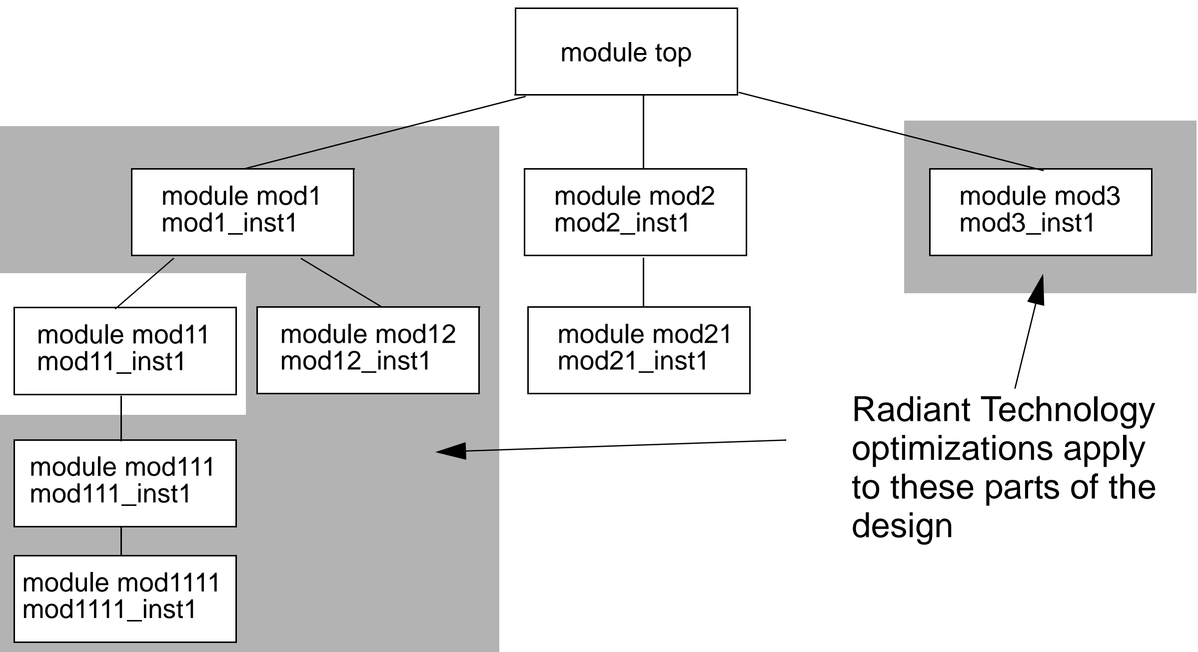

A tree statement with a depth of 0 is the equivalent of a module statement.

### 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 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`.

### Compilation

```
% vcs [vcs_options] Verilog_files
```

### Simulation

```
% simv [sim_options] +vcs+learn+pli
```

Step2: Using the compilation option `+applylearn+[tabfile]`.

### Compilation

```
% vcs [vcs_options] +applylearn+[tabfile] Verilog_files
```

### Simulation

```
% simv [sim_options]
```

## 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 10-1 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 10-2 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

Options like `-debug_pp`, `-debug`, and `-debug_all` disable VCS optimizations and also impact the performance. The `-debug_pp` option has less performance impact than the `-debug` or `-debug_all` options. The following table describes these options and their performance impact:

*Table 10-1 Performance Impact of -debug_pp, -debug, and -debug_all*

| Options | Description |
|---|---|
| `-debug_pp` | Use this option to generate a dump file. You can also use this option to invoke UCLI and DVE with some limitations. This has less performance impact when compared to -debug or -debug_all |
| `-debug` | Use this option if you want to use the `force` command at the UCLI prompt, and for more debug capabilities. |
| `-debug_all` | This option enables all debug capabilities, and therefore will have a huge performance impact. |

See the section "Compiling or 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 and simulate them along with the design.

Contact Synopsys Support Center (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.

# 11

# Using X-Propagation

This chapter includes the following sections:


- “Introduction to X-Propagation”
- “Using the X-Propagation Simulator”
- “X-Propagation Code Examples”
- “Support for Active Drivers in X-Propagation”
- “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 un-initialized 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 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 (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` 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-flip, 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 MX 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.

Verilog examples:

```
$$et_x_prop("tmerge");

$set_x_prop("xmerge");
```

```
$set_x_prop("vmerge");

$set_x_prop("xprop");
```

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. The Verilog `$set_x_prop` task can be called as many times as necessary in a simulation.

Another method to specify the Xprop merge mode is to use a Xprop configuration file. 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`.

### 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` for Verilog.

Report entries are created for the following HDL constructs:

- `if` statement in Verilog

- `case` statement in Verilog
- Body of an edge triggered `always` block in Verilog

Below is the format of a report entry in `xprop.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)
```

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 file `xprop.log`.

## Querying X-Propagation at Runtime

You can use the `$is_xprop_active` Verilog system function 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.

## 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.u1
```

## 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
|   out_of_bound_value = oob_value
|   module_ite
|   instance_item
|   tree_item

merge_function ::= tmerge |  xmerge
select_mode ::= resolution | dimensional | random
read_write_operation ::= read | write
oob_value ::= 0 | 1 | X

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.

### 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`

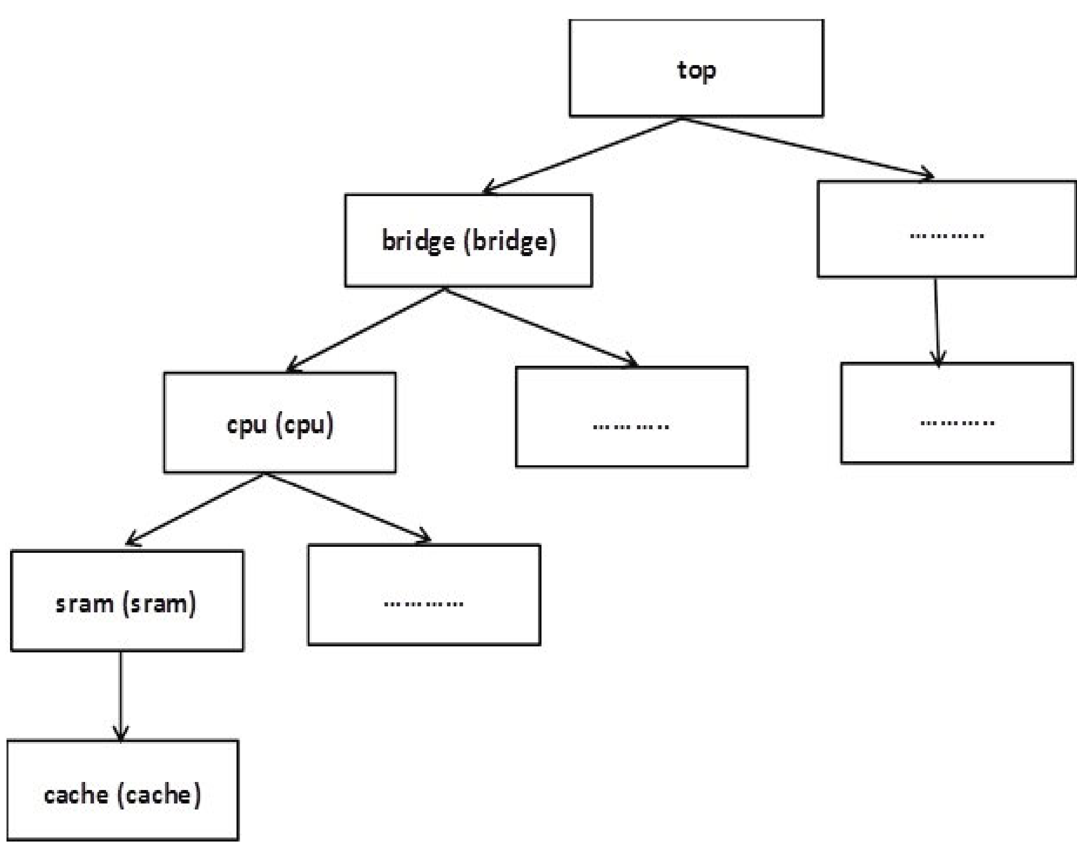
Apply the specified Xprop mode to all instances in the list and recursively to all the sub-instances.

`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 11-1 is a summary of the Xprop instrumentation control in a design hierarchy, when Xprop is enabled using various methods.

*Table 11-1    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 `$set_x_prop` can be used to change merge mode at runtime for instances that have been already compiled for Xprop<br>*Caution: Use sparingly* |
| Configuration file<br><br>example:<br>`vcs -xprop=cfg` | Instance/module specific control | Verilog task call `$set_x_prop` can be used to change merge mode at runtime for instances that have been already compiled for Xprop<br>*Caution: Use sparingly* |
| Pragma based<br><br>Verilog example:<br>`always (*xprop_off*)`<br>`@(posedge clk)` | Disabled Xprop for this process only | Process is disabled and cannot be enabled at runtime |

When a model use 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};
```

Note:

The wildcard * (asterisk) is only supported for Verilog.

As an example, consider the following code that needs to be simulated:

```
%cat   t.v:
`include "unit_functions.vh"
module top ();
     alldmtxd alldmtxd_xPropMe ();
     alldmtxd alldmtxd_FreeSpirit ();
endmodule

module  alldmtxd ();
     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};
```

## 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
```

## Support for XIndex Element Merging

VCS supports XIndex element merging in Verilog. This helps you to configure the simulator for out-of-bounds or XIndex. This significantly improves verification productivity by reducing debug effort required during gate-level simulations, thereby, reducing the time required to complete the verification cycle.

The following two operations are considered for XIndex element merging in Verilog:

- Write element merging semantics
- Read element merging semantics

The write element merging semantics operation merges RHS with each potentially written element. The read element merging semantics operation merges the elements of the set of potentially read values with each other.

The following are the methods of XIndex read and write element merging:

- Dimensional index merge
- Random index merge
- Index resolution merge

The models of read and write element merging have different accuracy and performance characteristics.

Also, these different models have Xprop merge modes (tmerge, xmerge, xpropOff) associated with them. The merge mode is not independently controllable, XIndex element merging uses the merge mode of the local context.

XIndex element merging considers all select expressions within an Xprop region as specified by the configuration file, including select expressions in `always` blocks, processes and continuous or concurrent assignments. However, XIndex element merging does not consider select expressions that are in instance port connections.

XIndex element merging does not result in any additional `YES` entries in the log file. All select expressions within Xprop-enabled regions are assumed to be instrumented. However, instance port expressions and XIndex expressions on a function call associated with output or inout expressions are not instrumented. Such unsupported expressions are logged in the `xprop.log` file with a `NO` entry.

## Index BSpace

Index BSpace is a set of values that is possible in the addressable space of designated array for an XIndex pattern. The addressable space of the array is index values within the zero-based power-of-2 space that encompasses the bounds of dimensions that are indexed.

For example, BSpace for the `X0X` pattern in range `[6:1]` is the set of minterms `{000, 001, 100, 101}`. In this case, the zero-based power-of-2 space for the range `6:1` is `7:0 (23-1:0)`. For read operations, index BSpace of an X pattern includes all minterms in the set. For write operations, index BSpace is further constrained to the subset of minterms that is 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'bx0x;
b = 1;
c = 2'b0x
array[a][b][c] = d;
```

For the assignment in the last statement, BSpace is the set of 3-tuples:

```
{(000, 1, 00), (000, 1, 01), (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.

## Addressing Models

While considering dimensional indexing in Xprop, addressing is modeled similar to gate-level implementation. Element merging considers every dimension that exists within a power-of-2 sized space. High-order bits that are unused in the range are masked away.

For non-power-of-2 based ranges, some minterms (values with the minimum number of bits required to uniquely address any element in a dimension range) may be out-of-bounds. For write element merging, these out-of-bounds minterms are always skipped, regardless of the element merging method. For read element merging, these out-of-bounds minterms result in `X` for `xmerge` and `vmerge` merge modes. For the `tmerge` merge mode, the result is `1`, `0`, or `X` depending upon the user setting that defines an out-of-bounds read value.

The section consists of the following subsections:

- “Unsigned Indexing”
- “Signed indexing”

### Unsigned Indexing

Dimensions where both bounds are positive (including `0`) are considered as unsigned indexing. For unsigned indexing, the power-of-2 space is `2**n-1:0`, where `n` is a smallest integer such that `2**n` is greater than the high bound of the dimension. For example, the range `6:1` results in an `n of 3`.

### Signed indexing

Dimensions where either or both bounds are negative are considered as 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 high bound and `-(2**(n-1))` is less than or equal to low bound. For example, the ranges `8:-8` and `7:-9` result in `n=5`, while range `7:-8` results in `n=4`.

## Element Merging Methods

Element merging methods apply within an Xprop instrumentation region and are not implemented in regions where Xprop is disabled.

The section consists of the following subsections:

- "Default Element Merging Method"
- "Dimensional Element Merging Method"
- "Randomized Element Merging Method"
- "Index Resolution Merging Method"

### Default Element Merging Method

The default XIndex element merging model follows SystemVerilog LRM semantics for read and write operations with `X` on the index.

For write merge mode, the assignment is skipped. For read merge mode, the result is always `X`.

### Dimensional Element Merging Method

The dimensional element merging method computes BSpace by converting an XIndex that has any `X` bits into an `xindex` with all `X`s in the index. Any index that does not contain an `X` is not changed. Therefore, an index with value `3'bx0x` is converted to `3'bxxx` prior to computing BSpace. Once BSpace is computed, the RHS of the assignment is merged with the element at each index in XIndex BSpace, similar to the resolution model. For read operation, the elements of BSpace are merged with each other to derive its result.

For write operation, `tmerge` merges values written at locations in BSpace. The `xmerge` mode writes `X` to all locations in BSpace. The `vmerge` mode skips the assignment.

For read operation, `tmerge` returns a value at each of the locations in BSpace merged with each other. `Vmerge` and `xmerge` return `X`.

### Randomized Element Merging Method

For the randomized merge mode, an in-bound random index is selected from XIndex BSpace. The write operation is evaluated using the selected index. The value written is computed according to the merge mode. For a read operation, the value at the randomized index is returned without merging.

For write operation, `tmerge` merges the RHS with a randomized element, `xmerge` writes an `X` to the randomized element, and `vmerge` skips the assignment.

For read operation, `tmerge` returns the randomized element and `vmerge` and `xmerge` return `X`.

### Index Resolution Merging Method

For write operation, the index resolution merge mode merges the RHS of the assignment with each element addressed by the minterms of XIndex BSpace.

If a BSpace minterm is out-of-bounds for that dimension, the element write is skipped for that minterm.

For read operation, the value of each index in BSpace is merged with each other to derive the result. If any BSpace index at a dimension is out-of-bounds, the value merged for the element is the value set using the `XINDEX_OUT_OF_BOUNDS_VALUE={1,0,X}` configuration file directive.

For write operation, `xmerge` writes an X and `tmerge` merges the value written with each BSpace element value. All masked indexes that are out-of-bounds are skipped.

For read operation, `tmerge` merges values of all locations in the BSpace with each other to get a result. Both `vmerge` and `xmerge` result in an X.

## 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

//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 Merging 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 Merging 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 Merging 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 11-2 are enabled:

Table 11-2 Assertions Enabled In Bounds Checking

| Assertion Type | Description | Default Behavior |
|---|---|---|
| xindex-wr | Write with indeterminate index (index containing X values) | Warning |
| xindex-rd | Read with indeterminate index (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.

## 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 MX 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.

# 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"
- "Case Statement"
  - "Verilog Example"
- "Edge Sensitive Expression"
  - "Verilog Example"
- "Latch"
  - "Verilog 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 11-3 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 11-3 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.

## 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 11-4    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 11-4 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.

## 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->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 11-5 Xprop Merge Mode Truth Table for D Flip-Flop

| clk | vmerge | tmerge | 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 11-5 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 11-3. Then, the merged value is assigned to `q`.

## 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 11-6 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 11-6 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 11-3. 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.

## Support for Active Drivers in X-Propagation

DVE supports active driver tracing functionality for the designs compiled with X-Propagation (using the `-xprop` option). DVE 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

DVE 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 11-1 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 11-1 as follows:

   ```
   % vcs comb_logic.sv -sverilog -xprop=tmerge -debug_pp
   ```

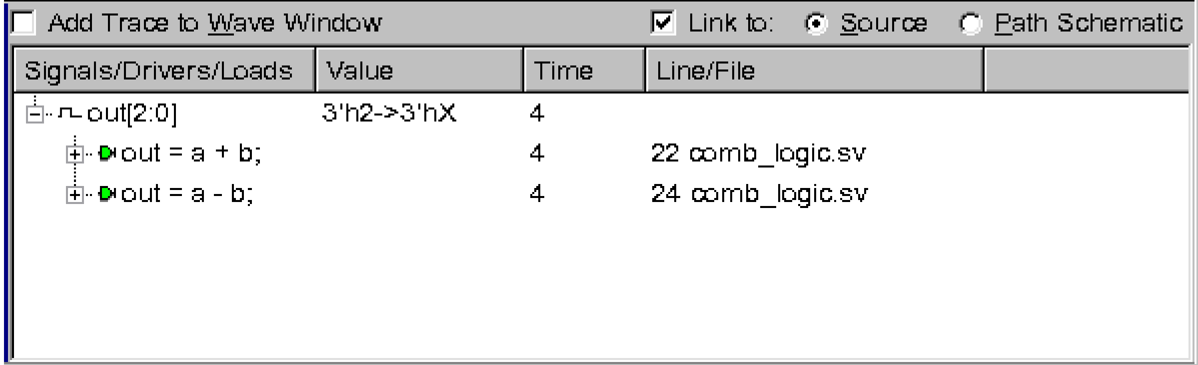
2. Invoke the DVE GUI using the following command:

   ```
   % simv -gui
   ```

3. Perform **Trace Drivers**.

   DVE displays active drivers for the combinational logic signals that cause Xprop in the **DriverLoad** Pane, as shown in Figure 11-1.

*Figure 11-1 Viewing Active Drivers of the Combinational Logic Signals*




DVE 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 11-2, to further trace the origin of X.

*Figure 11-2 Tracing Origin of X*

Add Trace to Wave Window    Link to: Source    Path Sc

| Signals/Drivers/Loads | Value | Time | Line/File |
|---|---|---|---|
| out[2:0] | 3'h2->3'hX | 4 | |
| out = a + b; | | 4 | 23 comb_logic.s |
| out = a - b; | | 4 | 25 comb_logic.s |
| a[1:0] | 2'h0->2'h1 | 4 | 17 comb_logic.s |
| b[1:0] | 2'h2->2'h1 | 4 | 17 comb_logic.s |
| op | 1'b1->1'bx | 4 | 17 comb_logic.s |
| #3 a = 1; b = 2'b01; op = 1'bx; | | 4 | 9 comb_logic.sv |

Note:

In the xmerge mode, all RHS contributors are displayed as inactive. In tmerge mode, the contributors with value change t hat can impact RHS value, are displayed as active. Otherwise, they are displayed as inactive.

## Latches

DVE supports active drivers tracing for Xprop in latches. DVE displays the control signals as contributors for the drivers.

For example, consider the following test case (`latch.sv`) that contains the latch signals:

*Example 11-2 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 11-2 as follows:

   ```
   % vcs latch.sv -sverilog -xprop=tmerge -debug_pp
   ```

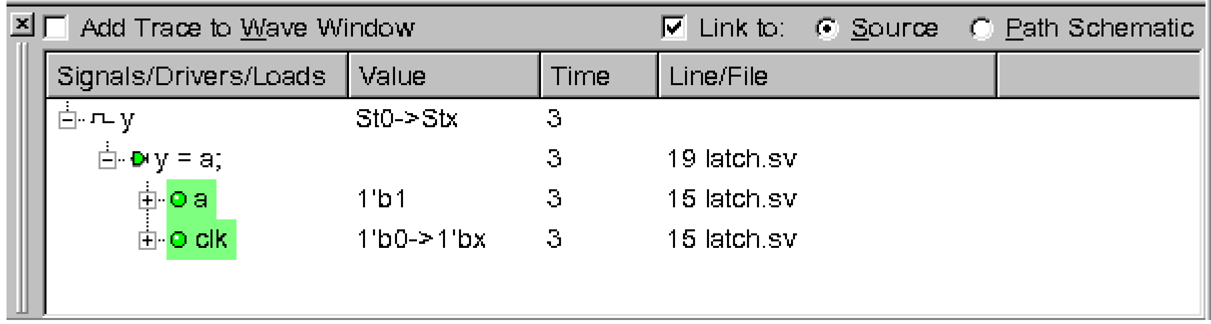
2. Invoke the DVE GUI using the following command:

   ```
   % simv -gui
   ```

3. Perform **Trace Drivers**.

   DVE displays active drivers for the latch signals that cause Xprop in the **DriverLoad** Pane, as shown in Figure 11-3.

*Figure 11-3    Viewing Active Drivers of the Latch Signals*




DVE displays both active and inactive contributor signals for the driver. You can expand the active contributor signal, as shown in Figure 11-4, to further trace the origin of X.

*Figure 11-4    Tracing Origin of X*

Add Trace to Wave Window    Link to: Source    Path Schematic

| Signals/Drivers/Loads | Value | Time | Line/File |
|---|---|---|---|
| y | St0->Stx | 3 | |
| y = a; | | 3 | 19 latch.sv |
| a | 1'b1 | 3 | 15 latch.sv |
| clk | 1'b0->1'bx | 3 | 15 latch.sv |
| #1 clk = 1'bx; | | 3 | 9 latch.sv |

## Flip-flops

DVE supports active drivers tracing for Xprop in flip-flops. DVE 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 11-3    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 11-3 as follows:

   ```
   % vcs flip_flop.sv -sverilog -xprop=tmerge -debug_pp
   ```

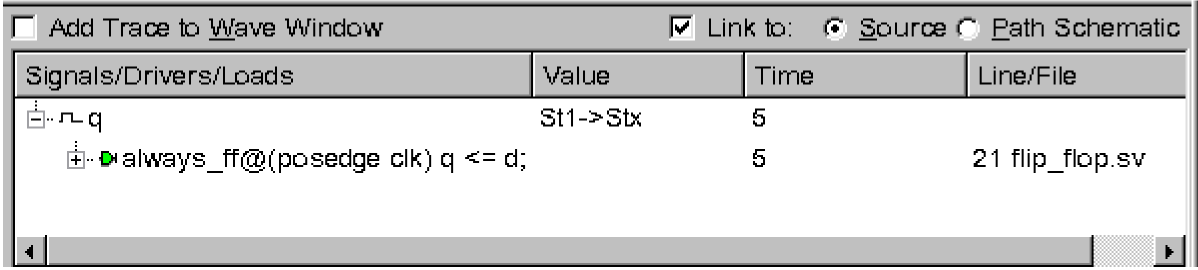
2. Invoke the DVE GUI using the following command:

   ```
   % simv -gui
   ```

3. Perform **Trace Drivers**.

   DVE displays active drivers for the flip-flops that cause Xprop in the **DriverLoad** Pane, as shown in Figure 11-5.

*Figure 11-5 Viewing Active Drivers of Flip-Flops*




DVE 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 11-6, to further trace the origin of X.

*Figure 11-6 Tracing Origin of X*

| Signals/Drivers/Loads | Value | Time | Line/File |
|---|---|---|---|
| clk | 1'b0->1'bx | 5 | |
| always #2 clk = ~clk; | | 5 | 5 flip_flop.sv |
| #1 clk = 1'bx; | | 5 | 10 flip_flop.sv |
| q | St1->Stx | 5 | |
| always_ff@(posedge clk) q <= d; | | 5 | 21 flip_flop.sv |
| clk | 1'b0->1'bx | 5 | 19 flip_flop.sv |
| d | 1'b1->1'b0 | 4 | 19 flip_flop.sv |

## 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.

## Limitations

1. X-Propagation is not supported with the following VCS features:

   - Multicore
   - $xzcheck
   - `+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, return values of function calls are also treated as non-constant expressions.

3.

# 12

# 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”

## 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 -xlrm Option for SDF Retain, Gate Pulse Propagation, and Gate Pulse Detection Warning”

## Using the Unified SDF Feature

The Unified SDF feature allows you to back-annotate SDF delays using the following compile-time option:

```
-sdf min|typ|max:instance_name:file.sdf
```

**Compilation**

```
% vcs -sdf min|typ|max:instance_name:file.sdf \
  [compile_options]
```

### Simulation

```
% simv [run_options]
```

For more information on specifying delays and SDF files, see "Options for Specifying Delays and SDF Files"

## 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:

### Compilation

```
% vcs [elab_options] top_cfg/entity/module
```

### Simulation

```
% simv [run_options]
```

For more details, see “Options for Specifying Delays and SDF Files”.

## Using the -xlrm 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 `-xlrm` 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 `-xlrm 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 `-xlrm 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 `-xlrm 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 `-xlrm 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 `-xlrm gd_pulsewarn` 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 `-xlrm gd_pulsewarn` 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
```

## 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 `$sdf_annotate` system task. You can also specify the precompiled SDF file in the `$sdf_annotate` system task. The `+csdf+precompile` option also supports zipped SDFs.

### Precompiling SDF Without Compiling Design Files

You can also precompile the SDF 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+./test.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 12-1.

*Table 12-1 VCS Simulation Delay Objects/Constructs*

| | SDF Constructs | VCS Simulation Object |
|---|---|---|
| **Delays** | | |
| | PATHPULSE | module path pulse delay |
| | GLOBALPATHPULSE | module path pulse reject/error delay |
| | IOPATH | module path delay |
| | PORT | module input port delay |
| | INTERCONNECT | module input port delay or, intermodule path delay when `+multisource_int_delays` is specified |
| | NETDELAY | module input port delay |
| | DEVICE | primitive and module path delay |
| **Timing-Checks** | | |
| | SETUP | `$setup` timing-check limit |
| | HOLD | `$hold` timing-check limit |
| | SETUPHOLD | `$setup` and `$hold` timing-check 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 |

*Table 12-1 VCS Simulation Delay Objects/Constructs*

| SDF Constructs | VCS Simulation Object |
| --- | --- |
| 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 specifies one of the following keywords:

`MINIMUM`

Annotates the minimum delay value.

`TYPICAL`

Annotates the typical delay value.

`MAXIMUM`

Annotates the maximum delay value.

`TOOL_CONTROL`

Delay value is determined by the command-line options of the Verilog tool (`+mindelays`, `+typdelays`, or `+maxdelays`)

The default for 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_MTM`
  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_MTM;
```

For example:

```
        SCALE_FACTORS=100:0:9;
        SCALE_TYPE=FROM_MTM;
        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`:

```
// test.v - test sdf annotation
`timescale 1ns/1ps
module test;
initial begin
     $sdf_annotate("./test.sdf",test, "./sdf.cfg",,,,);
end
wire  out1,out2;
wire  w1,w2;
reg in;
reg ctrl,ctrlw;
sub  Y (w1,w2,in,in,ctrl,ctrl);
sub  W (out1,out2,w1,w2,ctrlw,ctrlw);
initial begin
     $display("  i c ww oo");
     $display("ttt  n t 12 12");
     $monitor($realtime,,,in,,ctrl,,w1,w2,,out1,out2);
end
initial begin
     ctrl = 0;// enable
     ctrlw = 0;
     in = 1'bx;  //stabilize at x;
     #100 in = 1; // x-1
```

```
    #100 ctrl = 1; // 1-z
    #100 ctrl = 0; // z-1
    #100 in = 0; // 1-0
    #100 ctrl = 1; // 0-z
    #100 ctrl = 0; // z-0
    #100 in = 1'bx; // 0-x
    #100 ctrl = 1; // x-z
    #100 ctrl = 0; // z-x
    #100 in = 0; // x-0
    #100 in = 1; // 0-1
    #100 in = 1'bx; // 1-x
end
endmodule
`celldefine
module sub(o1,o2,i1,i2,c1,c2);
output o1,o2;
input  i1,i2;
input  c1,c2;
bufif0 Z(o1,i1,c1);
bufif0 (o2,i2,c2);
specify
    (i1,c1 *> o1) = (1,2,3,4,5,6);
    // 01 = 1, 10 = 2, 0z = 3, z1 = 4, 1z = 5, z0 = 6
    if (i2==1'b1) (i2,c2 *> o2) = (7,8,9,10,11,12);
         // 01 = 7, 10 = 8, z1 = 10, 1z = 11, z0 = 12
endspecify
subsub  X ();
endmodule
`endcelldefine
module subsub(oa,ob,ib,ia);
input ia,ib;output oa,ob;
specify
    (ia *> oa) = 99.99;
    (ib *> ob) = 2.99;
endspecify
endmodule

SDF File: test.sdf
(DELAYFILE
(SDFVERSION "3.0")
(DESIGN "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)))
))
)
)
SDF Configuration File: sdf.cfg
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:

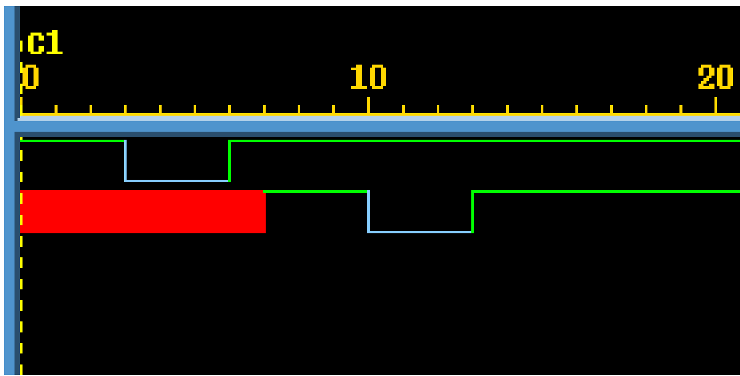
- "Transport and Inertial Delays"
- "Pulse Control"
- "Specifying the Delay Mode"

## 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 12-1 shows the waveforms for an input and output port of a module that models a buffer with a module path delay of seven-time units between these ports. The waveform on top is that of the input port and the waveform underneath is that of the output port. In this example, you have enabled transport delays for module path delays and specified that a pulse three-time units wide can propagate. For an explanation on how this is done, see "Enabling Transport Delays" and "Pulse Control".

*Figure 12-1 Transport Delay Waveforms*

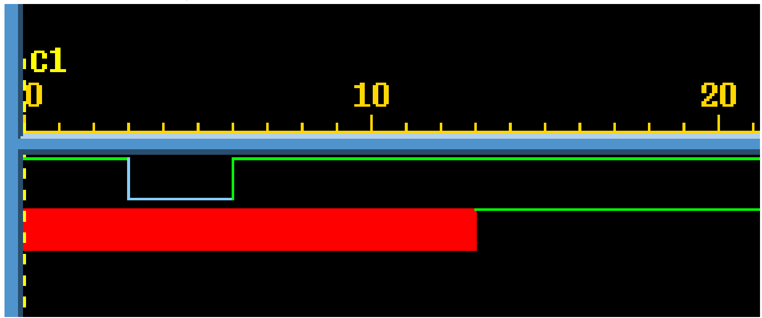



At time `0`, a pulse three-time unit wide begins on the input port. This pulse is narrower than the module path delay of seven-time units, but this pulse propagates through the module and appears on the output port after seven-time units. Similarly, another narrow pulse begins on the input port at time `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 12-2 shows the waveforms for the same input and output ports when you have not enabled transport delays for module path delays.

*Figure 12-2    Inertial Delay Waveforms*




The pulse that begins at time `0` that is three-time units wide does not propagate to the output port because it is narrower than the seven-time unit module path delay. 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 VCSto 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. With pulse control you can 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 that 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.

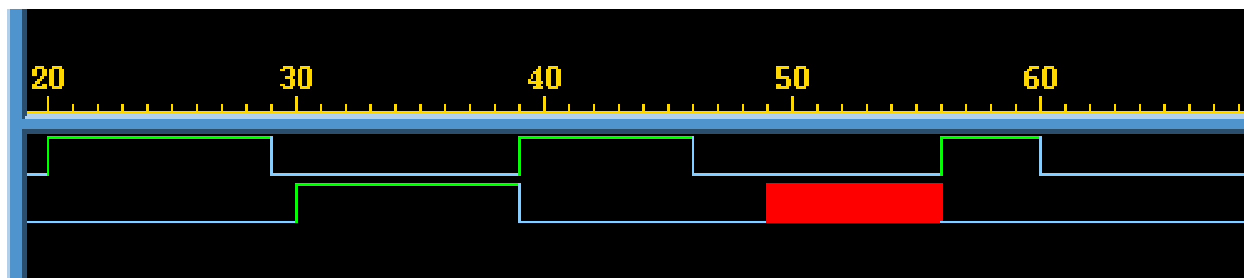
You can specify transport delays for INTERCONNECT delays on nets with the `+transport_int_delays`, `+pulse_int_e/`*`number`*, and `+pulse_int_r/`*`number`* options. You must include all three of these options.

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 12-3 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 12-3 Pulse Control With Transport Delays*




In the example illustrated in Figure 12-3, 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.

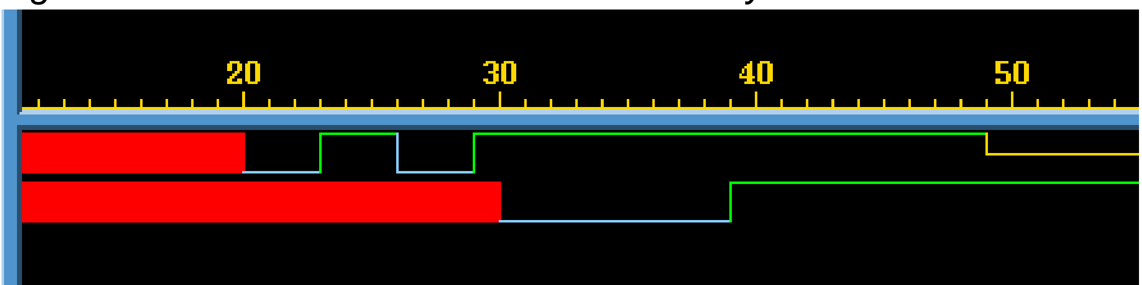
The second scheduled event is always tentative. If another event occurs on the input before the first event occurs on the output, that additional event on the input cancels the second scheduled event and schedules a new second event.

Figure 12-4 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 12-4 Pulse Control With Inertial Delays*



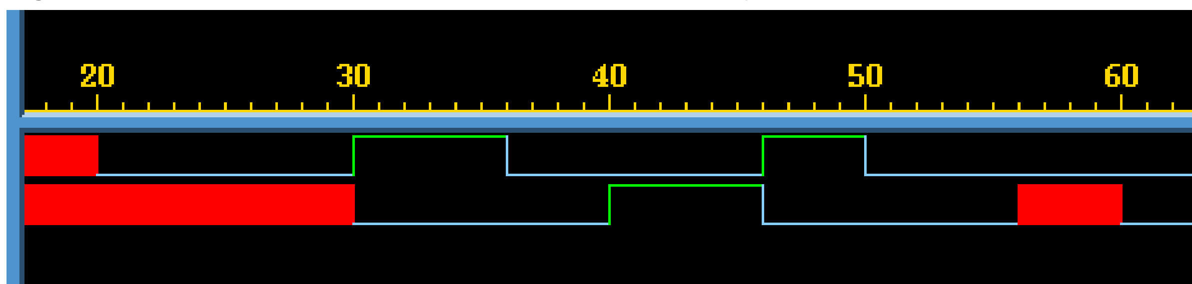

In the example illustrated in Figure 12-4, the following occurs:

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 12-5 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 12-5 Pulse Control With Inertial Delays and Narrow Pulses*




In the example illustrated in Figure 12-5, 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.

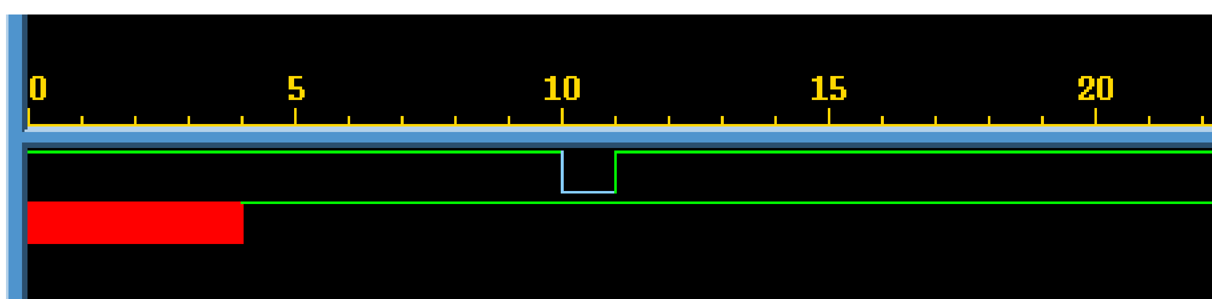
Pulse control sometimes blurs the distinction between inertial and transport delays. In this example, the results are the same if you also included the +transport_path_delays option.

## 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 12-6.

*Figure 12-6 Asymmetric Delays and Scheduling Problems*




In this example, you include the `+pulse_e/100` and `+pulse_r/0` options. The scheduling problem is that the leading edge of the pulse on the input, at time 10, schedules a transition to 0 on the output at time 16; but the trailing edge, at time 11, schedules a transition to 1 on the output at time 15.

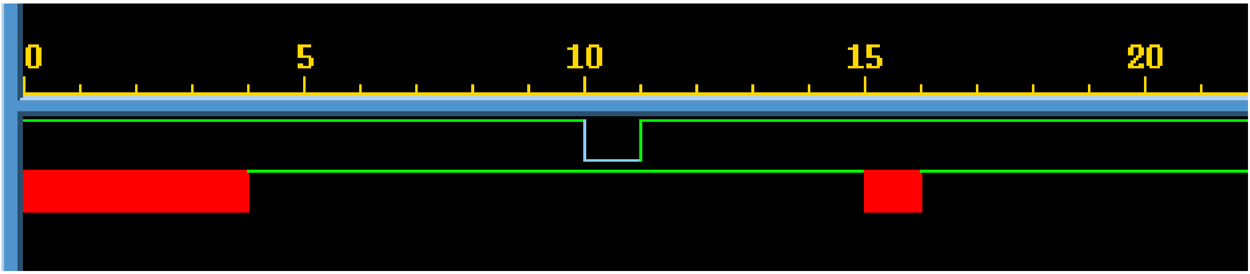
Obviously, the output has to end up with a value of 1 so VCS can not allow the events scheduled at time 15 and 16 to occur in sequence; if it did, the output ends up with a value of 0. This problem persists when you enable transport delays and whenever the percentage specified in the `+pulse_r/`*`number`* 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 12-7. Pulse on event behavior calls

for an X value pulse on the output after the delay and when there are asymmetrical delays scheduling events on the output that would be canceled by VCS , to output an X value pulse between those events instead.

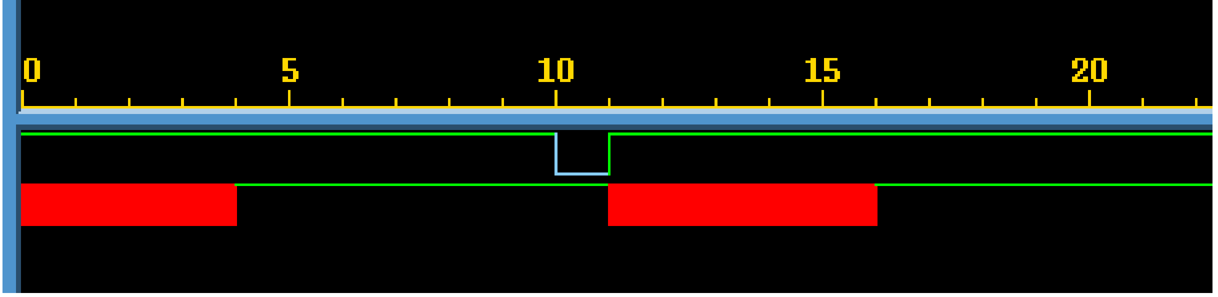
*Figure 12-7 Using +pulse_on_event*




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 12-8 illustrates pulse on detect behavior.

*Figure 12-8    Using +pulse_on_detect*



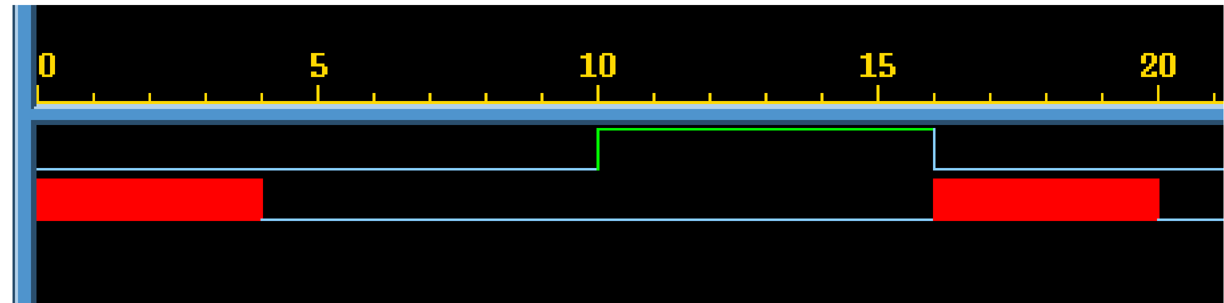

In this example, by including the `+pulse_on_detect` option, VCS causes the leading edge of the X value pulse on the output to begin at time 11. This is because of an unusual event that occurred on the output between times 15 and 16 because of the rise at simulation time 11.

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 12-9 shows the waveforms for the input and output port when you include the `+pulse_on_detect` option.

*Figure 12-9    Pulse on Detect Behavior Showing Multiple Transitions*




In the example illustrated in Figure 12-9, 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 12-1 shows such a conflict.

*Example 12-1 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 12-1, the module path delay is seven-time units, but the delay specifications distributed along that path add up to ten-time units.

If you include the `+delay_mode_path` compile-time option, VCS ignores the delay specifications in the primitive instantiation and continuous assignment statements and uses only the module path delay. In Example 12-1, it uses the seven-time unit delay for propagating signal values through the module.

If you include the `+delay_mode_distributed` compile-time option, VCS ignores the module path delays and uses the delay in the delay specifications in the primitive instantiation and continuous assignment statements. In Example 12-1, it uses the ten-time unit delay for propagating signal values through the module.

There are other modes that you can specify:

- If you include the `+delay_mode_unit` compile-time option, VCS ignores the module path delays and changes the delay specification in all primitive instantiation and continuous assignment statements to the shortest time precision argument of all the `` `timescale `` compiler directives in the source code. (The default time unit and time precision argument of the `` `timescale `` compiler directive is 1 s). In Example 12-1 the `` `timescale `` compiler directive has a precision argument of 1 ns. VCS might use this 1 ns as the delay, but if the module definition is used in a larger design and there is another `` `timescale `` compiler directive in the source code with a finer precision argument, then VCS uses the finer precision argument.
- If you include the `+delay_mode_zero` compile-time option, VCS changes all delay specifications and module path delays to zero.
- If you include none of the compile-time options described in this section, when, as in Example 12-1, the module path delay does not equal the distributed delays along the path, VCS uses the longer of the two.

## Using the Configuration File to Disable Timing

You can use the VCS configuration file to disable module path delays, specify blocks, and timing checks for module instances that you specify as well as all instances of module definitions that you specify. You use the instance, module, and tree statements to do this just as you do for applying Radiant Technology. 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 Radiant Technology (`+rad`).

You enable `timopt` with the `+timopt+`*`clock_period`* compile-time option, where the argument is the shortest clock period (or clock cycle) of the clock signals in your design. For example:

```
+timopt+100ns
```

This options specifies that the shortest clock period is 100ns.

`timopt` first displays the number of sequential devices that it finds in the design and the number of these sequential devices to which it might be able to apply optimizations. For example:

```
Total Sequential Elements : 2001
Total Sequential Elements 2001, Optimizable 2001
```

`timopt` then displays the percentage of identified sequential devices to which it can actually apply optimizations followed by messages about the optimization process.

```
TIMOPT optimized 75 percent of the design
Starting TIMOPT Delay optimizations
Done TIMOPT Delay Optimizations
DONE TIMOPT
```

The next step is to simulate the design and see if the optimizations applied by `timopt` produce a satisfactory increase in performance. If you are not satisfied, there are additional steps that you can take to get more optimizations from `timopt`.

