VCS®/VCSi™
User Guide

Version X-2005.06
August 2005

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

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2002 Synopsys, Inc. All rights reserved. This software and documentation are owned by Synopsys, Inc., and furnished under a license agreement. The software and documentation may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

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

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

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys, the Synopsys logo, Arcadia, BiNMOS-CBA, CMOS-CBA, COSSAP, DESIGN (ARROWS), DesignPower, DesignWare, dont_use, EPIC, ExpressModel, in-Sync, LM-1000, LM-1200, Logic Modeling, Logic Modeling (logo), Memory Architect, ModelAccess, ModelTools, PathMill, *PL*debug, RailMill, SmartLicense, SmartLogic, SmartModel, SmartModels, SNUG, SOLV-IT!, SourceModel Library, Stream Driven Simulator, Synopsys, Synopsys (logo), Synopsys VHDL Compiler, Synthetic Designs, Synthetic Libraries, TestBench Manager, and TimeMill are registered trademarks of Synopsys, Inc

3-D Debugging, AMPS, Behavioral Compiler, CBA Design System, CBA-Frame, characterize, Chip Architect, Compiled Designs, Core Network, Core Store, Cyclone, Data Path Express, DataPath Architect, DC Expert, DC Expert Plus, DC Professional, DelayMill, Design Advisor, Design Analyzer, Design Compiler, DesignSource, DesignTime, DesignWare Developer, Direct RTL, Direct Silicon Access, dont_touch, dont_touch_network, ECL Compiler, ECO Compiler, Embedded System Prototype, Floorplan Manager, Formality, FoundryModel, FPGA Compiler, FPGA *Express*, Frame Compiler, General Purpose Post-Processor, GPP, HDL Advisor, HDL Compiler, Integrator, Interactive Waveform Viewer, Library Compiler, LM-1400, LM-700, LM-family, Logic Model, ModelSource, ModelWare, Module Compiler, MS-3200, MS-3400, Power Compiler, PowerArc, PowerGate, PowerMill, PrimeTime, RTL Analyzer, Shadow Debugger, Silicon Architects, SimuBus, SmartCircuit, SmartModel Windows, Source-Level Design, SourceModel, SWIFT, SWIFT Interface, Synopsys Graphical Environment, Test Compiler, Test Compiler Plus, Test Manager, TestSim, Timing Annotator, Trace-On-Demand, VCS, VCSi, VHDL System Simulator, VirSim, Visualyze, Vivace, VSS Expert, and VSS Professional are trademarks of Synopsys, Inc.

Linux is a registered trademark of Linus Torvalds used by permission.

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

Printed in the U.S.A.
Document Order Number 34174-000 ZA
VCS/VCSi User Guide, Version X-2005.06

Table of Contents

1. Getting Started

What VCS Supports 1-2

Main Components of VCS 1-3

VCSi 1-5

VCS Workflow 1-6

Preparing to Run VCS 1-8

Obtaining a License 1-8

Setting Up Your Environment 1-10

Setting Up Your C Compiler 1-12

Compiling the simv Executable 1-13

Basic Compile-Time Options 1-14

Running a Simulation 1-18

Basic Runtime Options 1-18

Accessing The Discovery AMS Documentation 1-20

2. Modeling Your Design

Avoiding Race Conditions 2-2
Using and Setting a Value at the Same Time 2-2
Setting a Value Twice at the Same Time 2-3
Flip-Flop Race Condition 2-4
Continuous Assignment Evaluation 2-5
Counting Events. 2-6
Time Zero Race Conditions 2-6
Optimizing Testbenches For Debugging 2-7
Conditional Compilation 2-8
Enabling Debugging Features At Runtime. 2-10
Combining The Techniques 2-13
Avoiding the Debugging Problems from Port Coercion 2-14
Creating Models That Simulate Faster 2-15
Design Styles 2-16
Unacclerated Data Types, Primitives, and Statements 2-16
Inferring Faster Simulating Sequential Devices 2-18
Modeling Faster always Blocks 2-22
Implemented IEEE Std 1364-2001 Language Constructs 2-23
Comma Separated Event Control Expression Lists 2-24
Name-Based Parameter Passing 2-25
ANSI Style Port And Argument Lists 2-26
Initialize A Reg In Its Declaration 2-27
Conditional Compiler Directives. 2-27
Disabling Default Net Data Types 2-28

Signed Arithmetic Extensions 2-29
File I/O System Tasks 2-43
Passing Values From The Runtime Command Line 2-56
Indexed Part-Selects 2-56
Multi-Dimensional Arrays 2-58
Maintaining The File Name and Line Numbers 2-58
Implicit Event Control Expression Lists 2-59
The Power Operator 2-60
Attributes 2-61
Generated Instantiation 2-62
localparam Declarations 2-68
Constant Functions 2-69
Using the +v2k Compile-Time Option 2-70
Case Statement Behavior 2-72
Memory Size Limits in VCS 2-72
Using Sparse Memory Models 2-73
Obtaining Scope Information 2-74
Scope Format Specifications 2-75
Returning Information About The Scope 2-77
Compiler Directives 2-80
Compiler Directives for Cell Definition 2-80
Compiler Directives for Setting Defaults 2-81
Compiler Directives for Macros 2-81
Compiler Directives for Detecting Race Conditions 2-83
Compiler Directives for Delays 2-84

Compiler Directives for Backannotating SDF Delay Values . . 2-84
Compiler Directives for Source Protection 2-85
Compiler Directives for Controlling Port Coercion 2-86
General Compiler Directives 2-87
Unimplemented Compiler Directives 2-88
Unsupported Compiler Directives 2-89
System Tasks and Functions 2-90
SystemVerilog Assertions Severity 2-90
SystemVerilog Assertions Control 2-91
SystemVerilog Assertions 2-91
VCD Files 2-92
LSI Certification VCD and EVCD Files 2-94
VPD Files 2-97
SystemVerilog Assertions 2-103
Executing Operating System Commands 2-104
Log Files 2-105
Data Type Conversions 2-105
Displaying Information 2-106
File I/0 2-107
Loading Memories 2-109
Time Scale 2-110
Simulation Control 2-111
System Tasks for Timing Checks 2-111
PLA Modeling 2-114
Stochastic Analysis 2-114
Simulation Time 2-115

Probabilistic Distribution. 2-115
Resetting VCS 2-116
General System Tasks and Functions 2-116
IEEE Standard System Tasks Not Yet Implemented in VCS . 2-118
Nonstandard System Tasks Not Supported in VCS. 2-119
Verilog-XL System Tasks Not Supported in VCS. 2-119
Avoiding Circular Dependency 2-120
Designing with $lsi_dumpports for Simulation and Test 2-121
Dealing with Unassigned Nets. 2-122
Code values at time 0 2-123
Cross Module Forces and No Instance Instantiation 2-123
Signal Value/Strength Codes 2-125
Generating SAIF Files 2-126

3. Compiling Your Design

Incremental Compilation 3-2
Triggering Recompilation 3-3
Using Shared Incremental Compilation 3-4
The Direct Access Interface Directory. 3-6
64-32-Bit Cross-Compilation and Full 64-Bit Compilation 3-7
Identifying the Source of Memory Consumption 3-8
Minimizing Memory Consumption 3-9
Running a 64-32-Bit Cross-Compilation 3-10
Running a 64-Bit Compilation and Simulation 3-12
Optimizations To Reduce Memory Consumption 3-13

Initializing Memories and Regs 3-13
Allowing Inout Port Connection Width Mismatches 3-15
Using Lint 3-16
Changing Parameter Values From The Command Line 3-18
Making Accessing an Out of Range Bit an Error Condition 3-19
Compile-Time Options 3-21
Accessing Verilog Libraries 3-21
Incremental Compilation 3-23
Help and Documentation 3-26
SystemVerilog 3-26
OpenVera Native TestBench 3-28
Different Versions of Verilog 3-30
Initializing Memories and Regs 3-31
Using Radiant Technology 3-31
64 bit Compilation 3-32
Two State Simulation 3-32
Debugging 3-33
Finding Race Conditions 3-37
Starting Simulation Right After Compilation 3-38
Compiling OpenVera Assertions (OVA) 3-38
Compiling For Simulation With Vera 3-40
Compiling For Coverage Metrics 3-40
Discovery Visual Environment and UCLI 3-46
DVE 3-47
VirSim 3-47

Converting VCD and VPD Files 3-51
Specifying Delays 3-51
Compiling an SDF File 3-54
Profiling Your Design 3-56
File Containing Source Filenames and Options 3-57
Compiling Runtime Options into the simv Executable 3-58
Pulse Filtering 3-59
PLI Applications 3-60
Enabling and Disabling in Specify Blocks and Timing Checks 3-61
The VCS DirectC Interface 3-62
Negative Timing Checks 3-63
Flushing Certain Output Text File Buffers 3-64
Simulating SWIFT VMC Models and SmartModels 3-65
Controlling Messages 3-65
Cell Definition 3-67
Licensing 3-68
Controlling the Assembler 3-69
Controlling the Linker 3-70
Controlling the C Compiler 3-71
Source Protection 3-73
Mixed Analog/Digital Simulation 3-74
Changing Parameter Values 3-75
Specify the Time Scale 3-75
General Options 3-76
Compiling Runtime Options Into The simv Executable 3-81

Performance Considerations 3-83
Use Local Disk 3-83
Managing Temporary Disk Space on UNIX 3-84
Compile-Time Options that Impede or Accelerate VCS 3-85
Compiling for Debugging or Performance 3-88

4. Simulating Your Design

Runtime Options 4-2
Running DVE in Interactive Mode 4-2
Simulating OpenVera Testbenches 4-2
Simulating OpenVera Assertions 4-4
SystemVerilog Assertions 4-7
CLI Command File 4-10
UCLI 4-10
Specifying VERA Object Files 4-10
Coverage Metrics 4-11
Enabling and Disabling Specify Blocks 4-12
Specify When Simulation Stops 4-13
Recording Output 4-14
Controlling Messages 4-14
Discovery Visual Environment and UCLI 4-15
VPD Files 4-16
Controlling $gr_waves System Task Operations 4-18
VCD Files 4-19
Specifying Min:Typ:Max Delays 4-20
Flushing Certain Output Text File Buffers 4-21

Licensing 4-22
General Options. 4-23
Save and Restart. 4-25
Save and Restart Example 4-26
Save and Restart File I/O. 4-27
Save and Restart with Runtime Options 4-28
Restarting at The CLI Prompt 4-29
Specifying A Very Long Time Before Stopping Simulation 4-30
Passing Values From The Runtime Command Line 4-31
Performance Considerations. 4-32
Profiling the Simulation 4-33
The Top Level View 4-34
The Module View 4-35
The Instance View 4-36
The Module to Construct Mapping View 4-37
The Top Level Construct View 4-40
The Construct View Across Design 4-41
How VCS Prevents Time 0 Race Conditions 4-42
Protected and Portable Verilog Model 4-43

5. Debugging and Race Detection

Non-Graphical Debugging With The CLI. 5-2
Interactive Command Language 5-3
Command Files 5-7

Key Files 5-9
The Dynamic Race Detection Tool 5-10
Enabling Race Detection 5-12
Specifying The Maximum Size of Signals in Race Conditions 5-12
The Race Detection Report 5-13
Post Processing The Report 5-16
Debugging Simulation Mismatches 5-18
The Static Race Detection Tool 5-22

6. Using Radiant Technology

Compiling For Radiant Technology 6-1
Known Limitations 6-2
Potential Differences in Coverage Metrics 6-3
Compilation Performance with Radiant Technology 6-3
Applying Radiant Technology to Parts of The Design 6-3
The Configuration File Syntax. 6-4
Configuration File Statement Examples. 6-7

7. Using the PLI

The Functions in a PLI Application 7-5
Header Files for PLI Applications 7-6
The PLI Table File 7-7
The PLI Specifications 7-9
Specifying ACC Capabilities 7-12

Specifying the PLI Table File 7-20
PLI Object Compatibility with Verilog-XL 7-21
Globally Enabling ACC Capabilities 7-21
Enabling ACC Write Capabilities Using The Configuration File . 7-22
Using Only The ACC Capabilities That You Need. 7-25
Learning What ACC Capabilities Are Used 7-25
Compiling to Enable Only The ACC Capabilities You Need . 7-27
Limitations 7-28
Using VPI Routines 7-29
Support for the vpi_register_systf Routine. 7-31
PLI Table File For VPI Routines. 7-32
Integrating a VPI Application With VCS. 7-32
Writing Your Own main() Routine 7-34
Reading and Writing to Memories 7-35
acc_setmem_int. 7-38
acc_getmem_int 7-39
acc_clearmem_int 7-40
acc_setmem_hexstr. 7-45
acc_getmem_hexstr 7-49
acc_setmem_bitstr. 7-50
acc_getmem_bitstr. 7-51
acc_handle_mem_by_fullname 7-52
acc_readmem 7-53
acc_getmem_range 7-55

acc_getmem_size 7-56
acc_getmem_word_int 7-57
acc_getmem_word_range 7-58
Multi-Dimensional Arrays 7-59
tf_mdanodeinfo and tf_imdanodeinfo 7-60
acc_get_mda_range 7-62
acc_get_mda_word_range() 7-63
acc_getmda_bitstr() 7-65
acc_setmda_bitstr() 7-66
Probabilistic Distribution 7-67
vcs_random 7-68
vcs_random_const_seed 7-68
vcs_random_seed 7-69
vcs_dist_uniform 7-69
vcs_dist_normal 7-70
vcs_dist_exponential 7-71
vcs_dist_poisson 7-71
Returning a String Pointer to a Parameter Value 7-72
acc_fetch_paramval_str 7-72
Extended VCD Files 7-73
acc_lsi_dumpports_all 7-73
acc_lsi_dumpports_call 7-74
acc_lsi_dumpports_close 7-76
acc_lsi_dumpports_flush 7-77
acc_lsi_dumpports_limit 7-78

acc_lsi_dumpports_misc 7-79
acc_lsi_dumpports_off 7-80
acc_lsi_dumpports_on 7-81
acc_lsi_dumpports_setformat 7-83
acc_lsi_dumpports_vhdl_enable 7-84
Line Callbacks 7-85
acc_mod_lcb_add 7-85
acc_mod_lcb_del 7-87
acc_mod_lcb_enabled 7-88
acc_mod_lcb_fetch 7-89
acc_mod_lcb_fetch2 7-90
acc_mod_sfi_fetch 7-92
Source Protection 7-94
vcsSpClose 7-96
vcsSpEncodeOff 7-96
vcsSpEncodeOn 7-97
vcsSpEncoding 7-99
vcsSpGetFilePtr 7-100
vcsSpInitialize 7-101
vcsSpOvaDecodeLine 7-102
vcsSpOvaDisable 7-103
vcsSpOvaEnable 7-104
vcsSpSetDisplayMsgFlag 7-105
vcsSpSetFilePtr 7-106
vcsSpSetLibLicenseCode 7-107
vcsSpSetPliProtectionFlag 7-108

vcsSpWriteChar 7-109
vcsSpWriteString 7-110
Access Routine for Signal in a Generate Block. 7-112
acc_object_of_type 7-112
VCS API Routines 7-112
VcsInit() 7-112
VcsSimUntil() 7-113

8. Delays, Timing and SDF Files

Transport and Inertial Delays 8-2
Different Inertial Delay Implementations 8-4
Enabling Transport Delays. 8-7
Pulse Control. 8-7
Pulse Control with Transport Delays 8-9
Pulse Control with Inertial Delays 8-12
Specifying Pulse on Event or Pulse on Detect Behavior 8-16
Specifying The Delay Mode. 8-21
Using SDF Files 8-23
The $sdf_annotate System Task 8-24
Compiling The ASCII SDF File at Compile-Time 8-26
Precompiling An SDF File 8-29
Reading The ASCII SDF File During Runtime. 8-31
Replacing Negative Module Path Delays in SDF Files 8-35
Using The Shorter Delay in IOPATH Entries 8-36
Disabling CELLTYPE Checking in SDF Files 8-37

The SDF Configuration File 8-38
INTERCONNECT Delays 8-50
Multisource INTERCONNECT Delays 8-51
Single Source INTERCONNECT Delays 8-55
Min:Typ:Max Delays 8-55
Specifying Min:Typ:Max Delays at Runtime 8-57
Using The Configuration File To Disable Timing 8-58
Using The Timopt Timing Optimizer 8-59
Editing The timopt.cfg File 8-61

9. Negative Timing Checks

The Need For Negative Value Timing Checks 9-2
Negative Timing Checks for Asynchronous Controls 9-7
The $setuphold Timing Check Extended Syntax 9-8
The $recrem Timing Check Syntax 9-10
Enabling Negative Timing Checks 9-12
Other Timing Checks Use The Delayed Signals 9-14
Checking Conditions 9-18
Toggling The Notifier Register 9-19
SDF Backannotating to Negative Timing Checks 9-19
How VCS Calculates Delays 9-20
Using Multiple Non-Overlapping Violation Windows 9-22

10. Using Synopsys Models

SWIFT VMC Models and SmartModels Introduction 10-2
SWIFT Environment Variables 10-3
Generate Verilog Templates 10-4
Monitoring Signals in the Model Window 10-8
Using LMTV SmartModel Window Commands 10-10
Entering Commands Using The SWIFT Command Channel. 10-13
Loading Memories at The Start of Runtime 10-15
Compiling and Simulating a Model 10-15

Synopsys Hardware Models Introduction 10-17
Required Environment Variables 10-17
Using lmvc_template 10-18
Required VCS Command Line Options 10-19

11. Discovery Visual Environment

Primary DVE Components 11-2
Top Level Window 11-2
Source Window 11-6
Assertion Window 11-7
Wave Window 11-8
List Window 11-11
Schematic Window 11-12

12. VCD and VPD File Utilities

The vcdpost Utility 12-2
Scalarizing The Vector Signals 12-2

Uniquifying The Identifier Codes 12-3
The vcdpost Utility Syntax 12-4
The vcdiff Utility 12-5
Preparing to Run vcdiff 12-5
The vcdiff Utility Syntax 12-6
The vcat Utility 12-13
Generating Source Files From VCD Files 12-17
The vcsplit Utility 12-23

13. Two State Simulation

Value And Strength Mapping For Two State Simulation 13-3
Signals That Retain Four State Simulation 13-4
Data Types That Need Four State Simulation 13-5
Expressions That Require Four State Simulation 13-5
Signals in Case Expressions in Case Statements 13-7
Primitives With A Different Drive Strength 13-7
Signals in The Path For a Z Value 13-11
Undriven Signals Connected to Four State Signals 13-14
Ports That Connect To Four State Signals 13-15
Forced Signals 13-16
Continuously Assigned Signals In If Statement Expressions . 13-16
Parameters Retain Four State Simulation 13-16
Differences in Initialization 13-17
Initial Value of Connected Nets 13-18
Resolving Multiple Drivers 13-19

Changes In Simulation Behavior 13-21
Applying Different Stimulus 13-21
Different Initialization Causing Different Simulation Behavior 13-22
Missing Rising and Falling Edges 13-22
X and Z Values in Expressions 13-23
Tri-State Logic Gates 13-24
Some Strength Specifications Are Ignored 13-24
User-Defined Primitives Output Different Values 13-25
Infinite Value Expressed Differently 13-26
PLI Compatibility 13-27
Specifying Two State Simulation 13-27
Specifying Four State Simulation for Parts of Your Design 13-29
Using Metacomments 13-29
Using The Configuration File 13-30

14. Using OpenVera Assertions

Introducing OVA 14-2
Built-in Test Facilities and Functions 14-2
Using OVA Directives. 14-3
OVA Flow. 14-7
Checking OVA Code with the Linter Option. 14-7
Applying General Rules with VCS 14-8
Applying Magellan Rules for Formal Verification 14-16
Compiling Temporal Assertions Files 14-19

OVA Runtime Options 14-21
Functional Code Coverage Options 14-23
OpenVera Assertions Post-Processing 14-23
Overview 14-24
OVAPP Flow 14-24
Building and Running a Post-Processor 14-25
OVA Post-Processing CLI Commands 14-30
Using Multiple Post-Processing Sessions 14-31
Multiple OVA Post-Processing Sessions in One Directory . . 14-32
Viewing Output Results 14-40
Viewing Results in a Report File 14-40
Viewing Results with Functional Coverage 14-41
Viewing Results with VirSim 14-47
Using OVA with Third Party Simulators 14-50
Inlining OVA in Verilog 14-51
Specifying Pragmas in Verilog 14-52
Methods for Inlining OVA 14-52
Case Checking 14-61
General Inlined OVA Coding Guidelines 14-64
Using Verilog Parameters in OVA Bind Statements 14-65
Introduction 14-65
Use Model 14-66
Post-processing Flow 14-70
OVA System Tasks and Functions 14-71
Setting and Retrieving Category and Severity Attributes 14-72

Starting and Stopping the Monitoring of Assertions 14-73
Controlling the Response to an Assertion Failure 14-77
Display Custom Message for an Assertion Failure 14-78
Task Invocation from the CLI 14-79
Debug Control Tasks 14-80
Calls from within Code 14-81

15. Using SystemVerilog

SystemVerilog Data Types 15-3
Variable Data Types for Storing Integers 15-3
User Defined Data Types 15-4
Enumerations 15-5
Structures and Unions 15-8
SystemVerilog Arrays 15-12
Writing to Variables 15-16
Force and Release on SystemVerilog Variables 15-17
SystemVerilog Operators 15-28
New Procedural Statements 15-29
unique and priority if and case Statements 15-29
do while Statement 15-32
SystemVerilog Processes 15-32
The always_comb Block 15-33
The always_latch Block 15-35
The always_ff Block 15-36
Tasks and Functions 15-36

Tasks 15-36
Functions 15-38
SystemVerilog Assertions 15-41
Immediate Assertions 15-42
Concurrent Assertions Overview 15-43
Sequences 15-43
Properties 15-57
assert Statements 15-63
cover Statements 15-64
Action Blocks 15-67
Binding an SVA Module to a Design Module 15-68
The VPI for SVA 15-71
System Verilog Assertion Local Variable Debugging 15-72
Hierarchy 15-75
The `$root` Top-Level Global Declaration Space 15-75
New Data Types for Ports 15-77
Instantiation Using Implicit .name Connections 15-79
Instantiation Using Implicit .* Connections 15-79
New Port Connection Rules for Variables 15-80
Interfaces 15-81
Using Modports 15-85
Functions in Interfaces 15-87
Enabling SystemVerilog 15-88
Disabling Unique and Priority Warning Messages 15-88
Controlling How VCS Uses SystemVerilog Assertions 15-90

Compile-Time and Runtime Options for SystemVerilog Assertions. 15-90
Ending Simulation at a Specified Number of Assertion Failures. 15-94
Entering System Verilog Assertions as Pragmas. 15-95
Using SystemVerilog Assertions in an MX Design. 15-96
Options for SystemVerilog Assertion Coverage. 15-96
Reporting on SystemVerilog and OpenVera Assertions Coverage 15-98
The assertCovReport Report Files 15-109
Assertion Monitoring System Tasks. 15-116
Assertion System Functions 15-120
Using Assertion Categories 15-121

16. Using the VCS/SystemC Cosimulation Interface

Verilog Designs Containing SystemC Modules 16-3
Input Files Required. 16-4
Supported Port Data Types 16-5
Generating The Wrapper for SystemC Modules 16-6
Instantiating The Wrapper and Coding Style 16-9
Controlling Time Scale and Resolution in a SystemC Module 16-10
Compiling a Verilog Design Containing SystemC Modules . . 16-11
SystemC Designs Containing Verilog Modules 16-12
Input Files Required. 16-13
Generating The Wrapper for a Verilog Module 16-14
Instantiating The Wrapper 16-16

Compiling a SystemC Design Containing Verilog Modules . . 16-17
Elaborating The Design 16-19
Specifying Runtime Options. 16-20
Using a Port Mapping File 16-21
Using a Data Type Mapping File 16-22
Debugging the SystemC Code 16-24
Debugging The Verilog Code 16-25
Debugging Both the Verilog and SystemC Portions of a Design. 16-25
Using the Built-in SystemC Simulator 16-27
Using a Customized SystemC Installation. 16-28

Appendix A. VCS for the Verilog XL User

Race Debugging Differences. A-2
Support Tools. A-2
Evaluation of Expressions in Verilog-XL A-3
Initial vs. always Constructs. A-3
Incompatibilities with Verilog-XL A-4
System Tasks Not Implemented A-5
PLI A-5
CLI A-6
Tasks A-6
Port Coercion and Port Collapsing. A-7
TF Routine Compatibility A-7
Module Path Delays on Bidirectional Nets. A-8

Library Searching A-8
Event Control with Memories A-8
Disable Statements A-8
Case Expressions A-9
Assignment to Parameters A-10
Vectored Nets A-10
$dist_uniform A-10
Use of Conditioners in Timing Checks A-10
Module Input Port Delay (MIPD) A-11
SDF INTERCONNECT Delays A-11
Getting the Most Out of Compiled Simulation A-14
Working with Multiple Test Fixtures or Test Stimuli A-14
Alternate Interactive Commands A-17
Gate-level Timing A-17

Appendix B. Source Protection

Encrypting Source and SDF Files B-3
Encrypting Specified Regions B-4
Encrypting The Entire Source Description B-6
Encrypting SDF Files B-9
Specifying Encrypted Filename Extensions B-10
Specifying Encrypted File Locations B-11
Multiple Runs and Error Handling B-11
Permitting CLI/PLI Access to Encrypted Modules B-12
Simulating Encrypted Models B-12
Using the CLI B-13

Using System Tasks. B-14
Writing PLI Applications B-14
Mangling Source Files B-15
Creating A Test Case B-23
Preventing Mangling of Top-Level Modules B-25
Index IN-1

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 includes a "smart verification" solution that offers assertion-based verification and advanced coverage technology — all integrated within an industry-proven HDL simulation platform.

As the foundation for Synopsys' complete functional verification solution, VCS supports Verilog, VHDL, mixed-HDL and mixed-signal simulation for complex SoC designs.

This chapter covers the following topics:

- What VCS Supports
- Main Components of VCS

- VCS Workflow
- Preparing to Run VCS
- Compiling the simv Executable
- Running a Simulation

What VCS Supports

VCS provides a fully featured implementation of the Verilog language as defined in the *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995) and the *Standard Verilog Hardware Description Languag*e (IEEE Std 1364-2001).

VCS supports most of the design and assertion constructs in SystemVerilog. Not yet supported constructs are grayed out in the Synopsys_modified_SystemVerilog3.1aLRM.pdf file in the $VCS_HOME/doc/UserGuide directory.

VCS supports interfaces to a variety of other simulators and models, including (but not limited to) user PLI applications conforming to IEEE Std 1363-1995, delay calculators, SDF delay annotation, Synopsys Logic Modeling SmartModels®, and the ModelSource and LM-family hardware modelers.

Main Components of VCS

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

- SystemVerilog — an extensiob of the Verilog language that adds new design and assertion constructs
- OpenVera Assertions (OVA) — provides an easy and concise way to describe sequences of events and facilities to test for their occurrence.VCS natively compiles OVA. For details on OVA, see the chapter on using OVA and the language reference the *OpenVera Language Reference Manual: Assertions* volume. Many implemented SystemVerilog assertions constructs are functionally comparable to OpenVera assertion constructs.
- NTB — a test bench language that is a subset of the OpenVera testbench language. VCS can natively compile test bench files written in OpenVera testbench constucts into the simv executable file, along with Verilog source files and OpenVera Assertions (OVA) files. For details contact vcs_support@synopsys.com.
- Discovery Visualization Environment (DVE) is the new debugging environment. It is in the process of replacing VirSim.
- VirSim Graphical Debugging Environment — enables you to control an interactive simulation or to analyze saved results of simulation. You can use VirSim to trace signals of interest while viewing annotated values in the source code or schematic diagrams. You can also compare waveforms, extract specific signal information, and generate HDL test benches based on waveform outputs. For details, see the *VirSim User Guide*.

- Built-In Coverage Metrics — a comprehensive built-in coverage analysis functionality that includes condition, toggle, line, finite-state-machine, path and branch coverage. You can use Coverage Metrics to determine the quality or coverage of your verification test and focus on creating additional test cases. You only need to compile once to run both simulation and coverage analysis. For details, see the *VCS Coverage Metrics User Guide*.

- DirectC Interface — significantly improves ease-of-use and performance over existing PLI-based methods by allowing you to directly embed user-created C/C++ functions within your Verilog design description. VCS atomically recognizes C/C++ function calls and integrates them for simulation, thus eliminating the need to manually create PLI files. DirectC also eliminates the debugging and by bypassing PLI overhead, increases performance for simulating C/C++ code with Verilog. For details, see the *VCS DirectC User Guide*.

- VCS MX for Mixed HDL designs — allows performance optimizations to be applied to both the Verilog and VHDL parts of a design and enables you to extend your existing Verilog and VHDL flows.

- Mixed Signal Simulation — Synopsys provides NanoSim and VCS users who need to do mixed signal simulation with the *Discovery AMS: NanoSim-VCS User Guide* and the *Discovery AMS: Enhanced NanoSim-VCS User Guide*. See “Accessing The Discovery AMS Documentation” on page 1-20.

- Incremental Compilation — shortens the turnaround time from design modification by minimizing the amount of recompilation. This capability enables VCS to automatically compare the current design against the previously compiled database; it then recompiles only those portions of the design that have changed.

- 64-Bit Cross-Compilation and Full 64-Bit Compilation — VCS offers a choice of methodologies for high-capacity compilation and simulation. Its -comp64 option invokes a cross-compilation process that compiles a design on a 64-bit machine, which can then be simulated on a 32-bit machine. The -full64 option both compiles and simulates a design on a 64-bit machine.