If `timopt` 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 then you can use the following process to maximize `timopt` optimizations:

1. `timopt` writes a configuration file named `timopt.cfg` in the current directory that lists 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 { tfnpc } { 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.

## Negative Timing Checks

Negative timing checks are either `$setuphold` timing checks with negative setup or hold limits, or `$recrem` timing checks with negative recovery or removal limits.

This following sections describe their purpose, how they work, and how to use them:

- “The Need for Negative Value Timing Checks”

- “The $setuphold Timing Check Extended Syntax”
- “The $recrem Timing Check Syntax”
- “Enabling Negative Timing Checks”
- “Checking Conditions”
- “Toggling the Notifier Register”
- “SDF Back-Annotation to Negative Timing Checks”
- “How VCS Calculates Delays”

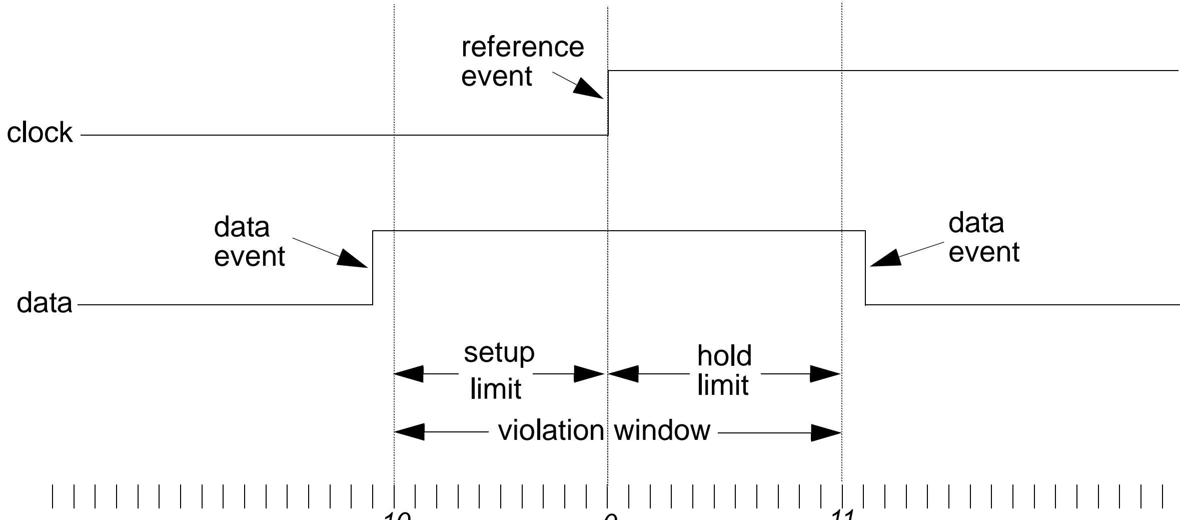

## The Need for Negative Value Timing Checks

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 that `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 12-10 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.

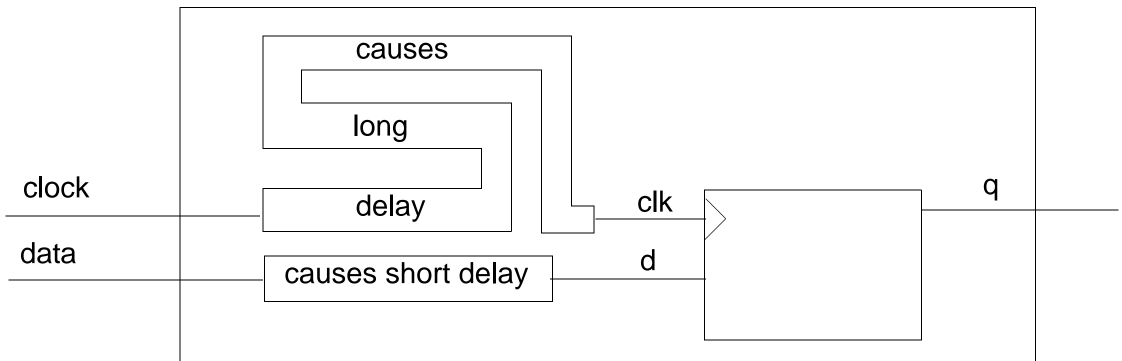
There are cases where the violation window cannot straddle the reference event at the inputs of an ASIC cell. Such a case occurs when:

- 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 12-11 ASIC Cell With Long Propagation Delays on Reference Events*

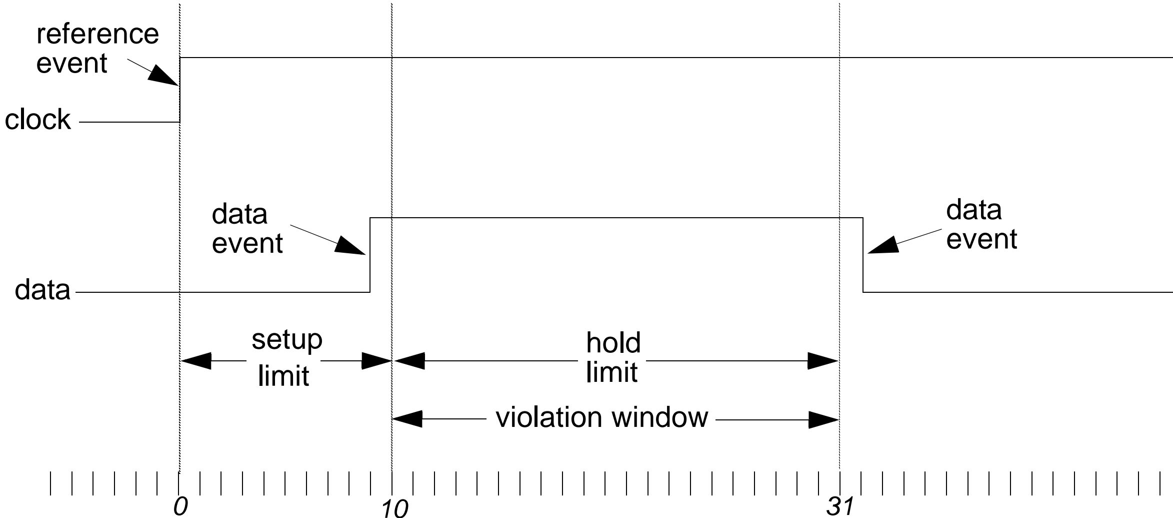



When this occurs, the violation window shifts at the cell boundary so that it no longer straddles the reference event. It shifts to the right when there are longer propagation delays on the reference event. This right shift requires a negative setup limit:

```
$setuphold (posedge clock, data, -10, 31, notifyreg);
```

Figure 12-12 illustrates this scenario.

*Figure 12-12 Negative Setup Limit*



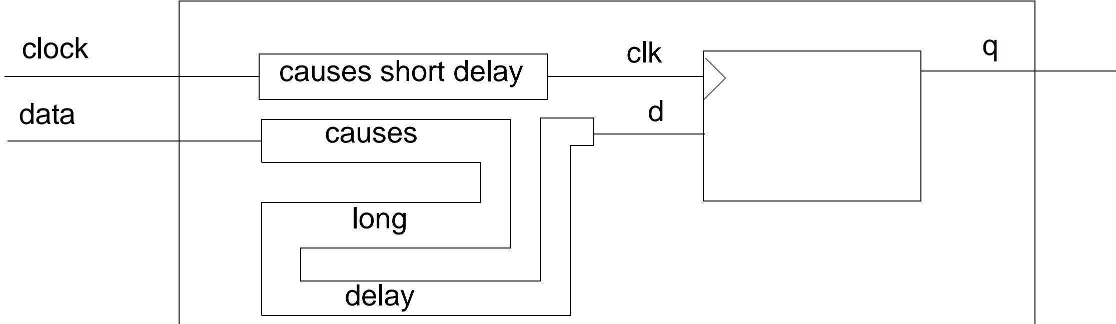

In this example, the `$setuphold` timing check is in the specify block of the top-level module of the cell. It specifies that there is a timing violation if there is a data event between `10` and `31` time units after the reference event on the cell boundary.

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 12-13 ASIC Cell With Long Propagation Delays on Data Events*

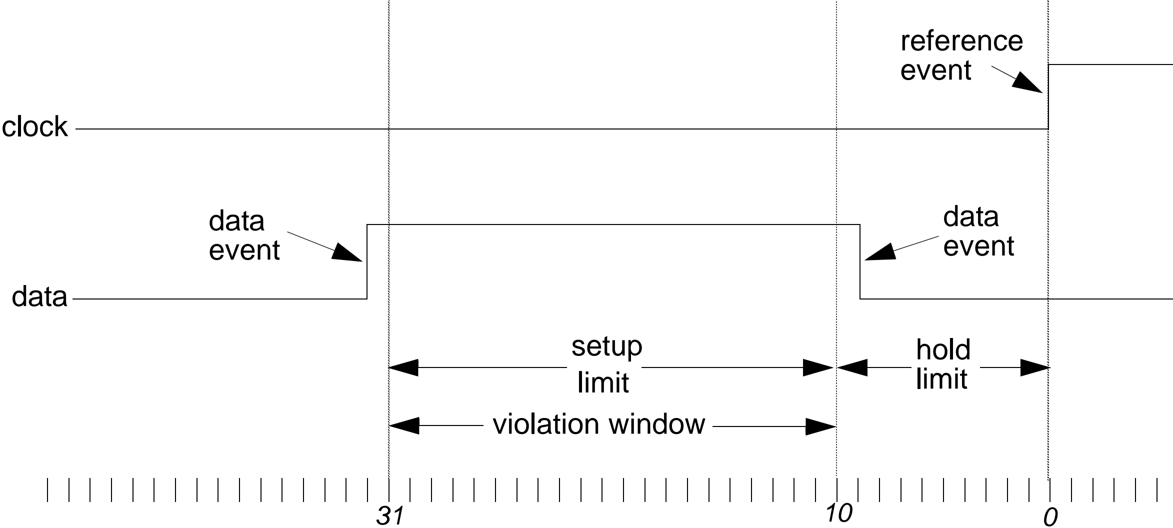



The violation window shifts to the left when there are longer propagation delays on the data event. This left shift requires a negative hold limit:

```
$setuphold (posedge clock, data, 31, -10, notifyreg);
```

Figure 12-14 illustrates this scenario.

*Figure 12-14 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, notifr1, 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 `$recrem` timing check, VCS compares or “checks” the time of a reference event with the time of a data event to see if there is a removal timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no removal timing violation.

`delayed_reference_signal`

The name of the delayed version of the reference signal, typically a control signal.

`delayed_data_signal`

The name of the delayed version of the data signal, typically a clock signal.

## Enabling Negative Timing Checks

To use a negative timing check you must include the `+neg_tchk` compile-time option when you compile your design. If you omit this option, VCS changes all negative limits to `0`.

If you include the `+no_notifier` compile-time option with the `+neg_tchk` option, you only disable notifier toggling. VCS still creates the delayed versions of the reference and data signals and displays timing violation messages.

Conversely, if you include the `+no_tchk_msg` compile-time option with the `+neg_tchk` option, you only disable timing violation messages. VCS still creates the delayed versions of the reference and data signals and toggles notifier regs when there are timing violations.

If you include the `+neg_tchk` compile-time option but also include the `+notimingcheck` or `+nospecify` compile-time options, VCS does not compile the `$setuphold` and `$recrem` timing checks into the `simv` executable. However, it does create the signals that you specified in the *delayed_reference_signal* and *delayed_data_signal* arguments, and you can use these to drive sequential devices in the cell. Note that there is no delay on these "delayed" 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 12-2 Notifier Register Example for Delayed Reference and Data Signals*

```
`timescale 1ns/1ns

module top;
    reg clk, d;
    reg rst;
    wire q;

    dff dff1(q, clk, d, rst);

    initial begin
     $monitor($time,,clk,,d,,q);
     rst = 0; clk = 0; d = 0;
     #100 clk = 1;
     #100 clk = 0;
     #10 d = 1;
     #90 clk = 1;
     #1 clk = 0; // width violation
     #100 $finish;
    end
endmodule

module dff(q, clk, d, rst);
    output q;
    input clk, d, rst;
    reg notif;

    DFF_UDP(q, d_clk, d_d, d_rst, notif);

    specify
     $setuphold(posedge clk, d, -10, 20, notif, , , d_clk,
                d_d);
     $setuphold(posedge clk, rst, 10, 10, notif, , , d_clk,
               d_rst);
     $width(posedge clk, 5, 0, notif);
    endspecify
endmodule
```

```
primitive DFF_UDP(q,data,clk,rst,notifier);
output q; reg q;
input data,clk,rst,notifier;

table
// clock  data rst   notifier  state  q
// ------------------------------
     r      0     0      ?      :  ?  :  0 ;
     r      1     0      ?      :  ?  :  1 ;
     f      ?     0      ?      :  ?  :  - ;
     ?      ?     r      ?      :  ?  :  0 ;
     ?      *     ?      ?      :  ?  :  - ;
     ?      ?     ?      *      :  ?  :  x ;
endtable
endprimitive
```

In this example, if you include the `+neg_tchk` compile-time option, the `$width` timing check uses the delayed version of signal `clk`, named `d_clk`, and the following sequence of events occurs:

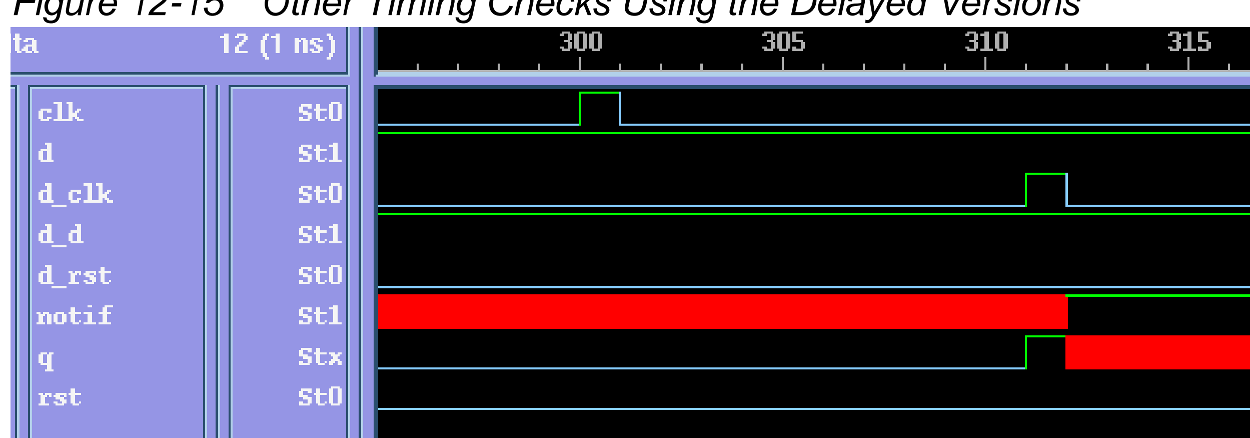
1. At time `311`, the delayed version of the clock transitions to `1`, causing output `q` to toggle to `1`.
2. At time `312`, the narrow pulse on the clock causes a width violation:

   ```
   "test1.v", 31: Timing violation in top.dff1
   $width( posedge clk:300,   : 301, limit: 5);
   ```

   The timing violation message looks like it occurs at time `301`, but you do not see it until time `312`.

3. Also at time `312`, reg `notif` toggles from `X` to `1`. This changes output `q` from `1` to `X`. There are no subsequent changes on output `q`.

*Figure 12-15    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:

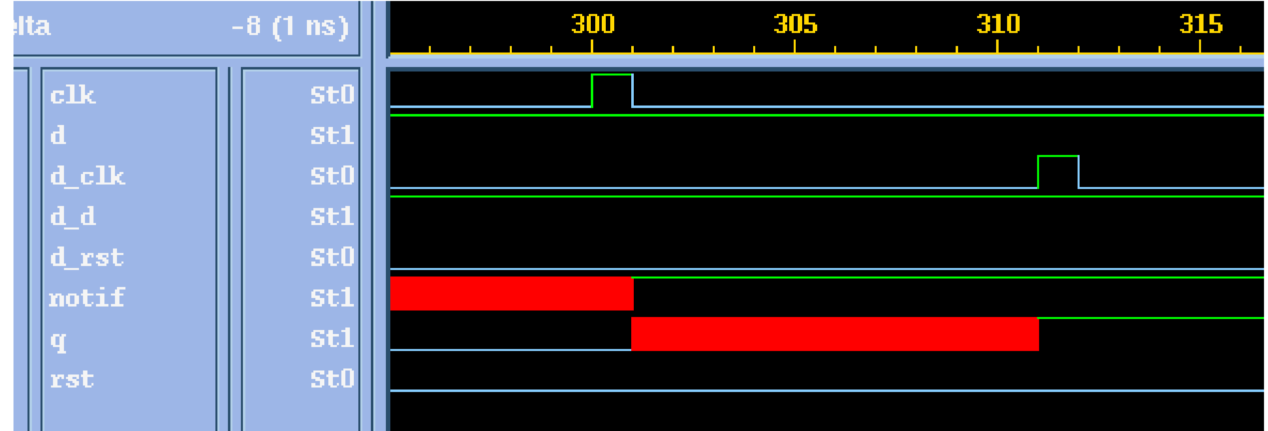
1. At time `301`, the narrow pulse on signal `clk` causes a width violation:

   ```
   "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 12-16 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 > (delay on del_clock - delay on del_data) > 10
```

There is no reason to make the delays on either of these delayed signals any longer than they have to be so the delay on `del_data` is `0` and the delay on `del_clock` is `11`. Any delay on `del_clock` between `11` and `19` time units would report a timing violation for the `$setuphold` timing check.

Multiple timing checks, that share reference or data events, and specified delayed signal names, can define a set of delay relationships. For example:

```
$setuphold(posedge CP,D,-10,20, notifier, , ,
           del_CP, del_D);
$setuphold(posedge CP,TI,20,-10, notifier, , ,
           del_CP, del_TI);
$setuphold(posedge CP,TE,-4,8, notifier, , ,
           del_CP, del_TE);
```

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 that 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 that 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.

# 13

# 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 res*pectively.

## Options For Coverage Metrics

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

Specifies compiling for the specified type or types of coverage. The argument specifies the types of coverage:

`line`

Compile for line or statement coverage.

`cond`

Compile for condition coverage.

`fsm`

Compile for FSM coverage.

`tgl`

Compile for toggle coverage.

`branch`

Compile for branch coverage

`assert`

Compile for SystemVerilog assertion coverage.

For more information on Coverage options, see the *Coverage Technology User Guide* in the *VCS Online Documentation*.

# 14

# Using OpenVera Native Testbench

OpenVera Native Testbench is a high-performance, single-kernel technology in VCS that enables:

- Native compilation of testbenches written in OpenVera and in SystemVerilog.
- Simulation of these testbenches along with the designs.

This technology provides a unified design and verification environment in VCS for significantly improving overall design and verification productivity. Native Testbench is uniquely geared towards efficiently catching hard-to-find bugs early in the design cycle, enabling not only completing functional validation of designs with the desired degree of confidence, but also achieving this goal in the shortest time possible.

Native Testbench is built around the preferred methodology of keeping the testbench and its development separate from the design. This approach facilitates development, debug, maintenance and reusability of the testbench, as well as ensuring a smooth synthesis flow for your design by keeping it clean of all testbench code. Further, you have the choice of either compiling your testbench along with your design or separate from it. The latter choice not only saves you from unnecessary recompilations of your design, it also enables you to develop and maintain multiple testbenches for your design.

This chapter describes the high-level, object-oriented verification language of OpenVera, which enables you to write your testbench in a straightforward, elegant and clear manner and at a high level essential for a better understanding of and control over the design validation process. Further, OpenVera assimilates and extends the best features found in C++ and Java along with syntax that is a natural extension of the hardware description languages. Adopting and using OpenVera, therefore, means a disciplined and systematic testbench structure that is easy to develop, debug, understand, maintain and reuse.

Thus, the high-performance of Native Testbench technology, together with the unique combination of the features and strengths of OpenVera, can yield a dramatic improvement in your productivity, especially when your designs become very large and complex.

This chapter includes the following topics:

- “Usage Model”
- “Key Features”

## Usage Model

As any other VCS applications, the usage model to simulate OpenVera testbench includes the following steps:

### Compilation

```
% vcs [ntb_options] [compile_options]  file1.vr file2.vr
  file3.v file4.v
```

### Simulation

```
% simv [run_options]
```

## Example

In this example, 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 <vera_defines.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
```

## Usage Model

### Compilation

```
% vcs  -ntb arb.v arb.vr arb.test_top.v
```

### Simulation

```
% simv
```

## Using Template Generator

To ease the process of writing a testbench in OpenVera, VCS provides you with a testbench template generator.

Use the following command to invoke the template generator on a Verilog design unit:

```
% ntb_template -t design_module_name [-c clock] design_file\
 [-vcs vcs_compile-time_options]
```

Where:

`-t` *design_module_name*

Specifies the top-level design module name.

*design_file*

Name of the design file.

`-c`

Specifies the clock input of the design.

`-template`

Can be omitted.

`-program`

Optional. Use it to specify program name.

`-simcycle`

Optional. Use this to override the default cycle value of 100.

`-vcs` *vcs_compile-time_options*

Optional. Use it to supply a VCS compile-time option. Multiple `-vcs` *`vcs_compile-time_options`* options can be used to specify multiple options. Use this option only for Verilog on top designs.

## Example

An example SRAM model is used in this demonstration of using the template generator to develop a testbench environment.

For details on the OpenVera verification language, refer to the *OpenVera Language Reference Manual: Native Testbench*.

### Design Description

The design is an SRAM whose RTL Verilog model is in the file `sram.v`. It has four ports:

```
- ce_N (chip enable)

- rdWr_N (read/write enable)

- ramAddr (address)

- ramData (data)
```

*Example 14-1 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'bzzzzzzzz;

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 <vera_defines.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

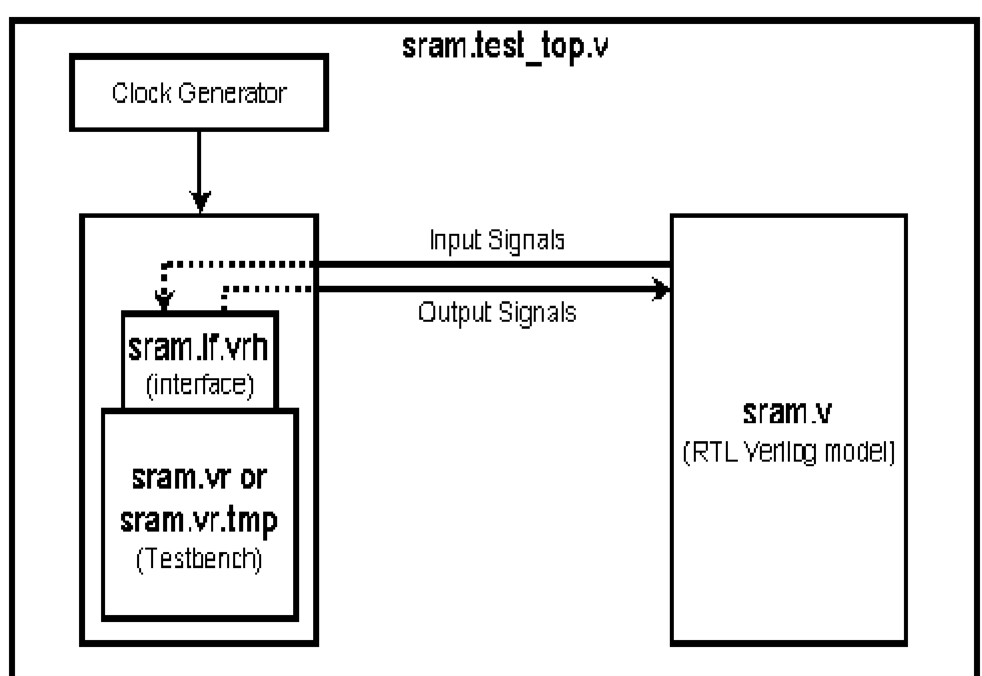
This is the top-level Verilog module that contains the testbench instance, the design instance, and the system-clock. The system clock can also provide the clock input for both the interface and the design. The following is the `sram.test_top.v` file that results from the previous command line:

```
//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

`ifdef emu
/* DUT is in emulator, so not instantiated here */
`else
  sram dut(
    .ce_N       (ce_N),
    .rdWr_N     (rdWr_N),
    .ramAddr    (ramAddr),
    .ramData    (ramData)
  );
`endif

  initial begin
    SystemClock = 0;
    forever begin
      #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
  end

endmodule
```

Figure 14-1 shows how the three generated files and the design connect and fit in with each other in the final configuration.

*Figure 14-1    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 14-2.

*Example 14-2    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'b00_0001; // declare, initialize address (for
                                // driving ramAddr during Write and
                                // Read)
reg [7:0] rand_bits;            // declare  rand_bits (for driving
                                // ramData during Write)
reg [7:0] data_result;          // declare data_result (for receiving
                                // ramData during Read)

@(posedge sram.clk);            // move to the first posedge of clock
rand_bits = random();           // 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 14-3.

### Interface for SRAM, sram.if.vrh

*Example 14-3*

```
#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 14-4 shows the modified top-level Verilog module for the SRAM.

*Example 14-4 Top-level Verilog Module, sram.test_top.v*

```
module sram_test_top;
  parameter simulation_cycle = 100;
  reg           SystemClock;
  wire          ce_N;
  wire          rdWr_N;
  wire [5:0]    ramAddr;
  wire [7:0]    ramData;
  wire         clk = SystemClock;/* Add this line. Interface
                clock input derived from the system clock*/

  `ifdef SYNOPSYS_NTB
  sram_test vshell(
    .SystemClock (SystemClock),
    .\sram.clk(clk),
    .\sram.ce_N (ce_N),
    .\sram.rdWr_N       (rdWr_N),
    .\sram.ramAddr      (ramAddr),
    .\sram.ramData      (ramData)
  );
`else
```

```
  vera_shell vshell(
    .SystemClock (SystemClock),
    .sram_ce_N  (ce_N),
    .sram_rdWr_N        (rdWr_N),
    .sram_ramAddr       (ramAddr),
    .sram_ramData       (ramData)
  );
`endif

  // design instance
  sram dut(
    .ce_N       (ce_N),
    .rdWr_N     (rdWr_N),
    .ramAddr    (ramAddr),
    .ramData    (ramData)
  );

  // system-clock generator
  initial begin
    SystemClock = 0;
    forever begin
      #(simulation_cycle/2)
        SystemClock = ~SystemClock;
    end
  end

endmodule
```

The top-level Verilog module contains the following:

- The system clock, `SystemClock`. The system clock is contained in the port list of the testbench instance.
- The declaration of the interface clock input, `clk`, and its derivation from the system clock.

- The testbench instance, `vshell`. The module name for the instance must be the name of the testbench program, `sram_test`. The instance name can be something you choose. The ports of the testbench instance, other than the system clock, refer to the interface signals. The period in the port names separates the interface name from the signal name. A backslash is appended to the period in each port name because periods are not normally allowed in port names.
- The design instance, `dut`.

### Compiling Testbench With the Design And Running

The VCS command line for compiling both your example testbench and design is the following:

### Compilation

```
% vcs  -ntb sram.v sram.test_top.v sram.vr
```

### Simulation

```
% simv
```

You will find the simulation output to be the following:

```
Cycle: 3 Time:  250
The SRAM is being written at ramAddr: 000001 with ramData:
10101100
Cycle: 6 Time:  550
The SRAM is being read at ramAddr: 000001 its ramData is:
10101100
$finish at simulation time                  550
           V C S   S i m u l a t i o n   R e p o r t
```

## Key Features

VCS supports the following features for OpenVera testbench:

- "Multiple Program Support"
- "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:

```
% vcs -ntb -ntb -ntb_opts
config=configfileVerilog_and_OV_files
```

## Configuration File

The configuration file contains the program construct.

The program keyword is followed by the OpenVera program file (`.vr` file) containing the testbench program and all the OpenVera program files needed for this program. For example:

```
//configuration file
program
     main1.vr
     main1_dep1.vr
     main1_dep2.vr
     ...
     main1_depN.vr
     [NTB_options ]

program
     main2.vr
     main2_dep1.vr
     main2_dep2.vr
     ...
     main2_depN.vr
     [NTB_options ]

program
     mainN.vr
     mainN_dep1.vr
     mainN_dep2.vr
     ...
     mainN_depN.vr
     [NTB_options ]
```

In this example, `main1.vr`, `main2.vr` and `mainN` files each contain a program. The other files contain items such as definitions of functions, classes, tasks and so on needed by the program files. For example, the `main1_dep1.vr`, `main1_dep2.vr` .... `main1_depN.vr` files contain definitions relevant to `main1.vr`. Files `main2_dep1.v`, `main2_dep2.vr` ... `main2_depN.vr` contain definitions relevant to `main2.vr`, and so forth.

## Usage Model for Multiple Programs

You can specify programs and related support files with multiple programs in two different ways:

1. Specifying all OpenVera programs in the configuration file
2. Specifying one OpenVera program on the command line, and the rest in the configuration file

Note:

- Specifying multiple OpenVera files containing the program construct at the VCS command prompt is an error.
- If you specify one program at the VCS command line and if any support files are missing from the command line, VCS issues an error.

### Specifying all OpenVera programs in the configuration file

When there are two or more program files listed in the configuration file, the VCS command line is:

```
% vcs -ntb -ntb_opts config=configfile
```

The configuration file, could be:

```
program main1.vr -ntb_define ONE
```

```
program main2.vr -ntb_incdir /usr/vera/include
```

### Specifying one OpenVera program on the command line, and the rest in the configuration file

You can specify one program in the configuration file and the other program file at the command prompt.

```
% vcs -ntb -ntb_opts config=configfile main2.vr
```

The configuration file used in this example is:

```
program main1.vr
```

In the previous example, `main1.vr` is specified in the configuration file and `main2.vr` is specified on the command line along with the files needed by `main2.vr`.

## NTB Options and the Configuration File

The configuration file supports different OpenVera programs with different NTB options such as `'include`, `'define`, or `'timescale`. For example, if there are three OpenVera programs `p1.vr`, `p2.vr` and `p3.vr`, and `p1.vr` requires the `-ntb_define VERA1` runtime option, and `p2.vr` should run with `-ntb_incdir /usr/vera/include` option, specify these options in the configuration file:

```
program p1.vr -ntb_define VERA1
program p2.vr -ntb_incdir /usr/vera/include
```

and specify the command line as follows.

```
% vcs -ntb -ntb_opts config=configfile p3.vr
```

Any NTB options mentioned at the command prompt, in addition to the configuration file, are applicable to all OpenVera programs.

In the configuration file, you may specify the NTB options in one line separated by spaces, or on multiple lines.

```
program file1.vr -ntb_opts no_file_by_file_pp
```

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
```

By default, files are processed in the order specified (or wildcard-expanded by the shell). This is a global option, and affects all Vera input files, including those preceding it, and those named in `-f file.list`.

When using the option `–ntb_opts print_deps` in addition to `–ntb_opts dep_check` with `vcs`, the reordered list of source files is printed on standard output. This could be used, for example, to establish a baseline for further testbench development.

For example, assume the following files and declarations:

```
b.vr: class Base {integer i;}
d.vr: class Derived extends Base {integer j;}
p.vr: program test {Derived d = new;}
```

File `d.vr` depends on file `b.vr`, since it contains a class derived from a class in `b.vr`, whereas `p.vr` depends on neither, despite containing a reference to a class declared in the former. The `p.vr` file does not participate in inheritance relationships. The effect of dependency ordering is to properly order the files `b.vr` and `d.vr`, while leaving files without class inheritance relationships alone.

The following command lines result in reordered sequences.

```
% vcs -ntb -ntb_opts dep_check d.vr b.vr p.vr
% vcs -ntb -ntb_opts dep_check p.vr d.vr b.vr
```

The first command line yields the order `b.vr d.vr p.vr,` while the second line yields, `p.vr b.vr d.vr`.

## Circular Dependencies

With some programming styles, source files can appear to have circular inheritance dependencies in spite of correct inheritance trees being cycle-free. This can happen, for example, in the following scenario:

```
a.vr: class Base_A {...}
      class Derived_B extends Base_B {...}
b.vr: class Base_B {...}
      class Derived_A extends Base_A {...}
```

In this example, classes are derived from base classes that are in the other file, respectively, or more generally, when the inheritance relationships project onto a loop among the files. This is, however, an abnormality that should not occur in good programming styles. VCS will detect and report the loop, and will use a heuristic to break it. This may not lead to successful compilation, in which case you

can use the `-ntb_opts print_deps` option to generate a starting point for manual resolution; however, if possible, the code should be rewritten.

### Dependency-based Ordering in Encrypted Files

As encrypted files are intended to be mostly self-contained library modules that the testbench builds upon, they are excluded from reordering regardless of dependencies (these files should not exist in unencrypted code). VCS splits Vera input files into those that are encrypted or declared as such by having the `.vrp` or `.vrhp` file extension or as specified using the `–ntb_vipext` option, and others. Only the latter unencrypted files are subject to dependency-based reordering, and encrypted files are prefixed to them.

Note:

> The `-ntb_opts dep_check` compile-time option specifically resolves dependencies involving classes and enums. That is, 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_filext` 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:

#### Compilation

```
% vcs -ntb -ntb_opts rvm [ntb_options] [compile_options]
file1.vr file2.vr  file3.v file4.v
```

#### Simulation

```
% simv [run_options]
```

For details on the use of RVM, see the *Reference Verification Methodology User Guide*. Though the manual descriptions refer to Vera, NTB uses a subset of the OpenVera language and all language specific descriptions apply to NTB.

Differences between the usage of NTB and Vera are:

- NTB does not require header files (`.vrh`) as described in the *Reference Verification Methodology User Guide* chapter “Coding and Compilation.”
- NTB parses all testbench files in a single compilation.
- The VCS command-line option `-ntb_opts rvm` must be used with NTB.

### Limitations

- The `handshake` configuration of notifier is not supported (since there is no handshake for triggers/syncs in NTB).
- RVM enhancements for assertion support in Vera 6.2.10 and later are not supported for NTB.
- If there are multiple consumers and producers, there is no guarantee of fairness in reads from channels, etc.

# 15

# 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”

- “Support for Overriding Parameter Values through Configuration”
- “Extensions to SystemVerilog”

For SystemVerilog assertions, see Chapter - "Using SystemVerilog Assertions".

## Use Model

The use model to compile and simulate your design with SystemVerilog files is as follows:

### Compilation

```
% vcs -sverilog [compile_options] Verilog_files
```

### Simulation

```
% simv [simv_options]
```

To analyze SV files, use the `-sverilog` option with `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”
- “Debugging UVM Testbench Designs Using DVE”
- “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
```

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
```

## 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
```

### 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
```

# 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_pp option to your command line:

```
% vcs -sverilog -debug_pp -ntb_opts uvm [compile_options] \
user_source_files_using_UVM
```

Note:

The -debug_pp 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_pp` option is not required and the HDL backdoor is enabled through cross-module references instead of the VPI. This provides better performance.

## 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 DVE 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 or DVE Transaction Recording" .

## Debugging UVM Testbench Designs Using DVE

DVE supports the debugging of UVM testbench designs and allows you to do the following. For more details, refer to the *Debugging UVM Testbench Designs* section in the *DVE User Guide*.

- View all available configurations in your design.
- View the `set/get` history of a configuration item.
- View all the predefined phases of common and UVM domain.
- Set breakpoints on the important phases or on the phase methods of `uvm_component`.
- View runtime arguments using the **Simulation Arguments** dialog box.
- Filter UVM object items in the **Watch** pane.

- View active and executed sequences.
- View the start and the end time of a sequence.
- View the executing thread of a sequence.
- Set a breakpoint on a sequence method.
- View the sequencer and the sequencer ID.
- View the definition and invocation of a sequence in the **Source View**.
- View the class instance of a sequence in the **Class** pane
- View the execution (start method) of a sequence in the **Stack** pane.
- View the transaction item of a sequence in the **Transaction** pane.
- View relations within the sequences in the **Transaction** pane.
- View a sequence object in the **References** dialog box.
- Add a sequence/sequencer object to the **Watch** pane.
- Add a sequencer stream to the **Wave View** from the **Transaction** pane.
- Navigate to the next/previous/parent/child for sequences and sequence items in the **Transaction** pane and the transaction waveforms.

## 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 any of the `-debug/-debug_pp/-debug_all` options during compilation. The phase recordings are stored in the simulation VPD file.

You can then use DVE to debug the UVM phases in **msglog** window. This is supported for both interactive and post-process 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]`

For example:

```
%vcs -ntb_opts uvm+rvm compile-time_options source_files
```

You can turn off the automatic inclusion of `uvm_vmm_pkg.sv` using the `+define+NO_VMM_UVM_INTEROP` option.

For example:

```
%vcs -ntb_opts uvm+rvm +define+NO_VMM_UVM_INTEROP
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
```

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.

## 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:

**Compilation**

```
% vcs -sverilog -ntb_opts rvm [compile-time_options]
Verilog_files
```

**Simulation**

```
% simv [simv_options]
```

To analyze SystemVerilog files using VMM, use the options `-sverilog` and `-ntb_opts rvm` with `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.

## 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>
```

## Recording OVM Transactions

The OVM version shipped with VCS has additional features that allows you to take advantage of VCS and DVE'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 any of the `-debug` options 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. The `-PP` option can be provided instead of the `-debug` options if only post process debug is desired.

To compile your OVM code, add the `-debug, -debug_pp, or -debug_all` option to your `vcs` command line.

For example:

```
% vcs -sverilog -ntb_opts ovm -debug[_pp/all]\
[compile-time_options]
```

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 DVE to debug the transactions and log messages. This is supported for both interactive and post-process 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 67
- “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”

## 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 15-1 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 15-1 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 no 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 no 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 15-2 illustrates the package export functionality:

*Example 15-2 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 15-3 shows the usage of these system tasks in an initial block

*Example 15-3    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 15-3, because the conditional expression `(Verilog_simulator == "VCS")` is true, VCS displays the following when it compiles and simulates Example 15-3:

```
Smart User!
```

For conditional expression `(Verilog_simulator == "Indecisive")`, VCS displays the following when it compiles and simulates Example 15-3:

```
Warning: "exp1.sv", 21: mod: at time 10
          2
         Expect a performance cost


Fatal: "exp1.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 1IEEE 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 15-4 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 15-5 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 15-6    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 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:

| | |
|---|---|
| `$ln(x)` | Natural logarithm |
| `$log10(x)` | Decimal logarithm |
| `$exp(x)` | Exponential |
| `$sqrt(x)` | Square root |
| `$pow(x,y)` | `x**y` |
| `$floor(x)` | Floor |
| `$ceil(x)` | Ceiling |
| `$sin(x)` | Sine |
| `$cos(x)` | Cosine |

| | |
|---|---|
| `$tan(x)` | Tangent |
| `$asin(x)` | Arc-sine |
| `$acos(x)` | Arc-cosine |
| `$atan(x)` | Arc-tangent |
| `$atan2(x, y)` | Arc-tangent of `x/y` |
| `$hypot(x, y)` | `sqrt(x*x+y*y)` |
| `$sinh(x)` | Hyperbolic sine |
| `$cosh(x)` | Hyperbolic cosine |
| `$tanh(x)` | Hyperbolic tangent |
| `$asinh(x)` | Arc-hyperbolic sine |
| `$acosh(x)` | Arc-hyperbolic cosine |
| `$atanh(x)` | Arc-hyperbolic tangent |
| `$clog2(n)` | 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 15-7 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 15-7   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] lwrd; integer intgr0;
    begin : func_main
      if ( lwrd > 0 )
        begin : ifloop
          lwrd = {lwrd >> 1};
          intgr0 = 1;
          while ( lwrd > 0 )
            begin : wloop
              lwrd = {lwrd >> 1};
              intgr0 = {intgr0 << 1};
            end : wloop
        end : ifloop
      else
        intgr0 = 0;   // return 0 when lwrd <= 0
      const_func = intgr0;
    end : func_main
 endfunction
endmodule
```

constant function call

constant function

## 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 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, VCShandles a missing external constraint block as an empty constraint block. This is not an error condition and VCSjust 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, VCScontinues 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 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 15-8 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 15-9   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 15-10 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 15-11 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.i1 is -1241023056


at 20
cc.st1.b1 is 3
cc.st1.i1 is -1877783293


at 30
cc.st1.b1 is 1
cc.st1.i1 is 629780255


at 40
cc.st1.b1 is 3
cc.st1.i1 is 469272579
```

Here is another code example:

*Example 15-12 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 15-13 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 15-14 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 15-15 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 15-16    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 subprocesses (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 subprocesses, not just the immediate child subprocesses, 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 subprocesses 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 subprocess 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 (subprocesses) 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 imported 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 15-17 and Example 15-8. This technique is also implemented for importing from a package to an interface.

*Example 15-17    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 15-7, 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 15-18 shows importing from packages in a program header.

*Example 15-18    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) .

## 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 15-19 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 MX, parameter overriding rules are not supported if the design hierarchy crosses the VHDL boundary.

## 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"
- "Single-Sized Packed Dimension Extension"
- "Covariant Virtual Function Return Types"
- "Self Instance of a Virtual Interface"


### Unique/Priority Case/IF Final Semantic Enhancements

The behavior of the compliance checking keywords `unique` and `priority` for `case` and for `if...else if...else` selection statements as defined in the 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 `-xlrm 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 `-xlrm 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

`-xlrm 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 15-20 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 `-xlrm uniq_prior_final`:

```
%> vcs -sverilog test.sv -R

Executing Case with condition value 0
RT Warning: More than one conditions match in 'unique case'
statement.
        "unique_case.sv", line 12, for top.
        Line    13 &    14 are overlapping at time     0.
Executing Case with cond value 0
RT Warning: More than one conditions match in 'unique case'
statement.
        "unique_case.sv", line 12, for top.
        Line    13 &    14 are overlapping at time     0.

 TIME 1 ns : cond value 1, a value 4
 Executing Case with cond value 1
RT Warning: More than one conditions match in 'unique case'
statement.
        "unique_case.sv", line 6, for top.
        Line     7 &     8 are overlapping at time     1.

 TIME 1 ns: cond value 0, a value 5
 Executing Case with cond value 0
RT Warning: More than one conditions match in 'unique case'
```

```
statement.
        "unique_case.sv", line 12, for top.
        Line    13 &    14 are overlapping at time     1.
```

### Simulation output with `-xlrm uniq_prior_final` compile-time option:

```
%> vcs -sverilog test.sv -xlrm uniq_prior_final -R
Executing Case with cond value 0:
RT Warning: More than one conditions match in 'unique case'
statement.
        "unique_case.sv", line 12, for top.
        Line    13 &    14 are overlapping at time     0.

 TIME 1 ns : cond value 1, a value 4
 Executing Case with cond value 1

 TIME 1 ns: cond value 0, a value 5
 Executing Case with cond value 0
RT Warning: More than one conditions match in 'unique case'
statement.
        "unique_case.sv", line 12, for top.
        Line    13 &    14 are overlapping at time     1.
```

*Example 15-21    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 `-xlrm`:

```
%> 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 `-xlrm uniq_prior_final`:

```
%> vcs -sverilog test.sv -xlrm 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 15-22   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 `-xlrm` 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 `-xlrm uniq_prior_final`:**

```
%> vcs -sverilog test.sv -xlrm 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 `-xlrm 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 15-23 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 `-xlrm 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 SIEEE 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` 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 -xlrm
uniq_prior_observed test.v

% simv
```

VCS MX 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.
```

## 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] un1;
enum logic [8] {IDLE, XX=8'bxxxxxxxx, 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 15-24 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)
```

# 16

# 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 subclasse(s). 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 bolde 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 16-1 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 16-1* ***before*** *Advice*

Target method:

```
class packet;
   task myTask();
        $display("Executing original code\n");
   endtask
```

```
endclass
```

Advice:

```
    before task myTask ();
             $display("Before in aoe1\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 aoe1\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 16-2* ***after*** *Advice*

Target method:

```
class packet;
   task myTask();
        $display("Executing original code\n");
   endtask
endclass
```

Advice:

```
    after task myTask ();
             $display("Before in aoe1\n");
    endtask
```

Weaving of the advice in the target method yields the following.

```
task myTask_newTarget();
     myTask();
     myTask_after();
endtask

task myTask();
        $display("Executing original code\n");
endtask

task myTask_after ();
        $display("After in aoe1\n");
endtask
```

As a result of weaving, all the method calls to `myTask()` in the input program code are replaced by method calls to `myTask_newTarget()`. Also, `myTask_newTarget()` replaces `myTask` as the target method for `myTask()`.

*Example 16-3* ***around** Advice*

Target method:

```
class packet;
   task myTask();
        $display("Executing original code\n");
   endtask
endclass
```

Advice:

```
around task myTask ();
        $display("Around in aoe1\n");
endtask
```

Weaving of the advice in the target method yields the following:

```
task myTask_around();
     $display("Around in 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()`. 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 16-4* ***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 aoe1\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 that 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 16-5 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 16-5, 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 16-5 leads to the expansion of code in the manner shown in Example 16-6. Advices are woven in the order of increasing precedence {2, 3, 4, 1} as explained earlier.

*Example 16-6 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

	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 16-6 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 16-7 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 16-7 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 16-8 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 16-8 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 16-9 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 16-9 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 16-10 shows the program in which `aoe1` dominates `aoe2`.

*Example 16-10 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 16-11 shows the program in which `aoe2` dominates `aoe1`.

*Example 16-11   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 16-11, 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 16-12* ***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 16-12 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 16-13* ***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 16-13. 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 16-14, the print method introduced by the extends directive hides the print method in the super class.

*Example 16-14 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 16-15 illustrates how the two hide permissions can change at different aspect items within an aspect extension.

*Example 16-15 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 forfieted

        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 16-16 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 16-16    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 16-17 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 16-17    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 16-18, the `extends` directive adds advices to the `packet::send` method.

*Example 16-18* ***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 16-18 is executed, the output is as follows:

```
Before sending packet
Sending data
After sending packet
```

In Example 16-19, extends directive `myaspect` adds advice to turn off constraint `c1` before each call to the `foo::pre_randomize` method.

*Example 16-19 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 16-20, extends directive `myaspect` adds advice to set a property named valid to 0 after each call to the `foo::post_randomize` method.

*Example 16-20   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 16-21 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 16-21 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 16-21 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
```

# 17

# 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 Debug Using DVE”
- “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 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 17-1 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 17-1, 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 17-2 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 17-2, 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 17-3   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 17-3, 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 17-4    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 17-4, 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 17-5 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 17-5, 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 17-6 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 17-6, 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 17-7   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 17-7, 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 17-8    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 17-8, 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 17-9    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 17-9, 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 17-10 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 17-1, 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 17-11 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 17-4, 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 17-12 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 17-12, 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 a1-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 17-13    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 activer 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 17-14 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 solver failed when solving following set of constraints

```
rand integer y; // rand_mode = ON
rand integer z; // rand_mode = ON
rand integer x; // rand_mode = ON
constraint c // (from this) (constraint_mode = ON)
{
( x < 1 ) ;
( x in { 3 , 5 , 7 : 11 } ) ;
}
```

You can use the `+ntb_enable_solver_trace_on_failure=0|1|2|3` runtime option as follows:

`0` Print a one-line failure message with no details.

| | |
|---|---|
| `1` | Print only the failure subset (this is the default). |
| `2` | Print the entire constraint problem and 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 $n^{th}$ partition of the $m^{th}$ 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:
01x0z11xzxxx11zx1xz0zx100zxxzzz0zxxzzzzxzzzxxzxzzzzzxzzzzz
xxzxxz
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 15th `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 debugging capability is also in DVE, including a similar solver trace capability to understand the details of a `randomize()` call and many graphical user interface features, such as cross-probing, search, and filters to make debugging constraints faster and easier. For more information, see the *DVE User Guide*.

## 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/VCSi 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 number = 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 MX 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.

## 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 Debug Using DVE

DVE supports constraint debugging and allows you to do the following:

- View static information about the constraint problem (including constraint blocks that are overridden due to class inheritance) in the Class Browser and Member Pane
- Browse dynamic information of the object being randomized, including `rand_mode`, `constraint_mode`, and object ID, in the Local Pane
- Use flexible solver breakpoint infrastructure to stop the simulation during interactive debug from the Breakpoint dialog box
  - Using randomize serial IDs: For example, stop the simulation at the 10th randomize call
  - Specifying the solver conditions on the randomization call. For example:
    - Stop the simulation when 'packet' class is randomized and when the solution has 'addr == 10' (addr is a rand data member of the packet class)

- Stop the simulation when there is constraint inconsistency from randomizations of any (*) class

- Step into the constraint solver after the simulation stops at a randomize statement, to easily navigate to the solution and relation spaces in the Constraints dialog box
  - Browse the solver trace in the Constraints dialog box showing partitioning of random variables, initial value ranges of random variables, and their current solved values
  - View the relationship between random variables and the constraints that establish such relationship
  - Find the random variables or constraint blocks of interest quickly using the “Search” feature with drag-and-drop, “add to search” option, and so on.
  - Change the radix for the variable and constraint expressions to improve readability
  - Extract test case for the randomize call or selected partitions in the randomize call
  - Debug inconsistency constraint – showing a minimal set of constraints causing the inconsistency
  - Debug default constraints – showing the overridden default variables and constraints
  - Debug soft constraints – showing which soft constraints are honored and which ones are dropped
- Interactively modify the constraint problem from the Local Pane without recompile
  - Control `rand_mode` of random variables

- Control `constraint_mode` of constraint blocks
- Add new constraint expressions to the constraint problem
- Disable selected constraint expressions from active constraint blocks

- Re-randomize on-the-fly without recompiling the design
  - Re-randomize the modified problem and step into the constraint solver
  - View the histogram of solutions for random variables after multiple re-randomizations to debug distribution from the Constraints dialog box.

For more details, refer to the “Debugging Constraints” section in the *DVE User Guide*.

## 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 MX 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 17-15 are equivalent inside the VCS constraint solver.

*Example 17-15    Guarded Expressions*

```
if (a | b | c)
     {
     obj.x == 10;
     }
```

-or-

```
(a | b | c) -> (obj.x == 10);
```

In Example 17-15, 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 17-16 shows a pure DPI function in C.

*Example 17-16 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 17-17 shows how to call a pure DPI function from constraints.

*Example 17-17    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 17-18 with the C code shown in Example 17-16 results in an error because the import DPI function is not explicitly marked as `pure`.

*Example 17-18    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 17-19 with the C code shown in Example 17-16 results in an error because `context` import DPI functions cannot be called from constraints.

*Example 17-19 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 17-16) 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 17-20 has undefined behavior (and may even cause a crash).

*Example 17-20 Non-pure DPI Function in C*

```
#include <stdio.h>
#include <stdlib.h>
#include "svdpi.h"

int readValueOfBFromFile(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 = readValueOfBFromFile("/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 17-21 shows a DPI function marked `pure` that is actually doing non-pure activities. This results in an error.

*Example 17-21 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 bit [5:2] arr1;
rand bit [-1:0][5:2] arr2;
constraint Cons1 {
foreach(arr1[i]) {
arr1[i] == 1;
}
}
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 17-22 The foreach Iterative Constraint for Packed Arrays*

all dimensions packed

```
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
```

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 17-23   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;
    } header;
   header hd;


function new();
        this.hd.p1 = new;
endfunction

endclass
```

**randomized object in a structure**

```
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'h10000000000.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 17-24 shows a soft constraint:

*Example 17-24 Soft Constraint*

```
class A;
   rand int x;
constraint c1 {
     soft x > 2; // soft constraint
}
endclass
```

Example 17-25 shows a hard constraint.

*Example 17-25 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:

priority(SC2) > priority(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 17-26, priority(SC2) > priority(SC1).

*Example 17-26 SC2 Higher Priority than SC1*

```
class A;
     rand int x;
constraint c1 {
     soft x > 10; // SC1
     soft x > 5; // SC2
     }
endclass
```

In Example 17-27, priority(SC2) > priority(SC1).

*Example 17-27 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 17-28, 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 17-28 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 17-28 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 17-29, priority(SC2) > priority(SC1).

*Example 17-29 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 priority(`SC2`) > priority(`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 17-1.

## 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 17-30, priority(SC3) > priority(SC2) > priority(SC1).

*Example 17-30 SC3 Higher Priority than SC1*

```
class A;
     rand int a;
```

Table 17-1 Examples with the Dominates_list

| **Dominates_list variations** | **Resulting AOP precedence and soft constraint priority** | **Solver result for x** |
|---|---|---|
| extends A_aop2(A) dominates (A_aop5);<br>constraint c2 {<br>soft x == 2; // SC2<br>}<br>constraint c3 {<br>soft x == 3; // SC3<br>}<br>endextends<br><br>extends A_aop4(A);<br>constraint c4 {<br>soft x == 4; // SC4<br>}<br>endextends<br><br>extends A_aop5(A);<br>constraint c5 {<br>soft x == 5; // SC5<br>}<br>endextends | • Precedence set by dominates: `A_aop2 > A_aop5`<br>• No specific precedence set between `A_aop2` and `A_aop4`. Therefore, the order of declaration depends on the precedence order between the two as follows: `A_aop4 > A_aop2`<br>• Overall precedence order is set as:`A_aop4 > A_aop2 > A_aop5`<br><br>This means the soft constraint from `A_aop5` is appended last to the class A, making this soft constraint (`SC5`) the highest soft priority. | 5 |

## Table 17-1 Examples with the Dominates_list

| **Dominates_list variations** | **Resulting AOP precedence and soft constraint priority** | **Solver result for x** |
|---|---|---|
| `extends A_aop2(A) dominates (A_aop4);`<br>`constraint c2 {`<br>`soft x == 2; // SC2`<br>`}`<br>`constraint c3 {`<br>`soft x == 3; // SC3`<br>`}`<br>`endextends`<br><br>`extends A_aop4(A);`<br>`constraint c4 {`<br>`soft x == 4; // SC4`<br>`}`<br>`endextends`<br><br>`extends A_aop5(A);`<br>`constraint c5 {`<br>`soft x == 5; // SC5`<br>`}`<br>`endextends` | • Precedence set by dominates: `A_aop2 > A_aop4`<br>• 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 > A_aop2`<br>• Overall precedence order:is set as `A_aop5 > A_aop2 > A_aop4`<br><br>This means the soft constraint from A_aop4 is appended last to the class A, making this soft constraint (`SC4`) the highest soft priority. | 4 |

Table 17-1 Examples with the Dominates_list

| Dominates_list variations | Resulting AOP precedence and soft constraint priority | Solver result for x |
|---|---|---|
| `extends A_aop2(A);`<br>`constraint c2 {`<br>`soft x == 2; // SC2`<br>`}`<br>`constraint c3 {`<br>`soft x == 3; // SC3`<br>`}`<br>`endextends`<br><br>`extends A_aop4(A) dominates (A_aop5;`<br>`constraint c4 {`<br>`soft x == 4; // SC4`<br>`}`<br>`endextends`<br><br>`extends A_aop5(A);`<br>`constraint c5 {`<br>`soft x == 5; // SC5`<br>`}`<br>`endextends` | • Precedence set by dominates: `A_aop4 > A_aop5`.<br>• 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 > A_aop2`.<br>• Overall precedence order is set as: `A_aop4 > A_aop5 > A_aop2`.<br><br>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. | 3 |

```
    rand int b;
    constraint c1 {
        soft a == 2; // SC1
    }
endclass

A objA;
    objA.randomize () with {
    soft a > 2; // SC2
    soft b == 1; // SC3
    }
```

## 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 17-31).

*Example 17-31 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 17-31, 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 17-32).

*Example 17-32 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 17-32, 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 17-32 with Example 17-33, where the `disable soft` constraint is omitted.

*Example 17-33 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 17-33, 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.

### Limitation

VCS has the following limitation on the usage of random variables in the guard expression of `disable soft` constraint:

- Random variables in the guard expression of `disable soft` constraint is not supported and displays an error message.

*Example 17-34 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 17-34, `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 17-35    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
```



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 17-35 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 17-36    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 17-37 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};

            // 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
```

## Limitations

The feature has the following limitations:

- Randomization of MDA is not supported with unique constraints.
- 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 17-38 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 17-39 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 17-40 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
```

# 18

# 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++.

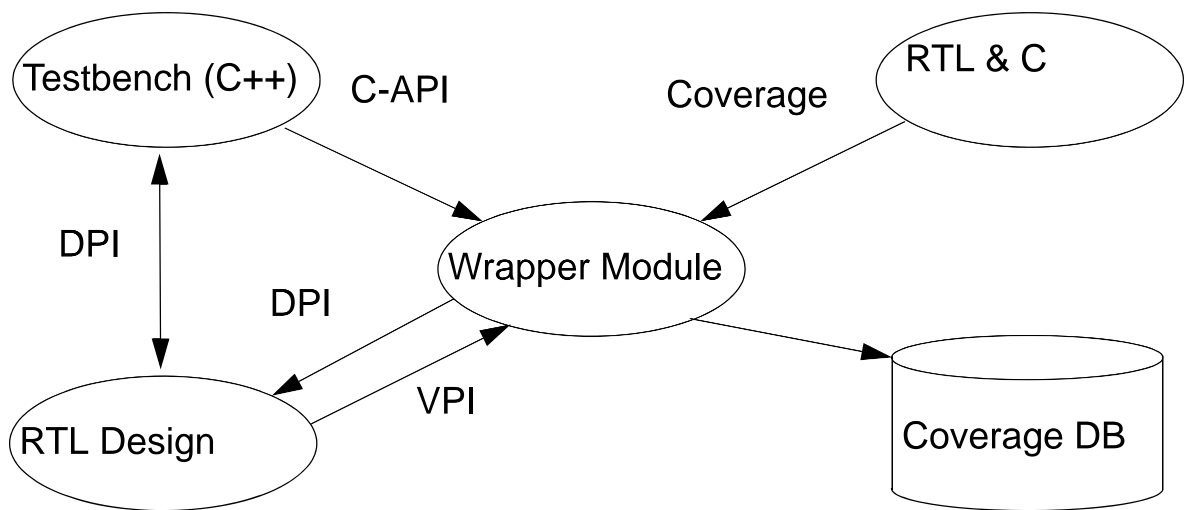
The SystemVerilog C/C++ interface feature provides an application programming interface (API) so that C/C++ code can use the SystemVerilog functional coverage infrastructure to verify its coverage.

Note:
: When you use the SystemVerilog C/C++ interface feature, you need to include the header file, `svCovgAPI.h`.

## SystemVerilog Functional Coverage Flow

Figure 18-1 illustrates the functional coverage flow:

*Figure 18-1 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 18-1 shows the mapping of the supported types:

*Table 18-1 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 *vl*);**

#### Parameters

*cgName*

Covergroup name.

*ciName*

Covergroup instance name (should be unique).

*refType*

`SV_SAMPLE_REF` or `SV_SAMPLE_VAL`.

*args*

A variable number of arguments for creating a new covergroup instance.

*vl*

Represents a C predefined data structure (`va_list`) for maintaining a list of arguments.

### Description

Create a covergroup instance using the covergroup and instance names. If 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).

# 19

# 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:

| SystemVerilog | OpenVera | 2/4 or 4/2 value conversion? | Change in sign? |
|---|---|---|---|
| `integer` | `integer` | N (equivalent types) | N (Both signed) |
| `byte` | `reg [7:0]` | Y | Y |
| `shortint` | `reg [15:0]` | Y | Y |
| `int` | `integer` | Y | N (Both signed) |
| `longint` | `reg [63:0]` | Y | Y |
| `logic [m:n]` | `reg [abs(m-n)+1:0]` | N (equivalent types) | N (Both unsigned) |
| `bit [m:n]` | `reg [abs(m-n)+1:0]` | Y | N (Both unsigned) |
| `time` | `reg [63:0]` | Y | N (Both unsigned) |

Note:

If a value or sign conversion is needed between the actual and formal arguments of a task or function, then the argument cannot be passed by reference.

## Arrays

Arrays can be passed as arguments to tasks and functions from SystemVerilog to OpenVera and vice versa. The formal and actual array arguments must have equivalent element types, the same number of dimensions with corresponding dimensions of the same length. These rules follow the SystemVerilog LRM.

- A SystemVerilog fixed array dimension of the form `[m:n]` is directly mapped to `[abs(m-n)+1]` in OpenVera.
- An OpenVera fixed array dimension of the form `[m]` is directly mapped to `[m]` in SystemVerilog.

Rules for equivalency of other (non-fixed) types of arrays are as follows:

- A dynamic array (or Smart queue) in OpenVera is directly mapped to a SystemVerilog dynamic array if their element types are equivalent (can be directly mapped).
- An OpenVera associative array with unspecified key type (for example, `integer a[]`) is equivalent to a SystemVerilog associative array with key type `reg [63:0]` provided the element types are equivalent.
- An OpenVera associative array with 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 Svobj=new();
SVobj.randomize();
obj.randomize() with
{obj.ri==SVobj.var;};
```

## Functional Coverage

There are some differences in functional coverage semantics between OpenVera and SystemVerilog. These differences are currently being eliminated by changing OpenVera semantics to conform to SystemVerilog. In 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.

### Compilation

```
% vcs [compile_options] -sverilog -ntb_opts interop
[other_NTB_options] file4.sv file5.vr file2.v file1.v
```

### Simulation

```
% simv [simv_options]
```

Note:

- If RVM class libs are used in the OpenVera code, use -ntb_opts rvm with the vlogan command.
- Using -ntb_opts interop -ntb_opts rvm with vcs, 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).

# 20

# 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”
- “Using SystemVerilog Constructs Inside vunits”
- “Calling $error Task When Else Block is Not Present”
- “Disabling Default Assertion Success Dumping in -debug_pp 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"

### Using VCS Checker Library

VCS provides you with SVA checkers, which can be directly instantiated in your Verilog source files. You can find these SVA checkers files in `$VCS_HOME/packages/sva_cg` directory.

This section describes the use model to compile and simulate the design with SVA checkers. For more information on SVA checker libraries and list of available checkers, see the *SystemVerilog Assertions Checker Library Reference Manual*.

### Instantiating SVA Checkers in Verilog

You can instantiate SVA checkers in your Verilog source just like instantiating any other Verilog module. For example, to instantiate the checker `assert_always`, specify 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:

### 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*.

## 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

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 compile and simulate the designs having inlined assertions is as follows:

### Compilation

```
% vcs -sv_pragma [compile_options] Verilog_files
```

### Simulation

```
% simv [run_options]
```

## 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 sub-divide the module into multiple modules and distribute assertions over those modules.

# Controlling SystemVerilog Assertions

SVAs can be controlled or monitored using:

- “Compilation and Runtime Options”
- “Concatenating Assertion Options” on page 11
- “Assertion Monitoring System Tasks”
- “Using Assertion Categories”

## Compilation and Runtime Options

VCS provides various compilation options to perform the following tasks:

- The following is the list of assertion options that are enabled when the `-assert enable_diag` compile-time 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` compile-time 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 enable dumping assertion information in a VPD file, use the `-assert dve` option. This option also allows you to view assertion information in the assertion pane in DVE (for more information, see the *DVE User Guide*.)
- 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*...

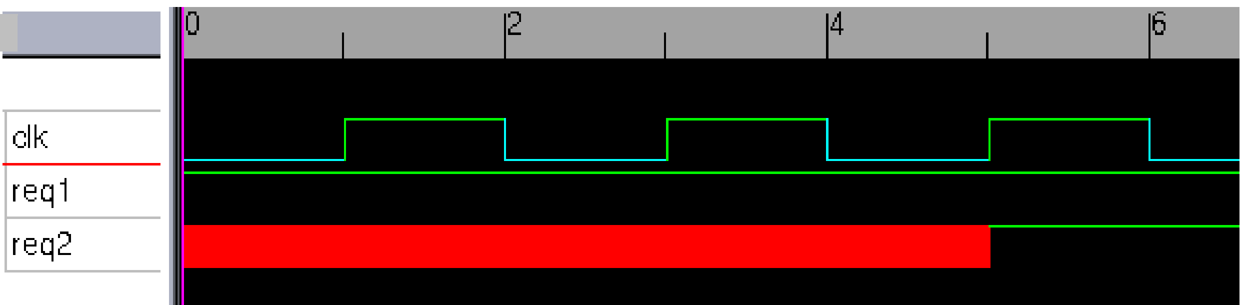
A comma separated list of assertions. If one of these assertions is not declared in the module definition containing this system task, specify it by its hierarchical name.

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
- Stopping and Restarting Assertions By Category
  - Starting and Stopping Assertions Using Assertion System Tasks

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 "Stopping and Restarting Assertions By Category" .

## Using Attributes

You can prefix an attribute in front of an `assert` statement to specify the category of the assertion. The attribute must begin with the `category` name and specify an integer value, for example:

```
(* category=1 *) a1: assert property (p1);
(* category=2 *) a2: assert property (s1);
```

The value you specify can be an unsigned integer from 0 to $2^{24}$ - 1, or a constant expression that evaluates to 0 to $2^{24}$ - 1.

You can use a `parameter`, `localparam`, or `genvar` in these attributes. For example:

```
parameter p=1;
localparam l=2;
.
.
.
(* category=p+1 *) a1: assert property (p1);
(* category=l *) a2: assert property (s1);

genvar g;
generate
for (g=0; g<1; g=g+1)
begin:loop
(* category=g *) a3: assert property (s2);
end
endgenerate
```

Note:

In a `generate` statement the category value cannot be an expression, the attribute in the following example is invalid:

```
genvar g;
generate
```

```
for (g=0; g<1; g=g+1)
begin:loop
(* category=g+1 *) a3: assert property (s2);
end
endgenerate
```

If you use a `parameter` for a category value, the parameter value can be overwritten in a module instantiation statement.

You can use these attributes to assign categories to both named and unnamed assertions. For example:

```
(* category=p+1 *) a1: assert property (p1);
(* category=1 *) assert property (s1);
```

The attribute is retained in a `tokens.v` file when you use the `-Xman=0x4` compile-time option and 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 20-1 stops the odd numbered categories

*Example 20-1 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 | 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 20-2 uses the *`globalDirective`* argument.

*Example 20-2    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:

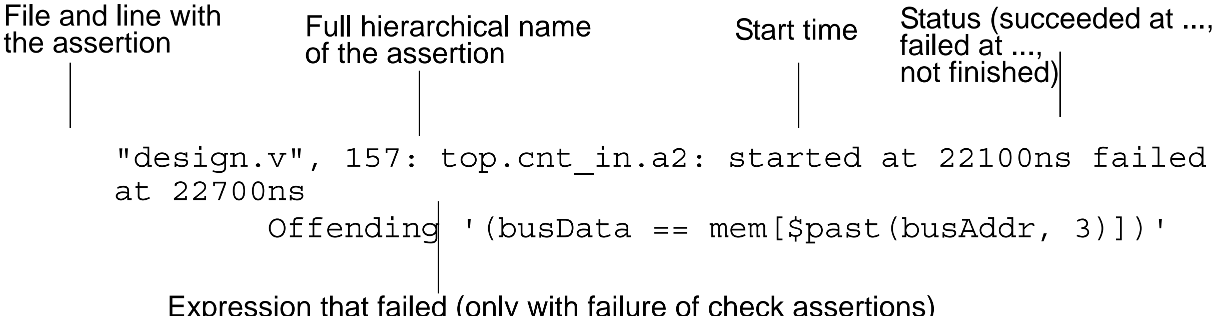
- Using a Report File
- Using DVE

For information on viewing assertions using DVE, refer to the "*Using the Assertion Pane*" chapter, in the DVE user 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:



## 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
```

This feature allows you to debug simple assertions directly from the failure message and avoids more complex debug process using DVE.

The values of the variables contained in the failing portion of a property are generated using the following formats:

*Table 20-1 Reporting Formats*

| **Variable Type** | **Format** |
|---|---|
| Simple scalar bit or logic variables | <var_name> = %b |
| Bitvector variables | <var_name> = %h |
| Int and integer variables | <var_name> = %0d |
| Real, realtime variables | <var_name> = %g |
| Sampled value function calls | <$sampled_function_call(argument list)> = %b<br><br>**Note:** 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 | <expr> = %h |
| User-defined function calls | <function_name>(list of var_name - value pairs) |
| 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 20-3,

*Example 20-3 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 20-4 test.ucli*

```
run 5
call {$uniq_prior_checkoff}
run 2
call {$uniq_prior_checkon}
run
exit
```

Consider Example 20-3 and Example 20-4. 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 20-5,

*Example 20-5    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_checkon` 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
  -xlrm 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))
                   0AS is called
                   0AC is called
                   0AC 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 -xlrm 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)
                   0AS is called
                   0AC is called
                   0AC 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 20-2. 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 20-2, you can use the `+lint=<ID>:<N>` option to control the display of messages.

*Table 20-2 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` |
| 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` |

| Assertion Rule Description | Lint Message |
| --- | --- |
| Using `(clk iff gate_expr)` inside an assert property | `Lint-[SVA-LCE] Logical 'AND' in clock expression` |
| 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 dve`, `-assert enable_diag`, `-debug`, `-debug_all`, `-debug_pp`, `-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 20-6.

*Example 20-6 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:

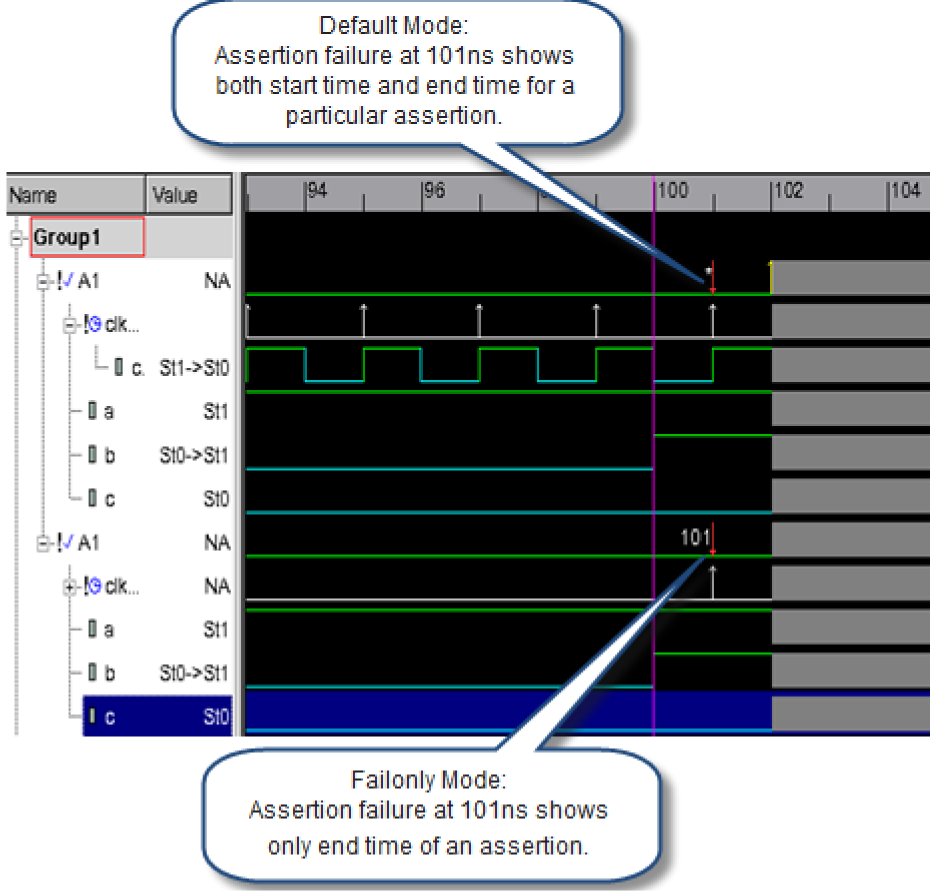
```
"test.v", 6: top.A1: started at 99ns failed at 101ns
        Offending 'c'
"test.v", 6: top.A1: started at 97ns failed at 101ns
        Offending 'c'
..........................
"test.v", 6: top.A1: started at 1ns failed at 101ns
        Offending 'c'
```

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 DVE:
  - Supports `-debug_pp` and `-debug_access` options.
  - DVE displays only one failure and reports the end time.
  - For debugging, it is better to use the default mode.

*Figure 20-1    Behavior with DVE*




- Behavior with coverage:
  - Reports unique matches only
  - The `-cm assert` option is not supported with the `-assert failonly` option.

  Consider Example 20-7.

*Example 20-7    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.

## 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>
```

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) 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_pp Option

By default, VCS does not dump the SystemVerilog assertion successes to the VPD file when the `-debug_pp` 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$^{TM}$-2012. For more details, see IEEE 1800$^{TM}$-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:

- Use `-assert dve` at compile time to enable debug for assertions. While basic debug support is available with this release, assertion tracing in DVE is not supported completely. DVE provides information, such as `start_time`, `end_time` for every attempt and statistics for every assertion/cover. DVE also groups all signals involved in an assertion on tracing an attempt. However, the extra hints that are provided for SVA constructs are not available for the new constructs as of now.
- 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

Use `-assert dve` at compile time to enable debug for assertions. While basic debug support is available with this release, assertion tracing in DVE is not supported completely. DVE provides information, such as `start_time`, `end_time` for every attempt and statistics for every assertion/cover. DVE also groups all signals involved in an assertion on tracing an attempt. However, the extra hints that are provided for SVA constructs are not available for the new constructs as of now.

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.

# 21

# Using Property Specification Language

VCS supports the Simple Subset of the IEEE 1850 Property Specification Language (PSL) standard. Refer to Section 4.4.4 of the *IEEE 1850 PSL LRM* for the subset definition.

You can use PSL along with SystemVerilog Assertions (SVA), SVA options, SVA system tasks, and OpenVera (OV) classes.

## Including PSL in the Design

You can include PSL in your design in any of the following ways:

- Inlining the PSL using the `//psl` or `/*psl */` pragmas in Verilog and SystemVerilog.
- Specifying the PSL in an external file using a verification unit (`vunit`).

## Examples

The following example shows how to inline PSL in Verilog using the `//psl` and `/*psl */` pragmas.

```
module mod;
  ....
  // psl a1: assert always {r1; r2; r3} @(posedge clk);

  /* psl
     A2: assert always {a;b} @(posedge clk);
     ...
  */
endmodule
```

The following example shows how to use `vunit` to include PSL in the design.

```
vunit vunit1 (verilog_mod)
{
  a1: assert always {r1; r2; r3} @(posedge clk);
}
```

# Use Model

If you inline the PSL code, you must compile it with the `-psl` option.

If you use `vunit`, you must compile the file that contains the `vunit` with the `-pslfile` option. You do not need to use this option if the file has the `.psl` extension.

### Compilation

```
% vcs -psl [vcs_options] Verilog_files
```

### Simulation

```
% simv
```

## Examples

To simulate the PSL code inlined in a Verilog file (`test.v`), execute the following commands:

```
% vcs -psl test.v
% simv
```

To simulate the `vunit` specified in an external file with the `.psl` extension (`checker.psl`), execute the following commands:

```
% vcs dev.v checker.psl
% simv
```

To simulate the `vunit` specified in an external file without the `.psl` extension (`checker.txt`), execute the following commands:

```
% vcs dev.v -pslfile checker.txt
% simv
```

To simulate both the PSL code inlined in a Verilog file (`test.v`), and the `vunit` specified in an external file (`checker.psl` or `checker.txt`), execute the following commands:

```
% vcs -psl -test.v checker.psl
% simv
```

or

```
% vcs -psl -test.v -pslfile checker.txt
% simv
```

## Using SVA Options, SVA System Tasks, and OV Classes

VCS enables you to use all assertion options with SVA, PSL, and OVA. For example, to enable PSL coverage and debug assertions while compiling the PSL code, execute the following commands:

```
% vcs  -psl -cm assert -debug -assert enable_diag test.v
% simv -cm assert -assert success
```

For information on all assertion options, see Appendix - Compile Time Options .

You can control PSL assertions in any of the following ways:

- Using the `$asserton`, `$assertoff`, or `$assertkill` SVA system tasks.
- Using NTB-OpenVera assert classes.

Note that VCS treats the `assume` PSL directive as the `assert` PSL directive.

Discovery Visualization Environment (DVE) supports PSL assertions. The PSL assertion information displayed by VCS is similar to SystemVerilog assertions.

## Limitations

The VCS implementation of PSL has the following limitations:

- VCS does not support binding `vunit` to an instance of a module or entity
- VCS does not support 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

# 22

# 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 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 Verilog RTL design under test in your testbench
- Verify a Verilog netlist after synthesis with the original SystemC testbench
- Write test benches in SystemC to check the correctness of Verilog designs
- Import legacy Verilog IP into a SystemC description

- Import third-party Verilog IP into a SystemC description
- Export SystemC IP into a Verilog environment when only a few of the design blocks are implemented in SystemC
- Use SystemC to provide stimulus to your design

The VCS /SystemC Co-simulation Interface creates the necessary infrastructure to co-simulate SystemC models with Verilog models. The infrastructure consists of the required build files and any generated wrapper or stimulus code. VCS writes these files in subdirectories in the `./csrc` directory. To use the interface, 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/VCS MX User Guide*.
- Examples of Verilog instantiated in SystemC and SystemC instantiated in Verilog 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 MX for your design, see the ***VCS/VCS MX SystemC User Guide*** available in the SolvNet online support site.

# 23

# 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 DirectC”
- Using SystemC - See Using SystemC
- Using SystemVerilog DPI routines - See the SystemVerilog LRM.

Note:
PLI1.0 refers to TF and ACC routines, and PLI2.0 refers to VPI.

# Using PLI

PLI is the programming language interface (PLI) between C/C++ functions and VCS. It helps 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 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 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 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 1.0 or PLI 2.0 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`

Disables the dumping of value change and simulation time data of modules defined under the `` `celldefine `` compiler directive into a VPD file created by the `$vcdpluson` system task. This capability is only used for batch simulation.

`persistent`

Checks if the specified function is defined in the PLI application, even if the corresponding system task or function is not used in the Verilog file. If the function is not found or defined in the PLI application, VCS exits with an undefined reference error message.

Note that if you use the `-debug`, `-debug_all`, or `-debug_pp` options during compilation, VCS performs these checks on every function mapped in the tab file.

To ignore this check, which is enabled by the these debug options or the `persistent` specification, set the `PERSISTENT_FLAG` environment variable to `1`.

**Example 1**

```
$val_proc call=val_proc check=check_proc misc=misc_proc
```

In this line, VCS calls the function named `val_proc` when it executes the associated user-defined system task named `$val_proc`. It calls the `check_proc` function at compile time to see if the user-defined system task has the correct syntax, and calls the `misc_proc` function in special circumstances, 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|sig_name|inst_name
[+]|%CELL|%TASK|*
```

Were,

acc

Keyword that begins a line for specifying access capabilities.

=|+=|-=|:=

Operators for adding, removing, or changing access capabilities. The operators in this syntax are as follows:

=

A shorthand for +=.

+=

Specifies adding the access capabilities that follow to the parts of the design that follow, as specified by module name,%CELL,%TASK, or * wildcard character.

-=

Specifies removing the access capabilities that follow from the parts of the design that follow, as specified by module name,%CELL,%TASK, or * wildcard character.

:=

Specifies changing the access capabilities of the parts of the design that follow, as specified by module name, %CELL,%TASK, or * wildcard character, to only those in the list of capabilities on this specification. A specification with this operator can change the capabilities specified in a previous specification.

capabilities

Comma-separated list of access capabilities. The capabilities that you can specify for the functions in your PLI specifications are as follows:

`r` or `read`

Reads the values of nets and registers in your design.

`rw` or `read_write`

Both reads from and writes to the values of registers or variables (but not nets) in your design.

`wn`

Enables writing values to nets.

`cbk` or `callback`

To be called when named objects (nets registers, ports) change value.

`cbka` or `callback_all`

To be called when named and unnamed objects (such as primitive terminals) change value.

`frc` or `force`

Forces values on nets and registers.

`prx` or `pulserx_backannotation`

Sets pulse error and pulse rejection percentages for module path delays.

`s` or `static_info`

Enables access to static information, such as instance or signal names and connectivity information. Signal values are not static information.

`tchk` or `timing_check_backannotation`

Back-annotates timing check delay values.

`gate` or `gate_backannotation`

Back-annotates delay values on gates.

`mp` or `module_path_backannotation`

Back-annotates module path delays.

`mip` or `module_input_port_backannotation`

Back-annotates delays on module input ports.

`mipb` or `module_input_port_bit_backannotation`

Back-annotates delays on individual bits of module input ports.

`module_names`

Comma-separated list of module identifiers or names.

Specifying modules enables, disables, or changes (depending on the operator) the ability of the PLI function to use the access capability in all instances of the specified module.

`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>`

Note:

- Signal names defined inside the generate block are not supported.
- Structure and interface type is not supported.

`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 flow is not supported.

`+`

Specifies adding, removing, or changing the access capabilities for not only the instances of the specified modules but also the instances hierarchically under the instances of the specified modules.

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.

`*`

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 might impede 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 23-1 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 23-2 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 },
accForceFlag };
  static s_setval_value value_s = {accIntVal};
  value_s.value.integer =2;
  acc_set_value(reg, &value_s, &delay_s);

}
```

*Example 23-3 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*[+]|%CELL|*

Here:

`acc`

Keyword that begins a line for specifying access capabilities.

`=|+=|-=|:=`

Operators for adding, removing, or changing access capabilities.

`capabilities`

Comma separated list of access capabilities.

`module_names`

Comma separated list of module identifiers. The specified access capabilities are added, removed, or changed for all instances of these modules.

`+`

Specifies adding, removing, or changing the access capabilities for not only the instances of the specified modules, but also the instances hierarchically under the instances of the specified modules.

`%CELL`

Specifies all modules compiled under the `` `celldefine `` compiler directive and all modules in the Verilog library directories and library files (as specified with the `-y` and `-v` options.)

`*`

Specifies all modules in the design. Using a wildcard character is no more efficient than using the `-debug` option with vcs.

The access capabilities and the interactive commands are as follows:

| ACC Capability | What it enables your PLI functions to do |
|---|---|
| `r` or `read` | For specifying “reads” in your design, it enables commands for performing the following:<br>• Creating an alias for another UCLI command (`alias`)<br>• Displaying UCLI help<br>• Specifying the radix of displayed simulation values (`oformat`)<br>• Displaying simulation values<br>• Descending and ascending the module hierarchy<br>• Depositing values on registers<br>• Displaying the set breakpoints on signals<br>• Displaying the port names of the current location, and the current module instance or scope, in the module hierarchy<br>• Displaying the names of instances in the current module instance or scope |

| ACC Capability | What it enables your PLI functions to do |
| --- | --- |
| | • Displaying the nets and registers in the current scope<br>• Moving up the module hierarchy<br>• Deleting an alias for another UCLI command<br>• Ending the simulation |
| `rw` or `read_write` | For specifying “reads and writes” in your design but `r` enables everything that `rw` does. A longer way to specify this capability is with the `read_write` keyword. |
| `cbk` or `callback` | Commands perform the following tasks:<br>• Setting a repeating breakpoint. In other words always halting simulation, when a specified signal changes value<br>• 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<br>• Removing a breakpoint from a signal<br>• Showing the line number or number in the source code of the statement or statements that causes the current value of a net<br>•A longer way to specify this capability is with the `callback` keyword. |
| `frc` or `force` | Commands perform the following tasks:<br>• Forcing a net or a register to a specified value so that this value cannot be changed by subsequent simulation events in the design<br>• Releasing a net or register from its forced value<br>A longer way to specify this capability is with the `force` keyword. |

### Example 1

The following specification enables many interactive commands including those for displaying the values of signals in specified modules and depositing values to the signals that are registers:

```
acc+=r:top,mid,bot
```

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+1 or +acc

Enables all capabilities except value change callbacks and delay annotation.

+acc+2

Enables capabilities of +acc and value change callbacks.

+acc+3

Enables capabilities of +acc and module path delay annotation.

+acc+4

Enables capabilities of +acc 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 PLI ACC write capabilities for all memories in the design, disabling them for the entire design, or enabling them for part or parts of the design hierarchy
- Four-state simulation for part of a design

The entries in the configuration file override the ACC write-enabling entries in the PLI table file.

The syntax of each type of statement in the configuration file to enable ACC write capabilities is as follows:

```
set writeOnMem;
```

OR

```
set noAccWrite;
```

OR

```
module {list_of_module_identifiers} {accWrite};
```

OR

```
instance {list_of_module_instance_hierarchical_names}
{accWrite};
```

OR

```
tree [(depth)] {list_of_module_identifiers} {accWrite};
```

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 about 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 23-1 Runtime Options to Enable PLI Learn*

| Option | Description |
|---|---|
| `+vcs+learn+pli` | Enables module level PLI learn |
| `+vcs+learn+pli+instpli` | Enables instance level PLI learn |
| `+vcs+learn+pli+sigpli` | Enables signal PLI learn at module level |
| `+vcs+learn+pli+sigpli`<br>`+vcs+learn+pli+instpli` | 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 23-1, Example 23-2, and Example 23-3.

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`, VCS ignores the debugging options.

## PLI Access to Ports of Celldefine and Library Modules

The `+nocelldefinepli` compile time option blocks debug access to celldefine and library modules. This option deletes (Programming Language Interface) PLI capabilities from the modules that are cell-defined or library modules.

However, you can access the ports inside such modules even in the presence of `+nocelldefinepli` optimization with an additional option `+ports.`

`+nocelldefinepli+1+ports`

Removes the PLI caps from the `` `celldefine `` modules and allows PLI access to port nodes and parameters.

`+nocelldefinepli+2+ports`

Removes the PLI caps from library and the `'celldefine` modules and allows PLI access to port nodes and parameters.

## Example

The following is a sample Verilog code in which the `dut` is a cell define module.

### test.sv

```
`celldefine
module ram (Addr, Data, CS, WE, OE);

parameter AddrSize = 4;
parameter WordSize = 1;

input [AddrSize-1:0] Addr;
inout [WordSize-1:0] Data;
input CS, WE, OE;

reg [WordSize-1:0] Mem [0:1<<AddrSize];

assign Data = (!CS && !OE)  ? Mem[Addr] : {WordSize{1'bz}};

always @(CS or WE)
  if (!CS && !WE)
    Mem[Addr] = Data;
```

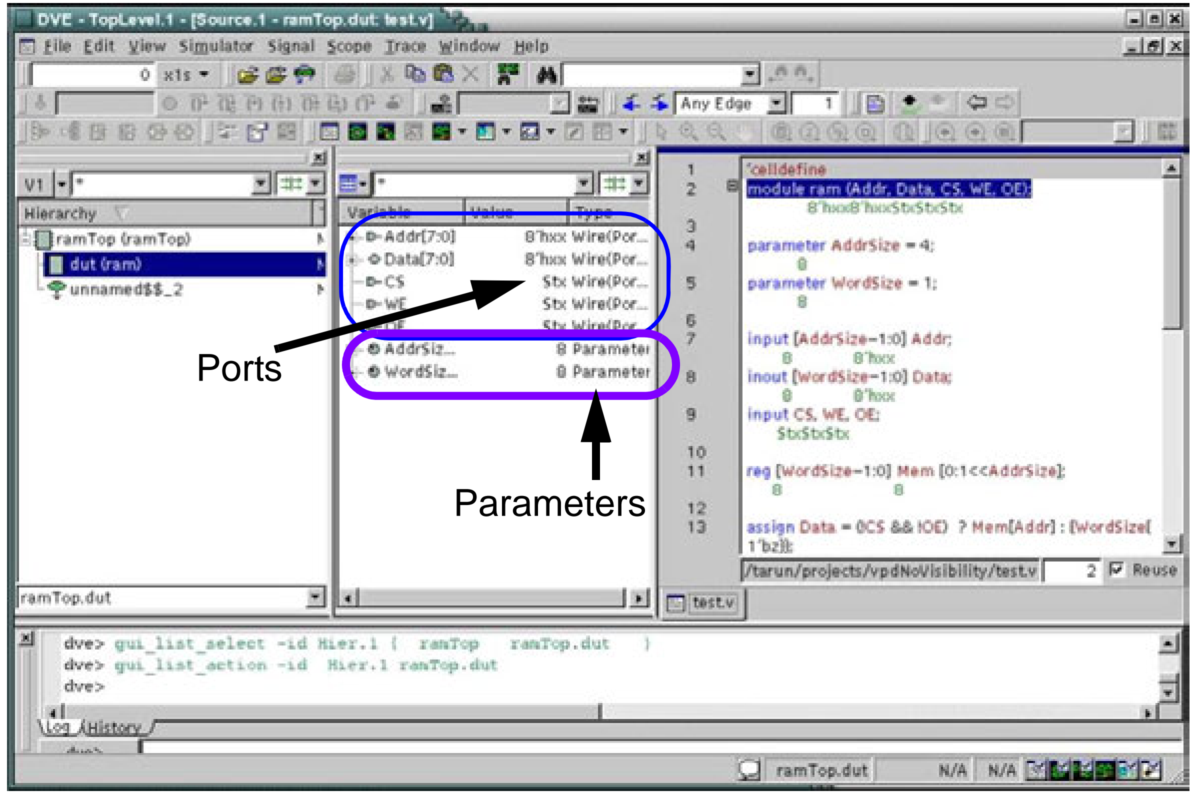
```
endmodule
`endcelldefine

module ramTop;
reg  [7:0] addr;
wire [7:0] data;
reg        cs, we, oe;
reg  [7:0] data_temp;

ram #(8,8) dut (addr, data, cs, we, oe);


assign data = (!cs && !we) ? data_temp : data;

initial begin
  $vcdpluson;
  $vcdplusmemon;
  repeat (10) begin
    #10;
    { cs, we, oe} = {$urandom%2, $urandom%2, $urandom%2};
   addr = {$urandom%2, $urandom%2, $urandom%2, $urandom%2,
$urandom%2, $urandom%2, $urandom%2, $urandom%2};
    data_temp = {$urandom%2, $urandom%2, $urandom%2,
$urandom%2, $urandom%2, $urandom%2, $urandom%2,
$urandom%2};
  end
end
endmodule
```

To compile this design code, use the following commands:

```
% vcs test.sv -debug_all -sverilog +nocelldefinepli+2+ports
% simv -gui &
```

### Visualization in DVE

In the following figure `Mem` which is an internal signal for the `ram` module is not shown in the **Data** pane. However other signals, which are ports or parameters, are displayed.




#### Limitations

Only Direct Kernel Interface (DKI) applications can access the ports, however, PLI applications cannot access the ports.

## Using VPI Routines

To enable VPI capabilities in VCS, use the `+vpi` compilation 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`.

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_compiletf
acc=mip,mp,gate,tchk,rw:test+
```

Note that this entry includes `acc=` even though the C functions in the PLI specification call VPI routines instead of PLI 1.0 routines. The syntax has not changed; you use the same syntax for enabling PLI 1.0 and PLI 2.0 routines.