VCSi

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

VCSi implementation requirements are summarized in the following:

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

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

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

VCS Workflow

The process of using VCS to simulate a Verilog model consists of two basic steps:

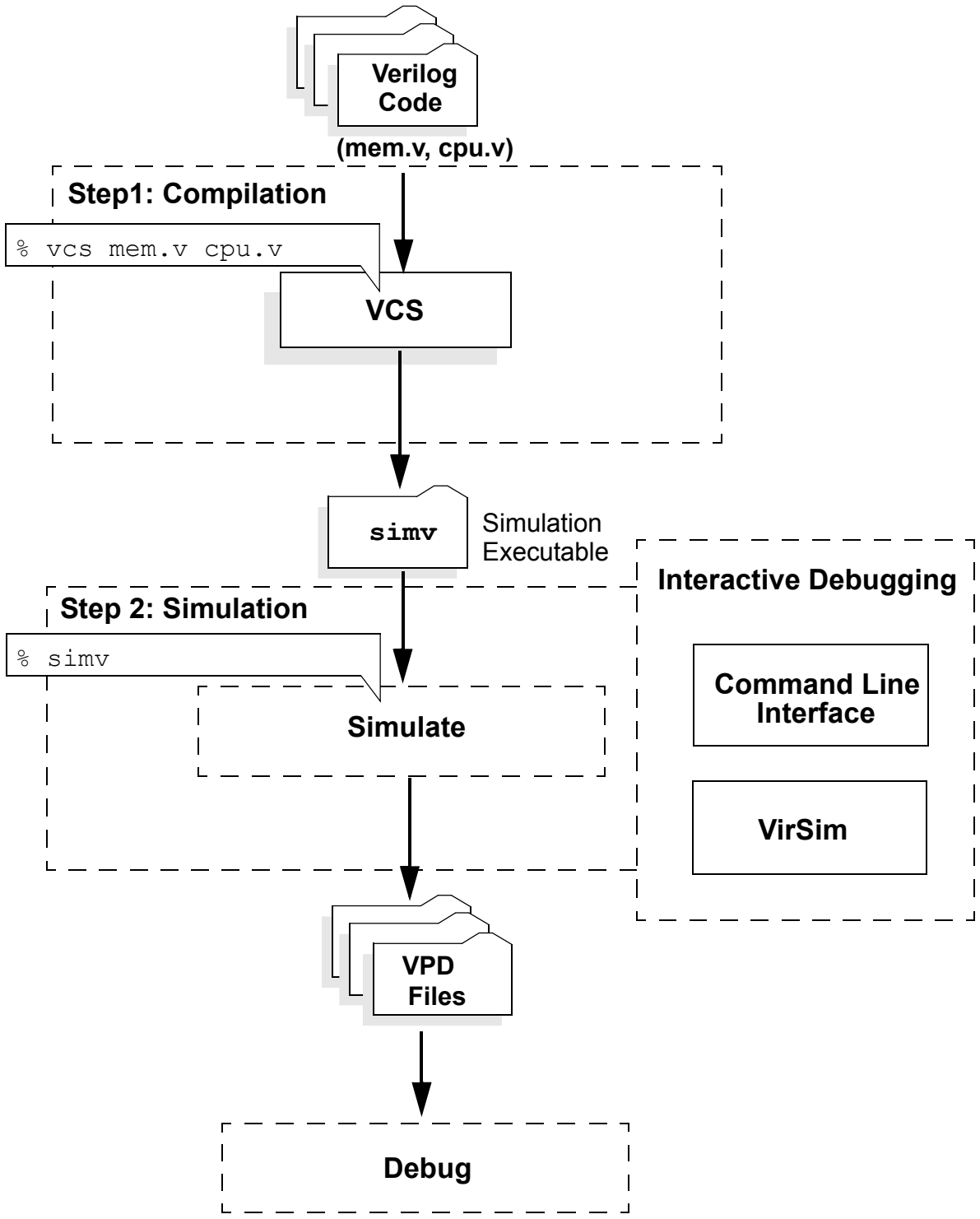
1. Compiling your source files into an executable binary file.
2. Running the executable binary file.

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

On Linux, Solaris, and HP platforms, you can use VCS to generate object code directly without generating C or assembly language files. Incremental compilation allow you to avoid compiling Verilog modules that have not changed since the last time you compiled them.

Figure 1-1 illustrates the VCS workflow.

Figure 1-1 Basic VCS Compilation and Simulation Flow



Preparing to Run VCS

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

- Obtaining a License
- Setting Up Your Environment
- Setting Up Your C Compiler

Obtaining a License

You must have a license before running VCS. To obtain a license contact your local Synopsys Sales Representative. Your Sales Representative will need the hostid for your machine. You may start this new license as follows:

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 the SNPSLMD_LICENSE_FILE environment variable to point to the license file, for example:

   ```
   setenv LM_LICENSE_FILE /u/edatools/vcs6.0/license.dat
   ```

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

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

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

   OR

   ```
   setenv LM_LICENSE_FILE /u/edatools/oldvcs/\
   viewlmgrd_license.dat:/u/edatools/vcs7.0/\
   snpslmd_license.dat
   ```

Setting Up Your Environment

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

- $VCS_HOME environment variable

  When you or someone at your site installed VCS, the installation created a directory that we call the *vcs_install_dir* directory. Define this environment variable to the pathname of the *vcs_install_dir* directory, for example:

  ```
  setenv VCS_HOME /u/net/eda_tools/vcs4.2
  ```

- PATH environment variable

  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 environment variable

  The definition can either be an absolute path name to a license file or to a port on the license server. Separate the arguments in this definition with colons in UNIX, for example:

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

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 you specified when you set the VCS_HOME environment variable. Specify a value other than 0 to enable the display, for example:
`setenv DISPLAY_VCS_HOME 1`

SYSTEMC
: Specifies the location of the SystemC simulator used with the VCS/SystemC cosimulation interface. See Chapter 16, “Using the VCS/SystemC Cosimulation Interface.”

TMPDIR
: The directory used by VCS and the C compiler to store temporary files during compilation.

VCS_CC
: Indicate the C compiler to be used. To use the gcc compiler specify:
`setenv VCS_CC gcc`.

VCS_COM
: Path to the VCS compiler executable named vcs1, not the compile script. If you receive a patch for VCS you might need to set this environment variable to specify the patch.
This variable is for solving problems that require patches from VCS and should not be set by default.

VCS_LOG
: Specifies the run-time log filename and location.

VCS_RUNTIME

Specifies which runtime library named libvcs.a VCS uses. This variable is for solving problems that require patches from VCS and should not be set by default.

VCS_SWIFT_NOTES

Enables the printf PCL command.
PCL is the Processor Control Language that works with SWIFT microprocessor models.
Set this environment variable's value to 1.

Setting Up Your C Compiler

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

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

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

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

Compiling the simv Executable

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

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

where:

source_files
: The Verilog, OpenVera assertions, or OpenVera testbench source files for your design separated by spaces.

source_or_object_files
: Optional C files (.c), object files (.o), or archived libraries (.a). These are DirectC or PLI applications that you want VCS to link into the binary executable file along with the object files from your Verilog source files.

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

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

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

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

Basic Compile-Time Options

This section outlines some of the basic compile-time options you can use to control how VCS compiles your Verilog source files. Detailed descriptions of all compile-time options are described in Chapter 3, "Compiling Your Design."

`+cli+[`*`module_identifier`*`=]`*`level_number`*

Enables command line interface (CLI) interactive debugging commands.

Using this option creates the Direct Access Interface Directory, by default named simv.daidir, in the directory where VCS creates the executable file. See “The Direct Access Interface Directory” on page 3-6.

The level number enables more and more commands. The level number can be any number between 1 and 4:

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

Specifies compiling for the specified type or types of coverage. The arguments specifies the types of coverage:

`line`
Compile for line or statement coverage.

`cond`
Compile for condition coverage.

`fsm`
Compile for FSM coverage.

`tgl`
Compile for toggle coverage.

`path`
Compile for path coverage.

`branch`
Compile for branch coverage

`assert`
Compile for SystemVerilog assertion coverage.

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

`-f` *`filename`*
Specifies a filename that contains a list of absolute pathnames for Verilog source files and compile-time options.

`+incdir+`*`directory`*`+`
Specifies the directory or directories that VCS searches for include files used in the `` `include `` compiler directive. More than one directory may be specified, separated by +.

`-I`
Compiles for interactive use.

`-line`
Enables source-level debugging tasks such as stepping through the code, displaying the order in which VCS executed lines in your code, and the last statement executed before simulation stopped. Typically you enter this option with a `+cli` option, for example

```
vcs +cli+1 -line
```

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

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

`+notimingcheck`
: 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.

`-ntb`
: Enables compilation of a OpenVera testbench file.

`-ova_file` *`filename`*
: Identifies an OVA file as input (this option is not required if the OVA file name contains a .ova extension.

`-P` *`pli.tab`*
: Compiles a user-defined PLI definition table file.

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

`-R`
: Runs the executable file immediately after VCS links together the executable file.

`-RI`
: Compiles for interactive use and starts VirSim.

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

`+v2k`
: Enables new language features in the proposed IEEE 1364-2001 standard. See “Implemented IEEE Std 1364-2001 Language Constructs” on page 2-23.

`-v` *`filename`*
: Specifies a Verilog library file. VCS looks in this file for module and UDP definitions for the module and UDP instances that VCS found in your source code when it did not find the corresponding module or UDP definitions in your source code.

`+vc`
: Enables extern declarations of C/C++ functions and calling these functions in your source code.

`-vera`
: Specifies the standard VERA PLI table file and object library.

`-y` *`directory`*
: Specifies a Verilog library directory. VCS looks in the source files in this directory for module and UDP definitions for the module and UDP instances that VCS found in your source code when it did not find the corresponding module or UDP definitions in your source code.

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

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

Running a Simulation

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

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

executable_file options

where:

executable_file
: The executable file that is output from the vcs command that compiles your source code and links your design with VCS to form the executable.

options
: Runtime options that specify how to simulate your design. Some of the basic runtime options are described in the following section, "Basic Runtime Options."

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

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

Basic Runtime Options

This section outlines some of the basic runtime options you can use to control how VCS compiles your Verilog source files. Detailed descriptions of all runtime options are described in Chapter 4, "Simulating Your Design."

`-cm line|cond|fsm|tgl|path|branch|assert`
: Specifies monitoring for the specified type or types of coverage. The arguments specifies the types of coverage:

`line`
Monitor for line or statement coverage.

`cond`
Monitor for condition coverage.

`fsm`
Monitor for FSM coverage.

`tgl`
Monitor for toggle coverage.

`path`
Monitor for path coverage.

`branch`
Monitor for branch coverage

`assert`
Monitor for SystemVerilog assertion coverage.

`-l` *`filename`*
All output of simulation is written to filename as well as to the standard output.

`+notimingcheck`
Disables timing check system tasks in your design. Using this option at runtime can improve the simulation performance of your design, depending on the number of timing checks that this option disables.

`-ova_cov`
Enables functional coverage reporting.

`-ova_report`
Generates an OVA report file in addition to printing results on screen.

`+vcs+learn+pli`

ACC capabilities enable debugging operations but they have a performance cost so you only want to enable them where you need them. This option keeps track of where you use them for debugging operations so that you can recompile your design and in the next simulation enable them only where you need them. When you use this option VCS writes the pli_learn.tab secondary PLI table file. You input this file when you recompile your design with the `+applylearn` compile-time option. See “Globally Enabling ACC Capabilities” on page 7-21.

Accessing The Discovery AMS Documentation

Read the Discovery AMS documentation for mixed signal simulation with VCS and NanoSim.

There are two ways to access the Discovery AMS documentation:\

- Using Synopsys Documentation on the Web
- Using the PDF files in the NanoSim installation

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

1. Go to www.synopsys.com.
2. Click on SOLVNET
3. Click on Documentation on the Web
4. Click the Go button next to Browse.
5. Click on NanoSim

6. Click on the document titles:

   Discovery AMS NanoSim-VCS User Guide or *Discovery AMS Enhanced NanoSim-VCS User Guide.*

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

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

   The Discovery AMS NanoSim-VCS User Guide also has information for the VCS MX user.

To access the PDF files in the NanoSim installation:

1. Change directories to the *NanoSim_installation_directory*/doc/ns/manuals directory.

2. Load either of the following files into the Acrobat reader:

   ensvcs.pdf for the Discovery AMS: Enhanced NanoSim-VCS User Guide.

   ns_vcs.pdf for the Discovery AMS: NanoSim-VCS User Guide

The Discovery AMS NanoSim-VCS User Guide also has information for the VCS MX user.

2

Modeling Your Design

This chapter begins with the following Verilog modeling techniques:

- Avoiding Race Conditions
- Optimizing Testbenches For Debugging
- Avoiding the Debugging Problems from Port Coercion
- Creating Models That Simulate Faster

This chapter then describes:

- Implemented IEEE Std 1364-2001 Language Constructs
- Case Statement Behavior
- Memory Size Limits in VCS
- Using Sparse Memory Models

This chapter then describes:

- Obtaining Scope Information
- Compiler Directives
- System Tasks and Functions
- Avoiding Circular Dependency
- Designing with $lsi_dumpports for Simulation and Test
- Generating SAIF Files

Avoiding Race Conditions

In Verilog, you can easily write models that have race conditions. A race condition is defined as a coding style for which there is more than one correct result. Race conditions are easy to code in Verilog, as seen in the text *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.

Some common race conditions are described in the following sections.

Using and Setting a Value at the Same Time

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

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

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

Flip-Flop Race Condition

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

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

The solution for this case is straightforward. Use the nonblocking assignment in the flip-flop to guarantee the ordering of assignment to the output 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 with the nonblocking form:

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

Beware that the following recoding does not resolve the race condition:

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

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

Continuous Assignment Evaluation

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

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

In VCS, this will display `0`, while in Verilog-XL, it will display `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 is the following:

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

The modification of `state` may propagate to `state0` before the `if` statement, causing unexpected behavior. You can avoid this by using the nonblocking assignment to `state` in the procedural code, to guarantee that `state` is not updated until the end of the time step, after the `if` statement has executed:

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

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, `count` may be 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 fix to this race is to make the code inside the `always` block insensitive to being called once or twice.

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 is occurring at time zero.

Fix this race condition by guaranteeing no transitions at time zero of any signals inside event triggers. Rewrite the clock driver in the above example to the following:

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

Optimizing Testbenches For Debugging

The principal purpose of a testbench is, of course, to apply stimulus and typically also to compare simulation results with expected results and to take some action when the actual and expected results differ, for example, halting or ending simulation so that you can try to identify the problem or at least not waste simulation time after encountering the problem.

Testbenches also typically execute debugging features, for example displaying text in certain situations as specified with the `$monitor` or `$display` system tasks. Another example of debugging features that are typically enabled in test benches are writing simulation history files during simulation so that you can view the results after the simulation. These simulation history files record the simulation times of value changes, among other things, of the signals in your design. These simulation history files can be either ASCII VCD files that you can input into a number of third party viewers or binary VPD files that you can input into DVE and VirSim. 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 or VirSim where DVE or VirSim will translate the VCD file to a VPD file and then display the results from the new VPD file.)

Now, as you might expect, debugging features significantly slow down the simulation performance of any logic simulator including VCS. This is particularly true for operations that have VCS display text on the screen and even more so for operations that have VCS write information to a file. For this reason, as stated a number of places throughout this user's guide, you'll want to be selective about where in your design and where in the development cycle of your design you enable debugging features. This section describes a number of techniques that you can use to be selective about enabling debugging features.

Conditional Compilation

Use `` `ifdef ``, `` `else ``, and `` `endif `` compiler directives in your test bench to specify compiling certain system tasks for debugging features when the `+define` compile-time option is on the command line (or when the `` `define `` compiler directive appears in the source code), for example:

```
initial
begin
`ifdef postprocess
$vcdpluson(0,design_1);
$vcdplustraceon(design_1);
$vcdplusdeltacycleon;
$vcdplusglitchon;
`endif
end
```

The vcs command line then includes the following:

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

The system tasks in this initial block are for recording several types of information in a VPD file for post-processing the design using DVE or VirSim. In this particular case this information is for all the signals in the design so the performance cost will be extensive. You would only want to do this sort of thing early in the development cycle of the design where you need to do plenty of debugging and finding bugs is more important than simulation speed.

The command line includes the `+define+postprocess` compile-time option telling VCS to compile the design with these system tasks compiled in the test bench.

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 test bench. Doing so will enable VCS to simulate your design much faster.

Advantages and Disadvantages

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

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

We recommend that you consider this technique as a way to prevent these system tasks from inadvertently remaining compiled into the test bench 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, for example:

```
initial
if ($test$plusargs("postprocess"))
begin
$vcdpluson(0,design_1);
$vcdplustraceon(design_1);
$vcsplusdeltacycleon;
$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 test bench and then enable the execution of the system tasks with the argument to the `$test$plusargs` system function as a runtime option, so for this example the simv command line is as follows:

```
simv +postprocess
```

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

There is a pitfall to this technique. This system function will match any +plusarg that has the function's argument as a prefix. For example, in the following code 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 test bench in order to stop VCS from writing this 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 into the test bench, or any system tasks that write to a file, requires VCS to compile the simv executable so that it is possible for it to write the 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 test bench, even when you don’t include the runtime option on the simv command line.

Using the `$test$plusargs` system function forces VCS to consider the worst case scenario — +plusargs will be used at runtime — and VCS generates the simv executable with the corresponding overhead to prepare for these +plusargs. The more fixed information VCS has at compile-time the more VCS can optimize simv for efficient simulation, 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 we recommend that if you use this technique that you plan to abandon it fairly soon in the development cycle and switch the conditional compilation technique for writing simulation history files or switch to a combination of the two techniques.

Combining The Techniques

Some users find that they have the greatest amount of control of the advantages and disadvantages of these techniques when they combine the two techniques, for example:

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

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

Avoiding the Debugging Problems from Port Coercion

In the previous generation of Verilog simulators, Verilog-XL®, there was the port collapsing algorithm where Verilog-XL removed certain ports so that Verilog-XL could simulate faster and use less memory. In Verilog-XL you could still refer to a collapsed port but internally, inside Verilog-XL, the port did not exist.

VCS replaces Verilog-XL so VCS must mimic port collapsing so that an old but reusable design, now simulated with VCS, will have the same simulation results. For this reason the default behavior of VCS is to “coerce” all ports to inout ports so designs will simulate the same as when their ports were collapsed by Verilog-XL.

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

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

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

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

Creating Models That Simulate Faster

When modeling your design, for faster simulation, use higher levels of abstraction. Behavioral and RTL models simulate much faster that gate and switch level models. This rule of thumb is nothing unique to VCS, in fact 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 statement that you should try to avoid.

VCS is optimized for simulating sequential devices. Under certain circumstances VCS will infer that an `always` block is a sequential device and simulate the `always` block much faster. This section also describes the inferencing rules for when VCS infers a sequential device for an `always` block.

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 will simulate the `always` block faster. This section also describes the guidelines for faster simulating `always` blocks that VCS infers to be combinatorial instead of sequential devices.

Design Styles

VCS can accelerate the simulation sequential device designs that can use edge-triggered flip-flops, transparent latches, and synchronous or asynchronous resets

Unacclerated 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 statement that you should try to avoid.

Avoid unacclerated data types

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

Data Type	Description in IEEE Std 1364-1995	Description in IEEE Std 1364-2001
time and realtime	Page 23-25	Page 22

Data Type	Description in IEEE Std 1364-1995	Description in IEEE Std 1364-2001
real	Page 23-25	Page 22
named event	Page 116	Page 138
trireg net	Page 18-22	Page 26
integer array	Page 24	Page 22

Avoid unaccelerated primitives

VCS cannot accelerate tranif1, tranif0, rtranif1, rtranif0, tran, and rtran switches. They are defined in IEEE Std 1364-1995 pages 65-66, 1364-2001 page 86.

Avoid calls to 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 faster simulation. For example:

```
assign net1 = flag1;
```

Simulates faster than:

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

Continuous assignment statements are described on IEEE Std 1364-1995 pages 50-53, 1364-2001 pages 69-70

Inferring Faster Simulating Sequential Devices

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

`always` constructs are defined on IEEE Std 1364-1995 pages 98-99, 1364-1995 page 118. Verilog users commonly use the term `always` block when referring to an `always` construct.

VCS can infer that an `always` block is a combinatorial or sequential device. This section describes when VCS infers a faster simulating sequential device for an `always` block.

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 1364-1995	Description in IEEE Std 1364-2001
force and release procedural statements	Page 105-106	Page 126-127
repeat statements	Page 111-113, see the other looping statements on these pages and consider them as an alternative.	Page 134-135, see the other looping statements on these pages and consider them as an alternative.
wait statements, also called level-sensitive event controls	Page 117	Page 141
disable statements	Page132-134	Page 162-164

Statement	Description in IEEE Std 1364-1995	Description in IEEE Std 1364-2001
fork-join block statements, also called parallel blocks	Page 121	Page 146-147

There is no restriction on blocking or nonblocking procedural assignment statements.

Using either of them does not prevent VCS from inferring a sequential device but in VCS blocking procedural assignment statements are more efficient.

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

Blocking and nonblocking procedural assignment statements are described on IEEE Std 1364-1995 pages 99-104, 1364-2001 pages 119-124.

Place task enabling statements in their own always block and use no delays

Tasks and task enabling statements are defined on IEEE Std 1364-1995 pages 125-128,1364-2001 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.

Event controls are defined on IEEE Std 1364-1995 pages 114-115, 1364-2001 page 138. Sensitivity lists are not defined in IEEE Std 1364-1995 but are mentioned in 1364-2001 page 139.

For correct inference, all sequential controls must be entered 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 does infer 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 false (1 or 0) value.

The following code examples illustrate this rule.

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.

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 when sel is true or false.

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

```
always @ (sel or a or b)
```

```
if (sel)
  y=a;
```

Here there is no simulation event when signal sel is false (0).

Modeling Faster always Blocks

No matter if VCS infers an always block to model a sequential device or not, there are modeling techniques you should use or avoid for faster simulation.

Place all signals being read in the sensitivity list

Once again, 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, or whose value changes trigger simulation events, in the sensitivity list event control.

Use blocking procedural assignment statements

In VCS blocking procedural assignment statements are more efficient.

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

Blocking and nonblocking procedural assignment statements are described on IEEE Std 1364-1995 pages 99-104, 1364-2001 pages 119-124.

Avoid force and release procedural statements

These statements are defined on IEEE Std 1364-1995 pages 105-106, 1364-2001 page 126-127. A few uses of these statements in combinatorial `always` blocks will not noticeably slow down simulation but their frequent use will a performance cost.

Implemented IEEE Std 1364-2001 Language Constructs

VCS has implemented some of the new features in the IEEE 1364-2001 Verilog standard:

- Comma Separated Event Control Expression Lists
- Name-Based Parameter Passing
- ANSI Style Port And Argument Lists
- Initialize A Reg In Its Declaration
- Conditional Compiler Directives
- Signed Arithmetic Extensions
- File I/O System Tasks
- Passing Values From The Runtime Command Line
- Indexed Part-Selects
- Multi-Dimensional Arrays
- Maintaining The File Name and Line Numbers
- Implicit Event Control Expression Lists
- The Power Operator

- Attributes
- Generated Instantiation
- localparam Declarations

Comma Separated Event Control Expression Lists

If you have multiple events in an event control expression list you can use commas or the keyword `or` to separate these events. For example the event controls in the following source code are valid:

```
always @ (r1,r2,r3)
begin
en=1;
@(r4 or r5 or r6) flag1=0;
end

always @ (r7 or r8, r9)
begin
⋮
```

If you use comma separated event control expression list, include the `+v2k` compile-time option.

Name-Based Parameter Passing

You can now use a name based instead of order based parameter passing in module instantiation statements. The following example shows the old way of parameter passing:

```
module top;
child #(1,20) c1();
child #(1,10) c2();
endmodule
```

```
module child;
parameter p0 = 1;
parameter p1 = 1;
parameter p2 = 1;
   ⋮
endmodule
```

Here you only want to change the value of the second declared parameter p1, but because it is second you must pass a value to the first declared parameter p0. In the new implementation you can do the following:

```
child #(.p1(20)) c1();
child #(.p1(10)) c2();
```

This works like name-based connections lists for ports in module instantiation statements.

If you use name-based parameter passing, include the `+v2k` compile-time option.

ANSI Style Port And Argument Lists

You can now use ANSI style port and argument lists for modules, tasks, and functions, for example:

```
module top(output reg [7:0] y,
           input wire [7:0] a,
           input c,
           inout [15:0] d);
```

There is no need for a separate port declaration, the direction and size of the port can be specified in the connection list following the keyword module. You can also specify a data type for a port in the connection list.

For task definitions, output argument declarations, but not input or inout argument declarations can now include a data type, for example:

```
task t1;
output reg [7:0] y;
output integer a;
output reg [7:0] b;
input c;
inout [15:0] d;
begin
  ⋮
end
endtask
```

Function declarations can now include a connection list for arguments in the header instead of separate argument declarations, for example:

```
function [17:0] func(input b, input c);
begin
        func = b + c;
end
endfunction
```

If you do use a separate port declaration you can now include a data type, for example:

```
output reg [7:0] r1;
```

If you use ANSI style port and argument lists, include the `+v2k` compile-time option.

Initialize A Reg In Its Declaration

You can now specify and initial value in any variable declaration, for example:

```
reg [15:0] r2 = 0;
```

You cannot initialize a port that you declared with a variable data type.

If you initialize a reg in its declaration, include the `+v2k` compile-time option.

Conditional Compiler Directives

You can now use the `` `ifndef `` (if not defined) and `` `elseif `` compiler directives with the `` `ifdef ``, `` `else ``, and `` `endif `` compiler directives, for example:

```
`define fb
module test1;
`ifdef fb
    `ifndef sn
        initial $display("fb is defined");
    `else
        initial $display("fb and sn defined");
    `endif
`elsif sn
    initial $display("sn defined, fb is not");
`else
    `ifndef lr
        initial $display("fb, sn, lr not defined");
    `elsif rl
        initial
          begin
             $display("fb and sn not defined");
             $display(" lr and rl defined");
          end
    `else
        initial $display("Only lr defined!");
    `endif
`endif
endmodule
```

If you use thes compiler directives, include the `+v2k` compile-time option.

Disabling Default Net Data Types

By default all signals that are not explicitly declared have the `wire` data type, however you can use the `` `default_nettype `` compiler directive to specify a different net data type, such as `wand` or `trior`, to be the default data type for undeclared signals. See “Compiler Directives for Setting Defaults” on page 2-81.

You might encounter debugging problems with this feature of the language. For example, when you declare a signal and then later in the code assign the value of an expression to that signal (and accidentally miss type the signal name in the assignment statement), VCS assumes this is an implicitly declared signal and gives it the default data type. During or after simulation you have to figure out why your declared signal is not getting the expression value.

Now you can specify that there is no default data type so all undeclared signals result in a syntax error. You do this with the `none` argument to the `` `default_nettype `` compiler directive:

```
`default_nettype none
```

The `none` argument is not a reserved keyword. You can use it in another context, for example in a text macro.

If you disable default net data types, include the `+v2k` compile-time option.

Signed Arithmetic Extensions

Verilog 2001 enables more data types to have positive, zero, and negative values, in other words signed values.You can specify that any net or reg have a signed value by including the keyword `signed` in the signal declaration. These signed values are pertinent only in the following types of expressions:

- Expressions containing the arithmetic operators:`+ - * / %`
- Expressions containing the relational operators: `< > <= >=`

If these expressions contain some signed and some unsigned operands, VCS converts the signed operands to unsigned operands, see “Comparing Signed And Unsigned Values” on page 2-33 and “Arithmetic With Signed And Unsigned Values” on page 2-36. For expressions containing other operators, the signed value is not pertinent, and VCS treats them as bit vectors with separate values for each bit.

The Verilog 2001 variables are `reg`, `real`, `integer`, `time`, and `realtime`. In the previous standard, IEEE 1364-1995, you could assign a negative value to all these variables, but if you assigned a negative value to a `reg`, the previous standard called for assigning an expression that consisted of the unary "-" operator and the positive value, and VCS assigned the value to the 2’s compliment of that negative value instead. Now with the `signed` keyword in the `reg` declaration you can also assign a negative value to a `reg` and the value will be a negative value. You don’t need, and cannot use, the `signed` keyword in the declaration of the `real`, `integer`, `time`, and `realtime` variables.

Note that the keyword `signed` immediately follows the data type, with the possible exception of the `trireg` data type, where it follows the optional charge strength specification if you specify a charge strength, for example:

```
reg signed [1:0] reg1;
wire signed [1:0] wire_1;
tri1 signed [1:0] tri1_1;
supply0 signed [1:0] supply0_1;
wand signed [1:0] wand_1;
trior signed [1:0] trior_1;
trireg (small) signed [1:0] trireg_1;
```

Specify that a port has a signed value as follows:

```
input signed [63:0] dat1;
```

The `signed` keyword immediately follows the keyword specifying the port's direction: `input`, `output`, or `inout`. Specifying signed values for a port means you don't have to explicitly declare the net that is that port. If a reg or integer is also a port and you want that port to output signed values, you can specify signed values either in the port declaration or the reg or integer.

You can specify that a user-defined function returns a signed value as follows. See "User-Defined Functions That Return Signed Values" on page 2-39, for example:

```
function signed [127:0] alu;
```

You can specify that a parameter have a signed value as follows. Also see "Using Signed Parameters" on page 2-39, for example:

```
parameter signed paramname = 10;
```

Using signed arithmetic extensions does not require the `+v2k` compile-time option.

Signed Constant Numbers

For signed numbers the MSB specifies the sign and is not part of the value. With signed values a 0 MSB represents a non-negative number (positive and 0 values), and a 1 MSB represents a negative number. The following table shows the binary representation of signed values assigned to a four bit signal:

Non-Negative Values		Negative Values	
Binary Representation	**Signed Value**	**Binary Representation**	**Signed Value**
0111	7	1111	-1
0110	6	1110	-2
0101	5	1101	-3
0100	4	1100	-4
0011	3	1011	-5
0010	2	1010	-6
0001	1	1001	-7
0000	0	1000	-8

You express a negative value in the 2's compliment of the non-negative value.

One way to look at how negative and non-negative values are specified is as follows:

- For non-negative values the MSB is 0, designating that it is a non-negative value, and the value itself is specified with 1 values for the other bits replacing the 0 values.

- For negative values the MSM is 1, designating that it is a negative value, and the value itself is specified with 0 values replacing the 1 values.

To use a radix notation in specifying a signed value, insert the letter `s` between the single quotation mark (tick) and the radix specifying character. For example, if you assign the following values to a 4-bit signal:

`4'sb1101`	Specifies -3 in binary format. The 1 value for the MSB specifies a negative value and the single 0 in the next to LSB bit specifies that the negative value is -3.
`4'so10`	Specifies -8 in octal format. The binary representation of the octal 10 value for four bits is 1000. Here the MSB is 1 so it is a negative value. The remaining zeroes specify that the negative value is -8.
`4'sd7`	Specifies a 7 in decimal format. The binary representation of the decimal 7 value for four bits is 0111. Here the MSB is 0 so it is a non-negative value. The remaining ones specify that the non-negative value is 7.
`4'she`	Specifies a -2 in hexadecimal format. The binary representation of the hexadecimal e value is 1110. Here the MSB is 1 so it is a negative value. The sole zero in the LSB specifies that the negative value is -2.

You can use an explicit sign with these signed constant numbers but in doing so you could negate a negative value.

Example:

```
signed_reg = -4'sd15;
```

The signed constant number `4'sd15`'s binary representation is 1111 so it is a negative value -1, however the preceding explicit negation sign (-) makes the value -(-1) or +1.

Comparing Signed And Unsigned Values

Some signed values appear to have different values but actually have matching values. For example:

```
(4'sb1110 == 8'sb11111110)
```

This expression evaluates to true. The two operands are signed values, the MSBs in these operands are ones so they both are negative values. As negative values the zeroes determine the value, and both operands have only one zero, each in the LSB, so the value of both operands is -2. This expression is just as true as the following expression:

```
(4'sb0001 == 8'sb00000001)
```

In this expression it is more apparent that the value of both operands is 1.

Non-negative signed values do equal their corresponding unsigned values. For example the following signals have equal values despite the fact that only one has a signed value and they have different bit widths:

```
reg signed [1:0] sr1 = 2'sb01;
reg [1:0] r1 = 2'b01;
reg r2 = 1'b1;
```

Therefore the following expressions are all true:

```
(sr1 == r1)
(sr1 == r2)
(r1 == r2)
```

Negative signed values are less than zero or positive signed or unsigned values, however this cannot be proved by the relational operators. In expressions that contain these operators and with both signed and unsigned operands, VCS converts the signed operands to unsigned operands.

With the following source code:

```
reg signed [1:0] sr1 = 2'sb10; // initial value -2
reg signed [1:0] sr2 = 2'sb11; // initial value -1
reg [1:0] r1 = 2'b01;          // initial value 1
reg r2 = 1'b0;                 // initial value 0
⋮
initial
begin
if(sr1 < sr2)
  $display("sr1 < sr2");
if(sr1 < r1)
  $display("sr1 < r1");
if(sr1 < r2)
  $display("sr1 < r2");
if(r2 < r1)
  $display("r2 < r1");
end
endmodule
```

VCS displays only the following:

```
sr1 < sr2
r2 < r1
```

In the comparison expression `(sr1 < r1)` and `(sr1 < r2)`, VCS converts the -2 value of signal sr1 to 2, so the expressions are not true. In the comparison expression `(sr1 < sr2)`, both operands have signed values so VCS does not convert the -2 value of sr1, so the expression is true.

Displaying Signed Values

The `%d` format specification can display negative signed values such as -1.

Example:

For the following code:

```
reg signed [1:0] sr1=2'sb11;
⋮
initial
$display("sr1=%0b %0o %0d %0h",sr1,sr1,sr1,sr1);
```

The `$display` system task displays the following:

```
sr1=11 3 -1 3
```

Arithmetic With Signed And Unsigned Values

Expressions using the `+`, `-`, `*`, `/`, and `%` operators work as you might expect when the operands are all signed values or all unsigned values. However when some operands are signed and others are not signed, VCS converts the signed values to unsigned values, for example:

```
reg signed [7:0] sr1 = 8'sb11111000; // initial value -8
reg signed [7:0] sr2 = 8'sb00000100; // initial value 4
reg [7:0] r1 = 4;                    // initial value 4
⋮
$display("sr1 + sr2 = %0d", sr1 + sr2);
$display("sr1 + r1 = %0d",sr1 + r1);
```

VCS displays the following:

```
sr1 + sr2 = -4
sr1 + r1 = 252
```

When VCS adds sr1 to sr2 both operands are signed so -8 + 4 = -4, but when VCS adds the signed signal sr1 to the unsigned signal r1, VCS converts the -8 signed value of sr1 to the unsigned value 248 and 248 + 4 = 252.

Arithmetic Shift Operators

With Verilog 2001 there are two types of shift operators, the logical shift operators >> and << which were in the previous standard, and the arithmetic shift operators >>> and <<<.

The arithmetic left shift operator <<< works the same way as the logical left shift operator <<. The values of the bits of the signal that is the left operand are reassigned or shifted to the bits toward the MSB position. The number of bits these values are shifted is specified by the integer which is the right operand. These operators assign zeros to the vacated bits.

For a signed signal the arithmetic left shift operator can shift a 1 value in place of a 0 value in the MSB. When it does it changes the sign of the signal from positive to a negative.

The arithmetic right shift operator >>> and the logical right shift operator both shift the values of signal that is the left operand toward the LSB position and the number of bits is also specified by the right operand integer. The difference between these operators is that the logical right shift operator always fills the vacated bits with zeroes, whereas the arithmetic right shift operator fills that vacated bits with what was the previous value of the MSB. This means that if the signal that is the left operand has a negative value, The vacated bits are filled with ones, preserving the negative value.

Example:

For the following code:

```
reg signed [7:0] sr1 = 8'sb10100000;

initial
begin
$monitor("sr1=%0d %b",sr1,sr1);
#1 sr1 = sr1 >>> 2;
#1 sr1 = sr1 >>> 2;
#1 sr1 = sr1 >>> 2;
#1 sr1 = sr1 >>> 2;
end
```

The `$monitor` system task displays:

```
sr1=-96 10100000
sr1=-24 11101000
sr1=-6 11111010
sr1=-2 11111110
sr1=-1 11111111
```

Note that the previous MSB does not have to indicate a negative value for the arithmetic right shift operator to fill with the previous value of the MSB. If for example the initial value of sr1 was 8'sbz0100000 the `$monitor` system task would display:

```
sr1=Z z0100000
sr1=Z zzz01000
sr1=Z zzzzz010
sr1=Z zzzzzzz0
sr1=Z zzzzzzzz
```

User-Defined Functions That Return Signed Values

Entering the `signed` keyword in a function declaration header specifies that VCS uses the return value as a signed value. It is not necessary for the assignment in the function declaration (to the internal register that has the same name as the function) to be an assignment of a signed value.

Example:

In the following code, wire w1 has an initial value of -2, the returned value of the function named funcname, even though the assignment inside the function declaration is not an assignment of a signed number.

```
function signed [1:0] funcname;
input in;
begin
funcname=2;
end
endfunction

wire signed [1:0] w1=funcname(1'b1);
```

Using Signed Parameters

Verilog module parameters have always been able to hold negative values, but specifying that a parameter is signed tells VCS to interpret a value passed to the parameter at compile-time to be a negative signed value. VCS stores parameters in 32 bits, so VCS will interpret a value to be negative if your code passes a 32 bit value with the MSB having a value of 1.

Example:

For the following code:

```
module top;
paramod #(.p1(32'b11111111111111111111111111111111),
          .sp1(32'b11111111111111111111111111111111)) pm1();
defparam top.pm1.p2=32'b11111111111111111111111111111111,
         top.pm1.sp2=32'b11111111111111111111111111111111;
endmodule

module paramod;
parameter p1 = -2,
          p2 = -2;
parameter signed sp1 = -2,
                 sp2 = -2;
initial
$display("p1=%0d p2=%0d sp1=%0d sp2=%0d",p1,p2,sp1,sp2);
endmodule
```

The `$display` system task displays the following:

```
p1=4294967295 p2=4294967295 sp1=-1 sp2=-1
```

The signed parameters have negative values and the unsigned ones do not, even though they were all passed the same 32 bit value.

Bit-Selects And Part-Selects of Signed Signals

Bit-selects and part-selects of signed signals are never signed themselves. When used in an expression they are only bit values and do not represent a signed value. However, when assigned to a signed signal, they can change the sign as well as the value of that signal.

Example:

```
reg signed [2:0] sr1 = 2'sb110;
```

```
reg signed [2:0] sr2;

initial
begin
if (sr1[2:1] == 3)
   sr2 = { sr1[1],1'b0,1'b1};
$display("sr2=%0d %b",sr2,sr2);
end
```

The part select in the conditional expression does not have a negative value. Its value is 3, so VCS executes the procedural assignment statement. VCS then displays:

```
sr2=-3 101
```

So assigning the bit-select gave this signal a negative value.

Concatenating Signed Values

The result of a concatenation of signed values is an unsigned value.

Example:

```
reg signed [2:0] sr1 = 3'sb101; // assign -3
reg signed [2:0] sr2 = 3'sb010; // assign +2
reg signed [2:0] sr3;
initial
begin
$display("sr1=%0d %b sr2=%0d %b",sr1,sr1,sr2,sr2);
$display("{sr1,sr2}=%0d",{sr1,sr2});
$display("{sr1[0],sr1[1],sr1[2]}=%0d",
          {sr1[0],sr1[1],sr1[2]});
sr3={sr1[0],sr1[1],sr1[2]};
$display("sr3=%0d %b",sr3,sr3);
end
```

In this example, the `$display` system tasks display:

```
sr1=-3 101 sr2=2 010
{sr1,sr2}=42
{sr1[0],sr1[1],sr1[2]}=5
sr3=-3 101
```

Concatenating sr1 with sr2 results in the unsigned 42 value. Reversing the bits of sr1, where the MSB remains a one, still results in an unsigned value of 5, but assigning these bits to sr3 assigns a -3 value because sr3 is a signed signal.

Type Casting System Functions

Verilog V2K has system functions for converting values to and from signed and unsigned values:

`$signed(`*`expression`*`)`
: Returns a signed value.

`$unsigned(`*`expression`*`)`
: Returns an unsigned value.

Example:

```
reg [3:0] r1;
reg signed [3:0] sr1;

initial
begin
r1 = $unsigned(-4);
sr1 = $signed(4'b1100);
$display("r1=%0d sr1=%0d",r1,sr1);
end
endmodule
```

In this example, the `$display` system task displays:

```
r1=12 sr1=-4
```

Limitations of Signed Arithmetic Extensions

You cannot use signed arithmetic extension with Radiant Technology. Do not compile a design containing signed arithmetic extensions with the `+rad` option.

There are also limitations on the use of signed signal as operands to the power `**` operator. See “The Power Operator” on page 2-60.

File I/O System Tasks

This section describes the new system tasks and functions for file I/O in the 1364-2001 standard and the modifications in this standard to the system tasks defined in the 1364-1995 standard.

To use these system tasks and function you don’t need to include the `+v2k` compile-time option.

$fopen

The `$fopen` system task can now return a file descriptor, in addition to a multi-channel descriptor, when it opens a file.

The file descriptor is a 32 bit value with the MSB reserved and always set so that other system tasks can see how the file was opened. The remaining bits hold a smaller number that differentiate it from other file descriptors for other open files. Unlike multi-channel descriptors, file descriptors cannot be combined through a bitwise or. Three file descriptors are always open: STDIN (with the value 32'h8000_0000), STDOUT (32'h8000_0001), and STDERR (32'h8000_0002). STDIN is open for read operations. STDOUT and STDERR are open for append operations.

You specify a file descriptor, instead of a multi-channel descriptor, with a type argument in the $fopen system task. The types you can specify are as follows:

Type Argument	Description
`"r"` or `"rb"`	Open for reading
`"w"` or `"wb"`	If file already exists, first change to zero length or create a new file for writing
`"a"` or `"ab"`	Append to an existing file; open a file for writing at the end of the file, or create a new file for writing
`"r+"`, `"r+b"`, or `"rb+"`	Open a file for an update (reading and writing)
`"w+"`, `"w+b"`, or `"wb+"`	If file already exists, first change to zero length or create for update
`"a+"`, `"a+b"`, or `"ab+"`	Append to an existing file; open a file for writing at the end of the file, or create a new file for writing

The `b` character in these arguments specifies a binary rather than a text file but this distinction is not made in UNIX systems.

The syntax for `$fopen` for a file descriptor is as follows:

```
integer file_descriptor_name = $fopen("filename","type");
```

The maximum number of files you can open using $fopen and multi-channel descriptors is 30. Before the implementation of the alternative file descriptor, this limit was 31.

$fscanf

The `$fscanf` system function reads characters in a file, interprets them according to the format specifications in a format string, and assigns these characters to a list of signal arguments.

Syntax:

```
integer = $fscanf(file_descriptor,"format",signalname...);
```

Example:

```
module fscan;
integer fd0,nfd;
reg [1:0] r1,r2,r3;
initial
begin
fd0=$fopen("fscanfile","w");
$fwrite(fd0,"r1=%d r2=%d r3=%d",0,1,2);
$fclose(fd0);
fd0=$fopen("fscanfile","r");
nfd=$fscanf(fd0,"r1=%d r2=%d r3=%d",r1,r2,r3);
$display("r1=%d r2=%d r3=%d",r1,r2,r3);
end
endmodule
```

The `$fwrite` system task writes these assignments in file fscanfile as specified by file descriptor fd0. The `$fscanf` system function makes the assignment in the same order.

The format specifications that you can enter in your format string are as follows:

`%b`	binary number
`%o`	octal number
`%d`	decimal number, can be signed

%h or %x	hexadecimal number
%f, %e, or %g	floating point number, can be signed
%v	Specifies a signal strength
%t	Specifies a timescale
%c	Specifies a single character
%s	Specifies a character string
%u	Specifies unformatted binary data
%z	Specifies reading unformatted binary data.

$sscanf

The $sscanf system function is similar to the $fscanf system function except that it reads from an input string rather than a file. This input string can be a character string or a signal with a bit width that is a multiple of eight bits, or in other words capable of holding an intact character string.

Syntax:

```
integer = $fscanf(input_string,"format",signalname...);
```

The format specifications you can enter in the format string are the same as those for the $fscanf system function.

Example:

```
module top;
reg [23:0] r1,r2,r3;
integer nfd1,nfd2;
initial
begin
r1="abc";
nfd1=$sscanf(r1,"%s",r2);
nfd2=$sscanf("def","%s",r3);
```

```
$display("r2=%s r3=%s",r2,r3);
end
endmodule
```

The `$display` system task displays:

```
r2=abc r3=def
```

$fread

The `$fread` system function reads binary data from a file and assigns it to a reg or memory. With memories you can specify a starting address and how many memory elements to write values to.

Syntax:

```
integer = $fread(reg,file_descriptor);

integer = $fread(memory,file_descriptor
[,start_address][,element_count]);
```

You can specify the element that `$fread` starts writing data to with the *`start_address`* argument. After writing data to this specified element, VCS writes to the next highest numbered element. You can specify how many elements of the memory VCS writes to with the *`element_count`* argument.

Example:

This example specifies writing to elements numbered 4, 5, and 6 in memory mem1.

```
int = $fread(mem1,fd0,4,3);
```

$sformat

The `$sformat` system function assigns a string value to a specified signal with a bit width that is a multiple of eight bits.

Syntax:

```
integer = $sformat(signalname,"format",arguments);
```

The format string is the same as the format string for `$fscanf`.

The arguments can be character strings or other signals with a bit width that is a multiple of eight bits. T

Example:

```
module sformat;
reg [23:0] r1,r2,r3;
integer i,j;
initial
begin
r1="abc";
i=$sformat(r2,"r1 is %s",r1);
j=$sformat(r3,"%s","xyz");
$display("r2=%s r3=%s",r2,r3);
end
endmodule
```

For this example, the `$display` system task displays:

```
r2=abc r3=xyz
```

$swrite

The `$swrite` system function is similar to the `$sformat` system function with one exception, `$swrite` can have more than one format string argument.

Syntax:

```
integer = $swrite(signalname,list_of_arguments...);
```

Example:

```
module swrite;
reg [23:0] r1,r2;
reg [47:0] r3;
integer i,j;
initial
begin
r1="abc";
r2="def";
i=$swrite(r3,"%s",r1,"%s",r2);
$display("r3=%s",r3);
end
endmodule
```

In this example, the `$swrite` system function has more than one format string. The `$display` system task displays:

```
r3=abcdef
```

There are also the `$swriteb`, `$swriteo`, and `$swriteh` system functions for assigning binary, octal, and hexadecimal values.

$fgetc

The `$fgetc` system function reads a character from a file.

Syntax:

```
integer = $fgetc(file_descriptor);
```

Example:

```
module testfgetc;

integer fd, c;
reg[0:7] r0;
reg[0:15]r1;

initial
begin
    fd = $fopen ("testfgetc.dat","w");
    $fwrite(fd, "aBc");
    $fclose (fd);

    fd = $fopen ("testfgetc.dat", "r");
    r1 = 0;
    c = 0;
    repeat (3)
     begin
          r1 = $fgetc(fd);
          $display("r1=%s", r1);
          c = c + 1;
    end
    $display ("Read %0d characters", c);
end

endmodule
```

In this example, the `$display` system tasks display the following:

```
r1= a
r1= B
r1= c
Read 3 characters
```

$ungetc

The `$ungetc` system function puts a character back into the input stream after it has been removed from there by the `$fgetc` system function.

Syntax:

```
integer = $ungetc(signalname,file_descriptor);
```

Example.

```
module io2;
reg [7:0] r1;
reg [7:0] r2;

integer fd0,nonfd0;

initial
begin

fd0=$fopen("io2file","wb");
$fwrite(fd0,"%s","hello_world");
$fclose(fd0);

fd0=$fopen("io2file","rb");

nonfd0=$fscanf(fd0,"%c",r1);
$display("just read character %c", r1);
r2=$fgetc(fd0);  // same thing
$display("just read character %c", r2);
nonfd0 = $ungetc(r2, fd0);
nonfd0 = $ungetc(r1, fd0);

nonfd0=$fscanf(fd0,"%c",r1);
$display("just read character %c", r1);
r2=$fgetc(fd0);  // same thing
$display("just read character %c", r2);


$fclose(fd0);
end
endmodule
```

In this example, the `$display` system tasks display the following:

```
just read character h
just read character e
just read character h
just read character e
```

The `$ungetc` system functions put back the h and e characters so that they could be displayed again.

$fgets

Reads characters from a file, and assigns them to a reg, until the reg is filled, or a newline character is read and assigned to the reg, or and end-of-file condition is found. If the reg is not an integral number of bytes in length (it contains a number of bytes and a partial byte) the most significant partial byte is not used in order to determine the size. If the reg does not have enough bytes there is an error condition.

You use this system function in an assignment to an integer. It returns the number of characters assigned to the reg unless there was an error condition, where it returns zero.

Syntax:

```
fgets(reg_name,file_descriptor)
```

Example:

```
module testfgets;
integer fd,int1,int2;
reg [23:0] r1;
reg [17:0] r2;
initial
begin
fd = $fopen("testfgets.dat","w");
$fwrite(fd,"aBc");
$fclose(fd);
```

```
fd=$fopen("testfgets.dat","r");
int1=$fgets(r1,fd);
int2=$fgets(r2,fd);
$display("r1=%s r2=%s int1=%0d int2=%0d",r1,r2,int1,int2);
end
endmodule
```

VCS displays the following:

```
r1=aBc r2=    int1=3 int2=0
```

$fflush

The `$fflush` system task writes all buffered data to their respective files.

Syntax:

```
$fflush([multichannel_descriptor|file_descriptor]);
```

If you specify a multi-channel of file descriptor, VCS writes all buffered data for the respective file to that file. Without arguments VCS writes all buffered data to their files.

$ftell

The `$ftell` system function returns the offset of a file. This offset is the position in the file, in bytes, that VCS has just written or read from.

Syntax:

```
integer = $ftell(file_descriptor);
```

Example:

```
module ftell;
integer fd0,nfd,offset;
reg [1:0] r1,r2,r3;

initial
begin
fd0=$fopen("ftellfile","w");
offset = $ftell(fd0);
$display("                          offset=%0d",offset);
$fwrite(fd0,"r1=%d r2=%d r3=%d",0,1,2);
offset = $ftell(fd0);
$display("                          offset=%0d",offset);
$fclose(fd0);
$system("more tellfile");
fd0=$fopen("ftellfile","r");
offset = $ftell(fd0);
$display("                          offset=%0d",offset);
nfd=$fscanf(fd0,"r1=%d r2=%d r3=%d",r1,r2,r3);
offset = $ftell(fd0);
$display("                          offset=%0d",offset);
$display("      r1=%d r2=%d r3=%d",r1,r2,r3);
offset = $ftell(fd0);
$display("                          offset=%0d",offset);
$fclose(fd0);
end

endmodule
```

In this example, the `$system` and `$display` system tasks display the following:

```
                        offset=0
                        offset=44
r1=          0 r2=          1 r3=          2
                        offset=0
                        offset=44
      r1=0 r2=1 r3=2
                        offset=44
```

Including spaces there are 44 characters in the file ftellfile.

$fseek

The `$fseek` system task sets the position of the next read or write operation.

Syntax:

```
integer = $fseek(file_descriptor,offset,operation);
```

Where:

offset — A number of bytes. The next read or write operations will be at the location specified by this number of bytes and the entry for the *operation* argument.

operation — Can be one of the following numbers:

- 0 — Next read or write is the number of bytes from the beginning of the file specified by the *offset* argument
- 1 — Next read or write is the current location plus the number of bytes specified by the *offset* argument
- 2 — Next read or write is the number of bytes from the end of the file specified by the *offset* argument

$rewind

The `$rewind` system function set the next read or write operation to the beginning of the file.

Syntax:

```
integer = $rewind(file_descriptor);
```

$ferror

The `$ferror` system function returns additional information about an error condition in file I/O operations.

Syntax:

```
integer = $ferror(file_descriptor,signalname);
```

The signal you specify must have at least 640 bits. VCS writes the error code string to this signal.

Passing Values From The Runtime Command Line

The `$value$plusargs` system function can pass a value to a signal from the simv runtime command line using a +plusarg, see “Passing Values From The Runtime Command Line” on page 4-31.

This system function does not require the `+v2k` compile-time option.

Indexed Part-Selects

You can now specify a part select by specifying a particular range from that bit toward the lower or higher numbered bits.

Syntax:

```
[start_bit+|-:range]
```

In this part-select a plus + specifies the part-select extends from the start bit to the higher numbered bits, the minus – specifies that the part-select extends from the start bit into the lower numbered bits.

Example:

```
module indexed;
reg [15:0] r1,r2;
reg [0:15] r3,r4;
initial
begin
$monitor("r1=%0b r2=%0b r3=%0b,r4=%0b",r1,r2,r3,r4);
r1[8+:5]=5'b11111;
r2[8-:5]=5'b11111;
r3[8+:5]=5'b11111;
r4[8-:5]=5'b11111;
end
endmodule
```

In this example, the `$monitor` system task displays the following:

```
r1=xxx11111xxxxxxxx r2=xxxxxxx11111xxxx
r3=xxxxxxxx11111xxx r4=xxxx11111xxxxxxx
```

Signal r1 has 1 values assigned to the 12-8 bits because the start bit was 8, the range 5, and the plus sign in the part select.

Signal r2 has 1 values assigned to the 8-4 bits because the start bit was 8, the range 5, and the minus in the part select.

Signal r3 has 1 values assigned to the 8-12 bits because the start bit was 8, the range 5, and the plus sign in the part select.

Signal r4 has 1 values assigned to the 4-8 bits because the start bit was 8, the range 5, and the minus in the part select.

If you use indexed part-selects, include the `+v2k` compile-time option.

Multi-Dimensional Arrays

You can declare multi-dimensional arrays of the `reg` and `integer` variable data type or any of the net data types.

Example:

```
module mult_dim_arrays;
reg [7:0] mem1 [3:0] [3:0];
integer int1 [2:0][10:1];
wire [31:0] w1 [2:0][2:0];

initial
begin
mem1[3][3]=8'b11111111;
int1[1][9]=36;
end
assign w1 [2][2]=mem1[3][3];
endmodule
```

The IEEE 1364-2001 standard extended the PLI VPI routines for multi-dimensional arrays but the current implementation of these routines in VCS do not support accessing multi-dimensional arrays.

If you use multi-dimensional arrays, include the `+v2k` compile-time option.

Maintaining The File Name and Line Numbers

The `` `line `` compiler directive specifies the source file the compiler directive is in and the line number it's on. Its syntax is as follows:

```
`line line_number "filename" level
```

You use this compiler directive, for example, if you use a preprocessor for your source code that you have developed or acquired from another company. When you use it the preprocessor passes this information to VCS so VCS error messages are about the original source code and not the output of the preprocessor.

The level parameter indicates whether an include file has been entered (value is 1), an include file is exited (value is 2), or neither has been done (value is 0).

If you use the `‘line` compiler directive, include the `+v2k` compile-time option.

Implicit Event Control Expression Lists

You can use the `*` wild card character in place of an event control expression list to represent all signals read to represent all signals whose values VCS reads in the execution of the statement or statements controlled by the event control. Typically these signals are in the right hand side of assignment statements but also can be those in function and task calls and in case and conditional expressions. The following is an example of an implicit event expression list:

```
always @*
begin
#5 r5 = r1 && r2;
#5 r6 = r3 && r4;
end
```

VCS executes the begin-end block when there is a value change in signals r1, r2, r3, or r4.

If you use an implicit event control expression list, include the `+v2k` compile-time option.

The Power Operator

The ** power operator raises the value of its first operand to the power of the second operand. For example:

```
initial
begin
r1=3;
r2=2;
int=r1 ** r2;
$display("int is %0d",int);
end
```

VCS displays:

```
int is 9
```

The result of the power operator is real if either operand is a real, integer,or a signed data type. If both operands are unsigned then the result shall be unsigned.

The result of the power operator is unspecified if the first operand is zero and the second operand is non-positive, or if the first operand is negative and the second operand is not an integral value.

When both operands are unsigned, the result of the power operator can be no longer than 1024 bits.

If you use the power operator, include the `+v2k` compile-time option.

Attributes

Attributes specify properties about objects, statements, and groups of statements in your Verilog code that can be used by various EDA tools, such as Design Compiler, to control how those tools work. VCS itself does not use attributes but stores this information so the attributes can be accessed from VCS using the VPI.

The following are examples of attributes:

```
(* optimize_power=1 *)
module dev (res,out,clk,data1,data2);
```

```
⋮
(* fsm_state=0 *) reg [3:0] reg2;
⋮
(* full_case=1, parallel_case = 0 *)
case (flag_1)
⋮
a = b + (* mode = "cla" *) c;
⋮
a = add (* mode = "cla" *) (b, c);
⋮
a = b ? (* no_glitch *) c : d;
```

VCS has implemented attributes as specified in the 1364-2001 standard with the following exceptions:

- The standard specifies “If the same attribute name is defined more than once for the same language element, the last attribute value shall be used...” but in VCS, if you do the following:

```
(* some_attribute=0, some_attribute=1 *)
```

VCS returns both the 0 and 1 value to an application that calls for the attribute from VCS.

- VCS has not implemented attributes on local parameters because VCS has not implemented local parameters.

The `+v2k` compile-time option enables VCS to store attributes.

Generated Instantiation

Note: The implementation of generate statements described in this document follows the 1364-2001 standard. An IEEE committee is currently examining generate statements. In future versions of the Verilog language, generate statements might function differently.

After a Verilog design has been parsed, but before simulation begins, the design must have the modules being instantiated linked to the modules being defined, the parameters propagated among the various modules, and hierarchical references resolved.This phase in understanding a Verilog description is termed *elaboration*.

Generate instantiations are resolved during elaboration because that is when the parameters associated with a module become defined, hence, allowing the definition of the generated statements and declarations.

Genvars are variables that are only defined during the evaluation of the generate instantiations and do not exist during simulation of a design.

All generate instantiations are coded within a module scope and require the keywords `generate` and `endgenerate`.

Generate statements allow control over the declaration of variables, functions and tasks, as well as control over instantiations. Generated instantiations are one or more modules, user-defined primitives, Verilog gate primitives, continuous assignments, initial blocks and always blocks. Generated declarations and instantiations can be conditionally instantiated into a design. Generated variable declarations and instantiations can be multiply instantiated into a design. Generated instances have unique identifier names and can be referenced hierarchically.

To support the interconnection between structural elements and/or procedural blocks, generate statements permit the following Verilog data types to be declared within the generate scope `net, reg, integer, real, time, realtime`, and `event`. Generated data types have unique identifier names and can be referenced hierarchically.

Parameter redefinition using by the ordered or named `parameter = value` assignment or `defparam` statements can also be declared within the generate scope. However, a `defparam` statement within the generate scope or within a hierarchy instantiated within the generate scope shall only modify the value of a parameter declared within the generate scope or within a hierarchy instantiated within the generate scope.

Tasks and functions declarations shall also be permitted within the generate scope, however not in a generate loop. Generated tasks and functions shall have unique identifier names and may be referenced hierarchically.

Module declarations and module items that shall not be permitted in a generate statement include parameters, input declarations, output declarations, inout declarations and specify blocks.

Connections to generated module instances are handled the same way as they are handled with normal module instances.

Generated statements are created using one of the following three methods: generate-loop, generate-conditional, or generate-case.

Examples

The example below shows how a module can be instantiated using the generate construct.

You can use the command line construct `+define` to pass a value for an OPERATION ranging from 0 to 3 to pick either MUL, DIV, ADD, and sub.

example:

```
% vcs instance.v +v2k +define+OPERATION=0
```

The example specifies the OPERATION to be passed as a parameter to a submodule control, regardless of whether a multiplier, adder, divider, or a subtractor is instantiated under the hierarchy top.submodule.

Example 2-1 Generated instance

```
module top;

reg [3:0] a,b;
wire [7:0] y;

`define OPERATION  0
```

```
submodule #(`OPERATION) submodule_instance ( a,b,y );

initial
begin
a = 8;
b = 6;
$display("The value of a,b,y are : a = %d b =  %d y = %d ",
a,b,y);
$finish;
end

endmodule

module submodule ( a,b,y);
input [3:0] a,b;
output [7:0] y;
wire [7:0] y;
wire [3:0] a,b;

parameter CONTROL = 1;

`define MUL 0
`define DIV 1
`define ADD 2
generate

    case (CONTROL)
    `MUL:  mul U (a,b,y);
    `DIV:  div    U (a,b,y);
       `ADD:  add      U (a,b,y);
    default: sub    U (a,b,y);
    endcase

endgenerate
endmodule


module mul ( a,b,y);
input [3:0] a,b;
output [7:0] y;
wire [7:0] y;
    assign y = a * b;
```

```
endmodule

module div ( a,b,y);
input [3:0] a,b;
output [7:0] y;
wire [7:0] y;
     assign y = a / b;
endmodule

module add ( a,b,y);
input [3:0] a,b;
output [7:0] y;
wire [7:0] y;
     assign y = a + b;
endmodule

module sub ( a,b,y);
input [3:0] a,b;
output [7:0] y;
wire [7:0] y;
     assign y = a - b;
endmodule
```

Example 2-2 Generate for loop

A forloop is used to demonstrate how to generate the following code:

```
add U[0].add_instance ( a[0], b[0], y[1:0]);

add U[1].add_instance ( a[1], b[1], y[3:2]);

add U[2].add_instance ( a[2], b[2], y[5:4]);
```

Note in the previous code that the name of the Named Block (in this case, `U`) is prepended to each instance name, along with a corresponding numbered deliniation (`U[0], U[1], U[2]`). This is required.

The example below also explains a Named Block.