This PLI table file is used for an example file named `set_mipd_delays_vpi.c`, which is available with *The Verilog PLI Handbook* by Stuart Sutherland, Kluwer Academic Publishers, Boston, Dordrect, 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;
   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;
   virtual ifc.tracker  modportIF2;

   C c;

   initial begin
`ifdef DUMP
     $vcdpluson;
`endif
      c = new(trkIF);
      c.test(modbusIF);
          modbusIF.ifci <= 10;
      #1
        $getVar;
        $display("end the first round\n");
      #1
        modbusIF.ifci <= 11;
         $getVar;
```

VI declared in Class scope

VI declared in Module scope

```
        $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_all
-sverilog

simv -gui &
```

To view how the virtual interface objects appear in DVE, see the *DVE User Guide*.

### Limitations

The following are the limitations with this functionality:

- Virtual Interface passed as a method port is not displayed in DVE.
- 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:

| | | |
|---|---|---|
| `cbEndOfSimulation` | `cbError` | `cbPliError` |
| `cbTchkViolation` | `cbSignal` | |

Also, the `cbValueChange` callback is not supported for the following objects:

- A memory or a memory word (index or element)
- `VarArray` or `VarSelect`

## Modified VPI Features

VCS complies with some of the constants that are standardized to comply with the IEEE LRM 1800-2009.

Table 23-2 provides the modified constants for the handle type returned by `vpi_get(vpiType, handle)`.

*Table 23-2 Modified Constants for the Handle Type Returned by `vpi_get(vpiType, handle)`*

| **In G-2012.09 and earlier releases** | **From H-2013.06 release** |
|---|---|

| vpiImmediateAssertType | vpiImmediateAssert |
|---|---|
| vpiImmediateAssumeType | vpiImmediateAssume |
| 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 23-3 provides the modified constants used in iterators.

*Table 23-3 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 when ref handle is vpiSequenceDecl |
| vpiIdentifier | vpiPropFormalDecl when ref handle is 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 23-4 provides the constants that are updated for the new value.

*Table 23-4    Constants That Are Updated for the New Value*

| Modified Constants |
|---|
| vpiPortType |
| vpiInterfacePort |
| vpiMember |
| vpiStructUnionMember |
| vpiAssertion |
| vpiClockingEvent |
| vpiDisableCondition |
| vpiIfOp |
| vpiIfElseOp |
| 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.

### Backwards Compatibility

When using the precompiled library, you can use the `-vpi_compliance=bc` runtime option for backwards compatibility. However, it is recommended to recompile the library with the latest VPI header files.

## Diagnostics for VPI PLI Applications

As defined in the LRM, VPI 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 option to make the PLI application to report errors without code modification:

- `-diag vpi`

For more information, see "Diagnostics for VPI 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 your design and C/C++ code using compile-time options for DirectC.

However, there are complications to this otherwise straightforward procedure.

DirectC allows the invocation of C++ functions that are declared in C++ using the `extern "C"` linkage directive. The `extern "C"` directive is necessary to protect the name of the C++ function from being mangled by the C++ compiler. Plain C functions do not undergo mangling, and therefore, do not need any special directive.

The declaration of these functions involves specifying a direction for the parameters of the C function, because, in the Verilog environment, they become analogous to Verilog tasks as well as functions. Verilog tasks are similar to void C functions in that they do not return a value. However, Verilog tasks do have input, output, and inout arguments, whereas C function parameters do not have explicitly declared directions. 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 23-5 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 23-6.

*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 23-5 C/C++ Function Return Types*

| Return Type | Specifies |
|---|---|
| `int` | The C/C++ function returns a value for type int. |
| `bit` | 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. |
| `reg` | The C/C++ function returns the value of a Verilog scalar reg. |
| `string` | The C/C++ function returns a pointer to a character string. |
| `pointer` | The C/C++ function returns a pointer. |
| `void` | The C/C++ function does not return a value. |

*Table 23-6 C/C++ Function Argument Directions*

| keyword | Specifies |
|---|---|
| `input` | The C/C++ function can only read the value or address of the argument. If you specify an input argument first, you can omit the `input` keyword. |

*Table 23-6 C/C++ Function Argument Directions*

| keyword | Specifies |
|---|---|
| `output` | The C/C++ function can only write the value or address of the argument. |
| `inout` | The C/C++ function can both read and write the value or address of the argument. |

*Table 23-7 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:

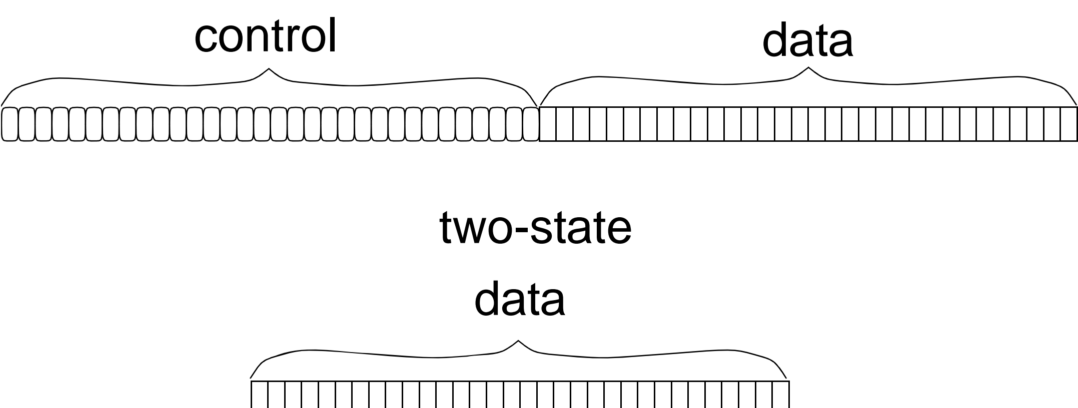
```
typedef unsigned int U;
typedef struct { U c; U d;} vec32;
```

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.

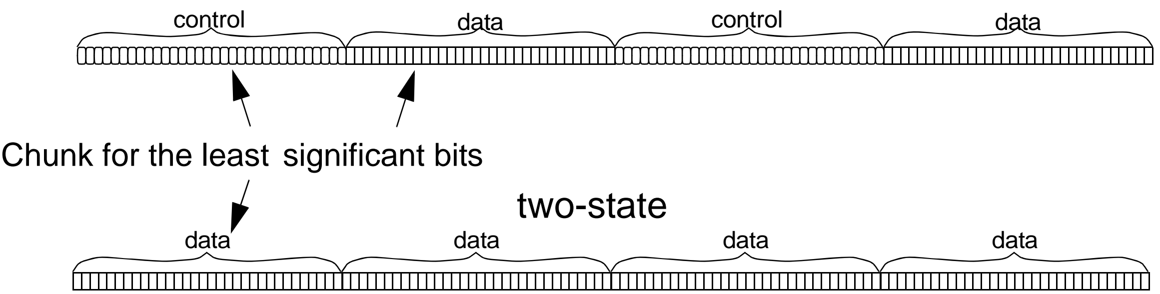
*Figure 23-1 Storing Vector Values*




Long vectors, more than 32 bits, have their value stored in more than one group of 32 bits and can be accessed by chunk. Short vectors, 32 bits or less, are stored in a single chunk.

For long vectors, the chunk for the least significant bits come first, followed by the chunks for the more significant bits.

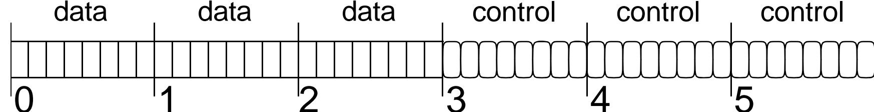
*Figure 23-2 Storing Vector Values of More than 32 Bits*




In an element in a Verilog memory, for each eight bits in the element, there is a data byte and a control byte with an additional set of bytes for remainder bit. 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 23-3 Storing Verilog Memory Elements in Machine Memory*




## Converting Strings

There are no *true* strings in Verilog, and a string literal, like "some_text," is 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_ConvertTo String(vec32 *, int, char *)
```

There are scenarios in which a string is created on the Verilog side and is passed to C code, and therefore, has to be converted from Verilog representation to C representation.

Consider the following example:

```
extern void WriteReport(string result_code, .... /* other
stuff */);
```

Example of 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 23-8 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 23-9 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 |

*Table 23-9    For Output and Inout Arguments*

| argument type | C/C++ formal parameter data type | Passed by |
|---|---|---|
| `reg[]` - any vector, including open vector | `vec32*` | reference |
| `array[]` - any array, 2 state or 4 state, including open array | `UB*` | reference |

In direct access, the return value of the function is always passed by value. The data type of the returned value is the same as the input argument.

### Example 1

Consider the following C/C++ function declared in the Verilog source code:

```
extern reg return_reg (input reg r1);
```

In this example, the function named `return_reg` returns the value of a scalar reg. The value of a scalar reg is passed to it. The header of the C/C++ function is as follows:

```
extern "C" scalar return_reg(scalar reti);
scalar return_reg(scalar reti);
```

If `return_reg()` is a C++ function, it must be protected from name mangling, as follows:

```
extern "C" scalar return_reg(scalar reti);
```

Note:

The `extern "C"` directive has been omitted in subsequent examples for brevity.

A scalar reg is passed by value to the function so the parameter is not a pointer. The parameter's type is scalar.

**Example 2**

Consider the following C/C++ function declared in the Verilog source code:

```
extern "C" bit [7:0] return_vector_bit (bit [7:0] r3);
```

In this example, the function named `return_vector_bit` returns the value of a vector bit. The `"C"` entry specifies direct access. Typically, a declaration includes this when some other functions use abstract access. The value of an 8-bit vector bit is passed to it. The header of the C/C++ function is as follows:

```
U return_vector_bit(U returner);
```

A vector bit is passed by value to the function because the vector bit is less than 33 bits so the parameter is not a pointer. The parameter's type is U.

**Example 3**

Consider the following C/C++ function declared in the Verilog source code:

```
extern void receive_pointer ( input pointer r6 );
```

In this example, the function named `receive_pointer` does not return a value. The argument passed to it is declared 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 your design for DirectC (by including the `+vc` compile-time option), VCS writes a file in the current directory named `vc_hdrs.h`. In this file, there are `extern` declarations for all the C/C++ functions that you declared in your Verilog code. For example, if you compile the Verilog code that contains all the C/C++ declarations in the examples in this section, the `vc_hdrs.h` file contains the following `extern` declarations:

```
extern void memory_rewriter(UB* mem2, /*OUT*/UB* mem1);
extern U return_vector_bit(U r3);
extern void receive_pointer(void* r6);
extern void incr(/*INOUT*/U* r7);
extern void* return_pointer();
extern scalar return_reg(scalar r1);
extern void reality(double* real1, /*OUT*/double* real2);
extern void receive_string(char* r5);
extern void passbig2(vec32* r8, /*OUT*/vec32* r9);
extern char* return_string();
extern void passbig1(U* r8, /*OUT*/U* r9);
```

These declarations contain the `/*OUT*/` comment in the parameter specification if its corresponding argument in your Verilog code is of type `output` in the declaration of the function.

These declarations contain the `/*INOUT*/` comment in the parameter specification if its corresponding argument in your Verilog code is of type `inout` in the declaration of the function.

You can copy from these `extern` declarations to the function headers in your C code. If you 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:

```
typdef 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("\ni1=%d i2=%d i3=%d i4=%d\n\n",i1,i2,i3,i4);
}
```

Parameters `h1`, `h2`, `h3`, and `h4` are `vc_handles` to regs `r1` and `r2` and memories `r3` and `r4`, respectively. The function 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("\ni1=%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("\ni1=%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:

  ```
  typdef 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 radixes 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" string FullPath(string filename);
// find full path to the file
// C string obtained from C domain

extern "A" void s2v(string, output reg[]);
// string-to-vector
// wrapper for vc_StringToVector().
```

```
`define FILE_NAME_SIZE 512

module Test;
 reg [`FILE_NAME_SIZE*8:1] file_name;
// this file_name will be passed to the Verilog code that
expects
// a Verilog-like string
.
.
.
 initial begin
s2v(FullPath("myStimulusFile"), file_name); // C-string to
Verilog-string
// bits of 'file_name' represent now 'Verilog string'
end
.
.
.
endmodule
```

The C code is as follows:

```
void s2v(vc_handle hs, vc_handle hv) {
          vc_StringToVector((char *)vc_getPointer(hs), hv);
         }
```

### void vc_VectorToString(vc_handle, char *)

Converts a vector value to a string value.

### int vc_getInteger(vc_handle)

Same as `vc_toInteger`.

### void vc_putInteger(vc_handle, int)

Passes an `int` value by reference through a `vc_handle` to a scalar reg or bit or a vector bit that is 32 bits or less. For example:

```
void putter (vc_handle h1, vc_handle h2, vc_handle h3,
vc_handle h4)
{
int a,b,c,d;
a=1;
b=2;
c=3;
d=9999999;

vc_putInteger(h1,a);
vc_putInteger(h2,b);
vc_putInteger(h3,c);
vc_putInteger(h4,d);
}
```

### vec32 *vc_4stVectorRef(vc_handle)

Returns a `vec32` pointer to a four-state vector. Returns NULL if the specified `vc_handle` is not to a four-state vector reg. For example:

```
typedef struct vector_descriptor {
        int width; /* number ofbits */
        int is4stte; /* TRUE/FALSE */
} VD;

void WriteVector(vc_handle file_handle, vc_handle a_vector)
{
    FILE *fp;
    int n, size;
    vec32 *v;
    VD    vd;
    fp = vc_getPointer(file_handle);

    /* write vector's size and type */
    vd.is4state = vc_is4stVector(a_vector);
    vd.width = vc_width(a_vector);
     size = (vd.width + 31) >> 5; /* number of 32-bit chunks */
      /* printf("writing: %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:

```
extern void big_2state( input bit [31:0] r1,
                        input bit [32:0] r2);

module test;
reg [31:0] r1;
reg [32:0] r2;
initial
begin
r1=4294967295;
r2=33'b100000000000000000000000000000010;
big_2state(r1,r2);
end
endmodule
```

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 chucks 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
     begin
          r1 [67:65] = 3’b111;
          r1 [64:33] = 32’bzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz;
          r1 [32:0] = 32’b00000000000000000000000000000000;
          copier(r1,r2);
          $display("r1=%0b\n",r1);
          $display("r2=%0b\n",r2);

     end
endmodule
```

In this example, there are two 68-bit regs. Values are assigned to all the bits of one reg and both of these regs are parameters to the C/C++ function named `copier`.

```
copier(vc_handle h1, vc_handle h2)
{
vec32 holder[3];
vc_get4stVector(h1,holder);
vc_put4stVector(h2,holder);
}
```

This function declares a `vec32` array of three elements named `holder`. It uses three elements because its parameters are 68-bit regs so you need an element for every 32 bits and one more for the remaining four bits.

The Verilog code displays the following:

```
r1=111zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz00000000000000000000000000000000

r2=111zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz00000000000000000000000000000000
```

### 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.

There 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;
            }
}
```

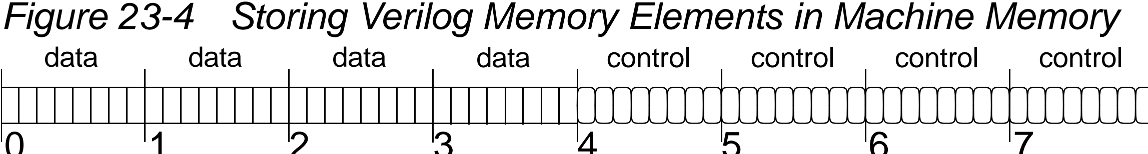
The purpose of the C/C++ function `mem_doer` is to copy the four elements in Verilog memory `memory1` into the 32 elements of `memory2`.

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 23-4 Storing Verilog Memory Elements in Machine Memory*




In an element in a Verilog memory, for each eight bits in the element there is a data byte and a control byte with an additional set of bytes for a remainder bit. 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_doer( inout reg [25:1] array [3:0]
memory1);

module top;
reg [25:1] memory1 [3:0];
initial
begin
memory1 [0] = 25'bz00000000xxxxxxxx11111111;
$display("memory1 [0] = %0b\n", memory1[0]);
mem_add_doer(memory1);
$display("\nmemory1 [3] = %0b", memory1[3]);
end
endmodule
```

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_doer(vc_handle h)
{
```

```
    U indx;
    UB *p1, *p2, t [8];

    indx = 0;
    p1 = vc_MemoryElemRef(h, indx);
    indx = 3;
    p2 = vc_MemoryElemRef(h, indx);
    memcpy(p2,p1,8);

    memcpy(t,p2,8);
    printf(" %d from t[0],  %d from t[1]\n",
            (int)t[0], (int) t[1]);
    printf(" %d from t[2],  %d from t[3]\n",
            (int)t[2], (int) t[3]);
    printf(" %d from t[4],  %d from t[5]\n",
            (int)t[4], (int)t[5]);
    printf(" %d from t[6],  %d from t[7]\n",
            (int)t[6], (int)t[7]);

}
```

C/C++ function `mem_elem_doer` uses the `vc_MemoryElemRef` routine to return pointers to addresses 0 and 3 in Verilog `memory1` and pass them to UB pointers `p1` and `p2`. The standard `memcpy` routine then copies the eight bytes for address 0 to address 3.

The remainder of the function is additional code to show you data and control bytes. The eight bytes pointed to by `p2` are copied to array `t` and then the elements of the array are printed.

The combined Verilog and C/C++ code displays the following:

```
memory1 [0] = z00000000xxxxxxxx11111111

 255 from t[0],  255 from t[1]
 0 from t[2],  0 from t[3]
 0 from t[4],  255 from t[5]
 0 from t[6],  1 from t[7]
```

```
memory1 [3] = z00000000xxxxxxxx11111111
```

As you can see, function `mem_elem_doer` passes the contents of the Verilog memory `memory1` element 0 to element 3.

In array `t`, the elements contain the following:

[0] The data bits for the eight 1 values assigned to the element.
[1] The data bits for the eight X values assigned to the element
[2] The data bits for the eight 0 values assigned to the element
[3] The data bit for the Z value assigned to the element
[4] The control bits for the eight 1 values assigned to the element
[5] The control bits for the eight X values assigned to the element
[6] The control bits for the eight 0 values assigned to the element
[7] The control bit for the Z value assigned to the element

### scalar vc_getMemoryScalar(vc_handle, U indx)

Returns the value of a one-bit memory element. For example:

```
extern void bitflipper (inout reg array [127:0] mem1);

module test;
reg mem1 [127:0];
initial
begin
mem1 [0] = 1;
$display("mem1[0]=%0d",mem1[0]);
bitflipper(mem1);
$display("mem1[0]=%0d",mem1[0]);
$finish;
end
endmodule
```

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 indx, char *)

This routine works similar to the `vc_putValue` routine except that is for passing values to a memory element instead of to a reg or bit. You enter an argument to specify the element (index) to which you want the routine to pass the value. For example:

```
#include <stdio.h>
#include "DirectC.h"

void check_vc_putvalue(vc_handle h)
{
  vc_putMemoryValue(h,0,"10xz");
}
```

### void vc_putMemoryValueF(vc_handle, U indx, char, char *)

This routine works 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 radixes 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_indx1,
vc_handle vh_indx2) {

  int sum = 0;
  int i1, i2, i3, i4, indx;

  i1 = vc_getInteger(vh_indx1);
  i2 = vc_getInteger(vh_indx2);
  /* loop over all possible indices for that slice */
  for (i3 = 0; i3 < vc_mdaSize(vh_array, 3); i3++) {

     for (i4 = 0; i4 < vc_mdaSize(vh_array, 4); i4++) {

         indx = vc_Index(vh_array, i1, i2, i3, i4);
         sum += vc_getMemoryInteger(vh_array, indx);
     }
  }
  return sum;
}
```

There are specialized, more efficient versions for two-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 23-10 summarizes all the access routines described in the previous section.

*Table 23-10 Summary of Access Routines*

| Access Routine | Description |
| --- | --- |
| `int vc_isScalar(vc_handle)` | Returns a 1 value if the `vc_handle` is for a one-bit reg or bit. It returns a 0 value for a vector reg or bit or any memory including memories with scalar elements. |
| `int vc_isVector(vc_handle)` | This routine returns a 1 value if the `vc_handle` is to a vector reg or bit. It returns a 0 value for a vector bit or reg or any memory. |
| `int vc_isMemory(vc_handle)` | This routine returns a 1 value if the `vc_handle` is to a memory. It returns a 0 value for a bit or reg that is not a memory. |
| `int vc_is4state(vc_handle)` | This routine returns a 1 value if the `vc_handle` is to a reg or memory that simulates with four states. It returns a 0 value for a bit or a memory that simulates with two states. |
| `int vc_is2state(vc_handle)` | This routine does the opposite of the `vc_is4state` routine. |
| `int vc_is4stVector(vc_handle)` | This routine returns a 1 value if the `vc_handle` is to a vector reg. It returns a 0 value if the vc_handle is to a scalar reg, scalar or vector bit, or to a memory. |
| `int vc_is2stVector(vc_handle)` | This routine returns a 1 value if the `vc_handle` is to a vector bit. It returns a 0 value if the vc_handle is to a scalar bit, scalar or vector reg, or to a memory. |
| `int vc_width(vc_handle)` | Returns the width of a `vc_handle`. |
| `int vc_arraySize(vc_handle)` | Returns the number of elements in a memory. |
| `scalar vc_getScalar(vc_handle)` | Returns the value of a scalar reg or bit. |
| `void vc_putScalar(vc_handle, scalar)` | Passes the value of a scalar reg or bit to a `vc_handle` by reference. |
| `char vc_toChar(vc_handle)` | Returns the 0, 1, x, or z character. |
| `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. |
| `char *vc_toString(vc_handle)` | Returns a string that contains the 1, 0, x, and z characters. |

| Access Routine | Description |
|---|---|
| `char *vc_toStringF(vc_handle, char)` | Returns a string that contains the 1, 0, x, and z characters and allows you to specify the format or radix for the display. The char parameter can be 'b', 'o', 'd', or 'x'. |
| `void vc_putReal(vc_handle, double)` | Passes by reference a real (double) value to a vc_handle. |
| `double vc_getReal(vc_handle)` | Returns a real (double) value from a `vc_handle`. |
| `void vc_putValue(vc_handle, char *)` | This function passes, by reference through the `vc_handle`, a value represented as a string containing the 0, 1, x, and z characters. |
| `void vc_putValueF(vc_handle, char, char *)` | This function passes by reference through the `vc_handle` a value for which you specify a radix with the third parameter. The valid radixes are 'b', 'o', 'd', and 'x'. |
| `void vc_putPointer(vc_handle, void*)`<br>`void *vc_getPointer(vc_handle)` | These functions pass, by reference to a `vc_handle`, a generic type of pointer or string. Do not use these functions for passing Verilog data (the values of Verilog signals). Use it for passing C/C++ data. `vc_putPointer` passes this data by reference to Verilog and `vc_getPointer` receives this data in a pass by reference from Verilog. You can also use these functions for passing Verilog strings. |
| `void vc_StringToVector(char *, vc_handle)` | Converts a C string (a pointer to a sequence of ASCII characters terminated with a null character) into a Verilog string (a vector with 8-bit groups representing characters). |
| `void vc_VectorToString(vc_handle, char *)` | Converts a vector value to a string value. |
| `int vc_getInteger(vc_handle)` | Same as `vc_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. |

| Access Routine | Description |
|---|---|
| vec32 *vc_4stVectorRef(vc_handle) | Returns a vec32 pointer to a four state vector. Returns NULL if the specified vc_handle is not to a four-state vector reg. |
| U *vc_2stVectorRef(vc_handle) | This routine returns a U pointer to a bit vector that is larger than 32 bits. If you specify a short bit vector (32 bits or fewer), this routine returns a NULL value. |
| void vc_get4stVector(vc_handle, vec32 *)<br>void vc_put4stVector(vc_handle, vec32 *) | Passes a four-state vector by reference to a vc_handle to and from an array in C/C++ function. vc_get4stVector receives the vector from Verilog and passes it to the array. vc_put4stVector passes the array to Verilog. |
| void vc_get2stVector(vc_handle, U *)<br>void vc_put2stVector(vc_handle, U *) | Passes a two state vector by reference to a vc_handle to and from an array in C/C++ function. vc_get2stVector receives the vector from Verilog and passes it to the array. vc_put4stVector passes the array to Verilog. |
| UB *vc_MemoryRef(vc_handle) | Returns a pointer of type UB that points to a memory in Verilog. |
| UB *vc_MemoryElemRef(vc_handle, U indx) | Returns a pointer to an element (word, address or index) of a Verilog memory. You specify the vc_handle of the memory and the element. |
| scalar vc_getMemoryScalar(vc_handle, U indx) | Returns the value of a one-bit memory element. |
| void vc_putMemoryScalar(vc_handle, U indx, scalar) | Passes a value, of type scalar, to a Verilog memory element. You specify the memory by vc_handle and the element by the indx parameter. |
| int vc_getMemoryInteger(vc_handle, U indx) | Returns the integer equivalent of the data bits in a memory element whose bit-width is 32 bits or less. |
| void vc_putMemoryInteger(vc_handle, U indx, int) | Passes an integer value to a memory element that is 32 bits or fewer. You specify the memory by vc_handle and the element by the indx parameter. |

| Access Routine | Description |
| --- | --- |
| void vc_get4stMemoryVector(vc_handle, U indx, vec32 *) | Copies the value in an Verilog memory element to an element in an array. This routine copies both the data and control bytes. It copies them into an array of type vec32. |
| void vc_put4stMemoryVector(vc_handle, U indx, vec32 *) | Copies Verilog data from a vec32 array to a Verilog memory element. |
| void vc_get2stMemoryVector(vc_handle, U indx, U *) | Copies the data bytes, but not the control bytes, from a Verilog memory element to an array in your C/C++ function. |
| void vc_put2stMemoryVector(vc_handle, U indx, U *) | Copies Verilog data from a U array to a Verilog memory element. |
| void vc_putMemoryValue(vc_handle, U indx, char *) | This routine works like the `vc_putValue` routine except that it is for passing values to a memory element instead of to a reg or bit. You enter an parameter to specify the element (index) you want the routine to pass the value to. |
| void vc_putMemoryValueF(vc_handle, U indx, char, char *) | This routine works like the `vc_putValueF` routine except that it is for passing values to a memory element instead of to a reg or bit. You enter an parameter to specify the element (index) you want the routine to pass the value to. |
| char *vc_MemoryString(vc_handle, U indx) | This routine works like the `vc_toString` routine except that it is for passing values to from memory element instead of to a reg or bit. You enter an parameter to specify the element (index) you want the routine to pass the value of. |
| char *vc_MemoryStringF(vc_handle, U indx, char) | This routine works like the `vc_MemoryString` function except that you specify a radix with the third parameter. The valid radixes are ’b’, ’o’, ’d’, and ’x’. |
| void vc_FillWithScalar(vc_handle, scalar) | This routine fills all the bits or a reg, bit, or memory with all 1, 0, x, or z values (you can choose only one of these four values). |

| Access Routine | Description |
| --- | --- |
| `char *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, ...)` | 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, 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 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` 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 wit 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
```

## Mixing Direct And Abstract Access

If you want some C/C++ functions to use direct access and others to use abstract access, you can do so by using a combination of "A" or "C" entries for abstract or direct access in the declaration of the function and the use of the +abstract suffix. The following table shows the result of these combinations:

| | **no +abstract suffix** | **include the +abstract suffix** |
|---|---|---|
| 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 will always look for this file in your current directory.
- Establish a link in the current directory to the $VCS_HOME/include/DirectC.h file.
- Include the -CC option as follows:

```
-CC "-I$VCS_HOME/include"
```

## Extended BNF for External Function Declarations

A partial EBNF specification for external function declaration is as follows:

```
source_text ::= description +
description ::= module | user_defined_primitive |
extern_function_declaration
extern_function_declaration ::= extern access_mode
extern_func_type extern_function_name (
list_of_extern_func_args ? ) ;
access_mode ::= ( "A" | "C" ) ?
```

Note:

If access mode is not specified, then the command line option `+abstract` rules; default mode is "C".]

```
extern_func_type ::= void | reg | bit |
DirectC_primitive_type | bit_vector_type
bit_vector_type ::= bit [ constant_expression :
constant_expression ]
list_of_extern_func_args ::= extern_func_arg
( , extern_func_arg ) *
extern_func_arg ::= arg_direction ? arg_type
optional_arg_name ?
```

Note:

Argument direction (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.

# 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
- SAIF System Tasks
- The Flows to Generate a Backward SAIF File
- 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

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

This section describes SAIF system tasks that you can use at the command line prompt.

`$set_toggle_region`

Specifies a module instance (or scope) for which VCS records switching activity in the generated SAIF file. Syntax:

```
$set_toggle_region(instance[, instance]);
```

`$toggle_start`

Instructs VCS to start monitoring switching activity.

Syntax:

```
$toggle_start();
```

`$toggle_stop`

Instructs VCS to stop monitoring switching activity.

Syntax

```
$toggle_stop();
```

`$toggle_reset`

Sets the toggle counter to 0 for all the nets in the current toggle region.

Syntax:

```
$toggle_reset();
```

`$toggle_report`

Reports switching activity to an output file.

Syntax:

```
$toggle_report("outputFile", synthesisTimeUnit,
                Scope);
```

This task has a slight change in native SAIF implementation compared to PLI-based implementation. VCS considers only the arguments specified here for processing. Other arguments have no meaning.

VCS does not report signals in modules defined under the `'celldefine` compiler directive.

`$read_lib_saif`

Allows you to read in a state dependent and path dependent (SDPD) library forward SAIF file. It registers the state and path dependent information on the scope. It also monitors the internal nets of the design.

Syntax:

```
$read_lib_saif("inputFile");
```

`$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(Scope);
  // initialization of Verilog signals
      ⋮
  $toggle_start;
  // testbench
      ⋮
  $toggle_stop;
  $toggle_report("outputFile", timeUnit,Scope);
end
```

The `$set_toggle_region(Scope)` 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 MX 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(Scope);
  // initialization of Verilog signals, and then:
  $toggle_start;
  // testbench
       ⋮
  $toggle_stop;
  $toggle_report("outputFile", timeUnit,Scope);
end
```

## 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

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.

## 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

You can use VCS to encrypt your HDL source files in such a way that they can be used only with VCS. This chapter describes how to use VCS to encrypt the source files for this purpose.

You can choose to encrypt only certain parts of your source files or entire files using either of the following methods:

- “IEEE Verilog Standard 1364-2005 Encryption”
- “128-bit Advanced Encryption Standard”
- “Skipping Encrypted Source Code”

## 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 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 *`protection_header_file`* that contains various protection pragmas.

VCS encrypts the following:

- Source files on the VCS command line
- 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, VCS encrypts only the regions specified by `'pragma protect 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 `-ipprotect` option to disable the 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"
- "Other Options for IEEE Std. 1364-2005 Encryption Mode"
- "How Protection Envelopes Work"
- "The VCS Public Encryption Key "
- "Creating Interoperable Digital Envelopes Using VCS - Example"
- "Discontinued -ipkey Option"

## The Protection Header File

The *protection_header_file* file may look like the following:

*Example 25-1 Sample IEEE Encryption Header File*

```
`pragma protect data_method = "aes128-cbc"
`pragma protect encoding = (enctype = "base64")
`pragma protect key_keyowner="Synopsys"
`pragma protect key_method="rsa"
`pragma protect key_keyname="SNPS-VCS-RSA-1"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDjJMv7PI1V+DJDaHZuVI
FbAXvr
6/tEpuM8cAKFuvpIoO6PE3DRqEwaHEJRyIsFnJnavVJ33+Kub54Cr/
9JCh6fnQht
AmKt/
nAznESOLExCKO1tmjYNCXLJ+QqWFoCuDuI4QS8Ruy1u3RwABCw7ESQwwIu
V
SZpOghOvjrPHzvlc0QIDAQAB
```

The following `` `pragma protect `` expressions are required inside the *protection_header_file* 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 the key name of keyowner.

`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 *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 expressions 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 these options as follows:

`-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 encrypts 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.

`-ipopt=outdir=`*`dir`*

Use this option to specify the target directory 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. 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 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 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. The option 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 the `` `pragma protect `` expression 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 `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.

---

## The VCS Public Encryption Key

The VCS base64 encoded RSA public key is:


```
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDjJMv7PI1V+DJDaHZuVI
FbAXvr
6/tEpuM8cAKFuvpIoO6PE3DRqEwaHEJRyIsFnJnavVJ33+Kub54Cr/
9JCh6fnQht
AmKt/
nAznESOLExCKO1tmjYNCXLJ+QqWFoCuDuI4QS8Ruy1u3RwABCw7ESQwwIu
V
SZpOghOvjrPHzvlc0QIDAQAB
```


The following `` `pragma protect `` expression identifies this key:

```
`pragma protect key_keyowner="Synopsys"
`pragma protect key_method="rsa"
`pragma protect key_keyname="SNPS-VCS-RSA-1"
```

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="Synopsys"
`pragma protect key_method="rsa"
`pragma protect key_keyname="SNPS-VCS-RSA-1"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDjJMv7PI1V+DJDaHZuVI
FbAXvr
6/tEpuM8cAKFuvpIoO6PE3DRqEwaHEJRyIsFnJnavVJ33+Kub54Cr/
9JCh6fnQht
AmKt/
nAznESOLExCKO1tmjYNCXLJ+QqWFoCuDuI4QS8Ruy1u3RwABCw7ESQwwIu
V
SZpOghOvjrPHzvlc0QIDAQAB
```

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 from its documentation. For VCS, it is this section on IEEE Verilog Std 1364-2005 Encryption. For other tools, an IP provider might need to contact its vendor.

The *protection_header_file* 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 = "IP Provider"
`pragma protect data_method = "aes128-cbc"
`pragma protect encoding = (enctype = "base64")

`pragma protect key_keyowner="Synopsys"
`pragma protect key_method="rsa"
`pragma protect key_keyname="SNPS-VCS-RSA-1"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDjJMv7PI1V+DJDaHZuVIFbAXvr
6/tEpuM8cAKFuvpIoO6PE3DRqEwaHEJRyIsFnJnavVJ33+Kub54Cr/9JCh6fnQht
AmKt/nAznESOLExCKO1tmjYNCXLJ+QqWFoCuDuI4QS8Ruy1u3RwABCw7ESQwwIuV
SZpOghOvjrPHzvlc0QIDAQAB

`pragma protect key_keyowner="VendorX"
`pragma protect key_method="rsa"
`pragma protect key_keyname="VENDORX-RSA-1"
`pragma protect key_public_key
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDjJMv7PI1V+DJDaHZuVIFbAXvr
6/tEpuM8cAKFuvpIoO6PE3DRqEwaHEJRyIsFnJnavVJ33+Kub54Cr/9JCh6fnQht
AmKt/nAznESOLExCKO1tmjYNCXLJ+QqWFoCuDuI4QS8Ruy1u3RwABCw7ESQwwIuV
SZpOghOvjrPHzvlc0QIDAQAB
```

(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 generates 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="IP Provider"
`pragma protect key_keyowner="Synopsys"
`pragma protect key_keyname="SNPS-VCS-RSA-1"
`pragma protect key_method="rsa"
`pragma protect encoding = (enctype = "base64",
line_length = 76, bytes = 128 )
`pragma protect key_block
fCY5ZM3A757rFLqRV/
Lk+hy8kPqXMnJ5rmr53Jnyv7u8nCxHUaZVqzmvWhp2pNwbJ+N
06jmd/
GF3KIexxUlD2nkF+tQEAtBAnHBvxweFAsBa43s1hRJW6TgXGD
FFktg5qa2b9clRWl92AggGSqmS+a1btkTZJ7PTjfanUrvtF3g
```

①

```
`pragma protect key_keyowner="VendorX"
`pragma protect key_keyname="VENDORX-RSA-N1"
`pragma protect key_method="rsa"
`pragma protect encoding = (enctype = "base64",
line_length = 76, bytes = 128 )
`pragma protect key_block
pNpqXq9REx09UGv+o62OuOYvoyf4mVIDIoaYfyZ6WDOEXZJq3rR
eZu+Jys7JYUhhUkHo638PP03pmnEasZjPXi9MqR/
tWCNeva5Ly0bEnkl2mrxqvOsvporedEyFx3swyQ48Kzq76rU7Qs
xlLz+mN3m97aaD/WusVe/Z0ozXtVo=
```

②

```
`pragma protect data_method="aes128-cbc"

`pragma protect encoding = (enctype = "base64",
line_length = 76, bytes = 176 )
`pragma protect data_block
+MW2QpXLShFRtT83KhWLYmbtcbKlE6jtCrr68RuPfNGys4r5cLDT
NGgytecJ1Br7WF6MXnS6NjRxpB7ZMEpN/
75UpcyVVUd3hOMVLVvQ+rrWtzVIPWa8td/
wvRA1qhQHVRC3QvW9UJWvOoAj6+6KPEi4TbZwMVFX5g/
J3XN4xASqClubQp+9sR2PJrpuWc3K
RN5dOZaq6Hmr0LVNbraNY4O8JwzNLOrR3gcQSul/86U=
```

(3)

(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 is used to encrypt 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.

## 128-bit Advanced Encryption Standard

VCS uses the 128-bit Advanced Encryption Standard (AES) to encrypt 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. To achieve this, perform the following steps:

**-protect128**

1. Enclose the Verilog code that you want to encrypt between `'protect128` and `'endprotect128` compiler directives.
2. Compile the files with the `-protect128` option. For example:

```
% vcs -protect128 testfile.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 `testfile.vp` when you execute the command listed above.

In the `.vp` files, VCS replaces `'protect128` and `'endprotect128` compiler directives with `'protected128` and `'endprotected128` compiler directives and encrypts the code in between these directives.

Note:

- If you specify `protect` and `protect128` compile options in the same `vcs` command, 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 in the same `vcs` command.

## 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 `testfile.vp` encrypted file in the `./out` directory:

```
% vcs -putprotect128 ./out -protect128 testfile.v
```

VCS creates a protected file in the specified directory. The '`./out/testfile.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 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 declaration 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$37A3F/8IWXLM[>/JJN8P\Q)Y=\FQ$J4M>
#.(31WZ'&(5&+%/L<RP0F+!$)E-U7!KA1Y!&5;S3>ID8RC):@*V>X
YZ1NC:S"/F]!NX0NKD"K8X5&4D_#)%PV(Y%PFO?4*96PED9&SI:PGMM
(J?GOD$%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$[*GB:L2U<9W,03Y&<B_1=DRWLJV;'OM'P];^[B5ZI
P"T)X^0Y.WRTK61I+D1_3=7\0D1C!%3+"(NR'K3$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"_^^0O8])%]O&3/7/P)O"?
[B@\,E<Y,N'"(&R 0530O;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 it 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 it does not encrypt port lists or UDP terminal lists.

```
module top;
parameter p1 = 3;
```

### The +protect Option

1. Enclose the Verilog and VHDL code that you want to encrypt between `'protect` and `'endprotect` compiler directives.

2. Compile the files with the `+protect` option.

   For example:

   ```
   % vcs +protect testfile.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 `testfile.vp` when you execute the command listed above.

   In the `.vp` file, VCS replaces `'protect` and `'endprotect` compiler directives with `'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\Lff9dBES3T<#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` compile option.

For example, the following command saves the `testfile.vp` encrypted file in the `./out` directory:

```
% vcs +putprotect ./out +protect 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)fXK
```

```
-@+EK?e=^Z\DLXU5XH0VQ19,>-9]]6+gDJ7Rf431EgL=7#>Y1V,9+3-
8&G>F[Q0C
4#B[FgQ#DUU<>_UR^I#D:eS(2+O15=^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]7XR0LRXFI;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 it 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 it 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` file using the `+autoincludeprotect` switch.

```
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 the `-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 VPD/FSDB for DVE/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 `` `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. Therefore, partial encryption is applied and the encrypted file is also enabled with debug capability (`-debug_all`) 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 `'unprotect` and the `'endunprotect` compiler directives. Enclose the VHDL code that you want decrypt between `-unprotect` and `-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.

# 26

# 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 itworks 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 26-1), use the `vcf` command with the `-verdi` option.

*Figure 26-1 Verdi GUI for VC Formal*

**Verification Targets: ALL**

| | status | depth | name | vacuity | witness | engine | type | elapsed_time |
|---|---|---|---|---|---|---|---|---|
| 1 | ✗ | 65 | fsm.a2 | ● | | b1 | assert | 00:00:01 |
| 2 | ✓ | | fsm.a_complete_frame | ● | | t1 | assert | 00:00:01 |
| 3 | ✗ | 81 | fsm.a_loop_break | ● | | b1 | assert | 00:00:02 |
| 4 | ✓ | | fsm.a_onehot | ● | | e2 | assert | 00:00:01 |
| 5 | ✓ | | fsm.c1 | ● | | e2 | cover | 00:00:01 |
| 6 | ✓ | 79 | fsm.c2 | | | b1 | cover | 00:00:02 |
| 7 | ✓ | 2 | fsm.c_blk_cnt | | | s1 | cover | 00:00:01 |
| 8 | ✓ | 1 | fsm.c_onehot | | | s1 | cover | 00:00:01 |

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
```

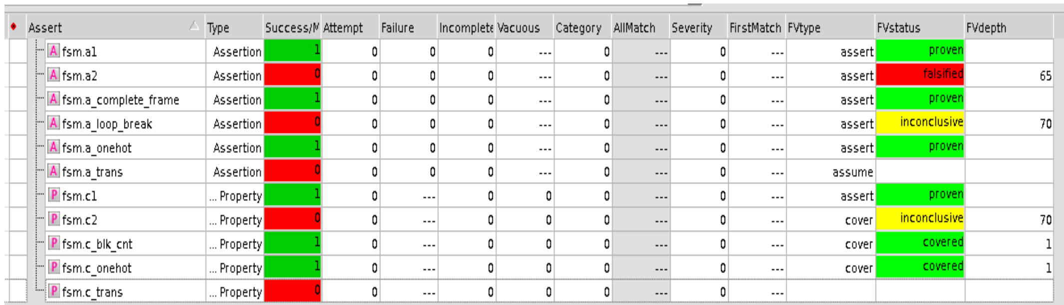
where,

`-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 26-2 Verdi Coverage Assert Pane*



| Assert | Type | Success/M | Attempt | Failure | Incomplete | Vacuous | Category | AllMatch | Severity | FirstMatch | FVtype | FVstatus | FVdepth |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| fsm.a1 | Assertion | 1 | 0 | 0 | 0 | --- | 0 | --- | 0 | --- | assert | proven | |
| fsm.a2 | Assertion | 0 | 0 | 0 | 0 | --- | 0 | --- | 0 | --- | assert | falsified | 65 |
| fsm.a_complete_frame | Assertion | 1 | 0 | 0 | 0 | --- | 0 | --- | 0 | --- | assert | proven | |
| fsm.a_loop_break | Assertion | 0 | 0 | 0 | 0 | --- | 0 | --- | 0 | --- | assert | inconclusive | 70 |
| fsm.a_onehot | Assertion | 1 | 0 | 0 | 0 | --- | 0 | --- | 0 | --- | assert | proven | |
| fsm.a_trans | Assertion | 0 | 0 | 0 | 0 | --- | 0 | --- | 0 | --- | assume | | |
| fsm.c1 | ... Property | 1 | 0 | --- | 0 | 0 | 0 | --- | 0 | --- | assert | proven | |
| fsm.c2 | ... Property | 0 | 0 | --- | 0 | 0 | 0 | --- | 0 | --- | cover | inconclusive | 70 |
| fsm.c_blk_cnt | ... Property | 1 | 0 | --- | 0 | 0 | 0 | --- | 0 | --- | cover | covered | 1 |
| fsm.c_onehot | ... Property | 1 | 0 | --- | 0 | 0 | 0 | --- | 0 | --- | cover | covered | 1 |
| 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.

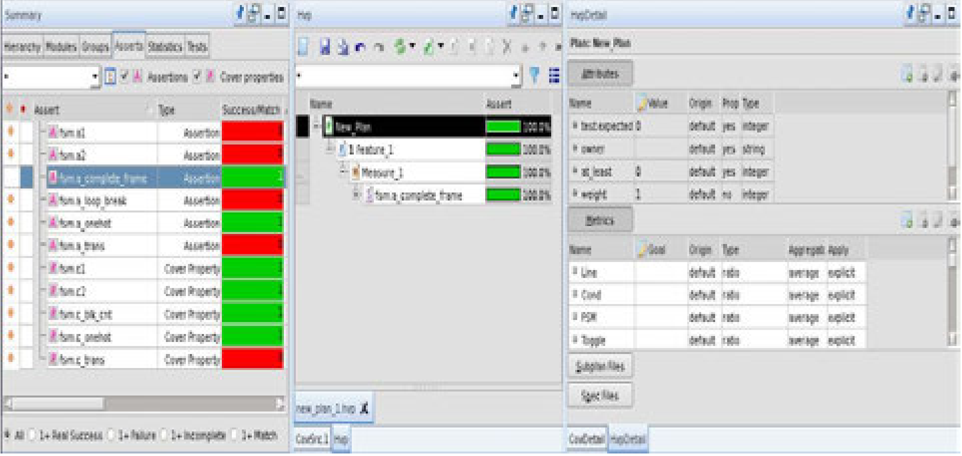
The `FVdepth` column is interpreted as follows:

- If `FVstatus` is Proven, Vacuous, or Uncoverable and its value is -1, it represents infinite depth.
- For other `FVstatus`, N>=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 26-3 shows VC Formal results in a verification plan with attributes and metrics.

*Figure 26-3 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.

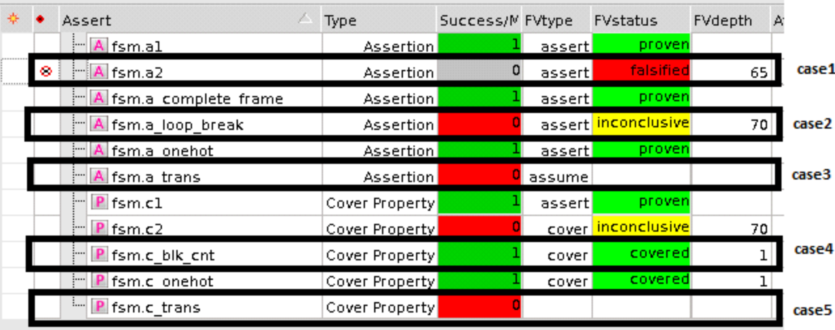
   After finding the matching region in the coverage database, the covered/coverable status is extracted from that database and added as a ratio for the `FVAssert` metric score. For the `FVAssert` metric, you can get `FVstatus` from FV annotations in the coverage database and compare `fvassert_expected_status`, 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 FVAssert metric score becomes 1/1. If expectation is fvassert_expected_status = inconclusive and fvassert_expected_depth = 60, the FVAssert 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/M | FVtype | FVstatus | FVdepth | |
|---|---|---|---|---|---|---|
| fsm.a1 | Assertion | 1 | assert | proven | | |
| fsm.a2 | Assertion | 0 | assert | falsified | 65 | case1 |
| fsm.a_complete_frame | Assertion | 1 | assert | proven | | |
| fsm.a_loop_break | Assertion | 0 | assert | inconclusive | 70 | case2 |
| fsm.a_onehot | Assertion | 1 | assert | proven | | |
| fsm.a_trans | Assertion | 0 | assume | | | case3 |
| fsm.c1 | Cover Property | 1 | assert | proven | | |
| fsm.c2 | Cover Property | 0 | cover | inconclusive | 70 | |
| fsm.c_blk_cnt | Cover Property | 1 | cover | covered | 1 | case4 |
| fsm.c_onehot | Cover Property | 1 | cover | covered | 1 | |
| 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 FVAssert 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".

# 27

# 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 27-1 shows the Certitude integration with VCS.

*Figure 27-1 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
   ```

   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 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_SYSCAN_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 MX LCA Features Guide*.

## Limitation

The feature has the following limitation:

- The DUT must be present entirely in a single partition.

# 28

# 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 option to generate a shell file based on your design topology:

**Table 0-1.**

| Option | Description |
| --- | --- |
| `-vlog` | Generates a Verilog shell file, *`filename`*`_vshell.v`. Use this option if your design is a Verilog-only design. |

- Compile your design and the *`filename`*`_vshell.v` file using the `-vera` option. This option is required to use Vera with VCS.
- Simulate the design by specifying the `.vro` file created in the first step using the `+vera_load` runtime option. You can also specify this `.vro` file in the `vera.ini` file in your working directory as shown in the following example:

```
vera_load = tb_top.vro
```

  See the *Vera User Guide* for more information.

## Usage Model

Use the following usage model to compile OpenVera code using Vera:

```
% vera -cmp [Vera_options] OpenVera_files
```

See the *Vera User Guide* for a list of Vera compilation options.

### Compilation

```
% vcs [compile_options] -vera verilog_filelist
filename_vshell.v
```

### Simulation

```
% simv [simv_options] +vera_load=file.vro
```

# 29

# Integrating VCS with Specman

The VCS ESI Adapter integrates VCS with the Specman Elite. This chapter describes how to prepare a stand-alone Verilog design for use with the ESI interface. See the *Specman Elite User Guide* for further information.

```
% 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”

- “Adding Specman Objects To DVE”
- “Version Checker for Specman”


## Type Support

The VCS ESI adapter supports the following Verilog Types:

- nets
- wires
- registers
- integers
- array of registers (verilog memory)

Other Verilog support:

- Verilog macros
- Verilog tasks
- Verilog functions
- Verilog events
- in/out/inout ports

# Usage Flow

This section explains how to integrate Specman with VCS.

## Setting Up The Environment

To set up the environment to run Specman with VCS:

- Set your `VCS_HOME` and `VRST_HOME` environment variables:

```
% setenv VCS_HOME [vcs_installation_path]
% set path = ($VCS_HOME/bin $path)
% setenv VRST_HOME [specman installation]
```

- Source your `env.csh` file for Specman:

```
% source ${VRST_HOME}/env.csh
```

  For 64-bit simulation, source your `env.csh` file as shown below:

```
% source ${VRST_HOME}/env.csh -64bit
```

## Specman e Code Accessing Verilog

Create the Verilog stub file `specman.v` and compile all Verilog files including `specman.v` as shown below:

```
% specman -c “load [top_e_file]; write stubs -verilog;”
```

Compile the design as given in the following table:

| Elaboration Mode | | Commands | Generated Executable |
|---|---|---|---|
| Compile | Execution with -o | "% sn_compile.sh -sim vcs \ -sim_flags "[compile-time_options] \ -debug 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 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 top_cfg/entity/module" -o <exe_name> " | <exe_name> |
| | Execution without -o | "% sn_compile.sh -sim vcs \ -sim_flags "[compile-time_options] \ -debug 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`.

## 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]
```

### Using specview

Set the environment variable `SPECMAN_OUTPUT_TO_TTY` as shown below:

```
% setenv SPECMAN_OUTPUT_TO_TTY 1
```

- In Compiled mode:

```
% specview -p "test -seed=1;" -sio simv -gui
```

- In Loaded mode:

```
% specview -p "load [top_e_file]; test -seed=1;" \
  -sio simv -gui
```

You can also specify VCS runtime options with `specview` or `specrun` as shown in the following examples:

*Example 29-1 To Invoke DVE Using specview*

The following command invokes the Specman GUI, as well as, DVE.

```
% specview -p "test -seed=1;" -sio simv -gui
```

Similarly, you can also use `-ucli` with `specview` to invoke simulation in UCLI mode.

*Example 29-2    To Invoke UCLI Using specrun*

The following command invokes the simulation in UCLI mode:

```
% specrun -p "test -seed=1;" simv -ucli -i include.cmd
```

Similarly, you can also use `-gui` with `specrun` to invoke DVE.

## Adding Specman Objects To DVE

Following are the steps involved to add e-objects to the DVE wave window:

- Compile the design. See “Usage Flow” .
- Create the `wave.ecom` file containing the list of e-objects to be added. For example:

```
wave exp sys.U_TbDut.My_Trans
wave event *.clk
```

- Simulate the design as shown below:

  - In Compiled mode:

```
% simv -gui -do run.do
```

Here, the `run.do` contains:

```
sn set wave -mode=manual dve
sn config wave -event_data=all_data
sn test
sn @wave
run 8 us
```

  - In Loaded mode:

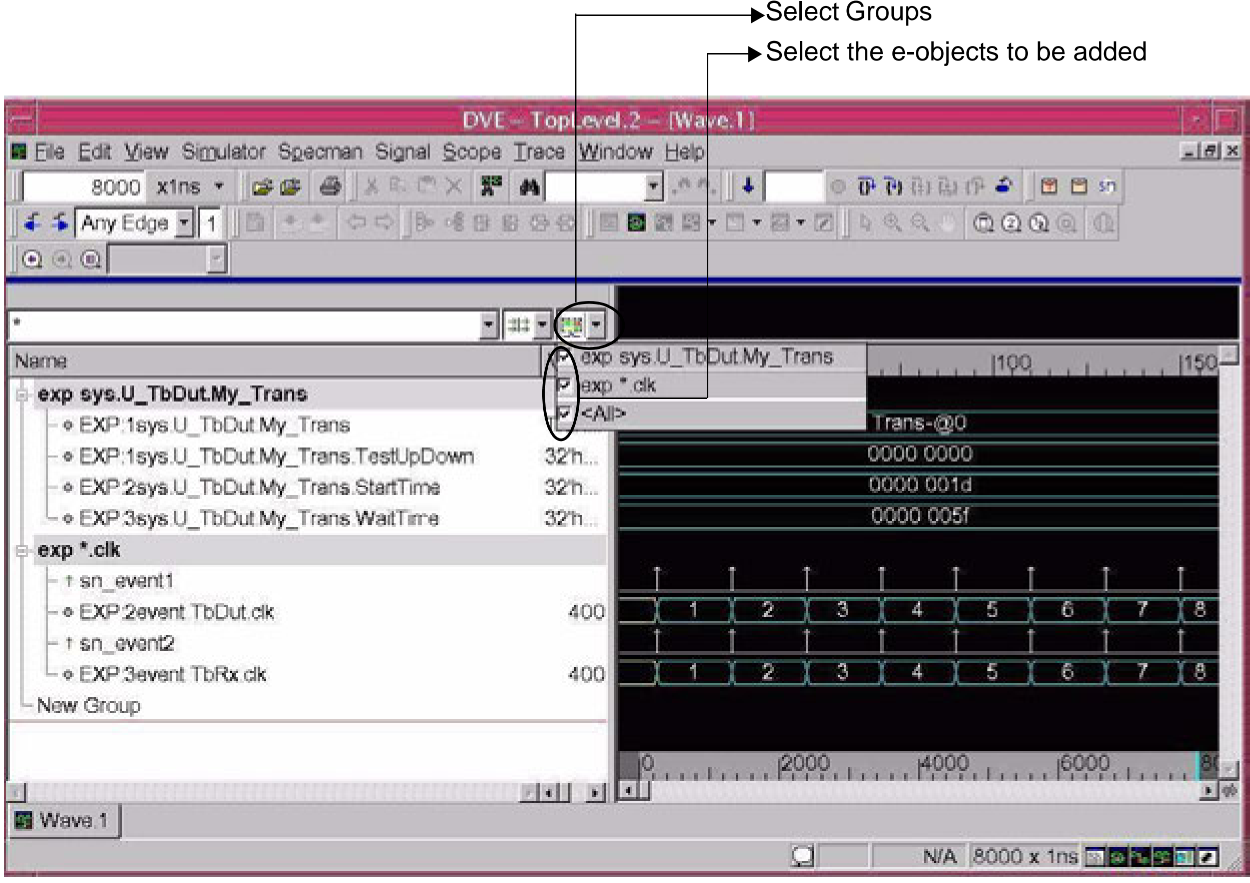
```
% simv -gui -do run.do
```

Here, the `run.do` contains:

```
sn set wave -mode=manual dve
sn config wave -event_data=all_data
sn load top_e_file.e
sn test
sn @wave
run 8 us
```

The `simv -gui -do run.do` command starts DVE, executes the UCLI commands specified in `run.do` and creates the `sn_wave_sys.tcl` session file.

- Now, load `sn_wave_sys.tcl` using **File > Load Session** and the dumped e-objects will be added to the Wave window automatically.
- Go to the Wave window and click on the groups icon to the side of the filter pane and select the e-objects to be added. See the figure shown below:



# 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.

## Use Model

### 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.

# 30

# 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
```

# 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:

```
parameter memory_spec = soma_file_path;
parameter init_file = "";
```

Note:

`memory_spec` and `init_file` are keywords.

# Use Model

This section describes the following topics:

- Use Model for Verilog Memory Models
- Execute Denali Commands at UCLI Prompt

## 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.

The use model is as follows:

### Compilation

```
% vcs -debug [vcs_options] verilog_filelist \
-P $DENALI/verilog/pli.tab $DENALI/verilog/denverlib.o
```

Note:

To compile the design in 64-bit mode, 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” .

# 31

# VCS and CustomSim Cosimulation

This chapter briefly describes the environment setup and usage model of the Synopsys CustomSim$^{TM}$ simulator and VCS mixed-signal simulations. Supported analog simulators also include HSIM and FineSim. You can use any one of the simulators to do mixed-signal simulations.

## Integrating VCS with CustomSim

VCS and CustomSim cosimulation allows a mixed-signal simulation solution, which enables simulating a design that is partly modeled in both analog and digital form. This section briefly describes the environment setup and usage model of VCS and CustomSim cosimulation mixed-signal simulations.

Note:

Unlike the VPI/PLI implementation of mixed-signal simulation previously used by CustomSim with VPI/PLI standard compliant digital simulators, CustomSim and VCS cosimulation uses a Direct Kernel Interface to exchange information between the CustomSim analog simulator and the VCS digital simulator. This approach provides more flexibility and better performance over VPI/PLI-based mixed-signal simulation.

It is recommended that before starting a mixed-signal simulation, both SPICE subcircuits and Verilog modules should be error-free (individually tested).

VCS and CustomSim cosimulation mixed-signal simulation supports:

- Verilog top-level netlists and SPICE top-level netlists.
- Donut partitioning, which is the arbitrary instantiation of SPICE subcircuits and Verilog modules under either SPICE or Verilog throughout the design hierarchy.
- Instance-based or cell-based partitioning.

For more information about CustomSim, see the Discovery AMS: *Mixed-Signal Simulation User Guide*. For information about CustomSim HSIM, see the CustomSim HSIM documentation. For more information about CutomSim FineSim, see the *FineSim User Guide: Pro and SPICE Reference*.

## Setting up the Environment

A working installation of VCS and a matching version of CustomSim are required to run mixed-signal simulation. The compatibility table for VCS and CustomSim versions that work together can be found at: https://solvnet.synopsys.com/retrieve/1463626.html.

## 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
```

Note:

If you set the SNPSLMD_LICENSE_FILE environment variable, then VCS ignores the LM_LICENSE_FILE environment variable.

## Required UNIX Paths and Variable Settings

To set the paths for CustomSim and VCS, do the following:

For CustomSim

```
% source XA_install_directory/CSHRC_platform
```

For CustomSim HSIM

```
% setenv HSIM_HOME HSIM_install_directory
% set path = ($VCS_HOME/bin $HSIM_HOME/bin $path)
% setenv HSIM_64 1
```

Unset the HSIM_64 variable 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

#### To Compile

The syntax to start the compile process is:

```
% vcs -ad=init_file verilog_source_files [other_vcs_options]
```

Where, `-ad=`*`init_file`* enables mixed-signal simulation. In the absence of the `init_file`, VCS looks for the default initialization file, `vcsAD.init`.

#### To Run Simulation

The syntax to run the mixed-signal simulation is:

```
% simv [runtime_options]
```

## Scheduling Analog-to-Digital Events in the NBA Region

VCS enables you to schedule a2d events in the non-blocking 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 digtal-to-digtal (d2d) node, then this feature will have no effect on the SPICE node.

# 32

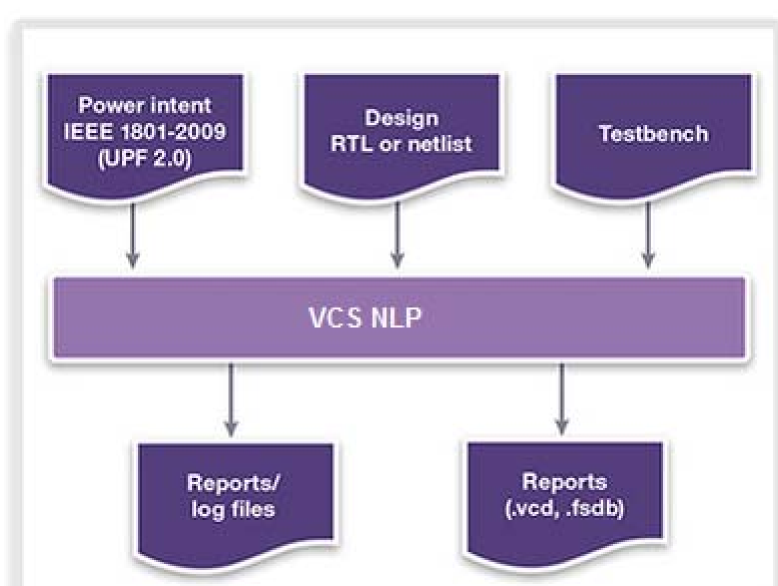
# Integrating VCS with Native Low Power (NLP)

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 32-1.

*Figure 32-1   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*.

# 33

# 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 `NOVAS_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/DVE with VCS

The following sections describe how to use the unified UVM library with Verdi or DVE transaction recorder and message catcher for the VCS simulator:

- Compilation
- Simulation

## Compilation

The compile flow is same for both DVE and Verdi transaction recording. It is not required to point `VCS_UVM_HOME` to UVM library in Verdi installation and recompile.

### Enabling FSDB or DVE Transaction Recording

Unified UVM library shipped with VCS has additional features that allow you to take advantage of VPD/FSDB transaction recording and DVE/Verdi transaction debugging capabilities.

For FSDB transaction recording, set `NOVAS_HOME` and `VCS_HOME` environment variables. With debug switches, such as `-debug_pp | -debug | -debug_all`, and NOVAS tab and PLI files, both Verdi transaction recorder and DVE transaction recorder are compiled.

You can link to the `novas.tab` and `pli.a` files of FSDB dumper using the following command:

```
% vcs -sverilog -ntb_opts uvm -debug_pp -P $NOVAS_HOME/share/
PLI/VCS/LINUX/novas.tab

$NOVAS_HOME/share/PLI/VCS/LINUX/pli.a
```

Note:

If the `NOVAS_HOME` variable is not set or `novas.tab` and `pli.a` files are not added, only the DVE transaction recorder is compiled.

### 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:

- The `-debug_access` option is a Limited Customer Availability (LCA) feature.
- You must use the `-ntb_opts uvm-1.2` option for UVM-1.2 code.
- You must set `NOVAS_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 sections 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
```

## Dumping Transactions or Messages in DVE Flow

To simulate, use `+UVM_TR_RECORD` to turn on transaction recording and use `+UVM_LOG_RECORD` to turn on recording of UVM report log messages:

```
% simv +UVM_TESTNAME=<your_uvm_test> +UVM_TR_RECORD \
+UVM_LOG_RECORD [simv_options]
```

You can then use DVE to debug the transactions and messages. This is supported for both interactive and post-process debug. The recorded streams with transactions and report logs are available in the VMM/UVM folder of the transaction browser.

Note:

If you have used the `UVM_TR_RECORD` feature with a previous version of VCS, then you must remove the `set_config_int("*", "recording_detail",UVM_FULL)` statement from your UVM code because it is no longer required.

# 34

# Integrating VCS with Verdi

This chapter describes VCS integration with Verdi in the following sections:


- Introduction
- Unified Compile Front End
- Dumping FSDB File for Various Flows
- Interactive and Post-Processing Debug
- Unified UCLI Dump Command

## Introduction

The increasing complexity and demand for quality and time to market of today's electronic devices has brought in a new set of challenges. Typically, majority of verification engineer's time is spent on debug. Simulation and debug are sequential activities, and once the simulation fails, debugging the failure is needed.

You use the desired debug tool to debug and fix the failure and may end up in executing redundant steps to setup the tool for debugging after the simulation is complete. For example, Verdi and VCS use different compilers to compile the HDL design and testbench that requires you to use different compiler scripts and compile the design twice to simulate and debug the design. In addition, the subset of supported HDL and the options for both the tools are different, so, you need to spend more time and effort to overcome these disparities.

The unified debug platform based on Verdi provides the ease of use, ease of migration, and helps in avoiding the time spent on redundant setups. The Unified Debug platform based on Verdi is tightly integrated to the VCS simulator and can be used for debugging.

The following topics describe the unified debug flow:

- Unified Compile Front End
- Dumping FSDB File for Various Flows
- Interactive and Post-Processing Debug
- Unified UCLI Dump Command
- UCLI FSDB Dump Commands

## Unified Compile Front End

Unified Compile front end is the integration of VCS and Verdi compilers to unify the compilation flows for simulation and debugging. Unified Compile front end uses VCS compiler scripts to compile the Knowledge Database (KDB) for Verdi. Consequently, only one common compiler script needs to be maintained for both VCS and Verdi, ensuring consistency between the two databases.

The benefits offered by Unified Compile front end are as follows:

- Single VCS and Verdi compilation
- Consistent HDL language support
- Consistency in utilizing or handling VCS and Verdi options

### Generating Verdi KDB with Unified Compile Front End

Unified Compile front end 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 the KDB. In case of VCS three-step flow, add the `-kdb` option in all the `vlogan/vcs` command lines.

When you specify the `–kdb` option, Unified Compile front end creates the Verdi 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>

% 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>
```

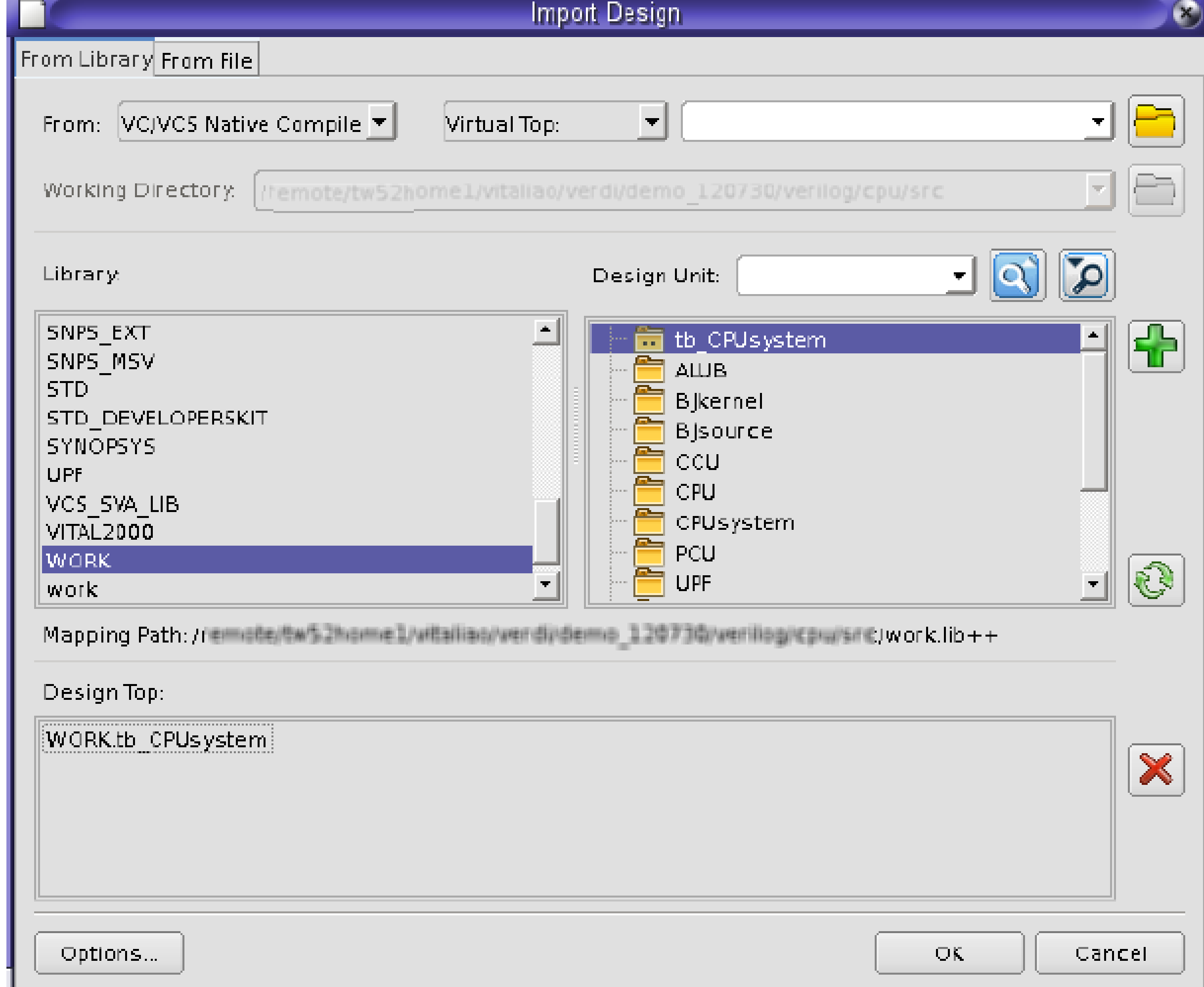
## Reading Compiled Design with Verdi

To read a compiled design, add the `-simflow` option to the Verdi command line. This imports the KDB compiled by the Unified Compile front end 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 34-1.

*Figure 34-1    Import Design Form*




You can also add the `-dbdir <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 `-dbdir` 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
```

Two-step compilation:

```
% vcs -sverilog -kdb top.v
```

Three-step compilation:

```
%vlogan top.v -kdb

%vcs top -debug_access+r -kdb
```

Import KDB:

```
% verdi -simflow -dbdir <path> -top top -nologo &
```

Invoke Verdi in interactive mode:

`% ./simv -verdi` or `./simv -gui=verdi`

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, Unified Compile front end flow is recommended to generate KDB 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, Unified Compile 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 unified compile flow.
- The Unified Compile front end 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 (`work.lib++`) is saved in the `work` directory in the current working directory.
- 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.setp` 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 Unified Compile front end:

- 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 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 VERDI_LIB $VERDI_HOME/share/PLI/VCS/<PLATFORM>
% setenv LD_LIBRARY_PATH $VERDI_LIB:$LD_LIBRARY_PATH
% 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 Verilog system tasks, or UCLI.

- Using Verilog System Tasks

  You can use the Verilog system tasks `$fsdbDumpfile()` and `$fsdbDumpvars()` in your Verilog design to dump an FSDB file (see “Using Verilog System Tasks” ).

- UCLI

  At UCLI prompt, you can use the UCLI commands `fsdbDumpfile` and `fsdbDumpvars` to dump an FSDB file.

Irrespective of whether you are using system tasks or UCLI commands, you must use one of the following options to enable FSDB dumping:

- `-debug_access`
- `-fsdb`
- `-P $VERDI_LIB/novas.tab $VERDI_LIB/pli.a`

## Using Verilog System Tasks

### Compilation

This can be done in following two ways:

- `% vcs -debug_access [elab_options] top_module/entity/cfg`
- `% vcs -fsdb [`*`compile_options`*`]` *`verilog_filelist`*

  Note:

  Usage of `-fsdb` is needed/recommended only when you use `-debug_pp`, `-debug`, or `-debug_all`.

```
% vcs -P $VERDI_LIB/novas.tab $VERDI_LIB/pli.a
[elab_options] top_module/entity/cfg
```

### 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 34-1 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 and simulate the above design is as follows:

### Compilation

```
% vcs -debug_access test.v
```

### Simulation

```
% simv
```

The above set of commands dumps all the instances in `test` into the `test.fsdb` file.

*Example 34-2 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 the design to use UCLI commands is as follows:

### Compilation

```
% vcs -debug_access test.v
```

### Simulation

```
% simv -ucli
ucli> fsdbDumpfile test.fsdb
ucli> fsdbDumpvars 0 test
ucli> run
ucli> quit
```

The above command dumps the whole design `test` into the `test.fsdb` file.

## Interactive and Post-Processing Debug

After the Verdi Knowledge Database (KDB) is generated using Unified Compile front end, the Unified Debug solution allows you to 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.

## Prerequisites

The following is the prerequisite to perform interactive simulation debug using Unified Debug solution:

- Generate Verdi KDB using Unified Compile front end. For more information, see "Unified Compile Front End" section.

The following are the prerequisites to perform post-process debug using Unified Debug solution:

- Generate Verdi KDB using Unified Compile front end. For more information, see "Unified Compile Front End" section.
- 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 `-verdi` or `-gui=verdi` option to invoke Verdi.

  For example,

  ```
  // invoke Verdi
  %> simv <simv_options> -verdi [-verdi_opts
  "<verdi_options>"]
  %> simv <simv_options> –gui=verdi [-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>"]
```

## 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 compiled by the Unified Compile front end, 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” .

## Limitations

The following is the limitation when performing power debug with UPF:

- The UPF file needs to be manually imported into Verdi both for Interactive and Post-Processing debug flows:
  - In interactive simulation debug flow, add the `-upf <UPF file>` option to import the UPF file.

    For example,

    ```
    %> vlogan -kdb <compile_options> <source files>

    %> vcs -kdb -upf <UPF file>

    %> simv -gui -upf <UPF file>
    ```

  - In post-processing debug flow, add the `-upf <UPF file>` option to import the UPF file.

    For example,

    ```
    %> vlogan -kdb <compile_options> <source files>
    ```

```
%> vcs -kdb -upf <UPF file>

%> simv

%> verdi -ssf novas.fsdb -simflow -simBin <simv_path/
simv> -upf <UPF file>
```

## Unified UCLI Dump Command

The UCLI `dump` command is enhanced to dump the Fast Signal Database (FSDB) file.

You can use the `dump` command to dump the Fast Signal Database (FSDB) file in addition to the VPD and EVCD file dumping. You can also perform the following operations using the `dump` command:

- Simultaneously open single VPD, EVCD, and FSDB dump files and manage them individually.
- Simultaneously open multiple FSDB dump files and manage them individually.

### Default Dump File

The default dump file for FSDB is `inter.fsdb`.

### Default Dump Type

The default dump type for VCS is VPD. You can use the following environment variable to change it to FSDB:

```
% setenv SNPS_SIM_DEFAULT_GUI verdi
```

## Use Model

### Use Model for FSDB Dumping

Following is the use model for FSDB dumping 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*.

`-add <list_of_nids>`

Specifies signals, scopes, or instances to be dumped. This command returns an integer value which increments after each call.

`-depth <levels>`

(Optional) Specifies the number of levels to be dumped. If the `-add` argument is specified, depth is calculated from the scope specified by the `-add` argument. If `-add` is not specified, depth is calculated from the current scope. The default value is 0, which means the entire design is down to the specified scope. Value 1 enables dumping only to the specified scope.

`-fid <fid>`

This argument specifies the file ID of the dump file to which the information must be dumped. The file ID, `<fid>`, is returned by the `dump -file` command.

`-aggregates`

This argument enables dumping complex data structures, such as VHDL records and arrays of records, and Verilog multi-dimensional arrays. You must use this argument along with the `-add` option.

### Example

```
ucli % dump -file test.fsdb -type FSDB
ucli % dump -add top.dut -aggregates
```

## Key Points to Note

- If a single dump file is open, you are not required to specify the `-fid` argument with the dump commands that follow the `dump -file` command.

  Example:

  `ucli% dump -file test.fsdb -type FSDB` (this command returns `FSDB0`)

  `ucli% dump –add / -depth 0` (this command dumps into the FSDB file `test.fsdb`)

  Note:

  `-type FSDB` can be omitted, if the default dump type is changed to FSDB, as described in “Default Dump Type” .

- If multiple dump files are open, you must specify the `-fid` argument with the dump commands that follow the second `dump -file` command.

  Example:

`ucli% dump -file test1.fsdb -type FSDB` (this command returns `FSDB0`)

`ucli% dump -file test2.fsdb -type FSDB` (this command returns `FSDB1`)

`ucli% dump –add/-depth 0 -fid FSDB0` (this command dumps into the FSDB file `test1.fsdb`)

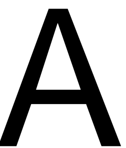
- During simulation, if the number of open dump files return to one, you can exclude the `-fid` argument. An error message is issued if the dump command is specified without the `-fid` argument when multiple dump files are open.

## UCLI FSDB Dump Commands

The following dump commands are supported only for the FSDB dump files:

- `dump -suppress_file`
- `dump -suppress_instance`
- `dump -enable`
- `dump -disable`
- `dump -glitch`
- `dump -opened`
- `dump -msv`

For more information on these commands, see the *Unified Command Line Interface User Guide*.



# VCS Environment Variables

This appendix covers the following topics:

- “Simulation Environment 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.

## Optional Environment Variables

VCS also includes the following environment variables that you can set in certain circumstances.

`DISPLAY_VCS_HOME`

Enables the display, at compile time, of the path to the directory specified in the `VCS_HOME` environment variable. Specify a value other than 0 to enable the display. For example:

```
setenv DISPLAY_VCS_HOME 1
```

`PERSISTENT_FLAG`

When set to 1, VCS disables the checks enabled by the `persistent` specification in the tab file. It also disables similar checks that are enabled by the `-debug`, `-debug_all`, or `-debug_pp` options. See the section “PLI Table File” .

`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_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.

## 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 `-xlrm 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 -xlrm env_expand
```

Content of `synopsys_sim.setup` file:

```
ELAB_EXPAND_ENV=TRUE
```

# B

# Compile-Time Options

The `vcs` command performs compilation of your design and creates a simulation executable. Compiled event code is generated and used by default. The generated simulation executable, `simv`, can then be used to run multiple simulations.

This section describes the `vcs` command and related options.

Syntax:

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

Here:

*`source_files`*

The Verilog or OVA source files for your design separated by spaces.

`source_or_object_files`

Optional C files (`.c`), object files (`.o`), or archived libraries (`.a`). These are DirectC or PLI applications that you want VCS to link into the binary executable file along with the object files from your Verilog source files. When including object files include the `-cc` and `-ld` options to specify the compiler and linker that generated them.

`options`

Compile-time options that control how VCS compiles your design.

This appendix lists the following:

- "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 Compile-time Options in a File"
- "Options for Compiling Runtime Options Into the Executable"
- "Options for PLI Applications"
- "Options to Enable the VCS DirectC Interface"
- "Options for Flushing Certain Output Text File Buffers"
- "Options for Simulating SWIFT VMC Models and SmartModels"
- "Options for Controlling Messages"
- "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"
- "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"
- "Option to Enable Bounds Check at Compile-Time"
- "Option to Enable Bounds Check at Runtime"
- "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:

`+liborder` 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.

## 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 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 construcs 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 time.

The following is the list of `keyword_argument` that are enabled when the `-assert enable_diag` compile-time 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` compile-time 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`

`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.

`funchier`

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.

Note:

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 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 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`. 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 5000 in the assertion expression will affect both compile-time and runtime performance. Therefore, VCS optimizes expression and issues a warning message as shown below:

```
Warning-[LDRF] Large delay or repetition found.
VCS will optimize compile time. However it may affect runtime.
Use '-assert disable_rep_opt' to disable this optimization.
"design.v", 156: (b_ce_idle [* 1:50000])
```

Use `-assert disable_rep_opt` to switch off the optimization and disable this message.

dumpoff

Disables the dumping of SVA information in the VPD file during simulation.

vpiSeqBeginTime

Enables you to see the simulation time that a SystemVerilog assertion sequence starts when using Debussy.

vpiSeqFail

Enables you to see the simulation time that a SystemVerilog assertion sequence doesn’t match when using Debussy.

+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_filext` *.ext*

Specifies an OpenVera file name extension. You can specify multiple file name extensions using the plus (+) character.

`-ntb_incdir` *directory_path*

Specifies the include directory path for OpenVera files. You can specify multiple include directories using the plus (+) character.

`-ntb_noshell`

Tells VCS not to generate the shell file. Use this option when you recompile a testbench.

`-ntb_opts` *keyword_argument*

The keyword arguments are as follows:

`ansi`

Preprocesses the OpenVera files in the ANSI mode. The default preprocessing mode is the Kernighan and Ritchie mode of the C language.

`check`

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'bxxxxxxxx;
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: \if_signal[3:0].

-ntb_shell_only

Generates only a .vshell file. Use this option when compiling a testbench separately from the design file.

-ntb_sfname *filename*

Specifies the file name of the testbench shell.

-ntb_sname *module_name*

Specifies the name and directory where VCS writes the testbench shell module.

-ntb_spath

Specifies the directory where VCS writes the testbench shell and shared object files. The default is the compilation directory.

-ntb_vipext *.ext*

Specifies an OpenVera encrypted-mode file extension to mark files for processing in OpenVera encrypted IP mode. Unlike the `-ntb_filext` option, the default encrypted-mode extensions `.vrp` and `.vrhp` are not overridden and will always be in effect. You can pass multiple file extensions at the same time using the plus (+) character.

## 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:

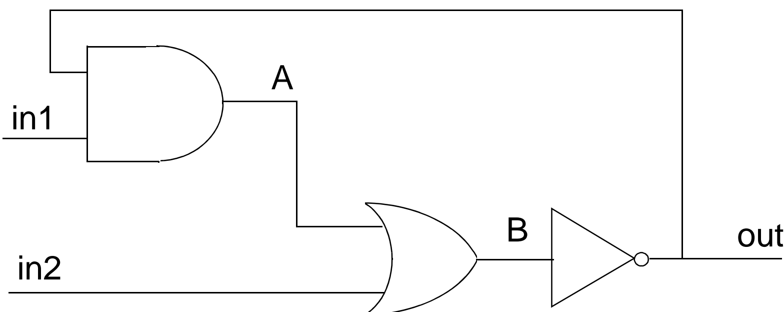
- reg
- bit
- integer
- int
- logic

For Example:

```
% vcs +vcs+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 B-1.

*Figure B-1 Combinational Logic With a Feedback Loop*




In Figure B-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+0` and `+vcs+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]
```

### 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 a 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 `x` value is to exclude an instance from the initial values specified by a different entry for a sub-hierarchy that includes 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

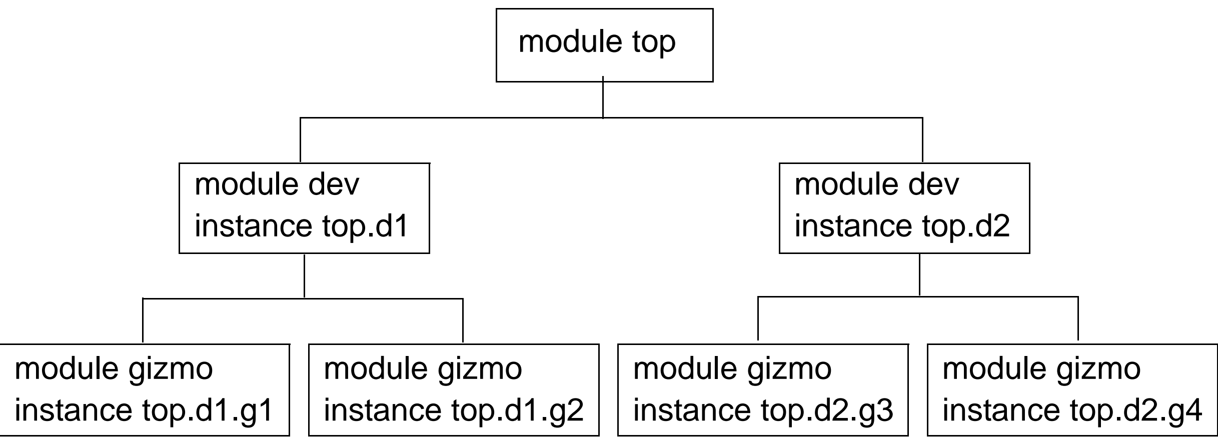
```
modtree module_name depth [x|z|0|1|random|
random seed_value]
```

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 B-2 is a hierarchical diagram of a small design.