```
module top;

reg [2:0] a, b;
wire [5:0] y;

genvar gv;

generate

     for ( gv =0 ; gv <=2; gv= gv +1 ) begin : U
      add add_instance ( a[gv], b[gv] , y[2*gv+1:2*gv]);
     end
endgenerate

initial
begin
{a , b} = 6'b100_110;

$display ( "%d %d %d", top.U[2].add_instance.y,
top.U[1].add_instance.y, top.U[0].add_instance.y );
$finish;
end

endmodule


module add (  a, b ,y );
input a;
input b;

output [1:0] y;
wire [1:0] y;

assign y = a + b;

endmodule
```

Limitations

- SDF back annotation is not implemented for generated instances.
- Does not work with Coverage Metrics.

Enabling

You enable the use of generate statements with the `+v2k` compile-time option.

---

localparam Declarations

A `localparam` in Verilog-2001 works the same way that a `parameter` does except for one exception, a `localparam` cannot be overridden by a `defparam` statement or a `parameter` value assignment in a module instantiation statement, for example:

```
module test;
parameter p1=1, p2=0;
⋮
dev #(.par1(p1),.locp1(p2))d1 (w1,r1);

defparam d1.par1=3;
defparam d1.locp1=1;
endmodule

module dev(output out, input in);
parameter par1=0;
localparam locp1=0;
⋮
endmodule
```

Module dev contains both a `parameter` and a `localparam`.

Module test instantiates module dev. The module instantiation statement for module dev contains two parameter value assignments. The assignment to `parameter` par1 is valid, but the assignment to `localparam` locp1 is invalid and results in an error condition.

Module test also contains two `defparam` statements. The first one, to `parameter` par1 is valid, but the second one, to `localparam` locp1 is invalid and also results in an error condition.

You enable the use of a localparam with the `+v2k` compile-time option.

Constant Functions

Constant functions are used to create values at elaboration time, before simulation starts. They are typically used for calculating parameter values.

```
module test;
parameter param1 = 8;
localparam lp1 = const_func(param1);

function integer const_func;
input in1;
integer i;
begin
for (i = 0; i < in1; i = i + 1)
const_func = i;
end
endfunction
endmodule
```

You can also have your code call a constant function during simulation.

Constant functions cannot contain any of the following:

- hierarchical references
- function calls that are not constant functions

- function calls to constant functions that are not also defined in the same module definition.
- system function calls

Using the +v2k Compile-Time Option

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

Std 1364-2001 Construct	**Require** `+v2k`
comma separated event control expressions: `always @ (r1,r2,r3)`	yes
name-based parameter passing: `modname #(.param_name(value)) inst_name(sig1,...);`	yes
ANSI-style port and argument lists: `module dev(output reg [7:0] out1, input wire [7:0] w1);`	yes
initialize a reg in its declaration: `reg [15:0] r2 = 0;`	yes
conditional compiler directives: `` `ifndef `` and `` `elseif ``	yes
disabling the default net data type: `` `default_nettype ``	yes
signed arithmetic extensions: `reg signed [7:0] r1;`	no
file I/O system tasks: `$fopen $fsanf $scanf` and more	no

Std 1364-2001 Construct	Require `+v2k`
passing values from the runtime command line: `$value$plusarg` system function	yes
indexed part-selects: `reg1[8+:5]=5'b11111;`	yes
multi-dimensional arrays: `reg [7:0] r1 [3:0] [3:0];`	yes
maintaining file name and line number: `` `line ``	yes
implicit event control expression lists: `always @*`	yes
the power operator: `r1=r2**r3;`	yes

attributes:
`(* optimize_power=1 *)`
`module dev (res,out,clk,data1,data2);`
	yes

`generate` statements	yes
localparam declarations	yes
Automatic tasks and functions `task automatic t1();`	requires the `-sverilog` compile-time option
constant functions `localparam lp1 = const_func(p1);`	yes
parameters with a bit range `parameter  bit [7:0][31:0] P = {32'd1,32'd2,32'd3,32'd4,32'd5,32'd6,32'd7,32'd8};`	requires the `-sverilog` compile-time option

Case Statement Behavior

The IEEE Std 1364-1995 and 1364-2001 standards for the Verilog language state that you can enter the question mark character (?) in place of the z character in casex and casez statements. The standard does not specify that you can also make this substitution in case statements and you might infer that this substitution is not allowed in case statements.

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

Memory Size Limits in VCS

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

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

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

Using Sparse Memory Models

If the simv executable needs large amounts of machine memory and If your design contains a large memory but your simulation only accesses a small number of elements in this memory you can significantly reduce the amount of machine memory VCS will need to simulate the large memory using a sparse memory model.

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

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

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

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

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

Note:

Sparse memory models can not 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 can not be used as a personality matrix in PLA system tasks.

Obtaining Scope Information

VCS has custom format specifications (not defined in IEEE Std 1364-1995 or 1364-2001) for displaying scope information. It also has system functions for returning information about the current scope.

Scope Format Specifications

The IEEE Std 1364-1995 and 1364-2001 describe the `%m` format specification for system tasks for displaying information such as `$write` and `$display`. The `%m` 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 the hierarchical name is the hierarchical name of the task declaration or function definition.

VCS has these additional format specifications for displaying scope information:

`%i`
Specifies the same as `%m` with one difference. When in a user-defined task, the hierarchical name is the hierarchical name the instance or named block containing the task enabling statement, not the hierarchical name of the task declaration. If the task enabling statement is in another user-defined task, the hierarchical name is the hierarchical name of the instance or named block containing the task enabling statement for this other user-defined task.

`%-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 is in the following:

- A named block — the hierarchical name is that of the module instance that contains the named block

- A user-defined task — the hierarchical name of the module instance containing the task enabling statement
- A user-defined function — the hierarchical name of the module instance containing the function definition

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

The following commented code example show 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.my_func"
$display("%-i");   // displays "top"
end
endfunction

dev1 d1 (r1);
endmodule

module dev1(inport);
input inport;
```

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

endmodule
```

Returning Information About The Scope

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

When VCS executes these system functions it does the following:

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

   The possible arguments are hierarchical names. If there are arguments it compares them from left to right with the current scope. If an argument matches the system function returns a 32-bit non-zero value. If none of the arguments match the current scope the system function returns a 32’b0 value.

The following is an example that contains these system functions.

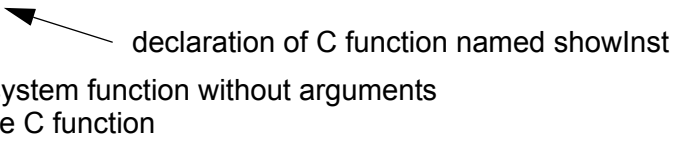
```
module top;
reg r1;
initial
r1=1;
dev1 d1(r1);
endmodule

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

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

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

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




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

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

In task t the following occurs:

1. the `$activeinst` system function returns a pointer to the current scope which is passed to the C function showInst. It is a pointer to a volatile or temporary char buffer containing the name of the instance.
2. A nested begin block executes only if the current scope is one of the two specified
3. VCS displays whether `$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 `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 `str` is pointing to. Notice that the function begins the information it displays with `DirectC:` so we 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]
```

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 contains these compiler directives and in the remaining source files that VCS subsequently compiles.

Compiler directives override compile-time options.

Compiler directives are not effective down the design hierarchy. A compiler directive written above a module definition effects how VCS compiles that module definition, but does not necessarily effect how VCS compiles module definitions instantiated in that module definition. if VCS has already compiled these module definitions, it does not recompile them. If VCS has not yet compiled these module definitions, the compiler directive does effect how VCS compiles them.

Compiler Directives for Cell Definition

`` `celldefine ``

Specifies that the modules under this compiler directive be tagged as “cell” for delay annotation. See IEEE Std 1364-1995 page 219,

1364-2001 page 350. Syntax:

```
`celldefine
```

`endcelldefine

Disables `celldefine. See IEEE Std 1364-1995 page 219, 1364-2001 page 350. Syntax:

```
`endcelldefine
```

Compiler Directives for Setting Defaults

`default_nettype

Sets default net type for implicit nets. See IEEE Std 1364-1995 page 219, 1364-2001 page 350. Syntax:

```
`default_nettype wire | tri | tri0 | wand | triand
| tri1 | wor | trior | trireg |none
```

See “Disabling Default Net Data Types” on page 2-28.

`resetall

Resets all compiler directives. See IEEE Std 1364-1995 page 225, 1364-2001 page 357. Syntax:

```
`resetall
```

Compiler Directives for Macros

`define

Defines a text macro. See IEEE Std 1364-1995 pages 220-222, 1364-2001 page 351. Syntax:

```
`define text_macro_name macro_text
```

`else

Used with `ifdef. Specifies an alternative group of source code lines that VCS compiles if the text macro specified with an `ifdef compiler directive is not defined. See IEEE Std 1364-1995 pages

222-224, 1364-2001 page 353. Syntax:

```
`else second_group_of_lines
```

`elseif

Used with `ifdef. Specifies an alternative group of source code lines that VCS compiles if the text macro specified with an ‘ifdef compiler directive is not defined but the text macro specified with this compiler directive is defined. See IEEE Std 1364-1995 pages 222-224, 1364-2001 page 353.Syntax:

```
`elseif text_macro_name second_group_of_lines
```

`endif

Used with `ifdef, specifies the end of a group of lines specified by the `ifdef or `else compiler directives. See IEEE Std 1364-1995 pages 222-224, 1364-2001 page 353. Syntax:

```
`endif
```

`ifdef

Specifies compiling the source lines that follow if the specified text macro is defined by either the `define compiler directive or the +define compile-time option. See IEEE Std 1364-1995 pages 222-224, 1364-2001 page 353. Syntax:

```
`ifdef text_macro_name group_of_lines
```

The character string VCS is a predefined text macro in VCS. So in the following source code:

```
`ifdef VCS
    begin
    // Block of code for VCS
      ⋮
   end
`else
   begin
   // Alternative block of code
      ⋮
   end
```

```
`endif
```

VCS compiles and executes the first block of code and ignores the second block even when you don't include `` `define VCS `` or `+define+VCS`.

When you encrypt source code VCS inserts `` `ifdef VCS `` before all encrypted parts of the code, see Appendix B,"Source Protection".

`` `ifndef ``

Specifies compiling the source code that follows if the specified text macro is not defined. See IEEE Std 1364-1995 pages 222-224, 1364-2001 page 353. Syntax:

`` `ifndef *text_macro_name group_of_lines* ``

`` `undef ``

Undefines a macro definition. See IEEE Std 1364-1995 page 222, 1364-2001 page 351. Syntax:

`` `undef *text_macro_name* ``

Compiler Directives for Detecting Race Conditions

`` `race ``

Specifies the beginning of a region in your source code where you want VCS to look for race conditions when you include the `-Xrace=0x1` compile time option. See "The Dynamic Race Detection Tool" on page 5-10.

`` `endrace ``

Specifies the end of a region in your source code where you want VCS to look for race conditions.

Compiler Directives for Delays

`delay_mode_path
: For modules under this compiler directive that contain specify blocks, ignore the delay specifications on all gates and switches and use only the module path delays and the delay specifications on continuous assignments. Syntax:

  ```
  'delay_mode_path
  ```

`delay_mode_distributed
: Ignore the module path delays specified in specify blocks in modules under this compiler directive and use only the delay specifications on all gates, switches, and continuous assignments. Syntax:

  ```
  'delay_mode_distributed
  ```

`delay_mode_unit
: Ignore the module path delays and change all the delay specifications on all gates, switches, and continuous assignments to the shortest time precision argument of all the `'timescale` compiler directives in the source code. The default time unit and time precision argument of the `'timescale` compiler directive is 1 ns. Syntax:

  ```
  'delay_mode_unit
  ```

`delay_mode_zero
: Change all the delay specifications on all gates, switches, and continuous assignments to zero and change all module path delays to zero. Syntax:

  ```
  'delay_mode_zero
  ```

Compiler Directives for Backannotating SDF Delay

Values

`vcs_mipdexpand
When back annotating SDF delay values from an ASCII text SDF file at runtime, as specified by the +oldsdf compile-time option which disables compiling the SDF file during compilation, if the SDF file contains PORT entries for the individual bits of a port, entering this compiler directive over the port declarations for these ports enables VCS to backannotate these PORT entry delay values. Similarly, entering this compiler directive over port declarations enables a PLI application to pass delay values to individual bits of a port.

As an alternative to using this compiler directive, you can use the +vcs+mipdexpand compile-time option, or you can enter the mipb ACC capability. Example:
$sdf_annotate call=sdf_annotate_call acc+=rw,mipb:*top_level_mod*+

When you compile the SDF file, which we recommend, the back annotation of the delay values for individual bits of a port does not require this compiler directive.

`vcs_mipdnoexpand
Turns off the enabling of backannotating delay values on individual bits of a port as specified by a previous `vcs_mipdexpand compiler directive.

Compiler Directives for Source Protection

`endprotect
Defines the end of code to be protected. See Appendix B,"Source

Protection".. Syntax:

```
`endprotect
```

`endprotected

Defines the end of protected code. See Appendix B,"Source Protection". Syntax:

```
`endprotected
```

`protect

Defines the start of code to be protected. See Appendix B,"Source Protection". Syntax:

```
`protect
```

`protected

Defines the start of protected code. See Appendix B,"Source Protection". Syntax:

```
`protected
```

Compiler Directives for Controlling Port Coercion

`noportcoerce

Do not force ports to inout. See Appendix A,"VCS for the Verilog XL User". Syntax:

```
`noportcoerce
```

`portcoerce

Coerce ports as appropriate (default). See Appendix A,"VCS for the Verilog XL User". Syntax:

```
`portcoerce
```

General Compiler Directives

Compiler Directive for Including a Source File

`include

Include source file. See IEEE Std 1364-1995 pages 224-225. Syntax:

`‘include "`*filename*"

Compiler Directive for Setting the Time Scale

`timescale

Set the time scale. See IEEE Std 1364-1995 pages 225-227, 1364-2001 page 357. Syntax:

`‘timescale` *time_unit / time_precision*

In VCS the default time unit is 1 s (a full second) and the default time precision is 1 s (also a full second).

Compiler Directive for Specifying a Library

`uselib

Search library for unresolved modules. You can specify either a library file or a library directory. Syntax:

`‘uselib file =` *filename*

or

`‘uselib dir =` *directory_name* `libext+.ext | libext=.ext`

Enter path names if the library file or directory is not in the current directory. For example:

```
`uselib file = /sys/project/speclib.lib
```

If specifying a library directory, include the `libext+.`*`ext`* keyword and append to it the extensions of the source files in the library directory, just like the `+libext+.`*`ext`* compile-time option, for example:

```
`uselib dir = /net/designlibs/project.lib libext+.v
```

To specify more than one search library enter additional `dir` or `file` keywords, for example:

```
`uselib dir = /net/designlibs/library1.lib dir=/
net/designlibs/library2.lib libext+.v
```

Here the `libext+.`*`ext`* keyword applies to both libraries.

Compiler Directive for Maintaining The File Name and Line Numbers

`` `line `*`line_number`* "*`filename`*" *`level`* ``

Maintains the file name and line number. See “Maintaining The File Name and Line Numbers” on page 2-58 and IEEE Std 1364-2001 page 358.

Unimplemented Compiler Directives

The following compiler directives are IEEE std 1364-1995 compiler directives that are not yet implemented in VCS.

```
`unconnected_drive
`nounconnected_drive
```

Unsupported Compiler Directives

The following compiler directives are not IEEE std 1364-1995 or 1364-2001 compiler directives but are implemented in Verilog-XL. VCS ignores these compiler directives.

```
`accelerate
`autoexpand_vectornets
`default_rswitch_strength
`default_switch_strength
`default_trireg_strength
`disable_portfaults
`expand_vectornets
`noaccelerate_ignored
`noexpand_ventornets
`noremove_gatenames
`noremove_netnames
`nosuppress_faults
`remove_gatenames
`remove_netnames
`suppress_faults
```

System Tasks and Functions

This section describes the system tasks and functions that are supported by VCS and then lists the system tasks that it does not support.

System tasks that are described in the IEEE Std 1364-1995 or 1364-2001 are listed with the page number of the description.

System Tasks that are described in the Acellera SystemVerilog 3.1 language reference manual are listed withthe page number of the description.

SystemVerilog Assertions Severity

`$fatal`
: Generates a runtime fatal assertion error. See the Accellera SystemVerilog 3.1 LRM, page 227.

`$error`
: Generates a runtime assertion error. See the Accellera SystemVerilog 3.1 LRM, page 227.

`$warning`
: Generates a runtime warning message. See the Accellera SystemVerilog 3.1 LRM, page 227.

`$info`
: Generates an information message. See the Accellera SystemVerilog 3.1 LRM, page 227.

SystemVerilog Assertions Control

`$assertoff`
: Tells VCS to stop monitoring any of specified assertions that start at a subsequent simulation time. See the Accellera SystemVerilog 3.1 LRM, page 228.

`$assertkill`
: Tells VCS to stop monitoring any of specified assertions that start at a subsequent simulation time, and stop the execution of any of these assertions that are now occurring. See the Accellera SystemVerilog 3.1 LRM, page 228.

`$asserton`
: Tells VCS to resume them monitoring of assertions that it stopped monitoring due to a previous `$assertoff` or `$assertkill` system task. See the Accellera SystemVerilog 3.1 LRM, page 228.

SystemVerilog Assertions

`$onehot`
: Returns true if only one bit in the expression is true. See the Accellera SystemVerilog 3.1 LRM, page 228.

`$onehot0`
: Returns true if at the most one bit of the expression is true (also returns true if none of the bits are true). See the Accellera SystemVerilog 3.1 LRM, page 228.

`$isunknown`
: Returns true if one of the bits in the expression has an X value. See the Accellera SystemVerilog 3.1 LRM, page 228.

VCD Files

VCD files are ASCII files that contain a record of a net or register's transition times and values. There are a number of third party products that read VCD files to show you simulation results either during or after simulation stops. VCS has the following system tasks for specifying the names and contents of these files:

`$dumpall`
: Creates a checkpoint in the VCD file. When VCS executes this system task, VCS writes the current values of all specified nets and registers into the VCD file, whether there is a value change at this time or not. See IEEE std 1364-1995 page 209, 1364-2001 page 327.

`$dumpoff`
: Stops recording value change information in the VCD file. See IEEE std 1364-1995 page 209, 1364-2001 page 326.

`$dumpon`
: Starts recording value change information in the VCD file. See IEEE std 1364-1995 page209, 1364-2001 page 326.

`$dumpfile`
: Specifies the name of the VCD file you want VCS to record.
Syntax:
`$dumpfile("`*`filename`*`");`

`$dumpflush`
: Emptied the VCD file buffer and writes all this data to the VCD file. See IEEE std 1364-1995 page 210, 1364-2001 page 328.

`$dumplimit`
: Limits the size of a VCD file. See IEEE std 1364-1995 page 209, 1364-2001 page 327.

`$dumpvars`

Specifies the nets and registers whose transition times and values you want VCS to record in the VCD file you specify with the `$dumpfile` system task. See IEEE std 1364-1995 page 208-209, 1364-2001 page 325-326.Syntax:

```
$dumpvars(level_number,module_instance |
net_or_reg);
```

You can specify individual nets or registers or specify all the nets and registers in an instance.

`$dumpchange`

Specifies to stop recording transition times and values in the current dump file and to start recording in a new file specified by this system task. Syntax:

```
$dumpchange("filename");
```

Code example:

```
$dumpchange("vcd16a.dmp");
```

`$fflush`

VCS stores VCD data in the operating system's dump file buffer and as simulation progresses, reads from this buffer to write to the VCD file on disk. If you need the latest information written to the VCD file at a specific time you 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 VCD to write the latest information to all these files at a particular time, use the

`$fflushall` system task. Syntax:

```
$fflushall;
```

`$gr_waves`

Produces a Value-Change-Dump 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);
```

LSI Certification VCD and EVCD Files

`$lsi_dumpports`

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

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

Syntax:

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

Code example:

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

If you would rather have the `$lsi_dumpports` system task

generate a EVCD file instead, include the `+dumpports+ieee` runtime option.

`$dumpports`

For creating an extended VCD file (EVCD file) as specified in IEEE Std. 1364-2001 pages 339-340.

You can, for example, input a EVCD file into TetraMAX for fault simulation.

EVCD files are similar to the simulation history files generated by the `$lsi_dumpports` system task for LSI certification, but there are differences in the internal statements in the file and the EVCD format is a proposed IEEE standard format whereas the format of the LSI certification file is specified by LSI.

In the past the `$dumpports` and `$lsi_dumpports` system tasks were both for generating 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,
"dumports.evcd");
```

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

`$dumpportsoff`

Suspends writing to files specified in `$lsi_dumpports` or `$dumpports` system tasks. You can specify a file to which VCS suspends writing or specify no particular file, in which case VCS

suspends writing to all files specified by `$lsi_dumpports` or `$dumpports` system tasks. See IEEE Std 1364-2001 page 340-341. Syntax:
`$dumpportsoff("filename");`

`$dumpportson`
Resume writing to the file after writing was suspended by a `$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 file to which writing was halted by any `$dumpportsoff` or `$dumpports` system task. See IEEE Std 1364-2001 page 340-341. Syntax:
`$dumpportson("filename");`

`$dumpportsall`
By default VCS writes to files only when a signal changes value. The `$dumpportsall` system task records the values of the ports in the module instances specified in the `$lsi_dumpports` or `$dumpports` system task whether there is a value change on these ports or not. You can specify the file to which you want VCS to record the port values for the corresponding module instance or specify no particular file, in which case VCS writes port values for all files opened by the `$lsi_dumpports` or `$dumpports` system task. See IEEE Std 1364-2001 page 351. Syntax:
`$dumpportsall("filename");`

`$dumpportsflush`
When writing files, VCS stores simulation data in a buffer during simulation from which it writes data to the file. If you want VCS to write all simulation data from the buffer to the file or files at a particular time, execute this `$dumpportsflush` system task. You can specify the file to which you want VCS to write from the buffer or specify no particular file, in which case VCS writes all data from the buffer to all files opened by the `$lsi_dumpports` or `$dumpports` system task. See IEEE Std 1364-2001 page

342. Syntax:
`$dumpportsfush("filename");`

`$dumpportslimit`
Specifies the maximum file size of the file specified by the `$lsi_dumpports` or `$dumpports` system task. You specify the file size in bytes. When the file reaches this limit VCS no longer writes to the file. You can specify the file whose size you want to limit or specify no particular file, in which case your specified size limit applies to all files opened by the `$lsi_dumpports` or `$dumpports` system task. See IEEE Std 1364-2001 page 341-342. Syntax:
`$dumpportslimit(filesize,"filename");`

VPD Files

VPD files are files that also 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 or VirSim graphical user interface to view the simulation results that VCS recorded in a VPD file. You cannot actually load a VCD file directly into DVE or VirSim, when you load a VCD file DVE or VirSim 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 or VirSim to step through the execution of your code if you specify recording this information.

VPD files are commonly used in post-processing, where VCS writes the VPD file during batch simulation and then you review the simulation results using DVE or VirSim.

There are system tasks that specify the information that VCS writes in the VPD file.

Note:
To use the system tasks for VPD files you must compile your source code with the `-I` or `-PP` compile-time options.

`$vcdplusautoflushoff`
Turns off the automatic “flushing” of simulation results to the VPD file whenever there is an interrupt such as when VCS executes the `$stop` system task. Syntax:
`$vcdplusautofloshoff;`

`$vcdplusautoflushon`
Tells VCS to “flush” or write all the simulation results in memory to the VPD file when ever there is an interrupt such as when VCS executes a `$stop` system task or when you halt VCS using the “.” (period) CLI command, UCLI stop command, or the Stop button the DVE or VirSim Interactive window. Syntax:
`$vcdplusautofloshon;`

`$vcdplusclose`
Tells VCS to mark the current VPD file as completed, and close that file. Syntax:
`$vcdplusclose;`

`$vcdplusdeltacycleon`
Enables delta cycle recording in the VPD file for post-processing. Syntax:
`$vcdplusevent(`*`net_or_reg`*`,"`*`event_name`*`", "<E|W|I><S|T|D>");`

Displays, in VirSim, a symbol on the signal’s waveform and in the Logic Browser. The *`event_name`* argument appears in the status bar when you click on the symbol.

`E|W|I` specifies severity. `E` for error, displays a red symbol, `W` for warning, displays a yellow symbol, `I` for information, displays a green symbol.
`S|T|D` specifies the symbol shape. `S` for square, `T` for triangle, `D` for diamond.
Enter no space between the `E|W|I` and the `S|T|D` arguments. Do not include angle brackets `< >`.
There is a limit of 244 unique events.

`$vcdplusfile`
Specifies the next VPD that VirSim opens during simulation, after it executes the `$vcdplusclose` system task and when it executes the next `$vcdpluson` system task. Syntax:
`$vcdplusfile("`*`filename`*`");`

`$vcdplusglitchon;`
Turns on checking for zero delay glitches and other cases of multiple transitions for a signal at the same simulation time. Syntax:
`$vcdplusglitchon;`

`$vcdplusflush`
Tells VCS to “flush” or write all the simulation results in memory to the VPD file at the time VCS executes this system task. Use `$vcdplusautoflushon` to enable automatic flushing of simulation results to the file when simulation stops. Syntax:
`$vcdplusflush;`

`$vcdplusmemon`
Records value changes and times for memories and multi-dimensional arrays. Syntax:
`system_task``(` *`Mda`* `[,` *`dim1Lsb`* `[,` *`dim1Rsb`* `[,` *`dim2Lsb`* `[,` *`dim2Rsb`* `[, ...` *`dimNLsb`* `[,` *`dimNRsb`*`]]]]]] );`

Mda

This argument specifies the name of the MDA to be recorded. It must not be a part select. If no other arguments are given, then all elements of the MDA are recorded to the VPD file.

dim1Lsb

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

dim1Rsb

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

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

dim2Lsb

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

dim2Rsb

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

dimNLsb

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

dimNRsb

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

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

- No arguments: The whole design will be traversed and all memories and MDAs will be recorded.

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

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

For more information, see the *VirSim User Guide*.

`$vcdplusmemoff`
: Stops recording value changes and times for memories and multi-dimensional arrays. Syntax is the same as the `$vcdplusmenon` system task.For more information, see the *VirSim User Guide*.

`$vcdplusmemorydump`
: Records (dumps) a snapshot of the values in amemory or multi-dimensional array into the VPD file. Syntax is the same as the `$vcdplusmenon` system task.For more information, see the *VirSim User Guide*.

`$vcdplusoff`
: Stops recording, in the VPD file, the transition times and values for the nets and registers in the specified module instance or individual nets or registers. Syntax:
  `$vcdplusoff[(`*`level_number`*`,`*`module_instance`* `|` *`net_or_reg`*`)];`

  where:

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

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

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

`$vcdpluson`
Starts recording, in the VPD file, the transition times and values for the nets and registers in the specified module instance or individual nets or registers. Syntax:

```
$vcdpluson[(level_number,module_instance |
net_or_reg)];
```

where:

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

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

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

`$vcdplustraceoff`
Stops recording, in the VPD file, the order of statement execution in the specified module instance. Syntax:

```
$vcdplustraceoff(module_instance);
```

`$vcdplustraceon`

Starts recording, in the VPD file, the order of statement execution in the specified module instance and the module instances hierarchically under it. Syntax:

```
$vcdplustraceon[(module_instance)];
```

SystemVerilog Assertions

IMPORTANT:

Enter these system tasks in an initial block. Do not enter these system tasks in an always block.

`$assert_monitor`

Analogous to the standard `$monitor` system task in that it continually monitors specified assertions and displays what is happening with them (you can have it only display on the next clock of the assertion). Its syntax is as follows:

```
$assert_monitor([0|1,]assertion_identifier...);
```

Where:

0

Specifies reporting on the assertion if it is active (VCS is checking for its properties) and for the rest of the simulation reporting on the assertion or assertions, whenever they start.

1

Specifies reporting on the assertion or assertions only once, the next time they start.

If you specify neither 0 or 1, the default is 0.

assertion_identifier...
A comma separated list of assertions. If one of these assertions is not declared in the module definition containing this system task, specify it by its hierarchical name.

`$assert_monitor_off`
Disables the display from the `$assert_monitor` system task.

`$assert_monitor_on`
Re-enables the display from the `$assert_monitor` system task.

Executing Operating System Commands

`$system`
Executes operating system commands. Syntax:
`$system("command");`

Code example:
```
$system("mv -f savefile savefile.1");
```

`$systemf`
Executes operating system commands and accepts multiple formatted string arguments. Syntax:
`$systemf("command %s ...","string",...);`

Code example:
```
int = $systemf("cp %s %s", "file1", "file2");
```

The operating system copies the file named file1 to a file named file2.

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 file you specify. 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 the `-l` runtime option or `$log` system task. Syntax:

```
$nolog;
```

Data Type Conversions

`$bitstoreal[b]`

Converts a bit pattern to a real number.
See IEEE std 1364-1995 pages 204-205, 1364-2001 page 310.

`$itor[i]`

Converts integers to real numbers.
See IEEE std 1364-1995 pages 204-205, 1364-2001 page 310.

`$realtobits`

Passes bit patterns across module ports, converting a real number to a 64 bit representation.
See IEEE std 1364-1995 pages 204-205, 1364-2001 page 310.

`$rtoi`

Converts real numbers to integers.
See IEEE std 1364-1995 pages 204-205, 1364-2001 page 310.

Displaying Information

`$display[b|h|0];`

Display arguments.
See IEEE std 1364-1995 pages 173-179, 1364-2001 pages 278-285.

`$monitor[b|h|0]`

Display data when arguments change value.
See IEEE Std 1364-1995 pages 179-180, 1364-2001 page 286.

`$monitoroff`

Disables the `$monitor` system task.
See IEEE std 1364-1995 pages 180-183, 1364-2001 page 286.

`$monitoron`

Re-enables the `$monitor` system task after it was disabled with the `$monitoroff` system task.
See IEEE std 1364-1995 pages 180-183.

`$strobe[b|h|0];`

Displays simulation data at a selected time.
See IEEE 1364-1995 page 179., 1364-2001 page 285.

`$write[b|h|0]`

Displays text.
See IEEE std 1364-1995 pages 173 -179, 1364-2001 pages 278-285.

File I/0

`$fclose`
Closes a file.
See IEEE std 1364-1995 pages 180-182, 1364-2001 pages 286-288.

`$fdisplay[b|h|0]`
Writes to a file.
See IEEE std 1364-1995 pages 180-182, 1364-2001 pages 288-289.

`$ferror`
System function that returns additional information about an error condition in file I/O operations, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 pages 294-295.

`$fflush`
Writes buffered data to files, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 page 294.

`$fgetc`
System function that reads that reads a character from a file, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 page 290.

`$fgets`
System function that reads a string from a file, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 page 290.

`$fmonitor[b|h|0]`
Writes to a file when an argument change value.
See IEEE std 1364-1995 page 181, 1364-2001 pages 287-288.

`$fopen`
Opens files.
See IEEE std 1364-1995 pages 180-182, 1364-2001 pages 286-

288. If using the 1364-2001 implementation, see “File I/O System Tasks” on page 2-43.

`$fread`
System function that reads binary data from a file, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 page 293.

`$fscanf`
System function that reads characters in a file, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 pages 290-293.

`$fseek`
System function that sets the position of the next read or write operation in a file, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 page 294.

`$fstrobe[b|h|0]`
Writes arguments to a file.
See IEEE std 1364-1995 pages 180-182, 1364-2001 pages 288-289.

`$ftell`
System function that returns the offset of a file, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 page 294.

`$fwrite[b|h|0]`
Writes to a file.
See IEEE std 1364-1995 pages 180-1822, 1364-2001 pages 88-289.

`$rewind`
System function that sets the next read or write operation to the beginning of a file, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 page 294.

`$sformat`
System function that assigns a string value to a specified signal,

see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 pages 289-290.

`$sscanf`
: System function that reads characters from an input stream, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 pages 290-293.

`$swrite`
: System function that assigns a string value to a specified signal, similar to the `$sformat` system function, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 pages 289-290.

`$ungetc`
: System functions that returns a character to the input stream, see “File I/O System Tasks” on page 2-43. See IEEE Std 1364-2001 page 290.

Loading Memories

`$readmemb`
: Loads binary values in a file into memories.
See IEEE std 1364-1995 pages 182-183, 1364-2001 pages 295-296.

`$readmemh`
: Loads hexadecimal values in a file into memories.
See IEEE std 1364-1995 pages 182-183, 1364-2001 pages 295-296.

`$sreadmemb`
: Loads specified binary string values into memories.
See IEEE std 1364-1995 page 641, 1364-2001 page 744.

`$sreadmemh`

Loads specified string hexadecimal values into memories.
See IEEE std 1364-1995 page 641, 1364-2001 page 744.

`$writememb`

The `$writememb` system task writes binary data in a memory to a file. Syntax:

```
$writememb ("filename",memory [,start_address]
[,end_address]);
```

Code example:

```
$writememb ("testfile.txt",mem,0,255);
```

`$writememh`

The `$writememh` writes hexadecimal data in a memory to a file. Syntax:

```
$writememh ("filename",memory [,start_address]
[,end_address]);
```

Time Scale

`$printtimescale`

Displays the time unit and time precision from the last `` `timescale `` compiler directive VCS reads before it reads the module definition containing this system task.
See IEEE std 1364-1995 pages 183-184, 1364-2001 pages 297-298.

`$timeformat`

Specifies how the `%t` format specification reports time information.
See IEEE std 1364-1995 pages 184-186, 1364-2001 pages 298-301.

Simulation Control

`$stop`

Halts simulation.

See IEEE std 1364-1995 page 187, 1364-2001 pages 301-302.

`$finish`

Ends simulation.

See IEEE std 1364-1995 page 187, 1364-2001 page 301.

System Tasks for Timing Checks

`$disable_warnings`

Disables the display of timing violations but does not disable the toggling of notifier registers. Syntax:

`$disable_warnings[(`*`module_instance`*`,...)];`

An alternative syntax is:

`$disable_warnings("timing"[,`*`module_instance`*`,...]);`

If you specify a module instance, this system task disables timing violations for the specified instance and all instances hierarchically under this instance.

If you omit module instances, this system task disables timing violations throughout the design.

Code example:

```
$disable_warnings(seqdev1);
```

`$enable_warnings`

Re-enables the display of timing violations after the execution of the `$disable_warnings` system task. This system task does not enable timing violations during simulation when you used the `+no_tchk_msg` compile-time option to disable them. Syntax:

`$enable_warnings[(`*`module_instance`*`,...)];`

An alternative syntax is:

`$enable_warnings("timing"[,`*`module_instance`*`,...]);`

If you specify a module instance, this system task enables timing violations for the specified instance and all instances hierarchically under this instance.

If you omit module instances, this system task enables timing violations throughout the design.

`$hold`

Reports a timing violation when a data event happens too soon after a reference event.

See IEEE std 1364-1995 pages 188-189, 1364-2001 pages 241-242.

`$period`

Reports a timing violation when an edge triggered event happens too soon to soon after the previous matching edge triggered event on a signal.

See IEEE Std 1364-1995 page 191, 1364-2001 pages 255-256.

`$recovery`

Reports a timing violation when a data event happens too soon after a reference event. Unlike the `$setup` timing check, the reference event must include the `posedge` or `negedge` keyword. Typically the `$recovery` timing check has a control signal, such as clear, as the reference event and the clock signal as the data event.

See IEEE 1364-1995 pages 191-192. 1364-2001 pages 245-246.

`$recrem`

Reports a timing violation if a data event occurs less than a specified time limit before or after a reference event. This timing check is identical to the `$setuphold` timing check except that typically the reference event is on a control signal and the data event is on a clock signal. You can specify negative values for the recovery and removal limits. The syntax is as follows:

`$recrem(`*`reference_event, data_event,`*

recovery_limit, removal_limit, notifier, timestamp_cond, timecheck_cond, delay_reference, delay_data);

See “The $recrem Timing Check Syntax” on page 9-10 for more information. See IEEE Std 1364-2001 pages 246-248.

`$removal`

Reports a timing violation if a the reference event, typically an asynchronous control signal, happens too soon after the data event, the clock signal. See IEEE Std 1364-2001 pages 244-245.

`$setup`

Reports a timing violation when the data event happens before and too close to the reference event.

See IEEE std 1364-1995 page 188, 1364-2001 page 241. This timing check also has an extended syntax like the `$recrem` timing check. This extended syntax is not described in IEEE std 1364-1995 or 1364-2001.

`$setuphold`

Combines the `$setup` and `$hold` system tasks.

See IEEE std 1364-1995 page 189 for the official description. There is also an extended syntax that is in IEEE Std 1364-2001 pages 242-244. This extended syntax is as follows:

`$setuphold(`*reference_event, data_event, setup_limit, hold_limit, notifier, timestamp_cond, timecheck_cond, delay_reference, delay_data*`);`

See “The $setuphold Timing Check Extended Syntax” on page 9-8 for more information.

`$skew`

Reports a timing violation when a reference event happens too long after a data event.

See IEEE 1364-1995 page 191, 1364-2001 pages 249-250.

`$width`

Reports a timing violation when a pulse is narrower than the specified limit.
See IEEE std 1364-1995 page 190, 1364-2001 pages 254-255.
VCS ignores the threshold argument.

PLA Modeling

`$async$and$array` to `$sync$nor$plane`

See IEEE Std 1364-1995 pages 197-200, 1364-2001 page 302.

Stochastic Analysis

`$q_add`

Places an entry on a queue in stochastic analysis.
See IEEE Std 1364-1995 page 201, 1364-2001 page 307.

`$q_exam`

Provides statistical information about activity at the queue.
See IEEE Std 1364-1995 page 201, 1364-2001 page 307.

`$q_full`

See IEEE Std 1364-1995 page 201, 1364-2001 page 307.

`$q_initialize`

Creates a new queue.
See IEEE Std 1364-1995 page 201, 1364-2001 page 306-307.

`$q_remove`

Receives an entry from a queue.
See IEEE Std 1364-1995 page 201, 1364-2001 page 307.

Simulation Time

`$realtime`

Returns a real number time.
See IEEE std 1364-1995 page 203, 1364-2001 pages 309-310.

`$stime`

Returns an unsigned integer that is a 32 bit time.
See IEEE std 1364-1995 page 203, 1364-2001 page 309.

`$time`

Returns an integer that is a 64 bit time.
See IEEE std 1364-1995 pages 202-203, 1364-2001 pages 308-309.

Probabilistic Distribution

`$dist_exponential`

Returns random numbers where the distribution function is exponential.
See IEEE std 1364-1995 page 206, 1364-2001 page 312.

`$dist_normal`

Returns random numbers with a specified mean and standard deviation.
See IEEE std 1364-1995 page 206, 1364-2001 page 312.

`$dist_poisson`

Returns random numbers with a specified mean.
See IEEE std 1364-1995 page 206, 1364-2001 page 312.

`$dist_uniform`

Returns random numbers uniformly distributed between parameters.
See IEEE std 1364-1995 page 206, 1364-2001 page 312.

`$random`

Provides a random number.
See IEEE std 1364-1995 page 205, 1364-2001 page 312.
Using this system function in certain kinds of statements might cause simulation failure, see “Avoiding the Debugging Problems from Port Coercion” on page 2-14.

Resetting VCS

`$reset`

Resets the simulation time to 0. See IEEE std 1364-1995 pages 638-639, 1364-2001 pages 741-742.

`$reset_count`

System function that keeps track of the number of times VCS executes the `$reset` system task in a simulation session. See IEEE std 1364-1995 pages 638-639, 1364-2001 pages 741-742.

`$reset_value`

System function that you can use to pass a value from before to after VCS executes the `$reset` system task in that you can enter a *`reset_value`* integer argument to the `$reset` system task and after VCS resets the simulation the `$reset_value` system function returns this integer argument. See IEEE std 1364-1995 pages 638-639, 1364-2001 pages 741-742.

General System Tasks and Functions

Checks for a Plusarg

`$test$plusargs`

Checks for the existence of a given plusarg on the runtime

executable command-line. Syntax:
`$test$plusargs("plusarg_without_the_+");`.

SDF Files

`$sdf_annotate`
Tells VCS to back annotate delay values from an SDF file to your Verilog design. See “The $sdf_annotate System Task” on page 8-24.

Counting the Drivers on a Net

`$countdrivers`
Counts the number of drivers on a net.
See IEEE std 1364-1995 pages 635-636, 1364-2001 page 738-739.

Depositing Values

`$deposit`
Deposits a value on a net or variable. This deposited value overrides the value from any other driver of the net or variable. The value propagates to all loads of the net or variable. A subsequent simulation event can override the deposited value. You cannot use this system task to deposit values to bit-selects or part-selects.
Syntax:
`$deposit(net_or_vaiable, value);`

The deposited value can be the value of another net or variable. You can deposit the value of a bit-select or part-select.

Fast Processing Stimulus Patterns

`$getpattern`

Provides for fast process of stimulus patterns.
See IEEE std 1364-1995 page 636-637, 1364-2001 page 739.

Saving and Restarting The Simulation State

`$save`

Saves the current simulation state in a file.
See IEEE std 1364-1995 pages 639-640, 1364-2001 pages 742-743, also see “Save and Restart” on page 4-25.

`$restart`

Restores the simulation to the state saved in the check file with the `$save` system task. Enter this system task at the CLI prompt instead of in the source code. You can also do this by entering the name of the check file at the system prompt. See IEEE std 1364-1995 pages 639-640, 1364-2001 pages 742-743, also see “Save and Restart” on page 4-25.

IEEE Standard System Tasks Not Yet Implemented in VCS

The following Verilog system tasks are included in the IEEE Std 1364-1995 and 1364-2001 standards but are not yet implemented in VCS:

`$dist_chi_square`	`$dist_erlang`
`$dist_t`	`$nochange`

Nonstandard System Tasks Not Supported in VCS

Annex F of the IEEE Std 1364-1995, annex C of the 1364-2001, describes system tasks that are not part of the IEEE Verilog standard. VCS has not implemented all of the system tasks described in Annex F. These unimplemented system tasks, described in Annex F, are as follows:

```
$incsave
$input
$key
$nokey
$list
$scale
$scope
$showscopes
$showvars
```

Verilog-XL System Tasks Not Supported in VCS

There are system tasks that are implemented in Verilog-XL that are not included in the IEEE Verilog standard and are not described in Annex F of the IEEE Std 1364-1995. These system tasks, which are not implemented in VCS, are as follows:

```
$cdc
$compare
$incpattern_read
$keepcommands
$list_forces
$lm_model
$reportprofile
$showallinstances
$showmodes
$cleartrace
$enable_warnings
$incpattern_write
$limexp
$listcounts
$nokeepcommands
$settrace
$showexpandednets
$showportsnotcollapsed
```

`$showvariables`	`$startprofile`
`$stopprofile`	`$strobe_compare`

Avoiding Circular Dependency

The `$random` system function has an optional seed argument. If you use this argument and 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);
```

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 messages 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 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 with 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 `$lsi_dumpports` with ATPG tools. ATPG tools many times strictly follow port direction and do not allow unidirectional ports to be driven from within the device. If care is not taken when writing the test fixture, the results of `$lsi_dumpports` will cause problems for ATPG tools.

Note:

Readers need to reference Signal Value/Strength Codes based on TSSI Standard Events Format State Character set at end of application note.

Dealing with Unassigned Nets

Lets use 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 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, it 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 uncommenting the line, // assign A = 1'bz;, in the above code. Now when 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 intended by the user. As a result, `$lsi_dumpports` makes an evaluation for drivers when all of the users intended assignments haven't been made. To correct this situation, the user needs to advance simulation time just enough to have their assignments take place, this can be accomplished by adding a #1 before `$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, the user has not specified an instance name for `$lsi_dumpports`. The syntax for `$lsi_dumpports` calls for an instance name. Since the user didn't instantiate module top in the test fixture, they are left specifying the MODULE name top. This will produce a warning message from VCS. Since top appears only once, that instance will be assumed.

The second problem comes from the cross module reference (XMR) used by the force command. Since the module test didn't instantiate top, the user had to use 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, the user needs to instantiate top inside of test, and drive the signal a from within test. This is done in the following way:

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

```
top u0 (a);
endmodule

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

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

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

Signal Value/Strength Codes

The enhanced state character set is expected to base on TSSI Standard Events Format State Character set with additional expansion to include more unknown states. The supported character set is described below

Test Bench 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 Test Bench)

L	low
H	high
X	unknown (don’t care)

T	tristate
I	low (2 or more DUT drivers active)
Test Bench 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)

Generating SAIF Files

Both Power Compiler and VCS generate SAIF (Switching Activity Interchange Format) files that you pass back and forth between VCS and Power Compiler so that Power Compiler can do power analysis and power optimization. You can do this during both RTL and gate-level simulation. These files are as follows:

- Power Compiler generates forward-annotation SAIF files that VCS reads. There are different types for RTL and gate-level simulation.
- VCS generates back-annotation SAIF files that Power Compiler reads.

See the *Power Compiler User Guide* and *Power Compiler Reference Manual* for information on how Power compiler generates and uses these SAIF files.

You use system tasks to tell VCS to generate back-annotation SAIF files. VCS automatically recognizes these system tasks, there are no compile-time options required to enable the recognition of these tasks. In earlier releases you used a PLI application to generate these SAIF files.

Note:

To enter these system tasks at the CLI prompt or in the Command field of the VirSim Interactive window, use the `+cli` compile-time option with the `2`,`3`, or `4` argument, for example `+cli+2`.

These system tasks are as follows:

`$set_toggle_region`

Specifies a signal or a module instance for which VCS records switching activity in the generated SAIF file. If you specify a module instance VCS records the signals in that instance. Syntax:

```
$set_toggle_region(signal_or_instance
[, signal_or_instance]);
```

`$toggle_start`

Tells VCS to start monitoring the switching activity in the instance specified by the previous `$set_toggle_region` system task. Syntax:

```
$toggle_start();
```

`$toggle_stop`

Tells VCS to stop monitoring switching activity in the instance specified by the previous `$set_toggle_region` system task. Syntax:

```
$toggle_stop();
```

`$toggle_reset`

Sets the toggle counter to 0 for all the nets in the instance specified in the previous `$set_toggle_region` system task. Syntax:

```
$toggle_reset();
```

`$toggle_report`

Reports switching activity that VCS has monitored in the specified back-annotation SAIF file. Syntax:

```
$toggle_report("back-annotation_SAIF_file",
synthesis_Time_Unit, module_instance);
```

Where:

`back-annotation_SAIF_file`

Is the SAIF file that VCS generates.

`synthesis_Time_Unit`

This optional real-number parameter is the time unit of your synthesis library. Express this time unit in scientific notation for seconds. For example, if your synthesis library's time unit is 10 picoseconds, enter `1.0e-11` for this argument.

`module_instance`

Specifies the instance in the design whose switching activity is recorded in the SAIF file.

`$set_gate_level_monitoring`

Turns on or off the monitoring of nets in the design. If you enter this system task, do so before the `$set_toggle_region` system task. Syntax:

```
$set_gate_level_monitoring("on" | "off" |
"rtl_on");
```

Where:

`on`
VCS monitors nets. VCS monitors a reg only if it is also an output port. This is the default monitoring policy in gate-level simulation.

`off`
VCS does not monitor net data types. This is the default monitoring policy in RTL simulation.

`rtl_on`
VCS monitors all reg data types during RTL simulation. VCS monitors net data types only if they are ports.

`$read_lib_saif`
Reads in a forward-annotation SAIF file for gate-level simulation. This file enables VCS to register the state dependent and path dependent (SDPD) information in the in the instance specified in the previous `$set_toggle_region` system task. Syntax:
`$read_lib_saif("`*`forward-annotation_SAIF_file`*`");`

`$read_rtl_saif`
Reads in a forward-annotation SAIF file for RTL simulation. Syntax:
`$read_rtl_saif("`*`forward-annotation_SAIF_file`*`"`
`[,` *`testbench_path_name`*`]);`

Where:

`forward-annotation_SAIF_file`
This file enables VCS to register synthesis invariant elements. By default, it doesn't register nets in the output back-annotation SAIF file.

`testbench_path_name`
The hierarchical name of the instance for which the forward-annotation SAIF file was written. Synopsys recommends that you always include this argument.

3

Compiling Your Design

This chapter describes the compilation process in detaul. Its sections include:

- Incremental Compilation, which includes shared incremental compilation, and a discussion of how a group of designers can use incremental compilation.
- The Direct Access Interface Directory.
- Compile-Time Options
- Performance Considerations
- Performance Considerations, including the practices that speed up compilation and the compile-time options that speed up or slow down compilation and simulation.

Incremental Compilation

VCS compiles your source code on a module by module basis. By default VCS re-compiles only the modules that have changed since last compilation, this is Incremental compilation.

Incremental compilation creates a subdirectory named csrc to store generated files from compilation. The generated files are as follows:

- The object file output from compilation. VCS links these files with the simulation engine to create the simv executable. If you are using native code generation, available only on Solaris, HP, and Linux, VCS generates these files directly.
- A Makefile that controls the compilation process.
- If you are not using native code generation, the intermediate C or assembly files that VCS generates for the modules in your design and then compiles into object files.
- In native code generation, files that describe the modules in the design. If you are not using native code generation, the descriptor information for the modules in your design is in comment lines in the intermediate C or assembly code files.

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

Compile-time options that affect incremental compilation all begin with "`-M`" see the “Incremental Compilation” on page 3-23.

We recommend that you avoid making changes to the contents of the csrc directory. However, some users may find it necessary to change the Makefile to fit the local environment; for example, to invoke a distributed C compile. If you don't want VCS to update (overwrite) the Makefile, enter the `-Mupdate=0` compile-time option and argument.

Triggering Recompilation

VCS re-compiles a module when you change its contents. The following conditions will also trigger re-compile a module:

- VCS version has changed
- Command-line options have changed
- Target of a hierarchical reference has changed
- Ports of a module instantiated in the module have changed
- Calls from a `$dumpvars` system task into the module have changed
- A compile time constant such as a parameter has changed

The following do not cause VCS to re-compile a module:

- Change of time stamp of any source file
- Change in filename or grouping of modules in any source file
- Unrelated change in another module in the same source file
- Nonfunctional changes such as comments or white space

Using Shared Incremental Compilation

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

To invoke shared incremental compilation your team uses the following compile-time options:

`-Mlib=`*`dir`*
: This option provides VCS with a central place to look for the descriptor information before it compiles a module and a central place to get the object files when it links together the executable. This option allows you to use the parts of a design that have been already tested and debugged by other members of your team without recompiling the modules for these parts of a design.

`-Mdir=`*`dir`*
: Specifies the pathname of a central directory where you want VCS to write the generated files such as option files. This option allows you to provide other members of your team with the generated files for new or improved modules in a large design so that other members don't have to compile those modules.

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

Alice is responsible for debugging the chip design and creating a shared design that Wally can use to debug his board design without having to recompile Alice's chip design.

Alice also needs to know what debug features Wally needs in the chip design. She meets with Wally and learns that Wally needs to dump the signals in a memory submodule in her chip design, so Alice adds a `$dumpvars` system task to her design:

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

Alice now compiles her design using the following vcs command line:

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

Alice includes the `-Mdir` option specifying a directory pathname into which VCS writes the generated files. Alice made sure that Wally could access this directory.

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

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

VCS does not recompile the chip design because Wally included the `-Mlib` option telling VCS where to look for descriptor information to compare to the chip design source files and this descriptor information told VCS that it didn't need to recompile the chip design. VCS used the generated files in this directory to create the simv executable for simulating the board design

Wally included the `-Mdir` option to specify a directory where VCS wrote the generated files for the board design but not the chip design. VCS also used the generated files in this directory to create the simv executable for simulating the board design. If he omitted this option, VCS would have written the generated files in a csrc directory in his current directory.

Wally and Alice used the same version of VCS. If Wally used a different version from the one used by Alice, VCS would recompile the chip design.

The Direct Access Interface Directory

When you compile your design with certain options VCS creates a Direct Access Interface directory in the same location as the simv executable file.

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

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

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

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

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

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

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

This section explains both of these methods.

Note the following:

- 64-bit machines have more capacity than 32-bit machines, but there is a performance trade-off.
- The 64-bit compilation and 64-32-bit cross-compilation processes are not available on all platforms, and not all features and capabilities of VCS work with them. For example, these features do not currently work with VCS MX. Specific requirements can change with each release so this information is documented in the *VCS Release Notes* ($VCS_HOME/Doc/ReleaseNotes).

- You can include PLI code in either the 64-32-bit cross-compilation or full 64-bit compilation processes. PLI code compiled with gcc generally works fine without any restrictions.

Identifying the Source of Memory Consumption

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

How VCS Indicates a Memory Problem

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

```
Error: out of virtual memory (swap space)
(v2ss1_16384x64cm16_func.v line 1053458)
Error: malloc(1937887600) returned 0: Not enough space


Doing SDF annotation .......
Error: malloc(400) returned  0: Not enough space


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


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

If you encounter one of these error messages, there are several alternative methods you can try that might help adapt to the memory requirements of your design. These methods, described briefly in the next section, Minimizing Memory Consumptionapply to cases in which you are simulating large, flat, gate-level designs with no timing or timing checks. If this is not the case, then you should proceed to the methodology described in ‘Running a 64-Bit Compilation and Simulation” on page 3-12.

Minimizing Memory Consumption

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

- Check the physical memory on your workstation. You can use the `top` utility for this. All current operating systems support 4 GB of memory, including Linux. Make sure you are using a machine with enough available memory (i.e. even though your machine may appear to have plenty of memory, if there are other processes concurrently running on that machine, you won't have access to the entire memory).
- Avoid using debug options like `+cli`, `-line`, `+acc` or any `-X...` switches.
- In the pli.tab files, beware of `acc` calls that call for global access (i.e. acc=rw:*)
- Minimize the amount of dumping. Instead of dumping the whole design, try to limit the scope of dumping to particular modules. Note that an overhead is still incurred if dumping is compiled in even though it may not be enabled until run-time using `$test$plusargs`.

- If dumping to a VPD file, use `+nocelldefinepli+`*`n`* to limit dumping to non-library modules.

For details on using these methods, see the application note "Guidelines for Simulating Large, Flat, Gate-Level Designs in VCS," available on Synopsys' SolvNet website.

Running a 64-32-Bit Cross-Compilation

If you tried the methods described in the previous section, Minimizing Memory Consumptionand yet continue to encounter compile-time memory consumption issues, then your next step is to try the 64-32-bit cross-compilation process. This process compiles a design using a 64-bit address range, which allows the compilation process to go beyond the 4 GB limit of a 32-bit application. The simv produced by this process is a 32-bit executable, which enables you to use your existing 32-bit PLI code and third-party applications during the simulation.