*Figure B-2 Design Hierarchy for Initializing from a Configuration File*




The following are example entries in a configuration file for the small design shown in Figure B-2.

```
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 `+vcs+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.*

+allmtm

Specifies compiling separate files for minimum, typical, and maximum delays when there are min:typ:max delay triplets in SDF files. If you use this option, you can use the +mindelays, +typdelays, or +maxdelays options at runtime to specify which compiled SDF file VCS uses. Do not use this option with the +maxdelays, +mindelays, or +typdelays compile-time options.

+charge_decay

Enables charge decay in trireg nets. Charge decay 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

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 zero.

+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.

`+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.

`+typdelays`

Specifies using the typical timing delays in min:typ:max delay triplets when compiling the SDF file. The *`mtm_spec`* argument to the `$sdf_annotate` system task overrides this option.

`+multisource_int_delays`

Enables the multisource INTERCONNECT feature, including transport delays with full pulse control.

`+nbaopt`

Removes all intra-assignment delays in all the 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 `$setuphold` and `$recrem` timing checks.

`+NTC2`

In `$setuphold` and `$recrem` timing checks, specifies checking the timestamp and timecheck conditions when the original data and reference signals change value instead of when their delayed versions change value.

## 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 Compile-time 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 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:

  `-f`  `-y`  `-l`  `-u`  `-v`

The `-f` option is not supported in the UUM flow.

`-file` *`filename`*

Specify a file that contains a list of source files and VCS compile-time options, including C source files and object files.

You can use this option to overcome the limitation of the `-f` compile-time option.

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 `-full64` 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 `$`.

## Options for Compiling Runtime Options Into the Executable

`+plusarg_save`

Some runtime options must be preceded by the `+plusarg_save` option for VCS to compile them into the executable.

`+plusarg_ignore`

Tells VCS not to compile the following runtime options into the simv executable. This option is used to counter the `+plusarg_save` option on a previous line.

## Options for PLI Applications

`+acc+`*`level_number`*

Enables PLI ACC capabilities for the entire design. The level number can be any number between 1 and 4:

`+acc` or `+acc+1`

Enables all capabilities except breakpoints and delay annotation.

`+acc+2`

Above, plus breakpoints.

`+acc+3`

Above, plus module-path delay annotation.

`+acc+4`

Above, plus gate delay annotation.

`+applylearn+`*`filename`*

Recompiles your design to enable only the ACC capabilities that you needed for the debugging operations you did during a previous simulation of the design.

`-e` *`new_name_for_main`*

Specifies the name of your `main()` routine. You write your own `main()` routine when you are writing a C++ application or when your application does some processing before starting the simv executable.

Note:

Do not use the `-e` option with the VCS/SystemC Cosimulation Interface.

`-P` *`pli.tab`*

Compiles a user-defined PLI definition table file.

`+vpi`

Enables the use of VPI PLI access routines.

`+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 any debug option (`-debug_all`, `-debug_pp`, or `-debug`), 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 `$dumpports` system task or the simulation history file for LSI certification specified by the `$lsi_dumpports` system task.

These options can also be entered at runtime. Entering them at compile-time modifies the `simv` executable so that it runs as if these options were always entered at runtime.

## Options for Simulating SWIFT VMC Models and SmartModels

`-lmc-swift`

Includes the LMC SWIFT interface.

`-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".

`+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.

## 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

`+nolibcell`

Does not define as a cell modules defined in libraries unless they are under the `` `celldefine `` compiler directive.

`+nocelldefinepli+0`

Enables recording in VPD files, the transition times and values of nets and registers in all modules defined under the `'celldefine` compiler directive or defined in a library that you specify with the `-v` or `-y` options. This option also enables full PLI access to these modules.

`+nocelldefinepli+1`

Disables recording in VPD files, the transition times and values of nets and registers in all modules defined under the `'celldefine` compiler directive. This option also disables full PLI access to these modules. Modules in a library file or directory are not affected by this option unless they are defined under the `'celldefine` compiler directive.

`+nocelldefinepli+2`

In VPD files, disables recording the transition times and values of nets and registers in all modules defined under the `'celldefine` compiler directive or defined in a library that you specify with the `-v` or `-y` options, whether the modules in these libraries are defined under the `'celldefine` compiler directive or not. This option also disables PLI access to these modules.

Disabling recording of transition times and values of the nets and registers in library cells can significantly increase simulation performance.

Note:

Disabling recording transitions in library cells is intended for batch simulation only and not for interactive debugging with DVE. Any attempt in DVE to access a part of your design for which VPD has been disabled may have unexpected results.

`+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. If VCS is unable to obtain the license after two attempts, it exits.

`-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.

`-l`*name*

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_definitions*t

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 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 "`*.sv`h" for SystemVerilog. Then the following VCS switches can be used to identify those file extensions as the differentiation between SystemVerilog and Verilog-AMS contexts:

```
% vcs -ams -ad +verilogamsext+vams \
+systemverilogext+sv+v+svh ...
```

## 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` compile-time 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` compile-time 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.

## 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 values through a file.

You must specify the file name, which contains the list of all parameters that should be overridden, with the `-gfile` option.

The syntax for `-gfile` option is as follows:

```
vcs top_level_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.

*path*: Specifies the absolute hierarchical path to the parameter value which is to be overridden.

Note:

The `-gfile` supports only VHDL syntax for hierarchical path representation.

All escaped identifiers in the Verilog path must be converted into VHDL extended identifiers. If the escaped identifier contains '\' characters, they must be escaped with another '\' character.

For example, consider the following Verilog hierarchical path for the parameter 'P1'.

```
top.dut.\inst1_\cpu .inst2.P1
```

The corresponding `generics_file` entry is as follows:

```
assign  'hffffffff /top/dut/\inst1_\\cpu\/
inst2/P1
```

All 'for-generate' and 'instance-array' parentheses must be round parentheses, and the path delimiter must be '/'. All instance paths must start with '/'.

**Example:**

You can override the parameter and generic values using the `-gfile` option as follows:

```
vcs test.v -gfile overrides.txt
```

Where, `overrides.txt` contains the following entries:

```
assign   'hffffffff /top/dut/\inst1_\\cpu\/
inst2/P1

assign    "DUMMY" /top/dut/\inst1_\\cpu\/inst2/
P2

assign    10.34 /top/dut/\inst1_\\cpu\/inst2/P3
```

**Supported Data Types:**

The following data types are supported 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.

## 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-[AOOBAW] 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-[AOOBAW] 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-OBAE] 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'bxxxxxxxx;
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.
```

---

# 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.

## Allow Zero or Negative Multiconcat Multiplier

`-noerror ZONMCM`

Changes the following errors to a warning condition, thus allowing VCS to create the simv executable after displaying the warning message:

```
Error-[ZMMCM] Zero multiconcat multiplier cannot be used in this context
              A replication with a zero replication constant is considered to have
              a size of zero and is ignored. Such a replication shall appear
              only within a concatenation in which at least one of the
              operands of the concatenation has a positive size.
        target : {0 {1'bx}}

 Error-[NMCM] Negative multiconcat multiplier
        target : {(-1) {1'bx}}
        "my_test.v", 6
```

VCS errors out if you use “0” or a negative number as a multiconcat multiplier. You can change that error to a warning message using this option.

## Specifying a VCD File

`+vcs+dumpvars`

A substitute for entering the `$dumpvars` system task, without arguments, in your Verilog code.

## 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). This option also requires the -debug_pp option.

## Enabling Identifier Search

You can use the following elaboration options to enable and control the Search Identifiers feature:

- -genid_db
- -nogenid_db
- Any debug option (-debug, -debug_all, -debug_pp)

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 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_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_all.

## Memories and Multi-Dimensional Arrays (MDAs)

`+memcbk`

Enables callbacks for memories and multi-dimensional arrays (MDAs). Use this option if your design has memories or MDAs, and you are doing any of the following:

- Writing a VCD or VPD file during simulation. For VCD files, at runtime, you must also enter the `+vcs+dumparrays` runtime option. For VPD files, you must also enter the `$vcdplusmemon` system task. VCD and VPD files are used for post-processing with DVE.
- Using the VCS/SystemC Interface.
- Writing an FSDB file for Debussy.
- Using any debugging interface application - VCSD/PLI (acc/vpi) that needs to use value change callbacks on memories or MDAs. APIs like `acc_add_callback`, `vcsd_add_callback` and `vpi_register_cb` need this option if these APIs are used on memories or MDAs.

Note:

The `+memcbk` option is enabled by default when any one of the following debug options is used at compile-time:

`-debug`, `-debug_pp`, or `-debug_all`

## 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. You do not see this change in the DVE Source View, but you do see it in all the other DVE windows.

## Defining a Text Macro

`+define+`*`macro`*`=`*`value`*`+`

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

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)
```

## 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 will then issue 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 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.

## 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
```

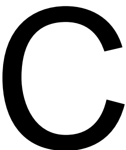
Dumps all the environment variables in the `vlogan_env_diag_<pid>.log` log file that is generated in the `AN.DB` directory.

```
% 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.



# 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 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 MX 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 is 2.

`+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 to continues to run after a constraint solver failure (default). |
| `1` | VCS exits on the first constraint solver error |

## 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:

`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.

`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.

`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+maxsucess=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_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 `StE` 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.

`+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.

`+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.

`+vpdfile+`*`file_name`*

Specifies the name of the output VPD file (default is `vcdplus.vpd`). You must include the full file name with the `.vpd` extension.

`+vpdfilesize+`*`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.

`+vpdfileswitchsize+`*`number_in_MB`*

Specifies a size for the vpd file. When the vpd file reaches this size, VCS closes this file and opens a new 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 +vpdfile+test.vpd +vpdfileswitchsize+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 `$vcdplus`*`xx`* system tasks and license checking. By default, VCS checks out a VPD PLI license if there is a `$vcdplus`*`xx`* 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

`+maxdelays`

Specifies using the maximum delays in min:typ:max delay triplets in module path delays and timing checks, if you compiled your design with the `+allmtm` compile time option. Also specifies using the maximum timing delays in min:typ:max delay triplets in an uncompiled SDF file.

If you compiled the SDF file with the `+allmtm` compile-time option, the `+maxdelays` option specifies using the compiled SDF file with the maximum delays.

Another use for this runtime option is to specify timing for SWIFT VMC and SmartModels when you also include the `+override_model_delays` runtime option.

`+mindelays`

Specifies using the minimum delays in min:typ:max delay triplets in module path delays and timing checks, if you compiled your design with the `+allmtm` compile-time option. Also specifies using the minimum timing delays in min:typ:max delay triplets in an uncompiled SDF file.

If you compiled the SDF file with the `+allmtm` compile time option, the `+mindelays` option specifies using the compiled SDF file with the minimum 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 Command line input*. All other VCS runtime options should be specified on the `simv` command line.

## 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 `seed_value` 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 +vcs+initreg+random option or the +vcs+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** |
|---|---|
| +vcs+initreg+config+*config_file* | Runtime configuration file overrides compile-time configuration file |
| +vcs+initreg+config+*config_file*<br>+vcs+initreg+*seed_value* | Uses specified seed for generating random initial values as defined in runtime configuration file |
| +vcs+initreg+config+*config_file*<br>+vcs+initreg+random | Issues a warning message, ignores +vcs+initreg+random |
| +vcs+initreg+config+*config_file*<br>+vcs+initreg+0 | Issues a warning message, ignores +vcs+initreg+0 |
| +vcs+initreg+config+*config_file*<br>+vcs+initreg+1 | Issues a warning message, ignores +vcs+initreg+1 |

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

`-E` *`program`*

Starts the *`program`* that displays the compile time options that were on the `vcs` command line when you created the `simv` (or `simv.exe`) executable file.

For example,`% simv -E echo`

You cannot use any other runtime option with the `-E` 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

`+vcs+ignorestop`

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

## Enabling User-defined Plusarg Options

`+`*`plus-options`*

User-defined runtime options to perform some operation when the option is on the `simv` command line. The `$test$plusargs` system task can check for such options.

## Enabling Overriding the Timing of a SWIFT SmartModel

`+override_model_delays`

Instead of using the DelayRange parameter definition in the template file, this option enables the `+mindelays`, `+typdelays`, and `+maxdelays` runtime options to specify the timing used by SWIFT SmartModels.

## 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” in Chapter “Compile-Time Options”.`+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” in Chapter “Compile-Time Options”

## 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 Rutime

`-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. 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 \
-I 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
```

# D

# Compiler Directives and System Tasks

This appendix describes:

- “Compiler Directives”
- “System Tasks and Functions”

## Compiler Directives

Compiler directives are commands in the source code that specify how VCS compiles the source code that follows them, both in the source files that contain these compiler directives and in the remaining source files that VCS subsequently compiles.

Compiler directives are not effective down the design hierarchy. A compiler directive written above a module definition affects how VCS compiles that module definition, but does not necessarily affect how

VCS compiles module definitions instantiated in that module definition. If VCS has already compiled these lower-level module definitions, it does not recompile them. If VCS has not yet compiled these module definitions, the compiler directive does affect how VCS compiles them.

Note:
Compile-time options override 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 | tri1 | 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 compiled 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
     :
   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 `+vcs+mipdexpand` compile-time option, or you can enter the `mipb` ACC capability. For example:

```
$sdf_annotate call=sdf_annotate_call
acc+=rw,mipb:top_level_mod+
```

When you compile the SDF file, which Synopsys recommends, you do not need to use this compiler directive to 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. VCShas 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_dumpports(top.middle1,"dumpports.dmp");
```

If you would rather have the `$lsi_dumpports` system task generate an extended VCD (EVCD) file instead, include the `+dumpports+ieee` runtime option.

`$dumpports`

Creates an EVCD file as specified in IEEE 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 `$lsi_dumpports` 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 `$dumpports` and `$lsi_dumpports` system tasks generated simulation history files for LSI certification and had identical syntax except for the name of the system task.

Syntax of the `$dumpports` system task is now:

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

You can specify more than one module instance.

Code example: `$dumpports(top.middle1,top.middle2,`
`"dumpports.evcd");`

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

`$dumpportsoff`

Suspends writing to files specified in `$lsi_dumpports` or `$dumpports` system tasks. You can specify a file to which VCS suspends writing or specify no particular file, in which case VCS suspends writing to all files specified by `$lsi_dumpports` or `$dumpports` system tasks. 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 `$dumpportsall` system task records the values of the ports in the module instances, which are specified by the `$lsi_dumpports` or `$dumpports` system task, irrespective of whether there is a value change on these ports or not. You can specify the file to which you want VCS to record the port values for the corresponding module instance or specify no particular file, in which case VCS writes port values in all files opened by the `$lsi_dumpports` or `$dumpports` system task. For more details, see IEEE Verilog LRM Std 1364-2005, Page 340.

Syntax: `$dumpportsall("`*`filename`*`");`

`$dumpportsflush`

VCS stores simulation data in a buffer during simulation from which it writes data to the file. If you want VCS to write all simulation data from the buffer to the file or files at a particular time, execute this `$dumpportsflush` system task. You can specify the file to which you want VCS to write from the buffer or specify no particular file, in which case VCS writes all data from the buffer to all files opened by the `$lsi_dumpports` or `$dumpports` system task. For more details, see IEEE Verilog LRM Std 1364-2005, Page 341.

Syntax: `$dumpportsfush("`*`filename`*`");`

`$dumpportslimit`

Specifies the maximum file size of the file specified by the `$lsi_dumpports` or `$dumpports` 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 `$lsi_dumpports` or `$dumpports` system task. For more details, see IEEE Verilog LRM Std 1364-2005, Page 340.

Syntax: `$dumpportslimit(filesize,"filename");`

## System Tasks for VPD Files

VPD files are files that store the transition times and values for nets and registers but they differ from VCD files in the following ways:

- You can use the DVE to view the simulation results that VCS recorded in a VPD file. You cannot actually load a VCD file directly into DVE. When you load a VCD file, DVE translates the file to VPD and loads the VPD file.
- They are binary format and therefore take less disk space and load much faster.
- They can also record the order of statement execution so that you can use the source window in DVE to step through the execution of your code if you specify recording this information.

VPD files are commonly used in post-processing, where VCS writes the VPD file during batch simulation. After that you can review the simulation results using DVE.

There are system tasks that specify the information that VCS writes in the VPD file.

Note:

To use the system tasks for VPD files, you must compile your source code with the `-debug_pp` option.

`$vcdplusautoflushoff`

Turns off the automatic flushing of simulation results to the VPD file whenever there is an interrupt, such as when VCS executes the `$stop` system task.

Syntax: `$vcdplusautoflushoff;`

`$vcdplusautoflushon`

Tells VCS to flush or write all the simulation results in memory to the VPD file whenever there is an interrupt, such as when VCS executes a `$stop` system task or when you halt VCS using the UCLI `stop` command, or using the **Stop** button on the DVE Interactive window.

Syntax: `$vcdplusautoflushon;`

`$vcdplusclose`

Tells VCS to mark the current VPD file as completed and close the file.

Syntax: `$vcdplusclose;`

`$vcdplusdeltacycleon`

The `$vcdplusdeltacycleon` task enables reporting of delta cycle information from the Verilog source code. It must be followed by the appropriate `$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`, DVE allows you to explicitly control the glitch detection.

Syntax: `$vcdplusdeltacycleon;`

Note:

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

### $vcdplusevent

The `$vcdplusevent` task allows you to record a unique event for a signal at the current simulation time unit.

Syntax

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

A symbol is displayed in DVE 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.

`$vcdplusfile`

Specifies the next VPD file that DVE opens during simulation after it executes the `$vcdplusclose` system task and when it executes the next `$vcdpluson` system task.

Syntax: `$vcdplusfile("`*`filename`*`");`

`$vcdplusglitchon`

Turns on the checking for zero delay glitches and other cases of multiple transitions for a signal at the same simulation time.

Syntax: `$vcdplusglitchon;`

`$vcdplusflush`

Tells VCS to flush or write all the simulation results in memory to the VPD file at the time VCS executes this system task. Use `$vcdplusautoflushon` to enable automatic flushing of simulation results to the file when the simulation stops.

Syntax: `$vcdplusflush;`

`$vcdplusmemon`

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.

`dim1Rsb`

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

Note:

The `dim1Lsb` and `dim1Rsb` arguments specify the range of the first dimension to be recorded. If no other arguments are 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 `$vcdplusmemon` 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_annotate`

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.  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. VCSexecuting function `f2` causes VCSto execute the function `f3`.

`#2` is a call of function `f2` in user-defined function `f1` at line number 10. VCSexecuting function `f1` causes VCSto execute the function `f2`.

`#3` is a call of function `f1` in the user-defined task `t` at line number 18. VCSexecuting task `t` causes VCSto execute the function `f1`.

`#4` is a task enabling statement for task `t` in the user-defined task `t1` at line number 22. VCSexecuting task `t1` causes VCSto execute the task `t`.

`#5` is a task enabling statement for task `t1` in the `begin-end` block named `B0`. VCSexecuting block `B0` causes VCSto execute the task `t1`.

`#6` is the instance of program test. VCSdoes 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 DVE or UCLI.

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
```

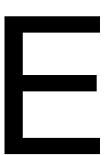
At runtime, VCS displays the following hierarchical information:

```
psstack = test.psstack.t2.f
```

## 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`



# 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: | `acc_setmem_int (memhand, value, row, start, length)` | | |
| Returns: | Type | Description | |
| | void | | |
| Arguments: | Type | Name | Description |
| | handle | memhand | Handle to memory |
| | int | value | The integer value written in binary format to the bits in the word. |
| | int | row | The memory array index. |
| | int | start | Bit number of the leftmost bit in the memory word where this routine starts writing the value. |
| | int | length | Starting with the start bit, specifies the total number of bits this routine writes to. |
| Related routines: | 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: | `acc_getmem_int (memhand, row, start, length)` | | |
| Returns: | Type | Description | |
| | int | Integer value of the bits in the memory word. | |
| Arguments: | Type | Name | Description |
| | handle | memhand | Handle to memory |
| | int | row | The memory array index |
| | int | start | Bit number of the leftmost bit in the memory word where this routine starts reading the value. |
| | int | length | Specifies the total number of bits this routine reads starting with the start bit. |
| Related routines: | acc_setmem_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_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: | `acc_clearmem_int (memhand)` | | |
| Returns: | Type | Description | |
| | void | | |
| Arguments: | Type | Name | Description |
| | handle | memhand | Handle to memory |
| 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 how to use `acc_getmem_int`, `acc_setmem_int`, and `acc_clearmem_int`:

- Example E-1 shows C code that includes a number of functions to be associated with user-defined system tasks.

- Example E-2 shows the PLI table for associating these functions with these system tasks.
- Example E-3 shows the Verilog source code containing these system tasks.

*Example E-1 C Source Code for Functions Calling acc_getmem_int, acc_setmem_int, and acc_clearmem_int*

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

void error_handle(char *msg)
{
  printf("%s",msg);
  fflush(stdout);
  exit(1);
}

void set_mem()
{
   handle memhand = NULL;
   int value = -1;
   int row = -1;
   int start_bit = -1;
   int len = -1;

   memhand = acc_handle_tfarg(1);
   if(!memhand) error_handle("NULL MEM HANDLE\n");
   value = acc_fetch_tfarg_int(2);
   row   =  acc_fetch_tfarg_int(3);
   start_bit = acc_fetch_tfarg_int(4);
   len =  acc_fetch_tfarg_int(5);

   acc_setmem_int(memhand, value, row, start_bit, len);
}

void get_mem()
{
   handle memhand = NULL;
```

```
    int row = -1;
    int start_bit = -1;
    int len = -1;
    int value = -1;

    memhand = acc_handle_tfarg(1);
    if(!memhand) error_handle("NULL MEM HANDLE\n");
    row  =  acc_fetch_tfarg_int(2);
    start_bit = acc_fetch_tfarg_int(3);
    len =  acc_fetch_tfarg_int(4);
    value = acc_getmem_int(memhand, row, start_bit, len);
    printf("getmem: value of word %d is : %d\n",row,value);
    fflush(stdout);
}

void clear_mem()
{
  handle memhand = NULL;

  memhand = acc_handle_tfarg(1);
  if(!memhand) error_handle("NULL MEM HANDLE\n");

  acc_clearmem_int(memhand);
}
```

The function with the `set_mem` identifier calls the IEEE standard `acc_fetch_tfarg_int` routine to get the handles for arguments to the user-defined system task that you associate with this function in the PLI table file. It then assigns the handles to local variables and calls `acc_setmem_int` to write to the specified memory in the specified word, start bit, for the specified length.

Similarly, the function with the `get_mem` identifier calls the `acc_fetch_tfarg_int` routine to get the handles for arguments to a user-defined system task and assign them to local variables. It then calls `acc_gtetmem_int` to read from the specified memory in

the specified word, starting with the specified start bit for the specified length. It then displays the word index of the memory and its value.

The function with the `clear_mem` identifier likewise calls the `acc_fetch_tfarg_int` routine to get a handle and then calls `acc_clear_mem_int` with that handle.

*Example E-2 PLI Table File*

```
$set_mem  call=set_mem acc+=rw:*
$get_mem  call=get_mem acc+=r:*
$clear_mem call=clear_mem acc+=rw:*
```

Here the `$set_mem` user-defined system task is associated with the `set_mem` function in the C code, as are the `$get_mem` and `$clear_mem` with their corresponding `get_mem` and `clear_mem` function identifiers.

*Example E-3 Verilog Source Code Using These System Tasks*

```
module top;
// read and print out data of memory
parameter start = 0;
parameter finish =9 ;
parameter bstart =1 ;
parameter bfinish =8 ;
parameter size = finish - start + 1;
reg [bfinish:bstart] mymem[start:finish];
integer i;
integer len;
integer value;

initial
 begin
 //  $set_mem(mem_name, value, row, start_bit, len)
   $clear_mem(mymem);

 // set values
```

```
 #1    $set_mem(mymem,  8,  2, 1, 5);
 #1    $set_mem(mymem, 32,  3, 1, 6);
 #1    $set_mem(mymem,  144,  4, 1, 8);
 #1    $set_mem(mymem,29,5,1,8);

 //  print values through acc_getmem_int
 #1 len = bfinish - bstart + 1;
 $display();
 $display("Begin Memory Values");
 for (i=start;i<=finish;i=i+1)
    begin
      $get_mem(mymem,i,bstart,len);
    end
   $display("End Memory Values");
   $display();

 // display values
 #1 $display();
 $display("Begin Memory Display");
 for (i=start;i<=finish;i=i+1)
    begin
      $display("mymem word %d is %b",i,mymem[i]);
    end
 $display("End Memory Display");
 $display();
end
endmodule
```

In this Verilog code, in the initial block, the following events occur:

1. The `$clear_mem` system task clears the memory.
2. Then the `$set_mem` system task deposits values in specified words, and in specified bits in the memory named `mymem`.
3. In a `for` loop, the `$get_mem` system task reads values from the memory and displays those values.

## acc_setmem_hexstr

You use the `acc_setmem_hexstr` access routine for writing the corresponding binary representation of a hexadecimal string to a Verilog memory.

**Table 0-1.**

| acc_setmem_hexstr | | | |
|---|---|---|---|
| Synopsis: | Writes a hexadecimal string to a word in a Verilog memory. | | |
| Syntax: | `acc_setmem_hexstr (memhand, hexStrValue, row, start)` | | |
| Returns: | Type | Description | |
| | void | | |
| Arguments: | Type | Name | Description |
| | handle | memhand | Handle to memory |
| | char * | hexStrValue | Hexadecimal string |
| | int | row | The memory array index |
| | int | start | Bit number of the leftmost bit in the memory word where this routine starts writing the string. |
| Related routines: | acc_setmem_int<br>acc_getmem_int<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 | | |

This routine takes a value argument which is a hexadecimal string of any size and puts its corresponding binary representation into the memory word indexed by *`row`*, starting at the bit number *`start`*.

## Examples

The following code examples illustrates the use of `acc_setmem_hexstr`:

- Example E-4 shows the C source code for an application that calls `acc_setmem_hexstr`.
- Example E-5 shows the contents of a data file read by the application.
- Example E-6 shows the PLI table file that associates the user-defined system task in the Verilog code with the application.
- Example E-7 shows the Verilog source that calls the application.

*Example E-4 C Source Code For an Application Calling acc_setmem_hexstr*

```
#include <stdio.h>
#include "acc_user.h"
#include "vcsuser.h"
#define NAME_SIZE 256
#define len 100
pli()
{
  FILE *infile;
  char memory_name[NAME_SIZE] ;
  char value[len];
  handle memory_handle;
  int row,start;

infile = fopen("initfile","r");
while ( fscanf(infile,"%s %s %d %d ",
        memory_name,value,&row,&start) != EOF )
   {
  printf("The mem= %s \n value= %s \n row= %d \n start= %d \n ",
           memory_name,value,row,start);
   memory_handle=acc_handle_object(memory_name);
   acc_setmem_hexstr(memory_handle,value,row,start);
```

```
    }
}
```

Example E-4 shows the source code for a PLI application that:

1. Reads a data file named `initfile` to find the memory identifiers of the memories it writes to, the hexadecimal string to be converted to its bit representation when written to the memory, the index of the memory where it writes this value, and the starting bit for writing the binary value.
2. Displays where in the memory it is writing these values
3. Calls the access routine to write the values in the `initfile`.

*Example E-5 The Data File Read by the Application*

```
testbench.U2.cmd_array 5 0 0
testbench.U2.cmd_array a5 1 4
testbench.U2.cmd_array a5a5 2 8
testbench.U1.slave_addr a073741824 0 4
testbench.U1.slave_addr 16f0612735 1 8
testbench.U1.slave_addr 2b52a90e15 2 12
```

Each line lists a Verilog memory, followed by a hex string, a memory index, and a start bit.

*Example E-6 PLI Table File*

```
$pli call=pli  acc=rw:*
```

Here the `$pli` system task is associated with the function with the `pli` identifier in the C source code.

*Example E-7 Verilog Source Calling the PLI Application*

```
module testbench;
  monitor U1 ();
  master  U2 ();
  initial begin
```

```
  $monitor($stime,,,
    "sladd[0]=%h sladd[1]=%h sladd[2]=%h load=%h
       cmd[0]=%h cmd[1]=%h cmd[2]=%h",
     testbench.U1.slave_addr[0],
     testbench.U1.slave_addr[1],
     testbench.U1.slave_addr[2],
     testbench.U1.load,
     testbench.U2.cmd_array[0],
     testbench.U2.cmd_array[1],
     testbench.U2.cmd_array[2] );
  #10;
  $pli();
  end
endmodule

module master;
  reg[31:0] cmd_array [0:2];
  integer i;
initial begin      //setup some default values
  for (i=0; i<3; i=i+1)
     cmd_array[i] = 32'h0000_0000;
end
endmodule

module monitor;
  reg load;
  reg[63:0] slave_addr [0:2];
  integer i;
initial begin //setup some default values
  for (i=0; i<3; i=i+1)
    slave_addr[i] = 64'h0000_0000_0000_0000;
  load = 1'b0;
end
endmodule
```

In Example E-7 module `testbench` calls the application using the `$pli` user-defined system task for the application. The display string in the `$monitor` system task is on two lines to enhance readability.

## acc_getmem_hexstr

You use the `acc_getmem_hexstr` access routine to get a hexadecimal string from a Verilog memory.

**Table 0-1.**

| acc_getmem_hexstr | | | |
|---|---|---|---|
| Synopsis: | Returns a hexadecimal string from a Verilog memory. | | |
| Syntax: | `acc_getmem_hexstr (memhand,hexStrValue,row,start,len)` | | |
| 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: | acc_setmem_int<br>acc_getmem_int<br>acc_setmem_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_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: | `acc_setmem_bitstr (memhand, bitStrValue, row, start)` | | |
| Returns: | Type | Description | |
| | void | | |
| Arguments: | Type | Name | Description |
| | handle | memhand | Handle to memory |
| | char * | bitStrValue | Bit string |
| | int | row | The memory array index |
| | int | start | Bit number of the leftmost bit in the memory word where this routine starts writing the string. |
| Related routines: | acc_setmem_int<br>acc_getmem_int<br>acc_setmem_hexstr<br>acc_getmem_hexstr<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 | | |

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: | `acc_getmem_bitstr (memhand,bitStrValue,row,start,len)` | | |
| 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: | acc_setmem_int<br>acc_getmem_int<br>acc_setmem_hexstr<br>acc_getmem_hexstr<br>acc_setmem_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_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_fullaname | | | |
|---|---|---|---|
| Synopsis: | Returns a handle to be used as a parameter to acc_readmem only | | |
| Syntax: | `acc_handle_mem_by_fullname (fullMemInstName)` | | |
| Returns: | Type | Description | |
| | handle | Handle to the instance | |
| Arguments: | Type | Name | Description |
| | char* | fullMemInstName | Hierarchical name for a memory |
| 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 | |
| | void | | |
| 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 E-8 C Source Code Calling Tacc_readmem and acc_handle_mem_by_fullname*

```
#include "acc_user.h"
#include "vcs_acc_user.h"
#include "vcsuser.h"

int test_acc_readmem(void)
{
  const char *memName = tf_getcstringp(1);
  const char *memFile = tf_getcstringp(2);
  handle mem = acc_handle_mem_by_fullname(memName);

  if (mem) {
    io_printf("test_acc_readmem: %s handle found\n",
memName);
    acc_readmem(mem, memFile, 'h');
  }
  else {
    io_printf("test_acc_readmem: %s handle NOT found\n",
               memName);
  }
}
```

*Example E-9 The PLI Table File*

```
$test_acc_readmem call=test_acc_readmem
```

*Example E-10 The Verilog Source Code*

```
module top;
reg [7:0] CORE[7:0];
initial $acc_readmem(CORE, "CORE");
initial $test_acc_readmem("top.CORE", "test_mem_file");
endmodule
```

## acc_getmem_range

You use the `acc_getmem_range` access routine to access the upper and lower limits of a memory.

**Table 0-1.**

| acc_getmem_range | | | |
|---|---|---|---|
| Synopsis: | Returns the upper and lower limits of a memory | | |
| Syntax: | `acc_getmem_range (memhandle, p_left_index,p_right_index)` | | |
| Returns: | Type | Description | |
| | void | | |
| Arguments: | Type | Name | Description |
| | handle | memhandle | Handle to a memory |
| | int* | p_left_index | Pointer to int |
| | int | p_right_index | Pointer to int |
| Related routines: | acc_setmem_int<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_size<br>acc_getmem_word_int<br>acc_getmem_word_range | | |

## 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: | `acc_getmem_size (memhandle)` | | |
| Returns: | Type | Description | |
| | int | The number of elements in a memory | |
| Arguments: | Type | Name | Description |
| | handle | memhandle | Handle to a memory |
| 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_word_int<br>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: | `acc_getmem_word_int (memhandle,row)` | | |
| 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: | 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_range | | |

## acc_getmem_word_range

You use the `acc_getmem_word_range` access routine to access the least significant bit of an element (or word, address, or row) and the length of the element.

**Table 0-1.**

| acc_getmem_word_range | | | |
|---|---|---|---|
| Synopsis: | Returns the least significant bit of an element and the length of the element | | |
| Syntax: | `acc_getmem_word_range (memhandle,lsb,len)` | | |
| Returns: | Type | Description | |
| | void | | |
| Arguments: | Type | Name | Description |
| | handle | memhandle | Handle to a memory |
| | int* | lsb | Pointer to the least significant bit |
| | int* | len | Pointer to the length of the element |
| Related routines: | acc_setmem_int<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 | | |

# 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: | `tf_mdanodeinfo(nparam, mdanodeinfo_p)`<br>`tf_imdanodeinfo(nparam, mdanodeinfo_p, instance_p)` | | |
| Returns: | Type | Description | |
| | mdanodeinfo_p * | The value of the second argument if successful; 0 if an error occurs | |
| Arguments: | Type | Name | Description |
| | int | nparam | Index number of the multi-dimensional array parameter |
| | struct t_tfmdanodeinfo * | mdanodeinfo_p | Pointer to a variable declared as the `t_tfmdanodeinfo` structure type |
| | char * | instance_p | 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.**

| acc_get_mda_range() | | | |
|---|---|---|---|
| Synopsis: | Gets all the ranges of the multi-dimensional array. | | |
| Syntax: | `acc_get_mda_range(mdaHandle, size, msb, lsb, dim, plndx, prindex)` | | |
| Returns: | Type | Description | |
| | void | | |
| Arguments: | Type | Name | Description |
| | handle | mdaHandle | Handle to the multi-dimensional array |
| | int * | size | Pointer to the size of the multi-dimensional array |
| | int * | msb | 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: | `acc_get_mda_range(mdaHandle, msb, lsb)` | | |
| Returns: | Type | Description | |
| | void | | |
| Arguments: | Type | Name | Description |
| | handle | mdaHandle | Handle to the multi-dimensional array |
| | int * | msb | Pointer to the most significant bit of a range |
| | int * | lsb | Pointer to the least significant bit of a range |
| Related routines: | tf_mdanodeinfo and tf_imdanodeinfo<br>acc_get_mda_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_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: | `acc_getmda_bitstr(mdaHandle, bitStr, dim, start, len)` | | |
| | Type | Description | |
| Returns: | void | | |
| | Type | Name | Description |
| Arguments: | 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: | tf_mdanodeinfo and tf_imdanodeinfo<br>acc_get_mda_range<br>acc_get_mda_word_range<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:

```
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: | `acc_setmda_bitstr(mdaHandle, bitStr, dim, start, len)` | | |
| Returns: | Type | Description | |
| | void | | |
| Arguments: | Type | Name | Description |
| | handle | mdaHandle | Handle to the multi-dimensional array |
| | char * | bitStr | Pointer to the bit string |
| | int * | dim | Pointer to the dimension in the multi-dimensional array |
| | int * | start | Pointer to the start element in the dimension |
| | int * | len | Pointer to the length of the string |
| Related routines: | tf_mdanodeinfo and tf_imdanodeinfo<br>acc_get_mda_range<br>acc_get_mda_word_range<br>acc_getmda_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:

```
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: | `vcs_random()` | | |
| Returns: | Type | Description | |
| | int | Random number | |
| Arguments: | Type | Name | Description |
| | None | | |
| Related routines: | vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_normal vcs_dist_exponential vcs_dist_poisson | | |

## vcs_random_const_seed

You use this routine to return a random number and you supply an integer constant argument as the seed for the random number.

**Table 0-1.**

| vcs_randon_const_seed | | | |
|---|---|---|---|
| Synopsis: | Returns a random number. | | |
| Syntax: | `vcs_random_const_seed(integer)` | | |
| Returns: | Type | Description | |
| | int | Random number | |
| Arguments: | Type | Name | Description |
| | int | integer | An integer constant. |
| Related routines: | vcs_random vcs_random_seed vcs_dist_uniform vcs_dist_normal vcs_dist_exponential vcs_dist_poisson | | |

## vcs_random_seed

You use this routine to return a random number and you supply a pointer argument.

**Table 0-1.**

| vcs_random_seed() | | | |
|---|---|---|---|
| Synopsis: | Returns a random number. | | |
| Syntax: | `vcs_random_seed(seed)` | | |
| Returns: | Type | Description | |
| | int | Random number | |
| Arguments: | Type | Name | Description |
| | int * | seed | Pointer to an int type. |
| Related routines: | vcs_random vcs_random_const_seed vcs_dist_uniform vcs_dist_normal vcs_dist_exponential vcs_dist_poisson | | |

## vcs_dist_uniform

You use this routine to return a random number uniformly distributed between parameters.

**Table 0-1.**

| vcs_dist_uniform | | | |
|---|---|---|---|
| Synopsis: | Returns random numbers uniformly distributed between parameters. | | |
| Syntax: | `vcs_dist_uniform(seed, start, end)` | | |
| Returns: | Type | Description | |
| | int | Random number | |
| Arguments: | Type | Name | Description |
| | int * | seed | Pointer to a seed integer value. |
| | int | start | Starting parameter for distribution range. |
| | int | end | Ending parameter for distribution range. |
| Related routines: | vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_normal vcs_dist_exponential vcs_dist_poisson | | |

## vcs_dist_normal

You use this routine to return a random number with a specified mean and standard deviation.

**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)` | |
| Returns: | Type | Description |
| | 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 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)` | | |
| | Type | Description | |
| Returns: | int | Random number | |
| | Type | Name | Description |
| Arguments: | int * | seed | Pointer to a seed integer value. |
| | int | mean | An integer that is the average value of the possible returned random numbers. |
| Related routines: | vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_normal vcs_dist_poisson | | |

## vcs_dist_poisson

You use this routine to return a random number with a specified mean.

**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 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 `char*`. | | |
| Syntax: | `acc_fetch_paramval_str(param_handle)` | | |
| Returns: | Type | Description | |
| | char* | string pointer | |
| Arguments: | Type | Name | Description |
| | handle | param_handle | Handle to a module parameter or specparam. |
| Related routines: | acc_fetch_paramval | | |

# 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_dumpports_call” for details.

`acc_lsi_dumpports_close`

Closes specified VCDE files. See “acc_lsi_dumpports_close” for details.

`acc_lsi_dumpports_flush`

Flushes cached data to the VCDE file on disk. See “acc_lsi_dumpports_flush” for details.

`acc_lsi_dumpports_limit`

Sets the maximum VCDE file size. See “acc_lsi_dumpports_limit” for details.

`acc_lsi_dumpports_misc`

Processes miscellaneous events. See “acc_lsi_dumpports_misc” for details.

`acc_lsi_dumpports_off`

Suspends VCDE file dumping. See “acc_lsi_dumpports_off” for details.

`acc_lsi_dumpports_on`

Resumes VCDE file dumping. See “acc_lsi_dumpports_on” for details.

`acc_lsi_dumpports_setformat`

Specifies the format of the VCDE file. See “acc_lsi_dumpports_setformat” for details.

`acc_lsi_dumpports_vhdl_enable`

Enables or disables the inclusion of VHDL drivers in the determination of driver values. See “acc_lsi_dumpports_vhdl_enable” for details.

## acc_lsi_dumpports_all

### Syntax

```
int acc_lsi_dumpports_all(char *filename)
```

### Synopsis

Adds a checkpoint to the file.

This is a PLI interface to the `$dumpportsall` system task. If the `filename` argument is NULL, this routine adds a checkpoint to all open VCDE files.

### Returns

The number of VCDE files that matched.

*Example E-11    Example of acc_lsi_dumpports_all*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
/* rut-roh, error ... */
}
acc_lsi_dumpports_limit(100000, outfile);
...
```

```
if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
     ...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
     ...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
     ...
```

### Caution

This routine may affect files opened by the `$dumpports` and `$lsi_dumpports` system tasks.

---

## acc_lsi_dumpports_call

### Syntax

```
int acc_lsi_dumpports_call(handle instance, char *filename)
```

### Synopsis

Monitors instance ports.

This is a PLI interface to the `$lsi_dumpports` task. The default file format is the original LSI format, but you can select the IEEE format by calling the `routine acc_lsi_dumpports_setformat()` prior to calling this routine. Your tab file will need the following `acc` permissions:

```
acc=cbka,cbk,cbkv:[<instance_name>|*].
```

### Returns

Zero on success, non-zero otherwise. VCS displays error messages through `tf_error()`. A common error is specifying a file name also being used by a `$dumpports` or `$lsi_dumpports` system task.

*Example E-12 Example of acc_lsi_dumpports_all*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);

if (acc_lsi_dumpports_call(instance, outfile)) {
     /* error */
}
```

### Caution

Multiple calls to this routine are allowed, but the output file name must be unique for each call.

For proper dumpports operation, your task's miscellaneous function must call `acc_lsi_dumpports_misc()` with every call it gets. This ensures that the dumpports routines sees all of the simulation events needed for proper update and closure of the 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 E-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 E-14    Example of acc_lsi_dumpports_flush*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...
```

```
if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
   /* add checkpoint (no need to enable dumping) */
   acc_lsi_dumpports_all(outfile);
   acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
   acc_lsi_dumpports_on(outfile);
...
```

---

## acc_lsi_dumpports_limit

### Syntax

```
int acc_lsi_dumpports_limit(unsigned long filesize, char
*filename)
```

### Synopsis

Sets the maximum VCDE file size.

This is a PLI interface to the `$dumpportslimit` task. If the `filename` is NULL, the file size is applied to all files.

### Returns

The number of files matched.

*Example E-15    Example of acc_lsi_dumpports_limit*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";
```

```
/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
     /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
     acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
...
```

### Caution

This routine may affect files opened by the `$dumpports` and `$lsi_dumpports` system tasks.

---

## acc_lsi_dumpports_misc

### Syntax

```
void acc_lsi_dumpports_misc(int data, int reason)
```

### Synopsis

Processes miscellaneous events.

This is a companion routine for `acc_lsi_dumpports_call()`.

For proper dumpports operation, your task's miscellaneous function must call this routine for each call it gets.

### Returns

No return value.

*Example E-16    Example or acc_lsi_dumpports_misc*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

void my_task_misc(int data, int reason)
{
   acc_lsi_dumpports_misc(data, reason);
    ...
}
```

## acc_lsi_dumpports_off

### Syntax

```
int acc_lsi_dumpports_off(char *filename)
```

### Synopsis

Suspends VCDE file dumping.

This is a PLI interface to the `$dumpportsoff` system task.  If the file name is NULL, dumping is suspended on all open files.

### Returns

The number of files that matched.

*Example E-17     of acc_lsi_dumpports_offExample*

```
#include "acc_user.h"
#include "vcs_acc_user.h"
```

```
handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
   acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
     acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
...
```

### Caution

This routine may suspend dumping on files opened by the `$dumpports` and `$lsi_dumpports` system tasks.

---

## acc_lsi_dumpports_on

### Syntax

```
int acc_lsi_dumpports_on(char *filename)
```

### Synopsis

Resumes VCDE file dumping.

This is a PLI interface to the `$dumpportson` system task. If the filename is NULL, dumping is resumed on all open files.

### Returns

The number of files that matched.

*Example E-18 Example of acc_lsi_dumpports_on*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile = "device.evcd";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile)) {
    /* rut-roh */
}
acc_lsi_dumpports_limit(100000, outfile);
...

if (time == yada_yada)
    acc_lsi_dumpports_off(outfile);
...

if (time == yada_yada_yada) {
    /* add checkpoint (no need to enable dumping) */
    acc_lsi_dumpports_all(outfile);
    acc_lsi_dumpports_flush(outfile);
}
...

if (resume_dumping_now)
    acc_lsi_dumpports_on(outfile);
```

```
...
```

### Caution

This routine may resume dumping on files opened by the `$dumpports` and `$lsi_dumpports` system tasks.