Setting up the Compiler and Linker

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

Memory Set-Up

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

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

```
% top

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

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

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

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

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

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

Specifying the Compiler, Linker, and -comp64 Option

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

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

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

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

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

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

```
vcs -comp64 Verilog_source_files
```

Running a 64-Bit Compilation and Simulation

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

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

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

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

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

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

```
vcs -full64 Verilog_source_files
```

Optimizations To Reduce Memory Consumption

If there is a very large module definition in a flat gate-level design memory consumption could be very high. If this is a problem for you one possible solution is the `+memopt` compile-time option. This option enables a number of optimizations that reduce the amount of memory needed during compilation. You should bear in mind that there could be a runtime performance cost using this compile-time option.

If you find that you are using up too much memory even when you include the `+memopt` option, that you are reaching the per process memory limits imposed by the system on the compilation process, try the `+memopt+2` compile-time option. The `+memopt+2` option spawns a second process for some of the memory optimizations.

When you use this option you will see entries in the compilation log file that tell you if these optimizations occurred. Check the log file after compilation.

You cannot include the `+memopt` compile-time options when you include any of the following compile-time options:

`+ad=`*`partition_file`* `+mhdl` `+race` `+racecd`

Initializing Memories and Regs

VCS has compile-time options for initializing all bits of memories and regs to the 0, 1, X, or Z value:

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

Initializes all bits of all memories in the design.

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

Initializes all bits of all regs in the design.

The `+vcs+initmem` option initializes regular memories and multi-dimensional arrays of the `reg` data type with more than two dimensions, for example:

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

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

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

When you use these options, to prevent race conditions, you should avoid the following :

- Assigning initial values to a regs in their declaration when the value assigned is not the same as the value specified with the `+vcs+initreg` option, for example:

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

- Assigning values to regs or memory elements at simulation time 0 when the value assigned is not the same as the value specified with the `+vcs+initreg` or `+vcs+initmem` option, for example:

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

Allowing Inout Port Connection Width Mismatches

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

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

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

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

Without the `+noerrorIOPCWM` compile-time option, VCS displays following error message:

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

VCS does not create the simv executable.

If you conclude the +noerrorIOPCWM compile-time option, VCS displays the following:

```
Warning-[IOPCWM] Inout Port connection width mismatch.
Connecting inout ports to
     mismatched width nets has unpredictable results and will
     not be permitted in future releases.
     The following 8-bit expression is connected to 16-bit
     port "pote" ofmodule "dev", instance "dev1".
     Expression: w1
     "exp1.v", line_number
```

VCS does create the simv executable.

Using Lint

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

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

This message is displayed because an entire vector reg, instead of a bit-select, was attached to an input terminal of a gate. In this message the text string GCWM is the ID of the message. You use the ID to enable or disable the message.

The syntax of the +lint option is as follows:

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

Where:

`no`	Specifies disabling lint messages with 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 specify exceptions.
`all`	Specifies enabling all lint 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:

`+lint=all,noGCWM`	Enables all lint messages except the message with the GCWM ID
`+lint=NCEID`	Enables the lint message with the NCEID ID.
`+lint=GCWM,NCEID`	Enables the lint messages with the GCWM and NCEID IIDs.
`+lint=none`	Disables all lint messages. This is the default.

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

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

```
Lint-[IRIMW] Illegal range in memory word

Lint-[NCEID} Non-constant expression in delay

Lint-[GCWM] Gate connection width mismatch

Lint-[CAWM] Continuous Assignment width mismatch

Lint-[IGSFPG] Illegal gate strength for pull gate
```

```
Lint-[TFIPC]  Too few instance port connections
lint-[IPDP] Identifier previously declared as port
lint-[PCWM] Port connect width mismatch
```

Changing Parameter Values From The Command Line

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

- `-pvalue`
- `-parameters`

The `-pvalue` option has the following syntax:

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

For example:

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

Note:
Do not enter the `-pvalue` option with the options you enter with the `-verilogcomp scs` compilation option in VCS MX.

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
```

Where:

`assign`
Is the keyword that starts a line in the file.

`value`
Is the new value of the parameter.

`path_to_the_parameter`
Specifies the 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 contests 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`.

Making Accessing an Out of Range Bit an Error Condition

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

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

There is no bit 2 in the declaration of reg r1.

VCS displays a warning but continues to compile the code and link together the simv executable. The following is an example of this warning message:

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

You can tell VCS to make accessing a bit or element that is outside the declared range to be an error condition, so VCS does not create the new simv executable, with the `+vcs+boundscheck` compile-time option. The following is an example of the error message VCS displays when you access an undeclared bit and enter this option:

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

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

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

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

Compile-Time Options

This section describes compile-time options you enter on the VCS command line and runtime options you can enter at compile time.

Accessing Verilog Libraries

`-v` *`filename`*

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

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

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

`+incdir+`*`directory`*`+`

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

`+libext+`*`extension`*`+`

Specifies that VCS only search the files in a library directory with the specified filename extensions. You can specify more than one extension, separating the extensions with a + character. For example, `+libext+.v+.V+` specifies searching the files in a library with either the .v or .V extension.

The order in which you add filename extensions to this option does not specify an order in which VCS searches files in the library with these filename extensions.

`+liborder`

Specifies searching for module definitions for unresolved module instances through the remainder of the library were 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 beginning to search libraries for module definitions for unresolved module instances beginning with the first library on the vcs command line.

+`libverbose`

Tells VCS to display a message when it finds a module definition in a source file in a Verilog library directory that resolves a module instantiation statement that VCS read in your source files, a library file, or in another file in a library directory. The message is as follows:

```
Resolving module "module_identifier"
```

VCS does not display this message when it finds a module definition in a Verilog library file that resolves a module instantiation statement.

Incremental Compilation

`-Marchive=`*`number_of_module_definitions`*

By default VCS compiles module definitions into individual object files and sends all the object files in a command line to the linker. Some platforms use a fixed-length buffer for the command line and if VCS sends too long a list of object files this buffer overflows and the link fails. A solution to this problem is to have the linker create temporary object files containing more than one module definition so there are fewer object files on the linker command line. You enable creating these temporary object files, and specify how many module definitions are in these files, with this option. Using this option briefly doubles the amount of disk space used by the linker because the object files containing more than one module definition are copies of the object files for each module definition. After the linker creates the simv executable it deletes the temporary object files.

`-Mupdate[=0]`

By default, VCS overwrites the Makefile between compilations. If you wish to preserve the Makefile between compilations, enter this option with the 0 argument.

Entering this option without the 0 argument, specifies the default condition, incremental compilation and updating the Makefile.

`-Mdefine=`*`name`*`=`*`value`*

Add a definition to the Makefile.

`-Mdelete`

Use this option for the rare occurrence when the `chmod -x simv` command in the make file can't change the permissions on an old simv executable. This option replaces this command with the `rm -f simv` command in the make file.

`-Mdirectory=`*`directory`*

Specifies the incremental compile directory. The default name for this directory is csrc, and its default location is your current directory. You can substitute the shorter `-Mdir` for `-Mdirectory`

`-Mfilename=`*`prefix`*

Base name (prefix) for C source and object files.

`-Minclude=`*`filename`*

Adds an include statement to the Makefile.

`-Mldcmd=`*`value`*

Format string used to invoke linker directly.

`-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 a design, see “Using Shared Incremental Compilation” on page 3-4.

You can specify more than one place for VCS to look for this descriptor information and object files. You can enter multiple entries of this option, for example:

```
vcs design.v -Mlib=/design/dir1 -Mlib=/design/dir2
```

Or you can specify more than one directory with this option, using a colon (:) as a delimiter between them, for example:

```
vcs design.v -Mlib=/design/dir1:/design/dir2
```

`-Mloadlist=`*`value`*

If *`value`* is `Yes`, directly invoke linker to link programs.

`-Mmakefile=`*`filename`*

Names the generated Makefile (default is Makefile).

`-Mmakeprogram=`*`program`*

Specifies the make program that analyzes dependencies, compiles what needs compiling, links, builds, and does the necessary steps to produce the executable file that you run to simulate your design with VCS.

Commonly used make programs include the make program that comes with your operating system and gmake from the GNU foundation. The make program from you operating system is the default make program.

If you are including options for the make program then include the entire argument in quotation makes.
Also you can shorten this option to `-Mmakep`.

The following example show including the `-j` option to the gmake program to specify parallel compilation:

```
-Mmakep="gmake -j 4"
```

In this example the 4 argument to the `-j` option for gmake specifies using four CPUs for parallel compilation. You can also use the VCS `-j`*`number_of_CPUs`* compile-time option to do this.

`-Mrelative_path`

Use this option if your linker has a limitation on the length of the linker line in the make file. If you specify a relative path with the `-Mlib` option, the `-Mrelative_path` option does not expand the relative path to an absolute path on the linker line in the make file.

`-Msrclist=`*`filename`*

Name of source list file that lists all object files created by VCS (default is filelist).

Help and Documentation

`-h` or `-help`

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

`-doc`

Starts a PDF file reader to display the PDF file for the VCS Documentation Navigator . This option tells VCS to execute the following system command:

```
"$PDF_READER $VCS_HOME/doc/UserGuide/navigator.pdf"
```

By default PDF_READER is set to the first acroread executable (the Adobe Acrobat reader) that it finds in the directories specified for your PATH environment variable. You can set the PDF_READER environment variable to other PDF file display tools, for example xpdf on Linux.

SystemVerilog

`-sverilog`

Enables the etensions to the Verilog language in the Accellera SystemVerilog specification.

`-ignore` *`keyword_argument`*

The keyword arguments are as follows:

`unique_checks`
Suppresses warning messages about `unique if` and `unique case` statements.

`priority_checks`
Suppresses warning messages about `priority if` and `priority case` statements.

`all`
Suppresses warning messages about `unique if`, `unique case`, `priority if` and `priority case` statements.

`-assert` *`keyword_argument`*

The keyword arguments are as follows:

`enable_diag`
Enables further control of results reporting on SystemVerilog assertions with runtime options.

`filter_past`
For assertions that are defined with the `$past` system task, ignore these assertions when the past history buffer is empty. For instance, at the very beginning of the simulation the past history buffer is empty. So the first sampling point and subsequent sampling points should be ignored until the past buffer has been filled with respect to the sampling point.

`dumpoff`
Disables the dumping of SVA information in the VPD file during simulation.

`-sv_pragma`
Tells VCS to compile the SystemVerilog Assertions code that

follows the `sv_pragma` keyword in a single line or multi-line comment.

OpenVera Native TestBench

`+debug_all`
Enables you to use the OpenVera testbench GUI.

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

`-ntb_cmp`
Compiles and generates the testbench shell (*file*.vshell) and shared object files. Use this option when compiling the .vr file separately from the design files.

`-ntb_define` *`macro`*
Specifies any OpenVera macro name on the command line. You can specify multiple macro names using the + delimiter.

`-ntb_filext .`*`ext`*
Specifies an OpenVera filename extension. You can specify multiple filename extensions using the + delimiter.

`-ntb_incdir` *`directory_path`*
Specifies the include directory path for OpenVera files. You can specify multiple include directories using the + delimiter.

`-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:

`check`
Reports error, during compilation or simulation, when there is an out-of-bound or illegal array access.

`dep_check`
Enables dependency analysis and incremental compilation. Detects files with circular dependencies and issues an error message when VCS cannot determine which file to compile first.

`no_file_by_file_pp`
By default, VCS does file by file preprocessing on each input file, feeding the concatenated result to the parser. This argument disables this behavior.

`print_deps[=`*`filename`*`]`
Enter this argument with the `dep_check` argument. This argument tells VCS to display the dependencies for the source files on the screen or in the file that you specify.

`tb_timescale=`*`value`*
Specifies an overriding timescale for the testbench. The timescale is in the Verilog format (for example, `10ns/10ns`).

`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].

You can enter more than one keyword argument, using the `+` delimiter, for example:

```
-ntb_opts use_sigprop+vera_portname
```

`-ntb_shell_only`
Generates only a .vshell file. Use this option when compiling a testbench separately from the design file.

`-ntb_sfname` *`filename`*
Specifies the filename of the testbench shell.

`-ntb_sname` *`module_name`*
Specifies the name and directory where VCS writes the testbench shell module.

`-ntb_spath`
Specifies the directory where VCS writes the testbench shell and shared object files. The default is the compilation directory.

`-ntb_vipext .`*`ext`*
Specifies an OpenVera encrypted-mode file extension to mark files for processing in OpenVera encrypted IP mode. Unlike the `-ntb_filext` option, the default encrypted-mode extensions .vrp, .vrhp are not overridden, and will always be in effect. You can pass multiple file extensions at the same time using the `+` delimiter.

`-ntb_vl`
Specifies the compilation of all Verilog files, including the design, the testbench shell file and the top-level Verilog module.

Different Versions of Verilog

`+systemverilogext+`*`ext`*
Specifies a filename extension for SystemVerilog source files. If you use a different filename extesion for the SystemVerilog part of your source code, and this option, you can omit the `-sverilog` option.

`+verilog2001ext+`*`ext`*

Specifies a filename extension for Verilog 2001 source files. If you use a different filename extesion for the Verilog 2001 part of your source code, and this option, you can omit the `+v2k` option.

`+verilog1995ext+`*`ext`*

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

Note:

You cannot enter all three of these options on the same command line.

Initializing Memories and Regs

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

Initializes all bits of all memories in the design, see “Initializing Memories and Regs” on page 3-13.

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

Initializes all bits of all regs in the design, see “Initializing Memories and Regs” on page 3-13.

Using Radiant Technology

`+rad`

Performs Rqdiiant Technology optimizations on your design.

`+optconfigfile+`*`filename`*

Specifies a configuration file that lists the parts of your design you want to optimize (or not optimize) and the level of optimization for

these parts. You can also use the configuration file to specify two state simulation and ACC write capabilities. See “Applying Radiant Technology to Parts of The Design” on page 6-3.

64 bit Compilation

`-full64`

Enables compilation and simulation in 64 bit mode, see “Optimizations To Reduce Memory Consumption” on page 3-13.

`-comp64`

Enable 64 bit compilation that generates a 32 bit simv executable, see “Optimizations To Reduce Memory Consumption” on page 3-13.

Two State Simulation

`+2state`

Enables 2-state simulation to achieve higher performance without having to alter designs or design methodology. See“Specifying Two State Simulation” on page 13-27.

`+warn2val`

Enables the display of warning messages about situations that may not be suitable for 2-state simulation. See “Specifying Two State Simulation” on page 13-27.

`+noinitnegedge`

In two state simulation a falling-edge-sensitive event control for a net in the “sensitivity list” position on an always block, in some situations, causes VCS to execute the always block at time 0, whereas VCS does not do so in four state simulation. To stop the execution of the always block at time 0 in two state simulation,

include the `+noinitnegedge` compile-time option. See “Specifying Two State Simulation” on page 13-27.

`+optconfigfile+`*`filename`*

Specifies a configuration file that lists the parts of your design you want to run in two and four state simulation. You can also use the configuration file to specify Radiant technology optimizations and ACC write capabilities.

Debugging

`-line`

Enables source-level debugging tasks such as stepping through the code, displaying the order in which VCS executed lines in your code, and the last statement executed before simulation stopped. Typically you enter this option with a `+cli` option, for example `vcs +cli+1 -line`

`+cli+[`*`module_identifier`*`=]`*`level_number`*

Enables command line interface (CLI) interactive debugging commands.

Using this option creates the Direct Access Interface Directory, by default named simv.daidir, in the directory where VCS creates the executable file. See “The Direct Access Interface Directory” on page 3-6.

The level number enables more and more commands. The level number can be any number between 1 and 4:

`+cli+1` or `+cli`

Enables commands that display the value of nets and registers and deposit values on registers

`+cli+2`

Same as above, plus enable callbacks on signals, that is commands that do something when there is a value change, for example

`break` *`signal_name`*.

`+cli+3`

Same as above, plus enable force and release of nets, for example

`force` *`net`*`=0`

This level number does not enable forcing values on registers.

`+cli+4`

Same as above, plus enables force and release of registers.

Using these options also creates the Direct Access Interface Directory, by default named simv.daidir, in the directory where VCS creates the executable file.

You can include a module identifier (name) in this option to enable a level of CLI commands for all instances of the module. If you want to enable CLI commands in other modules, enter another `+cli` option, for example:

```
vcs source.v +cli+dev1=4 +cli+dev2=4
```

This command line enables level four CLI commands in all instances of modules dev1 and dev2.

`+cliedit`

In UNIX, enables tcsh-like CLI mode. This mode enables you to use the GNU command line editing interface for entering CLI commands. With this mode, for example, entering Control - p displays the previous CLI command at the CLI prompt.

To use this option you first must download a zipped tar file from

our FTP site and unzip, extract this tar file, and set the GNURL_HOME environment variable. You may want to install this file to a location accessible to all users at your site. To do so:

1. Enter on a command line FTP FTP.synopsys.com
2. At the FTP Name prompt enter `anonymous`.
   This enables a guest login.
3. Enter your E-mail address as your password.
4. Enter the FTP command `cd TOOLS` to move to the directory that contains the tar file.
5. Enter the FTP command `bin`.
6. Enter the FTP command `get readline-2.0.tar.gz`. This downloads the file to your current directory.
7. Enter the FTP command `quit` to exit FTP.
8. Unzip the tar file by entering `gunzip readline-2.0.tar.gz`.
9. Extract the tar file by entering, for example:
   `tar xvf readline-2.0.tar`
   This creates the readline-2.0 directory. In that directory is the make_gnurl script.
10. Execute the make_gnurl script.
    This script builds the gnurl.o and other object files that you need for this option. It creates a subdirectory that it names after the UNIX platform you use, for example, sparc. It writes the gnurl.o and other object files in that subdirectory.

11. Set the GNURL_HOME environment variable to the subdirectory created by the script, for example:
    `setenv GNURL_HOME /u/eng/readline-2.0/sparc`
    Instead of setting this environment variable you can copy the gnurl.o and other object files in the subdirectory to the lib directory in your VCS installation.

    Documentation for this mode is included in the readline-2.0/doc directory.

`+acc+`*`level_number`*

Old method to enable PLI ACC capabilities for the entire design. Using this option is not recommended. For the new method, see “Specifying ACC Capabilities for VCS Debugging Features” on page 7-17.

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`*

Used in a subsequent simulation, this option re-compiles your design to enable only the ACC capabilities that you needed for the debugging operations you did, such as applying breakpoints to signals or stepping through the code in certain parts of your design, during a previous simulation of the design.

ACC capabilities enable debugging operations but slow down

your simulation so you only want to apply them where you need them.

You record where you last used them, in a previous simulation, with the `+vcs+learn+pli` runtime option.

The `+vcs+learn+pli` runtime option records where you used ACC capabilities in a file named pli_learn.tab. If you do not change the file's name or location, you can omit `+`*`filename`* from the `+applylearn` compile-time option.For more information, see "Using Only The ACC Capabilities That You Need" on page 7-25.

Finding Race Conditions

`+race`

Specifies that during simulation VCS generate a report of all the race conditions in the design and write this report in the race.out file. See "The Dynamic Race Detection Tool" on page 5-10.

`+race=all`

Analyzes the source code during compilation to look for coding styles that cause race conditions. See "The Static Race Detection Tool" on page 5-22

`+racecd`

Specifies that during simulation VCS generate a report of the race conditions in the design between the `'race` and `'endrace` compiler directives and write this report in the race.out file. See "The Dynamic Race Detection Tool" on page 5-10.

`+race_maxvecsize=`*`size`*

Specifies the largest vector signal for which the dynamic race detection tool looks for race conditions.

Starting Simulation Right After Compilation

`-R`
: Run the executable file immediately after VCS links together the executable file.

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

`-i` *`filename`*
: Specifies a file containing CLI commands that VCS executes when simulation starts. After the end of that file is reached, input commands are taken from the standard input. This option works only when you also include the `-R` and `-s` options. It is normally entered with the `+cli+`*number* compile-time option. This option is also accepted by the output simv executable; it is really a runtime option but it is frequently entered on the vcs command line.

Compiling OpenVera Assertions (OVA)

The following compile-time options are for OpenVera Assertions:

`-ovac`
: Starts the OVA compiler for checking the syntax of OVA files that you specify on the vcs command line. This option is for when you first start writing OVA files and need to make sure that they can compile correctly.

`-ova_cov`
: Enables viewing results with functional coverage.

`-ova_cov_events`
Enables coverage reporting of expressions.

`-ova_cov_hier` *`filename`*
Limits functional coverage to the module instances specified in *filename*.

`-ova_debug | -ova_debug_vpd`
Required to view results with DVE or VirSim.

`-ova_file` *`filename`*
Identifies *filename* as an assertion file. Not required if the file name ends with .ova. For multiple assertion files, repeat this option with each file.

`-ova_filter_past`
For assertions that are defined with the past operator, ignore these assertions where the past history buffer is empty. For instance, at the very beginning of the simulation the past history buffer is empty. So, a check/forbid at the first sampling point and subsequent sampling points should be igorned until the past buffer has been filled with respect to the sampling point.

`-ova_enable_diag`
Enables further control of result reporting with runtime options.

`-ova_inline`
Enables compiling of OVA code that is written inline with a Verilog design.

`-ova_lint`
Enables general rules for the OVA linter.

`-ova_lint_magellan`
Enables Magellan rules for the OVA linter.

Compiling For Simulation With Vera

`-vera`
Specifies the standard VERA PLI table file and object library.

`-vera_dbind`
Specifies the VERA PLI table file and object library for dynamic binding.

Compiling For Coverage Metrics

For more detailed information on these options see the *VCS /VCS MX Coverage Metrics User Guide*.

`-cm line|cond|fsm|tgl|path|branch|assert`
Specifies compiling for the specified type or types of coverage. The arguments specifies the types of coverage:

`line`
Compile for line or statement coverage.

`cond`
Compile for condition coverage.

`fsm`
Compile for FSM coverage.

`tgl`
Compile for toggle coverage.

`path`
Compile for path coverage.

`branch`
Compile for branch coverage

`assert`
Compile for SystemVerilog assertion coverage.

If you want VCS to compile for more than one type of coverage, use the plus (+) character as a delimiter between arguments, for example:

```
-cm line+cond+fsm+tgl
```

The `-cm` option is also a runtime option and an option on the cmView command line.

`-cm_assert_hier` *`filename`*
Limits assertion coverage to the module instances specified in *`filename`*. Specify the instances using the same format as VCS coverage metrics. If this option is not used, coverage is implemented on the whole design.

`-cm_cond` *`arguments`*
Modifies condition coverage as specified by the argument or arguments:

`basic`
Only logical conditions and no multiple conditions.

`std`
The default: only logical, multiple, sensitized conditions.

`full`
Logical and non-logical, multiple conditions, no sensitized conditions.

`allops`
Logical and non-logical conditions.

`event`
Signals in event controls in the sensitivity list position are conditions.

`anywidth`
Enables conditions that need more than 32 bits.

`sop`
Specifies condition SOP coverage.

`for`
Enables conditions in for loops.

`tf`
Enables conditions in user defined tasks and functions.

`sop`
Tells VCS that when it reads conditional expressions that contain the `^` bitwise XOR and `~^` bitwise XNOR operators, it reduces the expression to negation and logical AND or OR.

You can specify more than one argument. If you do use the `+` plus delimiter between arguments, for example:

```
-cm_cond basic+allops
```

`-cm_count`
Enables cmView to do the following:

- In toggle coverage, not just whether a signal toggled from 0 to 1 and 1 to 0, but also the number of times it so toggled.
- In FSM coverage, not just whether an FSM reached a state, had such a transition, but also the number of times it did.
- In condition coverage, not just whether a condition was met or not, but also the number of times the condition was met.
- In Line Coverage, not just whether a line was executed, but how many times.

`-cm_constfile` *`filename`*
Specifies a file listing signals and 0 or 1 values. VCS compiles for

line and condition coverage as if these signals were permanently at the specified values and you included the `-cm_noconst` option.

`-cm_dir` *`directory_path_name`*
Specifies and alternative name and location for the simv.cm directory. The `-cm_dir` option is also a runtime option and a `cmView` command line option.

`-cm_fsmcfg` *`filename`*
Specifies an FSM coverage configuration file.

`-cm_fsmopt` *`keyword_argument`*
The keyword arguments are as follows:

`allowTemp`
By default, the variable that holds the current state of the FSM must be directly assigned a numerical constant or the value of a variable that holds the next state of the FSM. This keyword allows FSM extraction when there is indirect assignment to the variable that holds the current state.

`optimist`
Specifies identifying illegal transitions when VCS extracts FSMs in FSM coverage. cmView then reports illegal transitions in report files.

`report2StateFsms`
By default VCS does not extract two state FSMs. This keyword tells VCS to extract them.

`reportvalues`
Specifies reporting the value transitions of the reg that holds the current state of a One Hot or Hot Bit FSM where there are parameters for the bit numbers of the signals that hold the current and next state. The default behavior is to identify these parameters as the states of the FSM and report assignments to their bits as state transitions.

`reportWait`
Enables VCS to monitor transitions when the signal holding the current state is assigned the same state value.

`reportXassign`
Enables the extraction of FSMs in which a state contains the X (unknown) value.

`-cm_fsmresetfilter` *`filename`*
Filters out transitions in assignment statements controlled by `if` statements where the conditional expression (following the keyword `if`) is a signal you specify in the file. This filtering out can be for the specified signal in any module definition or in the module definition you specify in the file. You can also specify in the file the FSM and whether the signal is true or false.

`-cm_hier` *`filename`*
When compiling for line, condition, toggle or FSM coverage, specifies a configuration file that specifies the module definitions, instances and subhierarchies, and source files you want VCS either to exclude from coverage or exclusively compile for coverage.

`-cm_ignorepragmas`
Tells VCS to ignore pragmas for coverage metrics.

`-cm_libs yv|celldefine`
Specifies compiling for coverage source files in Verilog libraries when you include the `yv` argument. Specifies compiling for

coverage module definitions that are under the `‘celldefine` compiler directive when you include the celldefine argument. You can specify both arguments using the plus (+) delimiter.

`-cm_line contassign`
Specified enabling line coverage for Verilog continuous assignments.

`-cm_name` *`filename`*
As a compile-time or runtime option, specifies the name of the intermediate data files. When starting cmView, specifies the name of the report files.

`-cm_noconst`
Tells VCS not to monitor for conditions that can never be met or lines that can never execute because a signal is permanently at a 1 0r 0 value. *See the VCS Coverage Metrics User Guide*.

`-cm_opfile` *`filename`*
Specifies a file that contains a list of signals that you want to also be treated as observation points.

`-cm_pp [gui]|[batch]`
Tells VCS to start cmView . It tells VCS to start cmView in batch mode to write reports by default.

`gui`
VCS starts the cmView graphical user interface to displaying coverage data.

`batch`
Specifies the default operation, writing reports in batch mode.

You enter cmView command line options to the right of this option and its oargument.

`-cm_resetfilter` *`filename`*
You can filter out od FSM coverage transitions in assignment

statements controlled by `if` statements where the conditional expression (following the keyword `if`) is a signal you specify in the file. This filtering out can be for the specified signal in any module definition or in the module definition you specify in the file. You can also specify in the file the FSM and whether the signal is true or false.

`-cm_scope "`*`argument`*`"`
Limits the scope of what part of the design VCS compiles for coverage. It takes an argument that is similar, but not identical to, a line in the configuration file for the `-cm_hier` option. The difference is the + (plus) and – (minus) follow the `tree`, `module`, `file` and `filelist` keywords instead of preceding them as they do in the configuration file. You can enter more than one `-cm_scope` option. The argument must be enclosed in quotation marks. The following is an example of the use of this option:

```
vcs -cm_scope "tree+top.inst1" -cm_scope "file-testshell.v
```

`-cm_tglfile` *`filename`*
Specifies displaying at runtime a total toggle count for one or more subhierarchies specified by the top-level module instance entered in the file. This option is also a runtime option.

Discovery Visual Environment and UCLI

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

`-debug`
Enables DVE and UCLI debugging.

`-debug_all`
Enables DVE and UCLI debugging inluding line stepping.

`-ucli`
Specifies UCLI mode at runtime.

`-gui`
Specifies starting the DVE GUI at runtime.

`-PP`
Enables you to enter in your Verilog source code system tasks like `$vcdpluson` to create a VPD file during simulation. This option minimizes net data details for faster post-processing.

DVE

`-debug`
Enables DVE and command line debugging. This option does not enable line stepping. '

`-debug_all`
Enables command line debugging including line stepping.

`-ucli`
Specifies UCLI mode at runtime.

`-gui`
Starts the DVE gui at runtime.

VirSim

The following options are for starting VirSim to run interactively with VCS:

`-RI`
Run Interactive. Compiles design for interactive use, invokes the VirSim graphical debugging system immediately after compilation, and pauses simulation at time zero.

`-RIG`

Run Interactive Debug. Like `-RI`, except you use an existing executable file, such as the simv file instead of compiling and linking a new one. Even though you have already compiled your source code, you must include the source code on the command line with this option. Also make sure that running VirSim was enabled when you created the simv or simv.exe file, you would have done that with the `-I`, or `-RI` compile-time options.

`+sim+`*`executable`*

If you are using the `-RIG` option and the executable is not named simv or simv.exe or not in the current directory, specify the name and location with this option.

`+cfgfile+`*`filename`*

Specifies using a configuration file that you recorded in a previous session with VirSim. A configuration file specifies what windows to open and what groups, expressions, etc. to use. You can also use the Load Configuration dialog to specify a configuration file.

`+vslogfile+`*`filename`*

Enables logging of VirSim commands. If you do not specify a filename, the log is automatically saved to your current directory as VirSim.log.

`+vslogfilesim+`*`filename`*

Enables logging of VCS communication messages. If you use both `+vslogfile` and `+vslogfilesim`, VirSim commands and VCS messages are saved to the same file. If you do not specify a filename, the log is automatically saved to your current directory as VirSim.log.

The following are options for creating a VPD file fof post-processing in VirSim.

`-I`

Does the following:

- Tells VCS to compile the simv executable so that VirSim can interactively control it when you start VCS and VirSim with the `-RIG` option, or you select the Sim -> Invoke Sim menu command in the Interactive window to bring up the Simulator Invocation Dialog and specify the executable in that dialog.
- Enables you to enter system tasks like `$vcdpluson` to create a VPD file, during simulation, in any of the following places:
  - in the Command field in the VirSim Interactive window
  - in the text interface at the VCS CLI command prompt
  - in your Verilog code

`-PP`

Enables you to enter in your Verilog source code system tasks like `$vcdpluson` to create a VPD file during simulation. This option minimizes net data details for faster post-processing. Faster than a VPD file produced by the `-I` or `-RI` option.