---

## acc_lsi_dumpports_setformat

### Syntax

```
int acc_lsi_dumpports_setformat(lsi_dumpports_format_type
format)
```

Where the valid `lsi_dumpports_format_types` are as follows:

USE_DUMPPORTS_FORMAT_IEEE

USE_DUMPPORTS_FORMAT_LSI

### Synopsis

Specifies the format of the VCDE file.

Use this routine to specify which output format (IEEE or the original LSI) should be used. This routine must be called before `acc_lsi_dumpports_call()`.

### Returns

Zero if success, non-zero if error. Errors are reported through `tf_error()`.

*Example E-19  Example of acc_lsi_dumpports_setformat*

```
#include "acc_user.h"
#include "vcs_acc_user.h"
```

```
handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile1 = "device.evcd1";
char *outfile2 = "device.evcd2";

/* use IEEE format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_IEEE);
if (acc_lsi_dumpports_call(instance, outfile1)) {
    /* error */
}

/* use LSI format for this file */
acc_lsi_dumpports_setformat(USE_DUMPPORTS_FORMAT_LSI);
if (acc_lsi_dumpports_call(instance, outfile2)) {
    /* error */
}
...
```

### Caution

The runtime plusargs `+dumpports+ieee` and `+dumpports+lsi` have priority over this routine.

The format of files created by calls to the `$dumpports` and `$lsi_dumpports` tasks are not affected by this routine.

---

## acc_lsi_dumpports_vhdl_enable

### Syntax

```
void acc_lsi_dumpports_vhdl_enable(int enable)
```

The valid enable integer parameters are as follows:

1 enables VHDL drivers

0 disables VHDL drivers

### Synopsis

Use this routine to enable or disable the inclusion of VHDL drivers in the determination of driver values.

### Returns

No return value.

*Example E-20   Example of acc_lsi_dumpports_vhdl_enable*

```
#include "acc_user.h"
#include "vcs_acc_user.h"

handle instance = acc_handle_by_name("test_bench.device",
0);
char *outfile1 = "device.evcd1";
char *outfile2 = "device.evcd2";

/* Include VHDL drivers in this report */
acc_lsi_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 E-21    Example of acc_mod_lcb_add*

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

/* VCS callback rtn */
void line_call_back(filename, lineno, userdata, tag)
char *filename;
int lineno;
char *userdata;
int tag;
{
    handle handle_mod = (handle)userdata;

    io_printf("Tag %2d, file %s, line %2d, module %s\n",
              tag, filename, lineno,
              acc_fetch_fullname(handle_mod));
}
```

```
/* register all modules for line callback (recursive) */
void register_lcb (parent_mod)
handle parent_mod;
{
   handle child = NULL;

   if (! acc_object_of_type(parent_mod, accModule)) return;

   io_printf("Registering %s\n",
             acc_fetch_fullname (parent_mod));

   acc_mod_lcb_add (parent_mod, line_call_back, parent_mod);

    while ((child = acc_next_child (parent_mod, child))) {
       register_lcb (child);
    }
}
```

---

## acc_mod_lcb_del

### Syntax

```
void acc_mod_lcb_del(handle handleModule,
                    void (*consumer)(), char *user_data)
```

### Synopsis

Unregisters a line callback routine previously registered with the `acc_mod_lcb_add()` routine.

### Returns

No return value.

*Example E-22    Example of acc_mod_lcb_del*

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"
```

```
/* VCS 4.x callback rtn */
void line_call_back(filename, lineno, userdata, tag)
char *filename;
int lineno;
char *userdata;
int tag;
{
   handle handle_mod = (handle)userdata;

   io_printf("Tag %2d, file %s, line %2d, module %s\n",
              tag, filename, lineno,
              acc_fetch_fullname(handle_mod));
}

/* unregister all line callbacks (recursive) */
void unregister_lcb (parent_mod)
handle parent_mod;
{
    handle child = NULL;

   if (! acc_object_of_type(parent_mod, accModule)) return;

   io_printf("Unregistering %s\n",
             acc_fetch_fullname (parent_mod));

  acc_mod_lcb_del (parent_mod, line_call_back, parent_mod);

   while ((child = acc_next_child (parent_mod, child))) {
        register_lcb (child);
    }
}
```

### Caution

The module handle, consumer routine, and user data arguments must match those supplied to the `acc_mod_lcb_add()` routine for a successful delete.

For example, using the result of a call such as `acc_fetch_name()` as the user data will fail, because that routine returns a different pointer each time it is called.

## acc_mod_lcb_enabled

### Syntax

```
int acc_mod_lcb_enabled()
```

### Synopsis

Test to see if line callbacks is enabled.

By default, the extra code required to support line callbacks is not added to a simulation executable. You can use this routine to determine if line callbacks have been enabled.

### Returns

Non-zero if line callbacks are enabled; 0 if not enabled.

*Example E-23 Example of acc_mod_lcb_enabled*

```
if (! acc_mod_lcb_enable) {
  tf_warning("Line callbacks not enabled.  Please recompile
with -line.");
}
else {
    acc_mod_lcb_add ( ... );
    ...
}
```

## acc_mod_lcb_fetch

### Syntax

```
p_location acc_mod_lcb_fetch(handle handleModule)
```

### Synopsis

Returns an array of breakable lines.

This routine returns all the lines in a module that you can set breakpoints on.

### Returns

The return value is an array of line number, file name pairs. Termination of the array is indicated by a NULL file name field. The calling routine is responsible for freeing the returned array.

```
typedef struct t_location {
   int line_no;
   char *filename;
} s_location, *p_location;
```

Returns NULL if the module has no breakable lines or is source protected.

*Example E-24    Example of acc_mod_lcb_fetch*

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

void ShowLines(handleModule)
handle handleModule;
{
   p_location plocation;

  if ((plocation = acc_mod_lcb_fetch(handleModule)) != NULL)
{
       int i;

     io_printf("%s:\n", acc_fetch_fullname(handleModule));

        for (i = 0; plocation[i].filename; i++) {
```

```
            io_printf("  [%s:%d]\n",
                        plocation[i].filename,
                        plocation[i].line_no);
        }
        acc_free(plocation);
    }
}
```

---

## acc_mod_lcb_fetch2

### Syntax

```
p_location2 acc_mod_lcb_fetch2(handle handleModule)
```

### Synopsis

Returns an array of breakable lines.

This routine returns all the lines in a module that you can set breakpoints on.

The tag field is a unique identifier used to distinguish `'include` files. For example, in the following Verilog module, the breakable lines in the first `'include` of the file `sequential.code` have a different tag than the breakable lines in the second `'include`. (The tag numbers will match the `vcs_srcfile_info_t->SourceFileTag` field. See the `acc_mod_sfi_fetch()` `routine` for details.)

```
module x;
initial begin
    'include sequential.code
    'include sequential.code
 end
endmodule
```

### Returns

The return value is an array of location structures. Termination of the array is indicated by a NULL filename field. The calling routine is responsible for freeing the returned array.

```
typedef struct t_location2 {
    int line_no;
    char *filename;
    int tag;
} s_location2, *p_location2;
```

Returns NULL if the module has no breakable lines or is source protected.

*Example E-25    Example of acc_mod_lcb_fetch2*

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

void ShowLines2(handleModule)
handle handleModule;
{
    p_location2 plocation;

    if ((plocation = acc_mod_lcb_fetch2(handleModule)) !=
NULL) {
        int i;

      io_printf("%s:\n", acc_fetch_fullname(handleModule));

       for (i = 0; plocation[i].filename; i++) {
           io_printf("  file %s, line %d, tag %d\n",
                      plocation[i].filename,
                      plocation[i].line_no,
                      plocation[i].tag);
        }
        acc_free(plocation);
    }
}
```

## acc_mod_sfi_fetch

### Syntax

```
vcs_srcfile_info_p acc_mod_sfi_fetch(handle handleModule)
```

### Synopsis

Returns the source file composition for a module. This composition is a file name with line numbers, or, if a module definition is in more than one file, it is an array of `vcs_srcfile_info_s` struct entries specifying all the file names and line numbers for the module definition.

### Returns

The returned array is terminated by a NULL `SourceFileName` field. The calling routine is responsible for freeing the returned array.

```
typedef struct vcs_srcfile_info_t {
    char *SourceFileName;
    int SourceFileTag;
    int StartLineNum;
    int EndLineNum;
} vcs_srcfile_info_s, *vcs_srcfile_info_p;
```

Returns NULL if the module is source protected.

*Example E-26 Example of acc_mod_sfi_fetch*

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

void print_info (mod)
handle mod;
{
    vcs_srcfile_info_p infoa;
```

```
    io_printf("Source Info for Module %s:\n",
              acc_fetch_fullname(mod));

   if ((infoa = acc_mod_sfi_fetch(mod)) != NULL) {
       int i;
        for (i = 0; infoa[i].SourceFileName != NULL; i++) {
             io_printf("  Tag %2d, StartLine %2d, ",
                       infoa[i].SourceFileTag,
                       infoa[i].StartLineNum);
             io_printf("EndLine %2d, SrcFile %s\n",
                       infoa[i].EndLineNum,
                       infoa[i].SourceFileName);
        }
        acc_free(infoa);
    }
}
```

## Access Routines for Source Protection

The `enclib.o` file provides a set of access routines that you can use to create applications which directly produce encrypted Verilog source code. Encrypted code can only be decoded by the VCS compiler. There is no user-accessible decode routine.

Note that both Verilog and SDF code can be protected. VCS knows how to automatically decrypt both.

VCS provides the following routines to monitor the port activity of a device:

`vcsSpClose`

This routine frees the memory allocated by `vcsSpInitialize()`. See “vcsSpClose”  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 E-27 outlines the basic use of the source protection routines.

*Example E-27   Using the Source Protection Routines*

```
#include <stdio.h>
#include "enclib.h"
void demo_routine()
{
    char *filename = "protected.vp";
    int write_error = 0;
    vcsSpStateID esp;
    FILE *fp;

    /* Initialization */

    if ((fp = fopen(filename, "w")) == NULL) {
        printf("Error: opening file %s\n", filename);
        exit(1);
    }

    if ((esp = vcsSpInitialize()) == NULL) {
        printf("Error: Initializing src protection
routines.\n");
        printf("       Out Of Memory.\n");
        fclose(fp);
        exit(1);
    }

   vcsSpSetFilePtr(esp, fp);  /* tell rtns where to write */

    /* Write output */

    write_error += vcsSpWriteString(esp,
                  "This text will *not* be encrypted.\n");

    write_error += vcsSpEncodeOn(esp);
    write_error += vcsSpWriteString(esp,
                   "This text *will* be encrypted.");
    write_error += vcsSpWriteChar(esp, '\n');

    write_error += vcsSpEncodeOff(esp);
```

```
    write_error += vcsSpWriteString(esp,
                   "This text will *not* be encrypted.\n");

    /* Clean up */

    write_error += fclose(fp);
    vcsSpClose(esp);

    if (write_error) {
        printf("Error while writing to '%s'\n", filename);
         }
}
```

### Caution

If you are encrypting SDF or Verilog code that contains include directives, you must switch off encryption (`vcsSpEncodeOff`), output the include directive and then switch encryption back on. This ensures that when the parser begins reading the included file, it is in a known (non-decode) state.

If the file being included has proprietary data it can be encrypted separately. (Don't forget to change the `` `include `` compiler directive to point to the new encrypted name.)

---

## vcsSpClose

### Syntax

```
void vcsSpClose(vcsSpStateID esp)
```

### Synopsis

This routine frees the memory allocated by `vcsSpInitialize()`. Call it when source encryption is finished on the specified stream.

### Returns

No return value.

*Example E-28   Example of vcsSpClose*

```
vcsSpStateID esp = vcsSpInitialize();
...
vcsSpClose(esp);
```

## vcsSpEncodeOff

### Syntax

```
int vcsSpEncodeOff(vcsSpStateID esp)
```

### Synopsis

This function performs two operations:

1. It inserts a trailer section that contains some closing information used by the decryption algorithm into the output file. It also inserts the `` `endprotected `` compiler directive in the trailer section.
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 E-29   Example of vcsSpEncodeOff*

```
vcsSpStateID esp = vcsSpInitialize();
FILE *fp = fopen("protected.file", "w");
int write_error = 0; *
if (fp == NULL) exit(1);
```

```
vcsSpSetFilePtr(esp, fp);

if (vcsSpWriteString(esp, "This text will *not* be encrypted.

    ++write_error;

if (vcsSpEncodeOn(esp)) ++write_error;
if (vcsSpWriteString(esp, "This text *will* be encrypted.

    ++write_error;

if (vcsSpEncodeOff(esp)) ++write_error;
if (vcsSpWriteString(esp, "This text will *not* be encrypted.

    ++write_error;

fclose(fp);
vcsSpClose(esp);
```

### Caution

You must call `vcsSpInitialize()` and `vcsSpSetFilePtr()` before calling this routine.

---

## vcsSpEncodeOn

### Syntax

```
int vcsSpEncodeOn(vcsSpStateID esp)
```

### Synopsis

This function performs two operations:

1. It inserts a header section which contains the `‘protected` compiler directive into the output file. It also inserts some initial header information used by the decryption algorithm.

2. It toggles the encryption flag to true so that subsequent calls to `vcsSpWriteString()` and `vcsSpWriteChar()` will have their data encrypted.

### Returns

Non-zero if there was an error writing to the output file, 0 if successful.

*Example E-30 Example of vcsSpEncodeOn*

```
vcsSpStateID esp = vcsSpInitialize();
FILE *fp = fopen("protected.file", "w");
int write_error = 0;

if (fp == NULL) exit(1);

vcsSpSetFilePtr(esp, fp);

if (vcsSpWriteString(esp, "This text will *not* be
encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOn(esp)) ++write_error;
if (vcsSpWriteString(esp, "This text *will* be
encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOff(esp)) ++write_error;
if (vcsSpWriteString(esp, "This text will *not* be
encrypted.\n"))
   ++write_error;
fclose(fp);
vcsSpClose(esp);
```

### Caution

You must call `vcsSpInitialize()` and `vcsSpSetFilePtr()` before calling this routine.

## vcsSpEncoding

### Syntax

```
int vcsSpEncoding(vcsSpStateID esp)
```

### Synopsis

Calling `vcsSpEncodeOn()` and `vcsSpEncodeOff()` turns encoding on and off. Use this function to get the current state of encoding.

### Returns

1 for on, 0 for off.

*Example E-31    Example of vcsSpEncoding*

```
vcsSpStateID esp = vcsSpInitialize();
FILE *fp = fopen("protected.file", "w");

if (fp == NULL) { printf("ERROR: file ..."); exit(1); }

vcsSpSetFilePtr(esp, fp);
...

if (! vcsSpEncoding(esp))
    vcsSpEncodeOn(esp)
...

if (vcsSpEncoding(esp))
    vcsSpEncodeOff(esp);

fclose(fp);
vcsSpClose(esp);
```

## vcsSpGetFilePtr

### Syntax

```
FILE *vcsSpGetFilePtr(vcsSpStateID esp)
```

### Synopsis

This routine just returns the value previously passed to the `vcsSpSetFilePtr()` routine.

### Returns

File pointer or NULL if not set.

*Example E-32 Example of vcsSpGetFilePtr*

```
vcsSpStateID esp = vcsSpInitialize();
FILE *fp = fopen("protected.file", "w");
if (fp != NULL)
    vcsSpSetFilePtr(esp, fp);
else
    /* doh! */

...

if ((gfp = vcsSpGetFilePtr(esp)) != NULL) {
    /* Add comment before starting encryption */
    fprintf(gfp, "\n// TechStuff Version 2.2\n");
    vcsSpEncodeOn(esp);
}
```

### Caution

Don't use non-`vcsSp*` routines (like `fprintf`) in conjunction with `vcsSp*` routines, while encoding is enabled.

## vcsSpInitialize

### Syntax

```
vcsSpStateID vcsSpInitialize(void)
```

### Synopsis

This routine allocates a source protect object.

Returns a handle to a malloc'd object which must be passed to all the other source protection routines.

This object stores the state of the encryption in progress. When the encryption is complete, this object should be passed to `vcsSpClose()` to free the allocated memory.

If you need to write to multiple streams at the same time (perhaps you're creating include or SDF files in parallel with model files), you can make multiple calls to this routine and assign a different file pointer to each handle returned.

Each call mallocs less than 100 bytes of memory.

### Returns

The `vcsSpStateID` pointer or NULL if memory could not be malloc'd.

*Example E-33 Example of vcsSpStateID*

```
vcsSpStateID esp = vcsSpInitialize();
if (esp == NULL) {
    fprintf(stderr, "out of memory\n");
    ...
}
```

### Caution

This routine must be called before any other source protection routine.

A NULL return value means the call to `malloc()` failed. Your program should test for this.

## vcsSpOvaDecodeLine

### Syntax

```
vcsSpStateID vcsSpOvaDecodeLine(vcsSpStateID esp, char
*line)
```

### Synopsis

This routine decrypts one line.

Use this routine to decrypt one line of protected IP code such as OVA code. Pass in a null `vcsSpStateID` handle with the first line of code and a non-null handle with subsequent lines.

### Returns

Returns NULL when the last line has been decrypted.

*Example E-34 Example of vcsSpOvaDecodeLine*

```
#include "enclib.h"

if (strcmp(linebuf, "‘protected_ip synopsys\n")==0) {
    /* start IP decryption */
    vcsSpStateID esp = NULL;
    while (fgets(linebuf, sizeof(linebuf), infile)) {
        /* linebuf contains encrypted source */
        esp = vcsSpOvaDecodeLine(esp, linebuf);
        if (linebuf[0]) {
            /* linebuf contains decrypted source */
```

```
                ...
            }
            if (!esp) break;  /* done */
        }
        /* next line should be `endprotected_ip */
        fgets(linebuf, sizeof(linebuf), infile);
        if (strcmp(linebuf, "`endprotected_ip\n")!=0) {
            printf("warning - expected `endprotected_ip\n");
        }
    }
```

## vcsSpOvaDisable

### Syntax

```
void vcsSpOvaDisable(vcsSpStateID esp)
```

### Synopsis

This routine switches to regular encryption. It tells VCS's encrypter to use the standard algorithm. This is the default mode.

### Returns

No return value.

*Example E-35 Example of vcsSpOvaDisable*

```
#include "enclib.h"
#include "encint.h"

int write_error = 0;
vcsSpStateID esp;

if ((esp = vcsSpInitialize()) printf("Out Of Memory");

vcsSpSetFilePtr(esp, fp); /* previously opened FILE* pointer
*/

/* Configure for OVA IP encryption */
vcsSpOvaEnable(esp, "synopsys");
```

```
if (vcsSpWriteString(esp, "This text will NOT be
encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOn(esp)) ++write_error;

if (vcsSpWriteString(esp, "This text WILL be encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOff(esp)) ++write_error;

if (vcsSpWriteString(esp, "This text will NOT be
encrypted.\n"))
    ++write_error;
/* Switch back to regular encryption */
vcsSpOvaDisable(esp);

if (vcsSpEncodeOn(esp)) ++write_error;

if (vcsSpWriteString(esp, "This text WILL be encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOff(esp)) ++write_error;

vcsSpClose(esp);
```

## vcsSpOvaEnable

### Syntax

```
void vcsSpOvaEnable(vcsSpStateID esp, char *vendor_id)
```

### Synopsis

Enables the OpenVera assertions (OVA) encryption algorithm. Tells VCS's encrypter to use the OVA IP algorithm.

### Returns

No return value.

*Example E-36 Example of vcsSpOvaEnable*

```
#include "enclib.h"
#include "encint.h"

int write_error = 0;
vcsSpStateID esp;

if ((esp = vcsSpInitialize()) printf("Out Of Memory");

vcsSpSetFilePtr(esp, fp); /* previously opened FILE* pointer
*/

/* Configure for OVA IP encryption */
vcsSpOvaEnable(esp, "synopsys");

if (vcsSpWriteString(esp, "This text will NOT be
encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOn(esp)) ++write_error;

if (vcsSpWriteString(esp, "This text WILL be encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOff(esp)) ++write_error;

if (vcsSpWriteString(esp, "This text will NOT be
encrypted.\n"))
   ++write_error;
/* Switch back to regular encryption */
vcsSpOvaDisable(esp);

if (vcsSpEncodeOn(esp)) ++write_error;

if (vcsSpWriteString(esp, "This text WILL be encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOff(esp)) ++write_error;

vcsSpClose(esp);
```

## vcsSpSetDisplayMsgFlag

### Syntax

```
void vcsSpSetDisplayMsgFlag(vcsSpStateID esp, int enable)
```

### Synopsis

This routine sets the DisplayMsg flag. By default, the VCS compiler does not display decrypted source code in its error or warning messages. Use this routine to enable this display.

### Returns

No return value.

*Example E-37 Example of vcsSpSetDisplayMsgFlag*

```
vcsSpStateID esp = vcsSpInitialize();
vcsSpSetDisplayMsgFlag(esp, 0);
```

## vcsSpSetFilePtr

### Syntax

```
void vcsSpSetFilePtr(vcsSpStateID esp, FILE *fp)
```

### Synopsis

This routine specifies the output file stream. Before using the `vcsSpWriteChar()` or `vcsSpWriteString()` routines, you must specify the output file stream.

### Returns

No return value.

*Example E-38 Example of vcsSpSetFilePtr*

```
vcsSpStateID esp = vcsSpInitialize();
```

```
FILE *fp = fopen("protected.file", "w");
if (fp != NULL)
    vcsSpSetFilePtr(esp, fp);
else
    /* abort */
```

## vcsSpSetLibLicenseCode

### Syntax

```
void vcsSpSetLibLicenseCode(vcsSpStateID esp, unsigned int
code)
```

### Synopsis

This routine sets the OEM library license code that will be added to each protected region started by `vcsSpEncodeOn()`.

This code can be used to protect library models from unauthorized use.

When the VCS parser decrypts the protected region, it verifies that the end user has the specified license. If the license does not exist or has expired, VCS exits.

### Returns

No return value.

*Example E-39    Example of vcsSpSetLibLicenseCode*

```
unsigned int lic_code = MY_LICENSE_CODE;
vcsSpStateID esp = vcsSpInitialize();
  ...

/* The following text will be encrypted and licensed */
vcsSpSetLibLicenseCode(esp, code);  /* set license code */
vcsSpEncodeOn(esp);            /* start protected region */
vcsSpWriteString(esp, "this text will be encrypted and
```

```
licensed");
vcsSpEncodeOff(esp);              /* end protected region */

/* The following text will be encrypted but unlicensed */
vcsSpSetLibLicenseCode(esp, 0);     /* clear license code */
vcsSpEncodeOn(esp);              /* start protected region */
vcsSpWriteString(esp, "this text encrypted but not
licensed");
vcsSpEncodeOff(esp);              /* end protected region */
```

### Caution

The rules for mixing licensed and unlicensed code is determined by your OEM licensing agreement with Synopsys.

The code segment in Example E-39 shows how to enable and disable the addition of the license code to the protected regions. Normally you would call this routine once, that is, after calling `vcsSpInitialize()` and before the first call to `vcsSpEncodeOn()`.

---

## vcsSpSetPliProtectionFlag

### Syntax

```
void vcsSpSetPliProtectionFlag(vcsSpStateID esp, int
enable)
```

### Synopsis

This routine sets the PLI protection flag. You can use it to disable the normal PLI protection that is placed on encrypted modules. The output files will still be encrypted, but CLI and PLI users will not be prevented from accessing data in the modules.

This routine only affects encrypted Verilog files. Encrypted SDF files, for example, are not affected.

### Returns

No return value.

*Example E-40   Example of vcsSpSetPliProtectionFlag*

```
vcsSpStateID esp = vcsSpInitialize();
vcsSpSetPliProtectionFlag(esp, 0); /* disable PLI protection
*/
```

### Caution

Turning off PLI protection will allow users of your modules to access object names, values, etc. In essence, the source code for your module could be substantially reconstructed using the CLI commands and ACC routines.

## vcsSpWriteChar

### Syntax

```
void vcsSpSetPliProtectionFlag(vcsSpStateID esp, int
enable)
```

### Synopsis

This routine writes one character to the protected file.

If encoding is enabled (see “vcsSpEncodeOn” ) 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 E-41 Example of vcsSpWriteChar*

```
vcsSpStateID esp = vcsSpInitialize();
FILE *fp = fopen("protected.file", "w");
int write_error = 0;

if (fp == NULL) exit(1);

vcsSpSetFilePtr(esp, fp);

if (vcsSpWriteChar(esp, 'a')) /* This char will *not* be
encrypted.*/
    ++write_error;

if (vcsSpEncodeOn(esp))
    ++write_error;

if (vcsSpWriteChar(esp, 'b')) /* This char *will* be
encrypted. */
    ++write_error;
if (vcsSpEncodeOff(esp))
    ++write_error;

fclose(fp);
vcsSpClose(esp);
```

### Caution

`vcsSpInitialize()` and `vcsSpSetFilePtr()` must be called prior to calling this routine.

## vcsSpWriteString

### Syntax

```
int vcsSpWriteString(vcsSpStateID esp, char *s)
```

#### Synopsis

This routine writes a character string to the protected file.

If encoding is enabled (see “vcsSpEncodeOn” ) 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 E-42 Example of vcsSpWriteString*

```
vcsSpStateID esp = vcsSpInitialize();
FILE *fp = fopen("protected.file", "w");
int write_error = 0;

if (fp == NULL) exit(1);

vcsSpSetFilePtr(esp, fp);

if (vcsSpWriteString(esp, "This text will *not* be
encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOn(esp)) ++write_error;
if (vcsSpWriteString(esp, "This text *will* be
encrypted.\n"))
    ++write_error;

if (vcsSpEncodeOff(esp)) ++write_error;
if (vcsSpWriteString(esp, "This text will *not* be
encrypted.\n"))
```

```
    ++write_error;

fclose(fp);
vcsSpClose(esp);
```

### Caution

`vcsSpInitialize()` and `vcsSpSetFilePtr()` must be called prior to calling this routine.

## Access Routine for Signal in a Generate Block

There is only one access routine for signals in generate blocks.

### acc_object_of_type

#### Syntax

```
bool acc_object_of_type(accGenerated, sigHandle)
```

#### Synopsis

This routine returns true if the signal is in a generate block.

#### Returns

1 - if the signal is in a generate block.

0 - if the signal is not in a generate block.

## VCS API Routines

Typically VCS controls the PLI application. If you write your application so that it controls VCS, you need these API routines.

## Vcsinit()

When VCS is run in slave mode, you can call this function to elaborate the design and to initialize various data structures, scheduling queues, etc. that VCS uses. After this routine executes, all the initial time 0 events, such as the execution of initial blocks, are scheduled.

Call the `vmc_main(int argc, char *argv)` routine to pass runtime flags to VCS before you call `VcsInit()`.

## VcsSimUntil()

This routine tells VCS to schedule a stop event at the specified simulation time and execute all scheduled simulation events until it executes the stop event. The syntax for this routine is as follows:

```
VcsSimUntil (unsigned int* t)
```

Argument `t` is for specifying the simulation time. It needs two words. The first [0] is for simulation times from 0 to $2^{32}$ -1, the second is for simulation times that follow.

If any events are scheduled to occur after time `t`, their execution must wait for another call to `VcsSimUntil`.

If `t` is less than the current simulation time, VCS returns control to the calling routine.

# Index

runtime information message generating D-12

## Symbols

B-99, C-8
-a filename B-95
-ams_discipline B-74
-ams_iereport B-74
-assert 15-40, B-10
  svaext 15-40
-bfl and -bom B-101
-bom and -bfl B-101
-C B-72
-c B-69
-CC B-71
-cc B-71
-CFLAGS B-71
-cm assert B-18
-cpp B-72
-debug 23-33, B-94
-debug_all B-94
-debug_pp B-94
-e name_for_main B-54
-E program runtime option C-34
-error 4-49, 4-51, 4-55, B-10, B-58, C-22
-error=PRIORITY C-20
-error=UNIQUE C-20
-extinclude B-29, D-9
-f filename B-50
-gui 2-6, 5-8
-h 2-3, B-9
-help 2-3, B-9
-ID 2-3, B-68
-jnumber_of_CPUs B-72
-l C-24
-l filename 2-7, B-94, C-22
-ld linker B-68
-LDFLAGS B-68
-lmc-swift B-58
-lmc-swift-template B-58
-lname B-69
-load 4-24, 23-39, B-56, C-37
-Makep B-8
-Mdir B-7
-Mdirectory B-7
-Mlib=dir B-7
-msg_config 4-49, 4-66
-Mupdate B-8
-nc B-59
-negdelay B-49
-noIncComp B-9
-ntb B-20
-ntb_define B-20
-ntb_filext B-21

-ntb_incdir B-21
-ntb_opts B-21
-ntb_sfname B-27
-ntb_vipext B-27
-o name B-97
-O number B-73
-O0 B-72
-ova_enable_case B-20
-override_timescale B-78
-P pli.tab B-54
-parameters 2-6, 4-32, B-75
-platform B-97
-PP D-21
-pvalue 2-6, 4-32, B-75
-q 2-6, B-62, C-23
-R 2-6, B-38, B-95
-simprofile B-50
-simprofile_dir_path 6-7
-suppress 4-49, B-59
-sysc B-90
-u B-95
-ucli 5-6
-uniq_prior maxfail=integer C-19
-V 2-7, B-62, C-23
-v 2-3, B-4
-Vt B-62
-Xman 25-23
-Xmangle 25-23
-Xnoman 25-24
-Xnomangle 25-24
-Xova B-20
-y 2-3, B-5
assert hier=file.txt B-18
‘celldefine B-66, B-67, D-2, D-3
‘default_nettype D-2
‘define D-3
‘delay_mode_distributed D-6
‘delay_mode_path D-6
‘delay_mode_unit D-6
‘delay_mode_zero D-7
‘else D-3
‘elseif D-3
‘endcelldefine D-2
‘endif D-4
‘endprotect 25-17
‘endprotected 25-17
‘endprotected128 25-17
‘endrace B-76
‘ifdef D-4
‘ifdef VCS D-4
‘ifndef D-5
‘include B-29, D-9
  specifying the search directories B-90
  with a different verion of Verilog B-29
‘line D-11
‘nounconnected_drive D-11
‘protect 25-17, 25-22
‘protect128 25-17
‘protected 25-17
‘race B-76
‘resetall D-3
‘timescale B-78, D-9
‘unconnected_drive D-11
‘undef D-6
‘uselib D-10
‘vcs_mipdexpand D-8
"A" specifier of abstract access 23-56
"C" specifier of direct access 23-56
**NC 3-15
/*synopsys translate_off*/ pragma B-100
/*synopsys translate_on*/ pragma B-100
//synopsys translate_off pragma B-100
//synopsys translate_on pragma B-100
%CELL 23-14, 23-20
%TASK 23-15
+abstract 23-131
+acc+2 B-53
+acc+3 B-54
+acc+4 B-54
+acc+level_number 23-24, B-53
+allhdrs 23-131
+allmtm B-39, C-27, C-28

+applylearn 23-28–23-36, C-34
+applylearn+filename B-54
+auto2protect 25-22
+auto3protect 25-22
+autoprotect 25-21
+charge_decay B-39
+define+macro=value 2-7, B-95
+delay_mode_distributed 12-38, B-40
+delay_mode_path 12-38, B-39
+delay_mode_unit 12-38, B-39
+delay_mode_zero 12-38, B-39
+deleteprotected 25-22
+incdir 2-4, B-89
+iopath+edge B-45
+libext 2-5, B-6
+liborder 2-5, B-6
+librescan B-6
+libverbose B-7, B-59
+lint 4-49, B-59
+lint=PWLNT
  B-19
+lint=sva 20-41
+list 23-131
+maxdelays B-39, B-42, C-27, C-35
+memcbk B-94
+mindelays B-39, B-42, C-27, C-35
+module module_identifier C-18
+multisource_int_delays B-43
+nbaopt B-43
+neg_tchk 12-59, 12-66, B-50
+no_notifier 12-59, B-47, C-20
+no_pulse_msg C-23
+no_tchk_msg 12-59, B-47, C-20
+nocelldefinepli+0 B-66
+nocelldefinepli+1 B-66
+nocelldefinepli+2 B-66
+noerrorIOPCWM B-91, B-92
+nolibcell B-66
+noportcoerce B-91
+nospecify 12-60, B-46
+notimingcheck 12-60, B-46, C-20
+ntb_cache_dir C-3
+ntb_delete_disk_cache C-3
+ntb_disable_cnst_null_object_warning C-3
+ntb_enable_checker_trace C-4
+ntb_enable_checker_trace_on_failure C-4
+ntb_enable_solver_trace_on_failure C-5
+ntb_enable_solver_trace_on_failure=value C-5
+ntb_exit_on_error C-5
+ntb_random_seed C-6
+ntb_random_seed_automatic C-6
+ntb_solver_array_size_warn C-7
+ntb_solver_debug 17-28, C-7
  extract 17-35, 17-38
  profile 17-33, 17-37
  serial 17-37
  trace 17-30, 17-33, 17-37
+ntb_solver_debug_dir C-8
+ntb_solver_debug_filter 17-30, 17-33, 17-35, C-8
+ntb_solver_mode C-9
+ntb_solver_mode=value C-9
+NTC2 12-65, B-50
+object_protect 25-34
+old_ntc B-50
+optconfigfile 10-6, B-38
+override_model_delays C-27, C-28, C-35
+pathpulse B-46
+pli_unprotected 25-22
+plusarg_ignore B-53
+plusarg_save B-53
+plus-options C-35
+protect file_suffix 25-22
+pulse_e/number 12-24, 12-26, 12-28, 12-33, 12-34, B-48
+pulse_int_e 12-23, 12-24, 12-26, 12-28, B-48
+pulse_int_r 12-23, 12-24, 12-26, 12-28, B-48
+pulse_on_detect 12-34, B-49
+pulse_on_event 12-33, B-48
+pulse_r/number 12-24, 12-26, 12-28, 12-33, 12-34, B-48
+putprotect+target_dir 25-22

+race=all 3-20, 3-21, B-76
+rad 10-6, B-38
+sdf_nocheck_celltype B-43
+sdfprotect file_suffix 25-23
+sdfverbose C-23
+systemverilogext B-28
+tetramax B-91
+timopt 12-40
+transport_int_delays 12-23, 12-26, 12-28, B-43
+transport_path_delays 12-23, 12-26, 12-28, B-44
+typdelays B-39, B-43, C-28, C-35
+UVM_LOG_RECORD 33-6
+UVM_TR_RECORD 33-6
+vc 23-130, B-56
+vcs+dumpfile+filename C-26
+vcs+dumpoff+t+ht C-26
+vcs+dumpon+t+ht C-26
+vcs+finish 5-27, C-21
+vcs+flush+all B-57, C-29
+vcs+flush+dump B-57, C-27, C-29
+vcs+flush+fopen B-57, C-29
+vcs+flush+log B-57, C-29
+vcs+ignorestop C-34
+vcs+initreg
  restricting initialization to either registers or memories" 4-31
+vcs+initreg+0|1|random| C-31
+vcs+initreg+0|1|random|seed_value C-31
+vcs+initreg+config 4-27, B-33, C-32
+vcs+initreg+config+config_file C-33
+vcs+initreg+random| C-33
+vcs+learn+pli ??–23-33, C-34
+vcs+lic+vcsi C-30
+vcs+lic+wait C-30
+vcs+loopdetect+number B-98, C-36
+vcs+loopreport+number B-98, C-35
+vcs+mipd+noalias C-36
+vcs+mipdexpand D-8
+vcs+nostdout C-24
+vcs+stop 5-27, C-21
+vcs+vcdpluson B-92
+vcsi+lic+vcs C-30
+verilog1995ext B-28
+verilog2001ext B-28
+vpddrivers C-26
+vpdfile 5-8
+vpdfileswitchsize 5-8
+vpdfileswitchsize+number_in_MB C-25
+vpdnoports C-26
+vpdportsonly C-26
+vpdupdate C-26
+vpi B-54
+vpi+1 B-54
+vpi+1+assertion B-55
+warn 3-53, 4-49, B-62
$assert_category_start 20-19, 20-23
$assert_category_stop 20-18
$assert_monitor 20-11, D-27
$assert_monitor_off 20-12, D-28
$assert_monitor_on 20-12, D-28
$assert_set_category 20-16, 20-19
$assert_set_severity 20-16
$assert_severity_stop 20-19
$assertcontrol 20-56
$assertkill D-12
$assertoff D-12
$asserton D-12
$async$and$array D-40
$bitstoreal D-30
$countdrivers D-43
$countunknown(expression) D-14
$countx(expression) D-13
$countz(expression) D-13
$deposit D-44
$disable_warnings D-36
$display D-31
$dist_exponential D-41
$dist_normal D-41
$dist_poisson D-41
$dist_uniform D-42

$dumpall D-14
$dumpfile D-15
$dumpflush D-15
$dumplimit D-15
$dumpoff D-14
$dumpon D-14
$dumpports D-17
$dumpports system task C-29
$dumpportsall D-19
$dumpportsflush D-19
$dumpportslimit D-19
$dumpportsoff D-18
$dumpportson D-18
$dumpvars D-15
$enable_warnings D-36
$error 20-50, D-12
$fatal 20-31, D-11
$fclose D-31
$fdisplay D-32
$ferror D-32
$fflush D-16, D-32
$fflushall D-16
$fgetc D-32
$fgets D-32
$finish D-36
$fmonitor D-32
$fopen B-57, D-32
  increasing the frequency of flushing B-57
$fopen system function C-29
  increasing the frequency of $fopen file, log file, and VCD file dumping C-29
  increasing the frequency of dumping to files opened by $fopen C-29
$fread D-33
$fscanf D-33
$fseek D-33
$fstobe D-33
$ftell D-33
$fwrite D-33
$get_initial_random_seed D-42
$getpattern D-44
$gr_waves D-16
$hold D-37
$info D-12
$isunknown D-13
$itor D-30
$log D-29
$lsi_dumpports 3-53–3-58, D-16
$lsi_dumpports system task C-29
$monitor D-31
$monitoroff D-31
$monitoron D-31
$nolog D-30
$onedriven D-14
$onedriven0 D-14
$onehot0 D-13
$past
  ignoring B-18
$period D-37
$printtimescale D-35
$q_add D-40
$q_exam D-40
$q_full D-40
$q_initialize D-40
$q_remove D-40
$random 3-52, D-42
$read_lib_saif 24-5
$readmemb 5-38, D-34
$readmemh 5-38, D-34
$realtime D-41
$realtobits D-30
$recovery D-38
$recrem D-38
  checking timestamp and timecheck conditions B-50
  diabling delayed versions of signals in other timing checks B-50
$removal D-38
$reset D-42
$reset_count D-42
$reset_value D-42
$restart D-44

$rtoi D-30
$save D-44
$sdf_annotate D-43
$set_gate_level_monitoring 24-4
$set_toggle_region 24-6
$setup D-39
$setuphold D-39
  checking timestamp and timecheck conditions B-50
  diabling delayed versions of signals in other timing checks B-50
$skew D-39
$sreadmemb D-34
$sreadmemh D-35
$stime D-41
$stop D-36
  ignoring C-34
$strobe D-31
$sync$nor$plane D-40
$system D-29
$systemf D-29
$test$plusargs C-35, D-43
$time D-41
$timeformat D-35
$ungetc D-34
$uniq_prior_checkoff system task 15-98, 15-99
$uniq_prior_checkon system task 15-98
$value$plusargs 5-22
$vcdplusautoflushoff D-21
$vcdplusautoflushon D-21
$vcdplusclose D-21
$vcdplusdeltacycleoff 9-12
$vcdplusdeltacycleon 9-11, D-21
$vcdplusevent D-22
$vcdplusfile D-23
$vcdplusflush D-23
$vcdplusglitchon D-23
$vcdplusmemoff 9-8, D-25
$vcdplusmemon 9-8, D-23
$vcdplusmemorydump 9-8, D-25
$vcdplusoff D-26
$vcdpluson B-93, D-26
$vcdplusxx system tasks
  ignoring C-25
$warning D-12
$width D-39
$write D-31
$writememb 5-38, D-35
$writememh 5-38, D-35


# A


-a filename B-95
"A" specifier of abstract access 23-56
+abstract 23-131
abstract access for C/C++ functions
  access routines for 23-82–23-125
  enabling with a compile-time option 23-131
  using 23-80–23-125
+acc+level_number 23-24, B-53
ACC capabilities 23-32, B-53
  applying in the design only where they are needed C-34
  cbk 23-12, 23-21
  cbka 23-12
  enabling debugging B-54
  frc 23-12, 23-21
  gate 23-13
  mip 23-13, C-36
  mipb 23-13
  mipd C-36
  mp 23-13
  prx 23-12
  r 23-12, 23-20
  recording where in the design they are needed C-34
  rw 23-12, 23-21
  s 23-12
  specifying 23-10–23-22
  tchk 23-13
acc_handle_simulated_net C-36
access routines for abstract access of C/C++ functions 23-82–23-125
Active time slot

changing UDP output evaluation to the NBA time slot 3-5
-ad B-73
Addressing Models 11-28
Signed indexing 11-29
Unsigned Indexing 11-28
AICMs
information messages B-74
+allhdrs 23-131
+allmtm B-39, C-27, C-28
alt_retain 12-6
-ams_discipline B-74
-ams_iereport B-74
ansi argument to -ntb_opts B-21
ANSI mode
in OpenVera files B-21
aop
advice
before/after/around 16-16
dominates 16-7
extends directive 16-3
placement element
after 16-11
around 16-11
C-34
+applylearn 23-28–23-36
arb.v 14-5
args PLI Specificaction 23-8
array
output and inout argument type 23-73
array index 17-52
array members 17-72
assembler
passing options to B-71
-assert 15-40, 20-50, B-10, C-10
svaext 15-40
assert 26-5
-assert assertion_block_identifier C-18
-assert failonly 20-44
-assert funchier 20-26, 20-27
-assert hier=file.txt B-18
-assert no_default_msg 20-29
–assert no_default_msg 20-29
–assert no_fatal_action 20-30
-assert quiet 20-29
-assert report 20-29
-assert success 20-11
-assert svvunit 20-49
$assert_monitor 20-11, D-27
$assert_monitor_off 20-12, D-28
$assert_monitor_on 20-12, D-28
assertion failure messages
controlling 20-28
assertion waring messages
suppressing B-9
Assertions
SystemVerilog
enabling or disabling a module or a hierarchy B-12
assertions
fatal error generating D-11
OpenVera B-20
blind signals B-27
bounds check in dynamic arrays B-21
bounds check in fixed-size arrays B-22
circular dependency check B-25
display on screen B-25
disabling default failure messages C-12
encrypted IP mode
filename extension B-27
encryption
tokens file B-26
file name extension B-21
file-by-file preprocessing
disabling B-25
include directory path B-21
including case violations in the global failure count C-20
interface ports named ifc_signal B-27
left padding in strings B-25
RVM enabling B-25
signal property access functions
enabling B-27
testbench shell
filename specifying B-27
generating only B-27

not generating B-21
testbench shell and shared object files
specifying the directory B-27
timescale B-26
VMM enabling B-25
Openvera
ANSI mode B-21
PSL
disabling default failure messages C-12
resume monitoring D-12
returning true if one bit is true D-13
returning true if one bit is X D-13
returning true if only one bit is true or no bits are true D-13
runtime error generating D-12
runtime information message generating D-12
runtime warning generating D-12
SystemVerilog
cover statements
disabling B-18
disabling B-18
disabling assertion failure messages C-12
but enabling summary information C-12
disabling default failure messages C-12
disabling from a file
specifying assertion block C-18
specifying module definitions C-18
disabling information in a VPD file B-19
dumping SVA in VPD file
disabling C-10
enabling and disabling from a file C-15
enabling assertion match (success) messages C-14
enabling from a file
specifying module definitions C-18
enabling runtime options B-11
enabling the -assert hier=file.txt runtime option for turning assertions off B-17
enabling vacuous success messages C-14
enhsnce reporting for assertions in functions B-12
excluding assertion failures with fail action blocks C-12
failure simulation time in Debussy B-19
generationg a report file C-13
adding more information C-14
ignoring $past B-18
maximum number of cover statement
specifying the total number of cover statements in the assertion coverage information C-11
monitoring for assertion coverage C-19
no match simulation time in Debussy B-19
not displaying the assert or cover statement summary C-11
not writing the program_name.db database file C-11
specifying configuration file B-12
specifying the maximum number of failures for each assertion C-11
specifying the maximum number of successes for each assertion C-11
specifying the number of failures for an assertion C-10
specifying the rotal number of assertion failures C-10
starting simulation time in Debussy B-19
turning off monitoring D-12
$assertkill D-12
$assertoff D-12
$asserton D-12
assert.report file C-13
adding more information C-14
assume 20-35, 26-5
$async$and$array D-40
+auto2protect 25-22
+auto3protect 25-22
auto-inserted connect modules (AICMs)
displaying information about B-74
+autoprotect 25-21
avoid using +verilogamsext+vams B-74