`-RI`

This option is for starting an interactive session but it also enables you to enter system tasks like `$vcdpluson` to create a VPD file, during simulation, in the following two places:

- in the Command field in the VirSim Interactive window
- in your Verilog code

`+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 that VirSim reads.

`+vpdfileswitchsize+`*`number_in_MB`*

Specifies a size for the vpd file. When the vpd file reaches this size, VCS closes this file and opens a new one with the same hierarchy as the previous vpd file. There is a number suffix added to all new vpd file names to differentiate them. For example:

```
simv +vpdfile+test.vpd +vpdfileswitchsize+10
```

The first vpd file is named test.vpd. When its size raches 10 MB, VCS starts a new file test_01.vpd, the third vpd file is test_02.vpd, and so on.

The following options are for post-processing a VPD file with VirSim:

`-RPP`

Run Post-Processing mode. Starts VirSim for post-process a VPD file. Requires a VPD file created by the `$vcdpluson` system task. You can specify the name of the VPD file with the `+vpdfile` option. In post-processing mode VirSim will need to read your source code to use the Logic Browser, so include the source code on the command line with this option.

`+vcdfile+`*`filename`*

Specifies the VCD file you want to use for post-processing. VCS translates the VCD file to a VPD file and loads the new VPD file into VirSim.

`+cfgfile+`*`filename`*

Specifies using a configuration file that you recorded in a previous session with VirSim. A configuration file specifies what windows to open and what groups, expressions, etc. to use.

`+vslogfile+`*`filename`*

Enables logging of VirSim commands to the specified file. If you do not specify a filename, the log is automatically saved to your current directory as VirSim.log.

+vpdfile+*filename*[+start+*start_time*+end+*end_time*]

For post-processing, specifies the VPD file you are using to display simulation results. The optional +start+*start_time* and +end+*end_time* arguments specify you only want VirSim to load and display the results from between these simulation times. For example, to load a .vpd simulation file from time 700 to time 1000 without source but with a .cfg configuration file in post simulation mode (-RPP):

```
vcs -RPP +vpdfile+vcdplus.vpd+start+700+end+1000
+cfgfile+default.cfg
```

You can also enter this option when generating a VPD file to specify the name of the VPD file.

Converting VCD and VPD Files

-vcd2vpd *vcd_filename vcdplus_filename*

Tells VCS to find and run the vcd2vpd utility that converts a VCD file to a VPD file. VCS inputs to the utility the specified VCD file and the utility outputs the specified VPD file.

-vpd2vcd *vcdplus_filename vcd_filename*

Tells VCS to find and run the vpd2vcd utility that converts a VPD file to a VCD file. VCS inputs to the utility the specified VPD file and the utility outputs the specified VCD file.

Specifying Delays

+allmtm

Specifies enabling the simv executable for the +mindelays, +typdelays, or +maxdelays options at runtime, instead of at compile-time, to specify which of the minimum, typical, or maximum delay values to use during simulation from min:typ:max

delay value triplets in module path delays and timing delays. This option is also used for compiling SDF files.

This option does not work for min:typ:max delay value triplets in other delay specification in your source code. Do not use this options with the `+maxdelays`, `+mindelays`, or `+typdelays` compile-time options. For more information see “Specifying Min:Typ:Max Delays at Runtime” on page 8-57.

`+charge_decay`
Enables charge decay in `trireg` nets. Charge decay will not work if you connect the `trireg` to a transistor (bidirectional pass) switch such as `tran`, `rtran`, `tranif1`, or `rtranif0`.

`+delay_mode_path`
For modules that contain specify blocks, ignores the delay specifications on all gates and switches and use only the module path delays and the delay specifications on continuous assignments.

`+delay_mode_zero`
Changes all the delay specifications on all gates, switches, and continuous assignments to zero and change all module path delays in specify blocks to zero.

`+delay_mode_unit`
Ignores the module path delays in specify blocks and change all the delay specifications on all gates, switches, and continuous assignments to the shortest time precision argument of all the `` `timescale `` compiler directives in the source code. The default time unit and time precision argument of the `` `timescale `` compiler directive is 1 s.

`+delay_mode_distributed`
Ignores the module path delays in specify blocks and use only the

delay specifications on all gates, switches, and continuous assignments.

`+maxdelays`
: Specifies using the maximum timing delays in min:typ:max delay triplets in delay specifications and also in delay entries in SDF files. See “Min:Typ:Max Delays” on page 8-55.

`+mindelays`
: Specifies using the minimum timing delays in min:typ:max delay triplets in delay specifications and also in delay entries in SDF files. See “Min:Typ:Max Delays” on page 8-55.

`+typdelays`
: Specifies using the typical timing delays in min:typ:max delay triplets in delay specifications and also in delay entries in SDF files. See “Min:Typ:Max Delays” on page 8-55.

`+multisource_int_delays`
: Enables the multisource interconnect feature, including transport delays with full pulse control. See “INTERCONNECT Delays” on page 8-50.

`+nbaopt`
: To make debugging in the waveform window easier, many users enter a `#1` intra-assignment delay in nonblocking procedural assignment statements, for example:

```
reg1 <= #1 reg2;
```

These delays impede the simulation performance of the design so after debugging you can remove these delays with this option.

Note:
: The `+nbaopt` option removes all intra-assignment delays in all the nonblocking assignment statements in the design, not just the `#1` delays.

`+old_iopath`

By default VCS replaces negative module path delays in SDF files with a 0 delay. If you include this option, VCS replaces these negative delays with the delay specified in a module's specify block.

`+transport_int_delays`

Enables transport delays for delays on nets with a delay backannotated from an INTERCONNECT entry in an SDF file. The default is inertial delays. See "Transport and Inertial Delays" on page 8-2, "'Enabling Transport Delays" on page 8-7" and "'INTERCONNECT Delays" on page 8-50" .

`+transport_path_delays`

Enables transport delays for module path delays. See "Transport and Inertial Delays" on page 8-2 and "'Enabling Transport Delays" on page 8-7".

Compiling an SDF File

`+allmtm`

Specifies compiling separate files for minimum, typical, and maximum delays when there are min:typ:max delay triplets in SDF files. If you use this option, you can use the `+mindelays`, `+typdelays`, or `+maxdelays` options at runtime to specify which compiled SDF file VCS uses. Do not use this options with the `+maxdelays`, `+mindelays`, or `+typdelays` compile-time options. See "Specifying Min:Typ:Max Delays at Runtime" on page 8-57.

`+csdf+precompile`

Precompiles your SDF file into a format that VCS can parse when it is compiling your Verilog code, see "Precompiling An SDF File" on page 8-29.

`+csdf+precomp+dir+`*`directory`*

Specifies the directory path where you want VCS to write the precompiled SDF file, see “Specifying an Alternative Name and Location” on page 8-30.

`+csdf+precomp+ext+`*`ext`*

Specifies an alternative to the "_c" character string addition to the filename extension of the precompiled SDF file, see “Specifying an Alternative Name and Location” on page 8-30.

`+maxdelays`

Specifies using the maximum timing delays in min:typ:max delay triplets when compiling the SDF file. See “Min:Typ:Max Delays” on page 8-55. The *`mtm_spec`* argument to the `$sdf_annotate` system task overrides this option, see “The $sdf_annotate System Task” on page 8-24.

`+mindelays`

Specifies using the minimum timing delays in min:typ:max delay triplets when compiling the SDF file. See “Min:Typ:Max Delays” on page 8-55. The *`mtm_spec`* argument to the `$sdf_annotate` system task overrides this option, see “The $sdf_annotate System Task” on page 8-24.

`+oldsdf`

Disables compiling the SDF file. Use this option if you cannot meet the limitations on compiled SDF files, see “Limitations on Compiling The SDF File” on page 8-26.

`+sdf_nocheck_celltype`

For a module instance to which an SDF file backannotates delay data, disables comparing the module identifier in the source code with the CELLTYPE entry in the SDF file. See “Disabling CELLTYPE Checking in SDF Files” on page 8-37.

`+typdelays`

Specifies using the typical timing delays in min:typ:max delay

triplets when compiling the SDF file. See “Min:Typ:Max Delays” on page 8-55. The *mtm_spec* argument to the `$sdf_annotate` system task overrides this option, see “The $sdf_annotate System Task” on page 8-24.

`+vcs+mipdexpand`

This option is used when back annotating SDF delay values from an ASCII text SDF file at runtime, as specified by the `+oldsdf` compile-time option which disables compiling the SDF file during compilation, if the SDF file contains PORT entries for the individual bits of a port, using this option enables VCS to backannotate these PORT entry delay values. Similarly, using this compile-time option enables a PLI application to pass delay values to individual bits of a port.

As an alternative to using this option, you can use the `` `vcs_mipdexpand `` compiler directive or you can enter the `mipb` ACC capability in your PLI table file, for example:

```
$sdf_annotate call=sdf_annotate_call
acc+=rw,mipb:top_level_mod+
```

When you compile the SDF file (recommended) the back annotation of the delay values for individual bits of a port does not require this option.

Profiling Your Design

`+prof`

Specifies that VCS writes the vcs.prof file during simulation that tells you the module definitions, module instances, and Verilog constructs in your design that use the most CPU time, see “Profiling the Simulation” on page 4-33.

`+vissymbols`

Makes symbols visible when you are using the prof or gprof program (not the VCS profiler that is enabled by the `+prof` option) to profile generated code.

File Containing Source Filenames and Options

`-f` *`filename`*

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

You can enter in this file are all the compile-time options that begins with a plus "+" such as `+2state` and `+cli+4` with three exceptions: do not enter the `+comp64`, `+full64`, or `+memopt` compile-time options.

You can also enter the following compile-time options that begin with a minus "-":

```
-f        -gen_asm       -gen_obj       -l
-line     -S      -u      -v      -y
```

You can also enter the runtime options in this file so that VCS compiles them into the simv executable, see "Compiling Runtime Options Into The simv Executable" on page 3-81

You cannot include C source or object file names, for PLI applications, in this file.

You can specify a pathname for the *filename* argument. Note that you can also enter the `-f` option in this file with the pathname of another file that also contains a list of absolute pathnames for Verilog source files and compile-time options.

`-F` *`filename`*

Similar to the `-f` option, but if you specify a pathname for the *filename* argument, you don't have to list absolute pathnames for the Verilog source files that you list in the file. VCS uses the path to this file as the path to the Verilog source files.
Nested files are not supported.

`-file` *`filename`*

This option is for problems you might encounter with entries in files specified with the `-f` or `-F` options. This file can contain more compile-time options and different kinds of files. It can contain options for controlling compilation and PLI options and object files.

You can also use escape characters and meta-characters in this file, like `$`, `\`, and `!` and they will expand, for example:

```
-CFLAGS '-I$VCS_HOME/include'
/my/pli/code/$PROJECT_VERSION/treewalker.o
-P /my/pli/code/$PROJECT_VERSION/treewalker.tab
```

You can comment out entries in this file with the Verilog `//` and `/* */` comment characters.

Compiling Runtime Options into the simv Executable

`+plusarg_save`

Some runtime must be preceded by the +plusarg_save option, either on the vcs command line or in the file specified with the -f or -F option, for VCS to compile them into the executable, see "Compiling Runtime Options Into The simv Executable" on page 3-81 for a list of these runtime options.

`+plusarg_ignore`

Tells VCS not to compile into the simv executable the runtime options that follow. This option is typically used in the file that you

specify with the `-f` compile-time option and entered to counter the `+plusarg_save` option on a previous line.

Pulse Filtering

`+pulse_e/`*`number`*

Flags as error and drive X 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 that the percentage of the module path delay by the *`number`* argument to the `+pulse_r/`*`number`* option.

`+pulse_r/`*`number`*

Rejects any pulse whose width is less than *number* percent of module path delay. The *`number`* argument is in the range of 0 to 100.

`+pulse_int_r`

Same as the existing `+pulse_r` option, except it applies only to interconnect delays.

`+pulse_int_e`

Same as the existing `+pulse_e` option, except it applies only to interconnect delays.

`+pulse_on_event`

Specifies that when VCS encounters a pulse shorter than the module path delay, VCS waits until the module path delay elapses and then drives an X value on the module output port and displays an error message. It drives that X value for a simulation time equal to the length of the short pulse or until another simulation event drives a value on the output port. See “Pulse Control” on page 8-7.

`+pulse_on_detect`

Specifies that when VCS encounters a pulse shorter than the

module path delay, VCS immediately drives an X value on the module output port, and displays an error message. It does not wait until the module path delay elapses.
It drives that X value until the short pulse propagates through the module or until another simulation event drives a value on the output port. See “Pulse Control” on page 8-7.

PLI Applications

`+applylearn+`*`filename`*
Used in subsequent compilations, re-compiles your design to enable only the ACC capabilities that you needed for the debugging operations you did during a previous simulation of the design. See more about this option in the section "‘Debugging” on page 3-33" and in “Using Only The ACC Capabilities That You Need” on page 7-25.

`-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 processes before starting the simv executable. See “Writing Your Own main() Routine” on page 7-34.

Note:
Do not use the `-e` options with the VCS/SystemC Cosimulation Interface.

`-P` *`pli.tab`*
Compiles a user-defined PLI definition table file. See Chapter 7, “Using the PLI.”

`+vpi`
Enables the use of VPI PLI access routines, see “Using VPI Routines” on page 7-29.

`-load` *`shared_library`*`:`*`registration_routine`*
Specifies the registration routine in a shared library for a VPI application, see ‘Integrating a VPI Application With VCS” on page 7-32.

`-use_vpiobj`
Used to specify the vpi_user.c file that enables you to use the vpi_register_systf VPI access routine.

Enabling and Disabling in Specify Blocks and Timing Checks

`+pathpulse`
Enables the search for the PATHPULSE$ specparam in specify blocks. This specparam is described on IEEE Std 1364-1995 pages 170-171.

`+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, after VCS has compiled these timing checks into the executable, to disable these timing checks, but the executable simulates faster if you include this option at compile-time so that the timing checks are not in the executable. If you need the delayed versions of the signals in negative timing checks but want faster performance, include this option at runtime. The delayed versions are not available if you

use this option at compile-time. See “Enabling Negative Timing Checks” on page 9-12.

VCS recognizes `+notimingchecks` to be the same as `+notimingcheck` when you enter it on the vcs or simv command line.

`+no_notifier`
Disables toggling of the notifier register that you specify in some timing check system tasks. This option does not disable the display of warning messages when VCS finds a timing violation that you specified in a timing check.

`+no_tchk_msg`
Disables display of timing violations but does not disable the toggling of notifier registers in timing checks. This is also a runtime option.

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 on the screen all the C/C++ functions that you called in your Verilog source code.

Negative Timing Checks

`+neg_tchk`
Enables negative values in timing checks. See “Negative Timing Checks” on page 9-1

`+old_ntc`
Prevents the other timing checks from using delayed versions of the signals in the `$setuphold` and `$recrem` timing checks, see “Other Timing Checks Use The Delayed Signals” on page 9-14.

`+NTC2`
In `$setuphold` and `$recrem` timing checks, specifies checking the timestamp and timecheck conditions when the original data and reference signals change value instead of when their delayed versions change value. See “Checking Conditions” on page 9-18.

`+overlap`
Enables accurate simulation of multiple non-overlapping violation windows for the same signals specified with negative delay values back annotated from an SDF file to timing checks. See “Using Multiple Non-Overlapping Violation Windows” on page 9-22.

Flushing Certain Output Text File Buffers

When VCS creates a log file, VCD file, or a text file specified with the `$fopen` system function, VCS writes the data for the file in a buffer and periodically dumps the data from the buffer to the file on disk. The frequency of these dumps varies depending on many factors including the amount of data that VCS has to write to the buffer as simulation or compilation progresses. If you need to see or use the latest information in these files more frequently than the rate at which VCS normally flushes this data, these options tell VCS to flush the data more often during compilation or simulation.

`+vcs+flush+log`
: Increases the frequency of flushing both the compilation and simulation log file buffers.

`+vcs+flush+dump`
: Increases the frequency of flushing all VCD file buffers.

`+vcs+flush+fopen`
: Increases the frequency of flushing all the buffers for the files opened by the `$fopen` system function.

`+vcs+flush+all`
: Shortcut option for entering all three of the `+vcs+flush+log`, `+vcs+flush+dump`, and `+vcs+flush+fopen` options

These options do not increase the frequency of dumping other text files, including the VCDE files specified by the `$dumpports` system task or the simulation history file for LSI certification specified by the `$lsi_dumpports` system task.

These options can also be entered at runtime. Entering them at compile-time modifies the simv executable so that it runs as if these options were always entered at runtime.

Simulating SWIFT VMC Models and SmartModels

`-lmc-swift`
Includes the LMC SWIFT interface. See “SWIFT VMC Models and SmartModels Introduction” on page 10-2.

`-lmc-swift-template`
Generates a Verilog template for a SWIFT Model. See “SWIFT VMC Models and SmartModels Introduction” on page 10-2.

Controlling Messages

`+libverbose`
Tells VCS to display a message when it finds a module definition in a source file in a Verilog library directory that resolves a module instantiation statement that VCS read in your source files, a library file, or in another file in a library directory. The message is 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. See “Using Lint” on page 3-16.

`-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; compile 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 will also be passed to run-time executable, just as if you entered `simv -V`

`-Vt`

Verbose mode; provide CPU time information. Like `-V`, but also prints the time used by each command. Use of the `-Vt` option can cause a slow-down of the simulation.

`+warn=[no]ID|none|all`

Enables or disables display of warning messages using warning message IDs. In the following warning message:

```
Warning-[TFIPC] Too few instance port connections
```

The text string TFIPC is the message ID. The syntax of this option is as follows:

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

Where:

no	Specifies disabling warning messages with the ID that follows. There is no space between the keyword no and the ID.
none	Specifies disabling all warning messages. IDs that follow, in a comma separated list, specify exceptions.
all	Specifies enabling all warning messages, IDs that follow preceded by the keyword `no`, in a comma separated list, specify exceptions

The following are examples that show how to use this option:

`+warn=noIPDW`	Enables all warning messages except the warning with the IPDW ID
`+warn=none,TFIPC`	Disables all warning messages except the warning with the TFIPC ID.
`+warn=noIPDW,noTFIPC`	Disables the warning messages with the IPDW and TFIPC IDs.
`+warn=all`	Enables all warning messages. This is the default.

Cell Definition

`+nolibcell`

Do not define as a cell modules defined in libraries unless they are under the `` `celldefine `` compiler directive.

`+nocelldefinepli+0`

Enables recording, in VPD files the transition times and values of nets and registers in all modules defined under the `` `celldefine `` compiler directive or defined in a library that you specify with the `-v` or `-y` compile-time options. This option also enables full PLI access to these modules. This option also overrides a `nocelldefinepli` entry in the `.tab` files in the *`vcs_install_dir`*`/virsimdir` subdirectory.

`+nocelldefinepli+1`

Disables recording, in VPD files the transition times and values of nets and registers in all modules defined under the `` `celldefine `` compiler directive. This option also disables full PLI access to these modules. Modules in a library file or directory are not affected by this option unless they are defined under the `` `celldefine `` compiler directive.

`+nocelldefinepli+2`

Disables recording, in VPD files the transition times and values

of nets and registers in all modules defined under the `celldefine compiler directive or defined in a library that you specify with the -v or -y compile-time options whether the modules in these libraries are defined under the `celldefine compiler directive or not. This option also disables PLI access to these modules.

Disabling recording of transition times and values of the nets and registers in library cells can significantly increase simulation performance.

As an alternative to using the +nocelldefinepli+1 option, you can add an entry in the virsims.tab AND virsims_pp.tab files (located in $VCS_HOME/virsims_support) to turn off VPD dumping for modules defined under the `celldefine compiler directive. Enter the keyword nocelldefinepli with appropriate spaces in the $virsim line, for example:

```
$virsim    data=0  check=vp_check_virsims
misc=vp_misc_virsims callback nocelldefinepli
acc+=rw,cbka:*
```

Note:

Disabling recording transitions in library cells is intended for batch-simulation only and not for interactive debugging with VirSim. Any attempt in VirSim to access a part of your design for which VPD has been disabled may have unexpected results.

Licensing

+vcs+lic+vcsi

Checks out three VCSi licenses to run VCS.

`+vcsi+lic+vcs`
Checks out a VCS license to run VCSi when all VCSi licenses are in use.

`+vcs+lic+wait`
Tells VCS to wait for a network license if none is available.

`+vcsi+lic+wait`
Tells VCSi to waid for a network license if none is available.

`-ID`
This option returns useful information about a number of things: the version of VCS that you have set the VCS_HOME environment variable to, the name of your work station, your work station's platform, the host ID of your work station (used in licensing), the version of the VCS compiler (same as VCS) and the VCS build date, and the VirSim version number.

Controlling the Assembler

`-gen_asm`
Assembly Code Generation mode. Use this option if you encounter problems in native code generation.
Not supported on IBM RS/6000 AIX.

`-S`
Old form of `-gen_asm`. Not supported on IBM RS/6000 AIX.

`-as` *`assembler`*
Selects an alternate assembler. Not supported on IBM RS/6000 AIX.

`-ASFLAGS` *`options`*
Passes options to assembler. Not supported on IBM RS/6000 AIX.

`-C`
Stops after generating the assembly code intermediate files. If you

also enter the `-gen_asm` option, does not assemble the assembly code files. Use this option if you want to assemble by hand. This option can also be used with the `-gen_c` options for disabling C code compilation.

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 proceed with compiling the source files and generates the intermediate C, assembly, or object files, then compile or assemble the C or assembly code, but not to link. 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*). Example: `-lm` (instructs VCS to include the math library).

`-syslib` *`libs`*

Specifies system libraries, for example `-syslib -ly -lc` includes the machine specific system libraries. Normally, libs entered on the vcs command line using `-l`*`name`* option are placed in front of libvcs.a on the link line. The `-syslib` option tells VCS to place the specified libs after libvcs.a on the link line.

Controlling the C Compiler

`-gen_c`
Generates C language code. This is the default in IBM RS/6000 AIX.

`-cc` *`compiler`*
Specifies an alternate C compiler

`-CC` *`options`*
Passes options to the C compiler or assembler.

`-CFLAGS` *`options`*
Pass options to C compiler. Multiple `-CFLAGS` are allowed. Allows passing of C compiler optimization levels.

`-cpp`
Specifies the C++ compiler.

Note:
If you are entering a C++ file, or an object file compiled from a C++ file, on the `vcs` command line, you must tell VCS to use the standard C++ library for linking. To do this enter the `-lstdc++` linker flag with the `-LDFLAGS` compile-time option, for example:

```
vcs source.v source.cpp -P my.tab \
-cpp /net/local/bin/c++ -LDFLAGS -lstdc++
```

`-j`*`number_of_processes`*
Specifies the number of processes that VCS forks for parallel compilation. There is no space between the "`j`" character and the number. You can use this option in any compilation mode: directly generating object files from the parallel compilation of your Verilog source files (`-gen_obj`, default on the HP, Solaris, and Linux platforms), generating intermediate assembly files (`-gen_asm`)

and then their parallel assembly, or generating intermediate C files (`-gen_c`) and their parallel compilation.

`-C`

Stops after generating the C code intermediate files if you also enter the `-gen_c` option, does not compile the C code files (`-gen_c` is the default on IBM RS/6000 AIX). Use this option if you want to compile by hand.

This option can also be used with the `-gen_asm` option for disabling assembly.

`-O0`

Suppress optimization for faster compilation (but slower simulation). Suppresses optimization both for how VCS writes intermediate C code files and how VCS compiles these files. This option is the uppercase letter "O" followed by a zero with no space between them.

`-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. `-O0` has special mention above.

`-override-cflags`

Tells VCS not to pass its default options to the C compiler.VCS has a number of C compiler options that it passes to the C compiler by default. The options it passes depends on the platform, whether it's a 64 or 64-32 bit compilation, whether it's an MX mixed-HDL design, and other factors. VCS passes these options and then passes the options you specify with the `-CFLAGS` compile-time option.

Source Protection

`+autoprotect[`*`file_suffix`*`]`
Creates a protected source file; all modules are encrypted.

`+auto2protect[`*`file_suffix`*`]`
Create a protected source file that does not encrypt the port connection list in the module header; all modules are encrypted.

`+auto3protect[`*`file_suffix`*`]`
Creates a protected source file that does not encrypt the port connection list in the module header or any parameter declarations that precede the first port declaration; all modules are encrypted.

`+deleteprotected`
Allows overwriting of existing files when doing source protection.

`+pli_unprotected`
Enables PLI and CLI access to the modules in the protected source file being created (PLI and CLI access is normally disabled for protected modules).

`+protect[`*`file_suffix`*`]`
Creates a protected source file, only encrypting `` `protect `` / `` `endprotect `` regions.

`+putprotect+`*`target_dir`*
Specifies the target directory for protected files.

`+sdfprotect[`*`file_suffix`*`]`
Creates a protected SDF file.

`-Xmangle=`*`number`*
Produces a mangled version of input, changing variable names to words from list.
Useful to get an entire Verilog design into a single file. Output is

saved in tokens.v file.You can substitute `-Xman` for `-Xmangle`

The argument *`number`* can be 1, 4, 12, or 28:

`-Xman=1`
Randomly changes names and identifiers, and removes comments, to provide a 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`.

Mixed Analog/Digital Simulation

`+ad=`*`partition_filename`*
Specifies the partition file that you use in mixed A/D simulation to specify the part of the design simulated by the analog simulator, the analog simulator you want to use, and the resistance mapping

information that maps analog drive resistance ranges to Verilog strengths.

`+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.

Changing Parameter Values

`-pvalue+`*`parameter_hierarchical_name=value`*
Changes the specified parameter to the specified value. See ‘Changing Parameter Values From The Command Line” on page 3-18.

`-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 you use the forward slash character (/) instead of a period as the delimiters. See ‘Changing Parameter Values From The Command Line” on page 3-18.

Specify the Time Scale

`-timescale=`*`time_unit/time_precision`*
By default if some source files contain the `` `timescale `` compiler directive and others don’t, and the ones that don’t precede the ones that do on the command line, this is an error condition and

VCS halts compilation. This option enables you to specify the timescale for the source files that don't contain this compiler directive and precede the source files that do.
Do not include spaces when specifying the arguments to this option.

`-override_timescale=`*`time_unit`*`/`*`time_precision`*
Overrides the time unit and a precision unit for all the `` `timescale `` compiler directives in the source code and, like the `-timescale` option, provides a timescale for all module definitions that precede the first `` `timescale `` compiler directive.
Do not include spaces when specifying the arguments to this option.

General Options

Enable Verilog 2001 Features

`+v2k`
Enables new language features in the proposed IEEE 1364-2001 standard. See 'Implemented IEEE Std 1364-2001 Language Constructs" on page 2-23.

Enable the VCS/SystemC Cosimulation Interface

`-sysc`
Tells VCS to look in the ./csrc directory for the subdirectories containing the interface and wrapper files needed by the interface to connect the Verilog and SystemC parts of the design.

Reduce Memory Consumption

`+memopt[+2]`

Applies optimizations to reduce memory, see “Optimizations To Reduce Memory Consumption” on page 3-13.

TetraMAX

`+tetramax`

Enter this option when simulatingTetraMAX’s testbench in zero delay mode.

Make Accessing an Undeclared Bit an Error Condition

`+vcs+boundscheck`

Changes reading from or writing to an undeclared bit to an error condition instead of a warning condition.

Treat Output Ports As Inout Ports

`+spl_read`

Tells VCS to treat output ports as “inout” in order to facilitate more accurate multi-driver contention analysis across module boundaries. This option can have an adverse impact on runtime performance.

Allow Inout Port Connection Width Mismatches

`+noerrorIOPCWM`

Changes the error condition, when a signal is wider or narrower than the inout port to which it is connected, to a warning condition, thus allowing VCS to create the simv executable after displaying the warning message.

Specifying a VCD File

`+vcs+dumpvars`

A substitute for entering the `$dumpvars` system task, without arguments, in your Verilog code.

Memories and Milti-Dimensional Arrays (MDAs)

`+memcbk`

Enables callbacks for memories and multi-dimensional arrays (MDAs). Use this option if your design has memories or MDAs and you are doing any of the following:

- Writing a VCD or VPD file during simulation. For VCD files, at runtime, you must also enter the `+vcs+dumparrays` runtime option. For VPD files you must enter the `$vcdplusmemon` system task. VCD and VPD files are used for post-processing with VirSim.
- Using the VCS/SystemC Interface
- Using VCS Coverage Metrics for any type of coverage
- Interactive debugging with VirSim
- Writing an FSDB file for Debussy
- Using any debugging interface application - VCSD/PLI (acc/vpi) that needs to use value change callbacks on memories or MDAs. APIs like acc_add_callback, vcsd_add_callback, and vpi_register_cb need this option if these APIs are used on memories or MDAs.

Specifying a Log File

`-l` *`filename`*

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

Hardware Modeling

`-lmc-hm`

Option for compiling a design that instantiates a hardware model. Including this option is an alternative to specifying the lmvc.tab PLI table file and the lmvc.o lm_sfi.a object file and library that you need for hardware modeling.

Changing Source File Identifiers to Upper Case

`-u`

Changes all the characters in identifiers to uppercase. It does not change identifiers in quoted strings such as the first argument to the `$monitor` system task.You do not see this change in the VirSim Source window but you do see it in all the other VirSim windows.