# B

Backward SAIF File 24-5
-bfl and -bom B-101
bidirectional registered mixed-signal net
dispalying a list of B-74
finishing compilation at B-74

bit
  C/C++ function argument type 23-59
  C/C++ function return type 23-58
  input argument type 23-72
  output and inout argument type 23-72
  reg data type in two-state simulation 23-55
$bitstoreal D-30
-bom and -bfl B-101
BoM file
  list of source files B-101
bounds check
  in OpenVera dynamic arrays B-21
  in OpenVera fixed-sise arrays B-22
buffer
  emptying into VCD files D-15

# C

-C B-72
C 17-52
-c 14-5, B-69
C code generating
  halt before compiling the generated C code B-72
  passing options to the compiler B-71
  specifying another compiler B-71
  specifying the optimization level B-73
  suppressing optimization for faster compilation B-73
C compiler
  not passing default options B-73
  optimization levels B-71
  passing options to B-71
  specifying B-71
C compiler, environment variable specifying the A-4
"C" specifier of direct access 23-56
C/C++ functions
  argument direction 23-57, 23-58
  argument type 23-57, 23-59
  calling 23-62–23-63
  declaring 23-55–23-61
  extern declaration 23-56
  in a Verilog environment 23-54–23-55
  return range 23-57
  return type 23-57, 23-58
  using abstract access 23-80–23-125
    access routines for 23-82–23-125
  using direct access 23-70–23-79
    examples 23-73–23-77
C++ compiler
  specifying B-72
call PLI specification 23-7
callbacks for memories and multi-dimensional arrays
  enabling B-94
calling C/C++ functions in your Verilog code 23-62–23-63
case pragmas
  enabling B-20
cbk ACC capability 23-12, 23-21
cbka ACC capability 23-12
-CC B-71
-cc B-71
cell
  for delay annotation
    disabling D-2
    specifying D-2
cell modules
  excluding from compilation B-66
‘celldefine B-66, B-67, D-2, D-3
CELLTYPE entries in SDF files
  disabling B-43
-CFLAGS B-71
-cg_coverage_control C-2
char*
  direct access for C/C++ functions
    formal parameter type 23-70
char**
  direct access for C/C++ functions
    formal parameter type 23-70
charge decay
  enabling B-39
+charge_decay B-39
check argument to -ntb_opts B-21
check PLI specification 23-7
check=all B-22

check=fixed B-22
checkers
  clocking 20-54
  instantiating 20-54
checkpoint
  in VCD files
    recording current values D-14
    start recording current values D-14
    stop recording current values D-14
circular dependency check check
  in OpenVera B-25
    display on screen B-25
class 17-70
classes
  inheritance between 17-72
clock signals 12-40–12-44
-cm 13-3, C-19
-cm assert B-18
code generation
  mixed code generation B-4
compiler directives D-1–D-11
  resetting D-3
compile-time options B-1–??
  displaying at runtime C-34
  specifying in a file B-50
compiling
  incremental compilation
    triggering ??–10-3
  omitting compilation between pragmas B-99
  verbose messages 2-7, B-62
  with ‘include and -extinclude B-29
compression
  disasbling for VPD files C-26
conditional expressions
  warning when evaluate to X or Z B-75
    filtering out false negatives B-75
configuration file
  for Radiant technology B-38
consistent behavior of class static properties 15-47
constraint solver
  array size warning C-7
  OpenVera
    trace information C-5
constraints
  conflicts 17-67
  constraint profiling 17-33, 17-37
  debugging C-7, C-8
  partitions 17-26
  strings in 17-66
  test case extraction 17-34, 17-35, 17-38
  unique 17-82
copyright information
  displaying C-23
copyright message
  suppressing B-59
$countdrivers D-43
cover 26-5
Coverage Database 26-3
coverage groups
  OpenVera
    enabling C-2
-cpp B-72

# D

data PLI specification 23-7
-debug 23-33, B-94
-debug_access+classdbg 4-12
-debug_access+dmptf 4-18
-debug_all B-94
debug_all, option 5-7, 5-8
-debug_pp 5-6, B-93, B-94
debug_pp, option 5-6
-debug_region 4-11
-debug_region=encrypt 4-16
-debug_region=stdpkg 4-12
-debug_region=tb 4-15
debug, option 5-7
Debussy B-94
  assertions
    failure simulation time B-19
    no match simulation time B-19
    stating simulation time B-19

declaring C/C++ functions in your Verilog code 23-55–23-61
default discrete discipline
  in VerilogAMS B-74
default net data type
  specifying D-2
‘default_nettype D-2
‘define D-3
+define+macro=value 2-7, B-95
delay mode distributed behavior B-40
delay mode path behavior B-39
delay mode unit behavior B-39
delay mode zero behavior B-39
delay values
  back annotating to your design D-43
‘delay_mode_distributed D-6
+delay_mode_distributed 12-38, B-40
‘delay_mode_path D-6
+delay_mode_path 12-38, B-39
‘delay_mode_unit D-6
+delay_mode_unit 12-38, B-39
‘delay_mode_zero D-7
+delay_mode_zero 12-38, B-39
delays C-27, C-28
  changing all delays to zero D-7
  changing all to 0 B-39
  ignoring all delays except gate, switch, and continuous assignment delays D-6
  ignoring all delays except module path delays D-6
  ignoring all module path delays and using for all other delay specifications the shortest time precision argument D-6
  ignoring module path delays and changing gate, switch, and continuous assignment delays to time precision B-39
  ignoring module path delays and using gate, switch, and continuous assignment delays B-40
  module path delays
    X value B-48
    with error message B-49
  on gates and switches
    ignoring B-39
  specifies using max of min|typ|max delays B-42
  specifies using min of min|typ|max delays B-42
  specifies using typ of min|typ|max delays B-43
  transport delays B-43, B-44
+deleteprotected 25-22
delta cycle information D-21
Denali 30-1
dep_check argument to -ntb_opts B-25
$deposit D-44
-deraceclockdata option 5-31
Design Description 14-6
-diag run_env B-102
-diag sys_task_mem 5-37, 5-38
diagnostics 6-113
direct access for C/C++ functions
  examples 23-73–23-77
  formal parameters
    types 23-70
  rules for parameter types 23-71–23-73
  using 23-70–23-130
DirectC
  abstract access
    specifying B-56
  enabling B-56
  listing the C/C++ functions B-56
  using pass by reference 23-69
  vc_hdrs.h file B-56
direction of a C/C++ function argument 23-58
directory for constraint solver profiles and testcases C-8
disable B-18
disable soft 17-80
disable_cover B-18
$disable_warnings D-36
Disabling XIndex Merging 11-31
$display D-31
DISPLAY_VCS_HOME A-3
$dist_exponential D-41
$dist_normal D-41
$dist_poisson D-41

$dist_uniform D-42
dmptf 4-6
double*
  direct access for C/C++ functions
    formal parameter type 23-70
DPI 17-52
$dumpall D-14
$dumpfile D-15
$dumpflush D-15
$dumplimit D-15
$dumpoff D-14
dumpoff B-19, C-10
$dumpon D-14
$dumpports D-17
$dumpportsall D-19
$dumpportsflush D-19
$dumpportslimit D-19
$dumpportsoff D-18
$dumpportson D-18
$dumpvars D-15
dynamic race detection B-76


# E


-e name_for_main B-54
-E program C-34
echo C-34
edge operator 20-55
edge sensitivity
  in SDF file IOPATH entries B-45
elaboration system tasks 15-40
Element Merging Methods 11-29
  Default 11-29
  Dimensional 11-30
  Index Resolution 11-31
  Randomized 11-30
‘else D-3
‘elseif D-3
enable_diag 20-11, B-11
enable_hier B-17
$enable_warnings D-36
enabling C-2
  only where used in the last simulation 23-32
enabling writing initialized values to a file 4-32
encryption
  all modules 25-21
    but not the module header 25-22
    but not the module header and parameter declarations 25-22
  enabling overwriting of existing files 25-22
  enabling PLI and UCLI access 25-22
  OpenVera
    tokens file B-26
  SDF files 25-23
  specifying the directory for encrypted files 25-22
  specifying with ‘protect ‘endprotect 25-22
‘endcelldefine D-2
‘endif D-4
ending simulation at a specified time C-21
‘endprotect 25-17
‘endprotected 25-17
‘endprotected128 25-17
Environment variables 1-7–1-8, A-1–??
-error 4-49, 4-51, 4-55, B-10, B-58, C-22
$error D-12
error messages
  changing to warning B-61
error_default_action_block 20-50
-error=PRIORITY C-20
-error=UNIQUE C-20
EVCD files D-17
  flushing the buffer D-19
  recording all port values D-19
  resume recording D-18
  specifying the file size D-19
  suspending D-18
executable
  specifying the name of B-97
-exitstatus 5-37
extended summary information
  displaying C-23
extends directive
  advice 16-4

introduction 16-4
extern declaration 23-56
extern declarations 23-77
-extinclude B-29, D-9

# F

-F B-51
-f filename B-50
fail action blocks C-12
fail-only 20-44
$fatal D-11
fatal assertion error generating D-11
$fclose D-31
$fdisplay D-32
$ferror D-32
$fflush D-16, D-32
$fflushall D-16
$fgetc D-32
$fgets D-32
-file 2-5, B-51
file
for runtime options C-30
file name extension in Verilog library directories
specifying B-6
files
grw.dump file D-16
tokens.v 25-23
VCD files
specifying the filename D-15
VPD
specifying at compile-time B-92
start recording D-26
filter_past B-18
$finish D-36
finish_maxfail=N C-10
$fmonitor D-32
$fopen B-57, D-32
increasing the frequency of flushing B-57
-force_list 5-9
foreach loops 17-57
four state Verilog data
stored in vec32 23-64–23-65
frc ACC capability 23-12, 23-21
$fread D-33
$fscanf D-33
FSDB files B-94
$fseek D-33
$fstobe D-33
$ftell D-33
function calls
context 17-54
DPI 17-52
non-pure 17-53
pure 17-52
$fwrite D-33

# G

gate ACC capability 23-13
gate-level
improving runtime performance B-99
gd_pulsewarn 12-9
generics
overriding with the -gfile elaboration option B-79
-genid_db B-93
$get_initial_random_seed D-42
$getpattern D-44
-gfile B-79
global_finish_maxfail=N C-10
globalDirective 20-23
gmake A-2
$gr_waves D-16
grw.dump file D-16
-gui 2-6, 5-8
-gui=dve 34-15
-gui=verdi 34-15

# H

-h 2-3, B-9
hard constraint 17-68
header and summary

suppressing C-23
-help 2-3, B-9
help with compile-time options, runtime options, and environment variables B-9
hier=file_name C-15
$hold D-37
-hsopt=gates B-99
-hsopt=racedetect 3-22
HVP 26-7

## I

-ID 2-3, B-68
ifc_signal
  OpenVera interface ports named B-27
‘ifdef D-4
‘ifndef D-5
-ignore B-9
+incdir 2-4, B-89
‘include D-9
  specifying the search directories B-90
including one source file in another D-9
incremental compilation B-7–B-9
  central place for descriptor information and object files B-7
  disabling B-9
  specifying the make path B-8
  updating the makefile B-8
incremental compile directory
  specifying B-7
Index BSpace 11-27
$info D-12
information messages
  about finding module definitrions in a library B-59
  lint B-59
-ignore B-9
initializing Verilog memories and variables
  in only parts of the design 4-26, 4-27
inout
  C/C++ function argument direction 23-59
input
  C/C++ function argument direction 23-58
int
  C/C++ function argument type 23-59
  C/C++ function return type 23-58
  direct access for C/C++ functions
    formal parameter type 23-70
  input argument type 23-72
  output and inout argument type 23-72
int*
  direct access for C/C++ functions
    formal parameter type 23-70
INTERCONNECT delays
  rejecting B-48
  SDF files B-43
    changing to transport delays B-43
    negative values enabling B-49
interface 14-7

    self() 15-105
Interface Description 14-14
internal disk cache for randomization
  delete before simulation C-3
  location C-3
intra-assingment delays
  removing B-43
IOPATH delays
  SDF files
    negative values enabling B-49
+iopath+edge B-45
Isolate
  Cost of Garbage Collection 6-114
$isunknown D-13
$itor D-30

## J

-jnumber_of_CPUs B-72

## K

-kdb 34-3
key word conflicts B-74
keywords

after 16-11
around 16-11
before 16-11
extends 16-3
virtuals 16-31

## L

-l C-24
-l filename 2-7, B-94, C-22
-ld linker B-68
-LDFLAGS options B-68
let operator 20-55
+libext 2-5, B-6
-libmap 4-38
+liborder 2-5, B-6
+librescan B-6
+libverbose B-7, B-59
licenses
enabling license queuing C-30
running VCS with three VCSi licenses C-30
running VCSi with a VCS license C-30
waiting for a license C-30
waiting for a network license C-30
'line D-11
Line-Based CPU Time Profiler 6-107
linker
linking a library to the executable B-69
linking by hand B-69
passing flags to B-68
specifying B-68
temporary object files B-69
linking
linking a specified library to the executable B-69
linking by hand B-69
passing options to the linker B-68
specifying another linker B-68
+lint 4-49, B-59
lint messages B-59
+list 23-131
LMC SWIFT interface
including B-58
-lmc-swift B-58
-lmc-swift-template B-58
-lname B-69
-load 4-24, 23-39, B-56, C-37
$log D-29
log file
appending to B-95
simulation
specifying C-22
log file buffers
increasing the frequency of flushing B-57
log file, environment variable specifying the A-4
log files
increasing the frequency of log file dumping C-29
increasing the frequency of log file, VCD file, and $fopen file dumping C-29
specifying compilation log file 2-7, B-94
specifying with a system task D-29
loops
specifying the maximum number of loops B-98, C-35, C-36
LSI certification D-16
EVCD files D-17
flushing the buffer D-19
icluding strength levels in the VCD file D-16
recording all port values D-19
resume recording D-18
specifying the file size D-19
suspends recording D-18
$lsi_dumpports 3-53–3-58, D-16

## M

macros
text macro defining B-95
text macros
defining D-3
else defining D-3
else if end D-4
elseif defining D-3
if defining D-4
if not defined D-5

undefining D-6
main() routine
specifying for PLI B-54
maintaining filename and line number D-11
make A-2
makefile
updating B-8
mangled file source protection 25-23
except module and port identifiers 25-24
-Marchive B-7, B-69
maxargs PLI specification 23-8
maxcover=N C-11
C-35
+maxdelays B-39, B-42, C-27
maxfail=N C-11
maxsuccess=N C-11
-mcg B-4
MDAs 17-58
-Mdir B-7
-Mdirectory B-7
mem_pli 6-17
mem_solver
argument to the profrpt -view option 6-13, 6-81
+memcbk B-94
memories
sparse memory models 3-44
Memory Constraint Solver view
in profiler reports 6-13, 6-81, 6-91
Memory Modeler - Advanced Verification (MMAV) 30-1
memory size limits 3-42
message control
by module definition 4-73
by source file 4-72
by sub-hierarchy 4-73
downgrading error messages 4-70
error messages 4-70
lint messages 4-68
suppressing messages 4-71
upgrading lint and warning messages 4-70
warning messages 4-69
messages
about finding module definitions in a library B-59
changing error to warning B-61
copyright message
suppressing B-59
lint B-59
quiet mode B-62
verbose mode B-62
including CPU time information B-62
warning
disabling B-62
-metadump 4-74
minargs PLI specification 23-8
C-35
+mindelays B-39, B-42, C-27
mip ACC capability 23-13
mipb ACC capability 23-13, D-8
MIPDs C-36
disabling connection upon MIPD delay annotation C-36
misc PLI specification 23-7
mixed analog/digital simulation
specifying B-73
mixed code generation B-4
mixed signal simulation
specifying B-73
-Mlib=dir B-7
-Mmakep B-8
module description , Verilog 14-15
-module module_identifier C-18
module path delays
allowing
in specific module instances 12-39
changing to tranport delays B-44
disabling B-46
disabling for an instance 12-39
suppressing
in specific module instances 12-39
X value B-48
X value with error message B-49
$monitor D-31
$monitoroff D-31

$monitoron D-31
mp ACC capability 23-13
-msg_config 4-49, 4-66
Multicore Technology ALP 8-1–8-5
multiple packed dimensions 17-58
+multisource_int_delays B-43
-Mupdate B-8

# N

+nbaopt B-43
-nbaudpsched B-99
**NC 3-15
-nc B-59
+neg_tchk 12-59, 12-66, B-50
negative multiconcat multiplier
  allowing B-92
negative timing checks B-49
-negdelay B-49
nets
  specifung defult data type D-2
no_default_msg C-12
-no_error ID+ID B-61
no_fatal_action C-12
no_file_by_file_pp argument to -ntb_opts B-25
+no_identifier C-20
+no_notifier 12-59, B-47
+no_pulse_msg C-23
+no_tchk_msg 12-59, B-47, C-20
+nocelldefinepli+1 B-66
nocelldefinepli PLI specification 23-8
+nocelldefinepli+0 B-66
+nocelldefinepli+2 B-66
nocovdb C-11
-noerror UPIMI+IOPCWM B-62
-xzcheck B-75
-nogenid_db B-93
-noIncrComp B-9
+nolibcell B-66
$nolog D-30
nonblocking assignments
  removing intra-assignment delays B-43
+noportcoerce B-91
nopostproc C-11
+nospecify 12-60, B-46
-notice 2-6
notifier register
  in timincheck system tasks
    disabling B-47
notifier registers, suppressing the toggling of C-20
+notimingcheck 12-60, B-46, C-20
'nounconnected_drive D-11
-ntb B-20
+ntb_cache_dir C-3
-ntb_define B-20
+ntb_delete_disk_cache C-3
+ntb_enable_solver_trace_on_failure C-5
+ntb_exit_on_error C-5
-ntb_filext B-21
-ntb_incdir B-21
-ntb_noshell B-21
-ntb_opts B-21
  print_deps B-25
  rvm B-25
  sv_fmt B-25
-ntb_opts no_file_by_file_pp 14-26
+ntb_random_seed C-6
+ntb_random_seed_automatic C-6
-ntb_sfname B-27
-ntb_shell_only B-27
-ntb_sname B-27
+ntb_solver_array_size_warn C-7
+ntb_solver_debug 17-28, C-7
  extract 17-35, 17-38
  profile 17-37
  serial 17-37
  trace 17-30, 17-37
+ntb_solver_debug_dir C-8
+ntb_solver_debug_filter 17-30, 17-33, 17-35, C-8
+ntb_solver_mode C-9
-ntb_spath B-27

-ntb_vipext 14-26, B-27
+NTC2 12-65, B-50

# O

-o name B-97
-O number B-73
-O0 B-72
object files
  enabling position independent code B-69
  specifying temporary B-69
+object_protect 25-34
+old_ntc B-50
$onehot
  $onehot D-13
$onehot0 D-13
OpenVera
  constraint solver mode C-9
  coverage groups C-2
  diagnostics
    when randomize() method called C-4
  enabling debugging
    when randomize() method called C-4
  exit on error C-5
  internal disk cache C-3
    delete before simulation C-3
  on null object handle of object randomized C-3
  trace information
    when randomize() returns 0 C-4
  trace information when constraint solver fails C-5
operating system commands, executing D-29
+optconfigfile 10-6, B-38
optimization
  suppresssing for faster compilation B-72
options for macro expansion B-96
output
  C/C++ function argument direction 23-59
OVA 20-28
–ova_enable_case 20-33, B-20
-ova_enable_case_maxfail C-20
–ova_enable_case_maxfail 20-32, C-20
-ova_inline B-20
–ova_inline 20-33, B-20
+override_model_delays C-27, C-28, C-35
-override_timescale B-78
-override-cflags B-73
Overriding Parameter Values
  Through Configuration File 15-89

# P

-P pli.tab 23-22, B-54
-p188_macro_expansion B-96
packed constraints 17-57
packed dimensions 17-57
-parallel 8-2
parallel compilation B-9, B-72
  disabling B-9
  specifying the number of forks B-72
-parallel_compile_off B-9
-parameters 2-6, 4-32, B-75
parameters
  overriding B-75, B-81
  overriding with the -gfile elaboration option B-79
partition compile
  limitations 8-5
partitions
  in constraints 17-26
pass by reference in DirectC 23-69
+pathpulse B-46
PATHPULSE$ specparam, enabling B-46
performance
  improving for gate-level designs B-99
$period D-37
persistent 23-9
PERSISTENT_FLAG A-3
-picarchive B-69
placement element
  after 16-11
  around 16-11
-platform B-97
platform directory in the VCS installation

returning B-97
PLI
ACC capabilities B-53
enabling debugging B-54
allowing access to ports and parameters B-67
disabling capabilities for ‘celldefine and library modules B-67
disabling capabilities for ‘celldefine modules B-67
enabling in encrypted files 25-22
specifying the name of your main() routine B-54
PLI library
loading dynamically at runtime 4-24, C-38
PLI specifications
args 23-8
call 23-7
check 23-7
data 23-7
maxargs 23-8
minargs 23-8
misc 23-7
nocelldefinepli 23-8
size 23-8
PLI table file 23-6–23-23, C-34
specifying B-54
pli_learn.tab C-34
+pli_unprotected 25-22
pli.tab file 23-6–23-23, C-36
specifying B-54
+plusarg_ignore B-53
+plusarg_save B-53
plusargs, checking for on the simv command line D-43
+plus-options C-35
pointer
C/C++ function argument type 23-59
C/C++ function return type 23-58
input argument type 23-72
output and inout argument type 23-72
port coercion
disabling B-91
position independent code
enabling B-69
-PP D-21
print_deps argument to -ntb_opts B-25
$printtimescale D-35
priority keyword 15-91
PRIORITY violations
limiting the number of C-19
procedure_prototype
example 16-29, 16-30
profile database
specifying the pathname 6-7
profiler
simulation 6-1
Profiler Report
Third-Party Shared Library 6-116
profiling B-50
profprt 6-19
profrpt 6-20
program_name.db database file
not writing C-11
proprietary message
suppressing C-23
‘protect 25-17, 25-22
+protect file_suffix 25-22
‘protect128 25-17
‘protected 25-17
prx ACC capability 23-12
PSL 20-28, 20-49
pulse error messages
suppressing C-23
+pulse_e/number 12-24, 12-26, 12-28, 12-33, 12-34, B-48
+pulse_int_e 12-23, 12-24, 12-26, 12-28, B-48
+pulse_int_r 12-23, 12-24, 12-26, 12-28, B-48
+pulse_on_detect 12-34, B-49
+pulse_on_event 12-33, B-48
+pulse_r/number 12-24, 12-26, 12-28, 12-33, 12-34, B-48
pulses
filtering out narrow pulses B-48
and flag as error B-48
on INTERCONNECT delays

INTERCONNECT delays
filtering out
SDF files
INTERCONNECT
delays
filtering out
B-48
rejecting narrow pulses B-48
on SDF INTERCONNECT delays B-48
X value B-48, B-49
+putprotect+target_dir 25-22
-pvalue 2-6, 4-32, B-75, B-81

## Q

-q 2-6, B-62, C-23
$q_add D-40
$q_exam D-40
$q_full D-40
$q_initialize D-40
$q_remove D-40
queue=blocking|nonblocking 31-4
quiet mode - suppressing
header and summary information C-23
proprietary message C-23
simulation report at the end of simulation C-23

## R

-R 2-6, B-38, B-95
r ACC capability 23-12, 23-13, 23-14, 23-20
-race B-76
race conditions
avoiding 3-2–3-8
continuous assignment evaluations 3-5
generating a report of B-76
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 3-20, 3-21, B-76
-racecd B-76
race.out file B-76
+rad 10-6, B-38
Radiant techology
configuration file B-38
enabling B-38
rand members 17-70
rand_mode() method 15-68
$random 3-52, D-42
random number generator
re-seeding C-6
random values
setting the seed C-6
after restore C-6
randomize() method 15-68
randomize() serial number 17-37
randomize()solver trace 17-28
randomized objects in a structure 17-61
$readmemb D-34
$readmemh D-34
real
C/C++ function argument type 23-59
input argument type 23-72
output and inout argument type 23-72
$realtime D-41
$realtobits D-30
$recovery D-38
$recrem D-38
checking timestamp and timecheck conditions B-50
diabling delayed versions of signals in other timing checks B-50
reg
C/C++ function argument type 23-59
C/C++ function return type 23-58
input argument type 23-72
output and inout argument type 23-72
reporting debug capabilities for each module 6-102
$reset D-42
$reset_count D-42
$reset_value D-42

‘resetall D-3
resetting
  keeping track of the number of resets D-42
  passing a value from before to after a reset D-42
  resetting VCS to simulation time 0 D-42
Resolving message upon instance resolution B-7
resolving module instances D-10
$restart D-44
restrict 20-35
RETAIN entries
  SDF files
    enabling B-44
return range of a C/C++ function 23-57
return type of a C/C++ function 23-57, 23-58
RTL Verilog example 14-7
$rtoi D-30
runtime assertion error generating D-12
runtime assertion warning generating D-12
runtime options
  compiling into the executable B-53
  prevent compiling into the executable B-53
  specifying in as file C-30
RVM B-25
rvm B-25
rw ACC capability 23-12, 23-14, 23-21

# S

s ACC capability 23-12
SAIF file dumping
  in Multicore ALP 8-3
-saif_opt 24-9
$save D-44
scalar
  direct access for C/C++ functions
    formal parameter type 23-70
scalar*
  direct access for C/C++ functions
    formal parameter type 23-70
scope randomize method 15-68
SDF 12-6
  optimistic mode 12-6
SDF backannotating
  enabling more than 10 warning and error messages C-23
SDF delay back-annotation
  disabling back-annotation to individual bits of an input port D-8
  to individual bits of an input port D-8
SDF files
  compiling separate files for min|typ|max delays B-39
  disabling CELLTYPE entries B-43
  encryption 25-23
  INTERCONNECT delays B-43
    changing to transport delays B-43
    negative values enabling B-49
    rejecting B-48
  INTERCONNECT entries
    negative values enabling B-49
  IOPATH delays
    negative values enabling B-49
  IOPATH entries
    edge sensitivirty B-45
    negavive values enabling B-49
  min|typ|max delays
    specified in a file B-38
  RETAIN entries
    enabling B-44
-sdf min|typ|max
  instance_name
    file.sdf B-38
$sdf_annotate D-43
+sdf_nocheck_celltype B-43
+sdfprotect file_suffix 25-23
-sdfretain 12-6, B-44
-sdfretain=warning B-44
SDFRT_IRV
  wanring B-44
+sdfverbose C-23
search order of Verilog library directories B-6
  rescan B-6
sequential devices
  inferring 3-34–3-38, 12-40–12-44

sequential UDPs
  changing output evaluation to the active region B-99
  changing output evaluation to the NBA tile slot 3-5
serial2trace.txt file 17-34
$setup D-39
$setuphold D-39
  checking timestamp and timecheck conditions B-50
  diabling delayed versions of signals in other timing checks B-50
signal property access functions
  OpenVera
    enabling B-27
-simBin 34-17
-simdir 34-6, 34-17
-simflow 34-4, 34-16
-simprofile 6-18, B-50
-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 6-18
simprofile_dir 6-19
-simprofile_dir_path 6-7
simulation
  immediately after compilation B-38
simulation report at the end of simulation
  suppressing C-23
simulation state
  saving D-44
simulation time slice based profiler 6-110
simv executable
  specifying a deifferent name B-97
single class 17-69
single packed dimension 17-57
size PLI specification 23-8
$skew D-39
skip_celldefine_scopes 24-9
-skip_translate_body B-99
soft constraint 17-68
soft constraints 17-67, 17-76
  disabling 17-67
  prioritization 17-69
soft keyword 17-68
solver trace reporting
  for the specified randomize() calls 17-37
SOMA 30-2
souce protection
  enabling overwriting of existing files 25-22
  enabling PLI and UCLI access 25-22
  encrypting all modules 25-21
    but not the module headers 25-22
    but not the module headers and parameter declarations 25-22
  specifying the directory for protected files 25-22
  specifying with ‘protect ‘endprotect 25-22
source file
  specifying in a file B-50
source files
  generating a list of B-101
source protection
  mangling 25-23
    except module and port identifiers 25-24
  SDF files 25-23
  specifying the end of the code to be protected 25-17
  specifying the end of the protected code 25-17
  specifying the start of the code to be protected 25-17
  specifying the start of the protected code 25-17
sparse memory models 3-44
specify blocks
  allowing
    in specific module instances 12-39
  disabling for an instance 12-39
  suppressing B-46
    in specific module instances 12-39
srandom(seed) system function C-6, C-7
$sreadmemb D-34
$sreadmemh D-35

state variables 17-50
Static Race Detection Tool B-76
std

randomize() method 15-68
$stimen D-41
C-34
$stop D-36
stopping simulation at a specified time C-21
string
C/C++ function argument type 23-59
C/C++ function return type 23-58
input argument type 23-72
output and inout argument type 23-72
strings
in constraints 17-66
$strobe D-31
-suppress 4-49, B-59
SV and RT assertions
browse, enable, and disable B-55
sv_fmt argument to -ntb_opts B-25
SVA 20-28, 20-49
-sverilog B-9
SWIFT SmartModels C-35
generating a template B-58
replaying DelayRange parameter definition with +mindelay, +typdelay, or +maxdelay C-35
$sync$nor$plane D-40
/*synopsys translate_off*/ pragma B-100
//synopsys translate_off pragma B-100
/*synopsys translate_on*/ pragma B-100
//synopsys translate_on pragma B-100
-sysc B-90
$system D-29
System Function Call
$clog2() 17-89
$size() 17-88
system tasks D-11–D-51, ??–D-51
disabling text output from C-24
IEEE standard system tasks not implemented D-51
SystemC
cosimulating with Verilog 1-2, 22-1
SystemC cosimulation B-94
enabling B-90
time resolution B-90
SYSTEMC_OVERRIDE A-3
$systemf D-29
SystemVerilog 17-67, 20-49
enabling B-9
randomized objects in a structure 17-61
specifying the filename extension B-28
unpacked dimensions B-93, D-27
SystemVerilog assertions 20-1–??
SystemVerilog LRM 17-52
+systemverilogext B-28

# T

-t 14-5
tb_timescale argument to -ntb_opts B-26
tchk ACC capability 23-13
temporary object files B-69
$test$plusargs D-43
testbench
OpenVera
enabling B-20
macro on command line B-20
timescale B-26
testbench template 14-7
+tetramax B-91
TetraMAX testbench simulation in zero delay mode B-91
text macros
defining B-95, D-3
else defining D-3
else if end D-4
elseif defining D-3
if defining D-4
if not defined D-5
undefining D-6
text output display from system tasks
disabling C-24
$time D-41
Time Constraint Solver view

in profiler reports 6-12, 6-81, 6-82
time precision
as delay specification D-6
time scale
for the compilation-unit scope B-77
overrideing the ‘timescale compiler directive f4rom the vcs command line B-78
specifying on the vcs command line B-77
time scale for time units and time precision D-9
time_pli 6-17, 6-19
time_solver
argument to the profrpt -view option 6-81
$timeformat D-35
-timescale B-77
‘timescale D-9
timescale
OpenVera testbench B-26
timing check system tasks
checking timestamp and timecheck conditions B-50
disabling B-46
in specific module instances 12-39
disabling delayed versions of signals B-50
disabling display of timing violations B-47
disabling toggling the notifier register B-47
enabling
in specific module instances 12-39
negative values enabling B-50
timing check system tasks, disabling B-46
timing checks
disabling C-20
disabling for an instance 12-39
suppressing the toggling of notifier registers C-20
timing violations
disabling B-47
disabling the display of C-20
timming checks
disabling the display of timing violations C-20
Timopt
the timing optimizer 12-40–12-44
+timopt 12-40
TMPDIR A-3
toggle coverage
in Multicore ALP 8-3
toggle_start_at_set_region 24-9
toggle_stop_at_toggle_report 24-9
tokens argument to -ntb_opts B-26
tokens.v file 20-18, 25-23
-top 4-44
top-level Verilog Module 14-7
transport delays B-43, B-44
+transport_int_delays 12-23, 12-26, 12-28, B-43
+transport_path_delays 12-23, 12-26, 12-28, B-44
C-35
+typdelays B-39, B-43, C-28

# U

U
direct access for C/C++ functions
formal parameter type 23-70
-u B-95
U*
direct access for C/C++ functions
formal parameter type 23-70
UB*
direct access for C/C++ functions
formal parameter type 23-70
UCLI
enabling in encrypted files 25-22
-ucli 5-6, 5-8
UCLI Commands for X-Propagation Control Tasks 11-41
UDPs
sequential UDPs
changing output evaluation to the active region B-99
changing output evaluation to the NBA time slot 3-5
unaccelerated
definitions and declarations 3-33–3-34
structural instance declarations 3-33
‘unconnected_drive D-11

‘undef D-6
$ungetc D-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 6-13
  mem_inst
    argument to the profrpt -view option 6-12
  mem_mod
    argument to the profrpt -view option 6-13
  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 6-12
  time_mod
    argument to the profrpt -view option 6-12
  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
-uniq_prior maxfail=integer C-19
uniq_prior_final compiler switch 15-91
unique constraints 17-82
unique keyword 15-91
UNIQUE violations
  limiting the number of C-19
uniquifying identifier codes in VCD files 9-21
-unit_timescale B-77
unpacked D-27
unpacked dimensions B-93, D-27
unused 26-5
-upf 34-17
uppercase
  changing Verilog identifiers to B-95
use_sigprop B-27
use_sigprop argument to -ntb_opts B-27
‘uselib D-10
user-defined plusarg enabling C-35
utility, vcsplit 9-47

# V

-V 2-7, B-62, C-23
-v 2-3, B-4, B-66
vacuous success message enabling C-14
$value$plusargs 5-22
+vc 23-130, B-56
VC Formal 26-2
VC Formal Assert 26-7
VC Formal Results 26-3
vc_2stVectorRef() 23-101
vc_4stVectorRef() 23-100
vc_argInfo() 23-123
vc_arraySize() 23-90
vc_FillWithScalar() 23-120
vc_get2stMemoryVector() 23-116
vc_get2stVector() 23-105
vc_get4stMemoryVector() 23-114
vc_get4stVector() 23-103
vc_getInteger() 23-99
vc_getMemoryInteger() 23-111
vc_getMemoryScalar() 23-110
vc_getPointer() 23-97
vc_getReal() 23-95
vc_getScalar() 23-90
vc_handle
  definition 23-80
  using 23-80–23-82

vc_hdrs.h file 23-77–23-78
  in DirectC B-56
vc_Index() 23-124
vc_Index2() 23-125
vc_Index3() 23-125
vc_is2state() 23-87
vc_is2stVector() 23-88
vc_is4state() 23-85
vc_is4stVector() 23-87
vc_isMemory() 23-85
vc_isScalar() 23-83
vc_isVector() 23-84, 23-126
vc_mdaSize() 23-125
vc_MemoryElemRef) 23-108
vc_MemoryRef() 23-105
vc_MemoryString() 23-118
vc_MemoryStringF() 23-119
vc_put2stMemoryVector() 23-117
vc_put2stVector() 23-105
vc_put4stMemoryVector() 23-116
vc_put4stVector() 23-103
vc_putInteger() 23-99
vc_putMemoryInteger() 23-114
vc_putMemoryScalar() 23-111
vc_putMemoryValue() 23-117
vc_putMemoryValueF() 23-117
vc_putPointer() 23-97
vc_putReal() 23-94
vc_putScalar() 23-90
vc_putValue() 23-95
vc_putValueF() 23-96
VC_STATIC_HOME 26-3
vc_StringToVector() 23-98
vc_toChar() 23-91
vc_toInteger() 23-91
vc_toString() 23-92
vc_toStringF() 23-94
vc_VectorToString() 23-99
vc_width() 23-89
vcat utility 9-33
VCD file
  specifying on the vcs command line B-92
VCD files
  checkpoint
    recording current values D-14
    start recording current values D-14
    stop recording current values D-14
  emptying or flushing the buffer D-15
  enabling VCD dumping for memories and multi-dimensional arrays C-27
  flushing the latest data to all open VCD files D-16
  flushing the latest data to the VCD file D-16
  for LSI certification D-16
  grw.dump file D-16
  including strength levels D-16
  increasing the frequency of flushing B-57
  increasing the frequency of VCD file dumping C-27, C-29
  LSI certification
    flushing the buffer D-19
    recording all port values D-19
    resume recording D-18
    specifying the file size D-19
    suspending D-18
  recording in another VCD file D-15
  specifing the time to turn on VCD dumping C-26
  specifying a limit to the VCD file size D-15
  specifying the filename D-15
  specifying the name of the VCD file C-26
  specifying the nets and variables recorded in the file D-15
  specifying the time to turn off VCD dumping C-26
VCD+ 9-2
  Advantages 9-2
  System Tasks
    $vcdplusdeltacycleoff 9-12
    $vcdplusdeltacycleon 9-11
    $vcdplusmemoff 9-8
    $vcdplusmemon 9-8
    $vcdplusmemorydump 9-8
vcdiff utility 9-23
  syntax 7-26, 9-23
$vcdplusautoflushoff D-21

$vcdplusautoflushon D-21
$vcdplusclose D-21
$vcdplusdeltacycleon D-21
$vcdplusevent D-22
$vcdplusfile D-23
$vcdplusflush D-23
$vcdplusglitchon D-23
$vcdplusmemoff D-25
$vcdplusmemon D-23
$vcdplusmemorydump D-25
$vcdplusoff D-26
$vcdpluson D-26
vcdpost utility 9-20
  syntax 9-22
vcf 26-2, 26-3
VCS
  predefined text macro D-4
VCS MX V2K Configurations and Libmaps 4-37
VCS_CC A-3
VCS_COM A-4
VCS_HOME B-68
VCS_LIC_EXPIRE_WARNING A-4
VCS_LOG A-4
'vcs_mipdexpand D-8
VCS_NO_RT_STACK_TRACE A-4
VCS_PRINT_INITREG_INITIALIZATION environment variable 4-32
VCS_SWIFT_NOTES A-4, A-5
+vcs+dumpoff+t+ht C-26
+vcs+dumpfile+filename C-26
+vcs+dumpon+t+ht C-26
+vcs+finish 5-27, C-21
+vcs+flush+all C-29
+vcs+flush+dump C-27, C-29
+vcs+flush+fopen C-29
+vcs+flush+log C-29
+vcs+ignorestop C-34
+vcs+initreg+config 4-27, B-33, C-32
+vcs+learn+pli ??–23-33, C-34
+vcs+lic+vcsi C-30
+vcs+lic+wait C-30
+vcs+mipd+noalias C-36
+vcs+mipdexpand D-8
+vcs+nostdout C-24
+vcs+stop 5-27, C-21
+vcs+vcdpluson B-92
vcsfind 1-10
+vcsi+lic+vcs C-30
vcsplit utility 9-47
$vcdpluson B-93
vec32
  storing four state Verilog data 23-64–23-65
vec32*
  direct access for C/C++ functions
    formal parameter type 23-70
vera_portname argument to -ntb_opts B-27
verbose mode - displaying
  compile-time and runtime numbers C-23
  copyright information C-23
  version and extended summary information C-23
-verdi 26-5, 34-15
verdi -cov 26-6
-verdi_opts 34-16
Verilog 4-37
  different versions
    filename extension B-28
  different versions'include
    filename extension B-29
Verilog 1995
  specifying the filename extension B-28
Verilog 2001
  specifying the filename extension B-28
Verilog identifiers
  changing to uppwecase B-95
Verilog library
  resolving module instances D-10
Verilog library directories
  displaying a message upon instance resolution B-7
  file name extensions
    specifying B-6
  specifying B-5

specifying the search order B-6
rescan B-6
Verilog library files
specifying B-4
Verilog model, example 14-7
Verilog module 14-7
Verilog module description 14-15
Verilog parameters
overriding B-75, B-81
overriding with the -gfile elaboration option B-79
+verilog1995ext B-28
+verilog2001ext B-28
VerilogAMS
defaule discrete discipline B-74
version number
returning B-68
VHDL generics
overriding with the -gfile elaboration option B-79
VHDL two state objects in Xprop 11-44
virtual interface
self instance 15-105
VMM B-25
void
C/C++ function return type 23-58
void*
direct access for C/C++ functions
formal parameter type 23-70
void**
direct access for C/C++ functions
formal parameter type 23-70
VPD file
specifying on the vcs command line B-92
VPD files D-20
buffer for
specifying the size of C-24
disable recording values for memories and MDAs D-25
disabling file compression C-26
disabling recording in transition times an values defined under ‘celldefine B-66
disabling recording in transition times an values defined under ‘celldefine or in a library B-66
enabling recording in transition times an values defined under ‘celldefine B-66
enabling VPD file locking C-26
ignoring $vcdplusxx system tasks C-25
marking as completed and closing D-21
record a unique event for a signal D-22
recording changes on the drivers of resolved nets C-26
recording delta cycle information D-21
recording only ports and their direction C-26
recording ports and their direction C-25
recording signals but not ports C-26
recording values for memories and MDAs D-23
records a snapshot of memories and MDAs D-25
specifying the name C-24
specifying the next VPD file D-23
specifying the size of C-24
start recording D-26
stop recording D-26
switching to record another VPD file C-25
turning off automatic flushing D-21
turning on automatic flushing D-21
turning on zero delay glitches D-23
write simulation results to the VPD file D-23
+vpdfile 5-8
+vpdfileswitchsize 5-8
VPI 17-55
specifying the registration routine in a shared library B-56
SV and RT assertions
browse, enable, and disable B-55
+vpi B-54
VPI PLI access routines
enabling B-54
vpiSeqBeginTime B-19
vpiSeqFail B-19
-Vt B-62
-vts_ignore_env=ENV1,ENV2,... B-78
vunit 20-49

# W


+warn 3-53, 4-49, B-62
$warning D-12
warning messages
  disabling B-62
  sover array size warning C-7
$width D-39
with expression in streaming 15-52
wn ACC capability 23-12
$write D-31
$writememb D-35
$writememh D-35


# X


XIndex Element Merging 11-25
-xlrm 12-5
-xlrm alt_retain 12-6
-xlrm env_expand A-5
-xlrm gd_pulseprop 12-8
-xlrm gd_pulsewarn 12-9
-xlrm uniq_prior_final compile switch 15-91
-xlrm_uniq_prior_observed 15-99
-Xman 25-23
-Xmangle 25-23
XMR 17-50
-Xnoman 25-24
-Xnomangle 25-24
-Xova 20-33
–Xova B-20
Xprop configuration file
  merge_mode 11-33
  select_method 11-33


# Y


-y 2-3, B-5, B-66


# Z


zero multiconcat multiplier
  allowing B-92