Defining a Text Macro

`+define+`*`macro`*`=`*`value`*`+`

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

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

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

```
‘uselib ‘USELIB libext+.v
```

Specifying the Name of the Executable File

`-o` *name*

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

Returning The Platform Directory Name

`-platform`

Returns the name of the *platform* directory in your VCS installation directory. For example, when you install VCS on a Solaris version 5.4 workstation, VCS creates a directory named sun_sparc_solaris_5.4 in the directory where you install VCS. In this directory are subdirectories for licensing, executable libraries, utilities, and other important files and executables. You need to set your path to these subdirectories. You can do so using this option:

```
set path=($VCS_HOME/bin\
$VCS_HOME/‘$VCS_HOME/bin/vcs -platform‘/bin\
$path)
```

Specifying Native Code Generation

`-gen_obj`

Generates object code; default on Solaris, HP and Linux platforms. Not supported on IBM RS/6000 AIX.

For Long Calls

`-B`

Generate long calls for large designs. HP 9000/700 models only.

Compiling Runtime Options Into The simv Executable

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

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

```
+cliecho            +no_pulse_msg
+sdverbose          +vcs+finish
+vcs+flush+all      +vcs+flush+dump
+vcs+flush+fopen    +vcs+flush+log
```

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

```
+cfgfile              +override_model_delays
+vcs+dumpoff          +vcs+dumpon
+vcs+dumpvarsoff      +vcs+grwavesoff
+vcs+ignorestop       +vcs+learn+pli
+vcs+mipd+noalias     +vcs+nostdout
```

```
+vcs+stop                +vera_load
+vera_mload              +vpdbufsize
+vpddrivers              +vpdfile
+vpdfilesize             +vpdnocompress
+vpdnostrengths          +vpdports
+vpdupdate
```

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

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

Several runtime options are also compile-time options. When these options appears on the vcs command line or in the file that you specify with the `-f` or `-F` compile-time option, even if you precede then with the `+plusarg_save` option, VCS considers them to be compile-time options so there is no way to compile these runtime options into the simv executable.

Runtime options are described in Chapter 4, “Simulating Your Design.”

Performance Considerations

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

Use Local Disk

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

If the csrc working directory is not located on an local disc, set the incremental compile directory so that it is. Or do: `cd` *`local_disk`* and run the compilation there. This ensures that the compilation directory can be rapidly accessed by VCS. If the link step takes more than a few seconds on a small design, then you know that you're accessing files over the network.

Managing Temporary Disk Space on UNIX

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

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

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

- If the `TMPDIR` environment variable is set to "." for example, all intermediate files will be placed in the working directory instead of /tmp, causing a slow down of compilation by up to 2x. Solution, use: `unsetenv TMPDIR`.

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

  ```
  setenv TMPDIR /bigdisk
  ```

Compile-Time Options that Impede or Accelerate VCS

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

Compile-Time Options that Slow Down Both Compilation and Simulation

`-line`	Enables line stepping
`+cli`	Enables breakpoints and forcing values
`+acc`	Enables PLI ACC capabilities.
`-I`	Compiles for interactive use.
`-RI`	Compiles for interactive use and start VirSim.
`-full64`	Compiles in 64 bit mode for 64 bit mode simulation

Compile-Time Options that Slow Down Simulation

`-RIG`	Starts VirSim using the current compilation instead of compiling any possible module changes
`+pathpulse`	Enables the `PATHPULSE$` specparam in specify blocks.
`+pulse_e`	Drives an X value on narrow pulses.
`+pulse_r`	Filters out narrow pulses.

`+pulse_int_e` Drives an X value on pulses narrower than interconnect delays.

`+pulse_int_r` Filters out pulses narrower than interconnect delays.

`+spl_read` Treats output ports as inout ports.

Compile-Time Options that Slow Down Compilation

`-gen_asm` Generates assembly code instead of direct generation of native code can increase compilation time up to 20%.

`-S` Generates assembly code instead of direct generation of native code can increase compilation time up to 20%.

`-gen_c` Generates C intermediate code instead of direct native code generation can increase compilation time as much as 3x.

`-comp64` Compiles in 64 bit mode for 32 bit simulation.

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

`-O3` or `-O4` When generating C code intermediate files and then compiling them, applies more than the default level of optimizations. Applying more optimizations slows down the C compiler but speeds up simulation.

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

`-O0` When generating C code intermediate files and then compiling them, turn off all optimizations. Turning off optimizations allows the C compiler to finish sooner but your design will simulate slower without these optimizations.

`-O1` When generating C code intermediate files and then compiling them, applies fewer optimizations. Applying fewer optimizations allows the C compiler to finish somewhat sooner but your design will simulate somewhat slower without these optimizations.

Compile-Time Options that Speed Up Both Compilation and Simulation

`+2state` Specifies two state simulation.

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

Compile-Time Options that Speed Up Simulation

`+delay_mode_zero`
Disables all delays can increase simulation speed but your design will simulate differently and some designs can get stuck in an infinite loop.

`+notimingcheck`
Ignores timing check system tasks.

`+nbaopt` Removes intra-assignment delays from nonblocking assignment statements

Compiling for Debugging or Performance

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

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

```
vcs -Mdir=csrc_debug -line -RI +cli+4 source.v
```

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

For faster simulation you enter the following vcs command line:

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

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

4

Simulating Your Design

This chapter describes the details of running a simulation. Its sections include:

- Runtime Options
- Save and Restart for saving a simulation at a certain state and the restarting the simulation at that state
- Specifying A Very Long Time Before Stopping Simulation
- Passing Values From The Runtime Command Line
- Performance Considerations
- Profiling the Simulation
- How VCS Prevents Time 0 Race Conditions
- Protected and Portable Verilog Model

Runtime Options

These runtime options are typically entered on the simv command line but some of them can be compiled into the simv executable at compile-time, see “Compiling Runtime Options Into The simv Executable” on page 3-81.

Running DVE in Interactive Mode

`-gui`
Invokes DVE at runtime.

`-ucli`
Invokes the UCLI debugger command line if issued at runtime.

`-l` *`logFilename`*
Captures simulation output, such as user input commands and responses to UCLI commands.

`-i` *`inputFilename`*
Reads interactive commands from a file, then switches to reading from standard command line input.

`-k` *`keyFilename`*
Writes interactive commands entered to *`keyFilename`*, which can be used by a later simv as `-i` *`inputFilename`*

Simulating OpenVera Testbenches

`+ntb_cache_dir`
Specifies the directory location of the cache that VCS maintains as an internal disk cache for randomization.

`+ntb_debug_on_error`

Causes the simulation to stop immediately when a simulation error is encountered. In addition to normal verification errors, This option halts the simulation in case of runtime errors as well.

`+ntb_enable_solver_trace=`*`value`*

Enables a debug mode that displays diagnostics when VCS executes a randomize() method call. Allowed values are:

0 - Do not display (default).

1 - Displays the constraints VCS is solving.

2 - Displays the entire constraint set.

`+ntb_enable_solver_trace_on_failure[=value]`

Enables a mode that displays trace information only when the VCS constraint solver fails to compute a solution, usually due to inconsistent constraints. When the value of the option is 2, the analysis narrows down to the smallest set of inconsistent constraints, thus aiding the debugging process. Allowed values are 0, 1, 2. The default value is 2.

`+ntb_exit_on_error[=`*`value`*`]`

Causes VCS to exit when value is less than 0. The value can be:

0: continue

1: exit on first error (default value)

N: exit on nth error.

When value = 0, the simulation finishes regardless of the number of errors.

`+ntb_load=path_name_to_libtb.so`

Specifies loading the testbench shared object file libtb.so.

`+ntb_random_seed=`*`value`*

Sets the seed value used by the top level random number

generator at the start of simulation. The random(seed) system function call overrides this setting. The value can be any integer number.

`+ntb_solver_mode=`*`value`*
: Allows choosing between one of two constraint solver modes. When set to 1, the solver spends more pre-processing time in analyzing the constraints, during the first call to randomize() on each class. Subsequent calls to randomize() on that class are very fast. When set to 2, the solver does minimal pre-processing, and analyzes the constraint in each call to randomize(). Default is 2.

`+ntb_stop_on_error`
: Causes the simulation to stop immediately when a simulation error is encountered, turning it into a cli debugging environment. In addition to normal verification errors, this option halts the simulation in case of runtime errors. The default setting is to execute the remaining code within the present simulation time.

Simulating OpenVera Assertions

The following are runtime options for simulation OpenVera Assertions:

`-ova_quiet [1]`
: Disables printing results on screen. The report file is not affected. With the 1 argument, only a summary is printed on screen.

`-ova_report [`*`filename`*`]`
: Generates a report file in addition to printing results on screen. Specifying the full path name of the report file overrides the default report name and location.

`-ova_verbose`

Adds more information to the end of the report including assertions that never triggered and attempts that did not finish, and a summary with the number of assertions present, attempted, and failed.

`-ova_name` *`name | pathname`*

Specifies an alternative name or location and name for the ./simv.vdb/scov/results.db and ./simv.vdb/reports/ova.report files. You use this option if you want data and reports from a series of simulation runs. It is a way of keeping VCS from overwriting these files from a previous simulation.

If you just specify a name the alternatively named files will be in the default directories. If you specify a pathname, with an argument containing the slash character /, you specify a different location and name for these files, for example:

```
-ova_name /net/design1/ova/run2
```

This example tells VCS to write run2.db and run2.report in the /net/design1/ova directory.

Runtime options for controlling how VCS writes its report on OpenVera Assertions. You can use them only if you compiled with the `-ova_enable_diag` compile-time option.

`-ova_filter`

Blocks reporting of trivial if-then successes. These happen when an if-then construct registers a success only because the if portion is false (and so the then portion is not checked). With this option, reporting only shows successes in which the whole expression matched.

`-ova_max_fail` *N*
: Limits the number of failures for each assertion to *N*. When the limit is reached, the assertion is disabled. *N* must be supplied, otherwise no limit is set.

`-ova_max_success` *N*
: Limits the total number of reported successes to *N*. *N* must be supplied, otherwise no limit is set. The monitoring of assertions continues, even after the limit is reached.

`-ova_simend_max_fail` *N*
: Terminates the simulation if the number of failures for any assertion reaches *N*. *N* must be supplied, otherwise no limit is set.

`-ova_success`
: Enables reporting of successful matches in addition to failures. The default is to report only failures.

Runtime options for functional coverage(enabled by the `-ova_cov` compile-time option):

`-ova_cov`
: Enables functional coverage reporting.

`-ova_cov_name` *filename*
: Specifies the file name or the full path name of the functional coverage report file.

`-ova_cov_db` *filename*
: Specifies the path name of the initial coverage file. The initial coverage file is needed to set up the database.

SystemVerilog Assertions

`-assert` *keyword_argument*

The keyword arguments are as follows:

`dumpoff`
Disables the dumping of SVA information in the VPD file during simulation.

`filter`
Blocks reporting of trivial implication successes. These happen when an implication construct registers a success only because the precondition (antecedent) portion is false (and so the consequent portion is not checked). With this option, reporting only shows successes in which the whole expression matched.

`finish_maxfail=`*N*
Terminates the simulation if the number of failures for any assertion reaches *N*. *N* must be supplied, otherwise no limit is set.

`global_finish_maxfail=`*N*
Stops the simulation when the total number of failures, from all SystemVerilog assertions, reaches *N*.

`maxfail=`*N*
Limits the number of failures for each assertion to *N*. When the limit is reached, the assertion is disabled. *N* must be supplied, otherwise no limit is set.

`maxsuccess=`*N*
Limits the total number of reported successes to *N*. *N* must be supplied, otherwise no limit is set. The monitoring of assertions continues, even after the limit is reached.

`nopostproc`

Disables the display of the SVA coverage summary at the end of simulation. This summary looks like this for each `cover` statement:

"*source_filename*.v", *line_number*: *cover_statement_hierarchical_name number* attempts, *number* total match, *number* first match, *number* vacuous match

`quiet 0 | 1`

`0`

Disables messages, in standard output, about assertion failures

`1`

Disables messages, in standard output, about assertion failures, but displays a summary of them at the end of simulation. The never triggered assertions are also reported

`report[=`*`path/filename`*`]`

Generates a report file in addition to printing results on your screen. By default this file's name and location is ./assert.report, but you can change it to where you want by entering the filename path name argument.

The filename can start with a number or letter. The following special characters are acceptable in the filename: `%`, `^`, and `@`. Using the following unacceptable special characters: `#`, `&`, `*`, `[]`, `$`, `()`, or `!` has the following consequences:

- A filename containing `#` or `&` results in a filename truncation to the character before the `#` or `&`.
- A filename containing `*` or `[]` results in a `No match` message.

- A filename containing `$` results in an `Undefined variable` message.
- A filename containing `()` results in a `Badly placed ()'s` message.
- A filename containing `!` results in an `Event not found` message.

`success`
Enables reporting of successful matches, and successes on `cover` statements, in addition to failures. The default is to report only failures.

`verbose`
Adds more information to the end of the report specified by the `report` keyword argument and a summary with the number of assertions present, attempted, and failed.

You can enter more than one keyword, using the plus `+` separator, for example:

```
-assert maxfail=10+maxsucess=20+success+filter
```

`-cm assert`
Specifies monitoring for System Verilog assertions coverage. Like at compile-time, `-cm` is not a new runtime option but the `assert` argument is new.

`-cm_assert_dir` *`path/filename`*
Specifies the path and filename of an initial coverage file. An initial coverage file is needed to set up the database. By default, an empty coverage file is loaded from the following directory: simv.vdb/snps/fcov.

CLI Command File

`-i` *`filename`*
: Specifies a file containing CLI commands that VCS executes when simulation starts. After VCS reaches the end of that file VCS takes commands from the standard input. Normally used along with the `-s` runtime option and a `+cli+`*`number`* compile-time option. A typical file for this option is the vcs.key file.

`-k` *`filename`* `|` `off`
: Specifies an alternative name or location for the vcs.key file into which VCS writes the CLI and DVE or VirSim interactive commands that you enter during simulation. The `off` argument tells VCS not to write this file.

`+cliecho`
: Specifies that VCS display the CLI commands in a file that you specify with the `-i` option as VCS executes these CLI commands.

UCLI

`-ucli`
: Enables the use of UCLI commands.

Specifying VERA Object Files

`+vera_load=`*`filename`*`.vro`
: Specifies the VERA object file.

`+vera_mload=`*`filename`*
: Specifies a text file that contains a list of VERA object files.

Coverage Metrics

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

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

`line`
Monitor for line or statement coverage.

`cond`
Monitor for condition coverage.

`fsm`
Monitor for FSM coverage.

`tgl`
Monitor for toggle coverage.

`path`
Monitor for path coverage.

`branch`
Monitor for branch covarage

`assert`
Monitor for SystemVerilog assertion coverage

If you want VCS to monitor for more than one type of coverage, use the plus (+) character as a delimiter between arguments, for example:

```
simv -cm line+cond+fsm+tgl+path
```

The `-cm` option is also a compile-time option and an option on the `cmView` command line.

`-cm_dir` *`directory_path_name`*
: Specifies and alternative name and location for the simv.cm directory. The `-cm_dir` option is also a compile-time option and a `cmView` command line option.

`-cm_glitch` *`period`*
: Specifies a glitch period during which VCS does not monitor for coverage caused by value changes. The period is an interval of simulation time specified with a non-negative integer.

`-cm_log` *`filename`*
: As a compile-time or runtime option, specifies a log file for monitoring for coverage during simulation. As a `cmView` command line option, specifies a log file for writing reports.

`-cm_name` *`filename`*
: As a compile-time or runtime option, specifies the name of the intermediate data files. On the `cmView` command line, specifies the name of the report files.

`-cm_tglfile` *`filename`*
: Specifies displaying at runtime a total toggle count for one or more subhierarchies specified by the top-level module instance entered in the file. This option is also a compile-time option.

Enabling and Disabling Specify Blocks

`+no_notifier`
: Suppresses the toggling of notifier registers that are optional arguments of system timing checks. The reporting of timing check violations is not affected. This is also a compile-time option.

`+no_tchk_msg`

Disables the display of timing violations but does not disable the toggling of notifier registers in timing checks. This is also a compile-time option.

`+notimingcheck`

Disables timing check system tasks in your design. Using this option at runtime can improve the simulation performance of your design, depending on the number of timing checks that this option disables.

You can also use this option at compile-time. Using this option at compile-time tells VCS to ignore timing checks when it compiles your design so the timing checks are not compiled into the executable and this results in a faster simulating executable than one that includes timing checks that are disabled by this option at runtime.

If you need the delayed versions of the signals in negative timing checks but want faster performance, include this option at runtime. The delayed versions are not available if you use this option at compile-time. See “Enabling Negative Timing Checks” on page 9-12.

VCS recognizes `+notimingchecks` to be the same as `+notimingcheck` when you enter it on the vcs or simv command line.

Specify When Simulation Stops

`+vcs+stop+`*`time`*

Stop simulation at the *`time`* value specified. The *`time`* value must be less than 2^{32} or 4,294,967,296.

`+vcs+finish+time`

Ends simulation at the *time* value specified. The *time* value must be also less than 2^{32}.

For both of these options there is a special procedure for specifying time values larger than 2^{32}, see the section "Specifying A Very Long Time Before Stopping Simulation" on page 4-30 .

Recording Output

`-a` *filename*

Specifies appending all messages from simulation to the bottom of the text in the specified file as well as displaying these messages in the standard output.

`-l` *filename*

Specifies writing all messages from simulation to the specified file as well as displaying these messages in the standard output. Usage is letter "`l`" (lowercase "`L`") for log file.

Controlling Messages

`-q`

Quiet mode. Suppress printing of VCS header and summary information.
Suppresses proprietary message at beginning
Suppresses VCS Simulation Report at the end (time, CPU time, data structure size, and date)

`-V`

Verbose mode. Print VCS version and extended summary information.

Prints VCS compile and run-time version numbers, and copyright information, at start of simulation.

`+no_pulse_msg`

Suppresses pulse error messages, but not the generation of StE values at module path outputs when a pulse error condition occurs.

You can enter this run-time option on the vcs command line.

You cannot enter this option in the file you use with the `-f` compile-time option.

`+sdfverbose`

By default VCS displays no more than ten warning and ten error messages about back annotating delay information from SDF files. This option enables the display of all back annotation warning and error messages.

This default limitation on back annotation messages applies only to messages displayed on the screen and written in the simulation log file. If you specify an SDF log file in the $sdf_annotate system task, this log file receives all messages.

`+vcs+nostdout`

Disables all text output from VCS including messages and text from `$monitor` and `$display` and other system tasks. VCS still writes this output to the log file if you include the `-l` option.

Discovery Visual Environment and UCLI

`-gui`

Starts the DVE GUI.

`-ucli`
arts the UCLI debugger command line

`-l` *`log_filename`*
Specifies a log file that contains the commands you entered and the responses from VCS and DVE.

`-i` *`input_filename`*
Specifies a file containing UCLI commands. VCS executes these at the start of simulation.

`-k` *`key_filename`*
File where VCS records the UCLI commands it executes. You can use this file as input with the -i options in a subsequent simulation.

VPD Files

VPD files are simulation history files that VCS writes during simulation and DVE or VirSim reads after simulation to show you the simulation results. These are the runtime options that specify the size and contents of the VPD. You specify these runtime options on the vcs command line.

`+vpdbufsize+`*`number_of_megabytes`*
To gain efficiency, VPD uses an internal buffer to store value changes before saving them on disk. This option modifies the size of that internal buffer. The minimum size allowed is what is required to share two value changes per signal. The default size is the size required to store 15 value changes for each signal but not less than 2 megabytes.

Note:
The buffer size automatically is increased as needed to comply with the above limit.

`+vpdfile+`*`filename`*

Specifies the name of the output VPD file (default is `vcdplus.vpd`). You must include the full file name with the `.vpd` extension.

`+vpdfilesize +`*`number_of_megabytes`*

Creates a VPD file, which has a moving window in time while never exceeding a specified file size *`number_of_megabytes`*. When the VPD file size limit is reached, VPD will continue saving simulation history by overwriting older history.

File size is a direct result of circuit size, circuit activity, and the data being saved. Test cases show that VPD file sizes will likely run from a few megabytes to a few hundred megabytes. Many users can share the same VPD history file, which may be a reason for saving all time value changes when you do simulation. You can save one history file for a design and overwrite it on each subsequent run.

`+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.

`+vpddrivers`

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

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

`+vpdports`

By default, VPD does not store the port type for each signal, and the Hierarchy Browser views all signals as internal and not connected to a port.

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

`+vpdnocompress`

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

`+vpdnostrengths`

By default, VPD stores strength information on value changes to the VPD file. Use of this option may lead to slight improvements in simulator performance.

Controlling $gr_waves System Task Operations

`-grw` *`filename`*

Sets the name of the `$gr_waves` output file to the specified *filename*.

Default filename is grw.dump.

`+vcs+grwavesoff`

Suppress `$gr_waves` system tasks.

VCD Files

`-vcd` *`filename`*

Set name of `$dumpvars` output file to filename.The default filename is verilog.dump. A `$dumpfile` system task in the Verilog source code will override this option.

`+vcs+dumpoff+`*`t`*`+`*`ht`*

Turn 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`*

Suppress `$dumpvars` system task until time *`t`*. *`ht`* is the high 32 bits of a time value greater than 32 bits.

`+vcs+dumpvarsoff`

Suppress `$dumpvars` system tasks.

`+vcs+dumparrays`

Enables recording memory and multi-dimensional array values in the VCD file. You must also have used the `+memcbk` compile-time option.

Specifying Min:Typ:Max Delays

`+maxdelays`

Specifies using the maximum delays in min:typ:max delay triplets in module path delays and timing check 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 and Synopsys hardware models 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 check if you compiled your design with the `+allmtm` compile-time option.

Also specifies using the minimum timing delays in min:typ:max delay triplets in an uncompiled SDF file.

If you compiled the SDF file with the `+allmtm` compile-time option, the `+mindelays` option specifies using the compiled SDF file with the minimum delays.

Another use for this runtime option is to specify timing for SWIFT VMC and SmartModels and Synopsys hardware models 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 check 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 your source code and in uncompiled SDF files (unless you specify otherwise with the *`mtm_spec`* argument to the `$sdf_annotate` system task, see the section "The $sdf_annotate System Task" on page 8-24), and 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 and Synopsys hardware models when you also include the `+override_model_delays` runtime option.

Flushing Certain Output Text File Buffers

When VCS creates a log file, VCD file, or a text file specified with the `$fopen` system function, VCS writes the data for the file in a buffer and periodically dumps the data from the buffer to the file on disk. The frequency of these dumps varies depending on many factors including the amount of data that VCS has to write to the buffer as simulation or compilation progresses. If you need to see or use the latest information in these files more frequently than the rate at which VCS normally dumps this data, these options tell VCS to dump the data more frequently. How much more frequently also depends on many factors but the increased frequency will always be significant.

`+vcs+flush+log`
: Increases the frequency of dumping both the compilation and simulation log files.

`+vcs+flush+dump`
: Increases the frequency of dumping all VCD files.

`+vcs+flush+fopen`
: Increases the frequency of dumping all files opened by the `$fopen` system function

`+vcs+flush+all`
: Increases the frequency of dumping all log files, VCD files, and all files opened by the `$fopen` system function

These options do not increase the frequency of dumping other text files including the VCDE files specified by the `$dumpports` system task or the simulation history file for LSI certification specified by the `$lsi_dumpports` system task.

These options can also be entered at compile-time. There is no particular advantage to entering them at compile-time.

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`
: Wait for network license if none is available when the job starts.

General Options

See The Compile-Time Options Used To Create The Executable

`-E` *`program`*

Starts the *program* that displays the compile-time options that were on the vcs command line when you created the simv (or simv.exe) executable file. For example:

```
simv -E echo
simv -E echo > simvE.log
```

You cannot use any other runtime options with the `-E` option.

Stop Simulation When The Executable Starts

`-s`

Stops simulation just as it beings, and enters interactive mode. Use with the `+cli+`*`number`* option.

Record Where ACC Capabilities are Used

`+vcs+learn+pli`

ACC capabilities enable debugging operations but they have a performance cost so you only want to enable them where you need them. This option keeps track of where you use them for debugging operations so that you can recompile your design and in the next simulation enable them only where you need them. When you use this option VCS writes the pli_learn.tab secondary PLI table file. You input this file when you recompile your design with the `+applylearn` compile-time option. See the section "Globally Enabling ACC Capabilities" on page 7-21 .

Suppress the $stop System Task

`+vcs+ignorestop`

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

User-Defined Plusarg Options

`+`*`plus-options`*

User-defined run-time options, which can be checked for by the `$test$plusargs` system task to perform some operation when the option is on the simv command line. For an example of checking for a user-defined plusarg run-time option see the section "Enabling Debugging Features At Runtime" on page 2-10 .

Enable Overriding The Timing of a SWIFT SmartModel

`+override_model_delays`

Enables the `+mindelays`, `+typdelays`, and `+maxdelays` runtime options to specify the timing used by SWIFT SmartModels instead of using the DelayRange parameter definition on the template file. See the section "Changing The Timing of A Model" on page 10-16 .

acc_handle_simulated_net PLI routine and MIPD annotation

`+vcs+mipd+noalias`

For the PLI routine acc_handle_simulated_net, aliasing of a loconn net and a hiconn net across the port connection is disabled if MIPD delay annotation happens for the port. If you specify acc capability: mip or mipb in the pli.tab file, such aliasing is disabled only when actual MIPD annotation happens.

If during a simulation run, acc_handle_simulated_net is called before MIPD annotation happens, a warning message is issued. 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.

acc_handle_simulated_net is defined on IEEE Std 1364-1995 page 366.

Save and Restart

VCS provides a save and restart feature that allows checkpoints of the simulation to be saved at arbitrary times. The resulting checkpoint files can be executed at a later time, causing simulation to resume at the point immediately following the save.

Benefits of save and restart include:

- Regular checkpoints for interactively debugging problems found during long batch runs
- Use of plusargs to start action such as `$dumpvars` on restart
- Executes common simulation system tasks such as `$reset` just once in a regression

Restrictions of save and restart include:

- Requires extra Verilog code to manage the save and restart
- Must duplicate start-up code if handling plusargs on restart

- File I/O suspend and resume in PLI applications must be given special consideration

Save and Restart Example

A simple example of save and restart is given below to demonstrate the basic functionality.

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

```
% 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 to compile the example Source file:

```
vcs test.v
```

Now run the simulation:

```
simv
```

VCS displays the following:

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

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

```
test.chk
```

VCS displays the following:

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

Save and Restart File I/O

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

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

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

Save and Restart with Runtime Options

If your simulation behavior depends on the existence of run-time `plusargs` or any other run-time 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 run-time events after the restart action. Depending on the complexity of your environment and your usage of the save and restart feature, this can be a significant task.

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

A more detailed example follows to illustrate this. The first run optimizes simulation speed by omitting the `+dump` flag. If a bug is found, the latest checkpoint file is run with the `+dump` flag to enable signal dumping.

```
// file test.v
module dumpvars();
task processargs;
     begin
```

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

Restarting at The CLI Prompt

The `$restart` system task has been implemented to allow you to enter it at the CLI prompt. You enter it with the check file name created by the `$save` system task, for example:

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

Specifying A Very Long Time Before Stopping Simulation

You can use the `+vcs+stop+`*`time`* runtime option to specify the simulation time when VCS halts simulation if the *`time`* value you specify is less that 2^{32} or 4,294,967,296. You can also use the `+vcs+finish+`*`time`* runtime option to specify when VCS not just halts but ends simulation but also under the proviso that the time value be less than 2^{32}.

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

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

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

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

   This difference is the first argument.

   You can let VCS do some of this work for you, in this example using the following source code:

```
module wide_time;
time wide;
initial
begin
wide = 64'd10_000_000_000;
```

```
$display("Hi=%0d, Lo=%0d", wide[63:32],
wide[31:0]);
end
endmodule
```

VCS displays:

```
Hi=2,Lo=1410065408
```

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

   In this example:

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

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

   In this example, you round down to 2.

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

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

Passing Values From The Runtime Command Line

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

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

The *plusarg_format* argument specifies a user-defined runtime option for passing a value to the specified signal. It specifies the text of the option and the radix of the value that you pass to the signal.

The following code example contains this system function:

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

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

If you enter the following simv command line:

```
simv +r1=10
```

The `$monitor` system task displays the following:

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

Performance Considerations

When you simulate your design you can look for ways to improve the simulation performance. There are run-time options that enable VCS to simulate faster or slower. There is also a technique called profiling that tells you which modules in your design take the most CPU time.

Runtime Options that Accelerate or Impede Simulation

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

`+vcs+dumpvarsoff`Suppress `$dumpvars` system tasks.

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

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

The run-time options that specify writing to a file that slow down simulation. These run-time options are as follows:

`-a` *`filename`* All output of simulation is appended to filename as well as sent to the standard output.

`-l` *`filename`* All output of simulation is written to filename as well as to the standard output.

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

Profiling the Simulation

If you include the `+prof` compile-time option when you compile your design, VCS generates the vcs.prof file during simulation. This file contains a profile of the simulation. It reports the following:

- The module instances in the hierarchy that use the most CPU time
- The module definitions whose instances use the most CPU time
- The Verilog constructs in those instances that use the most CPU time

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

The profile data in the vcs.prof file is organized into a number of “views” of the simulation. They are as follows:

- Top Level View
- Module View
- Instance View
- Module Construct Mapping View
- Top Level Construct View
- Construct View Across Design

The Top Level View

This view shows you how much CPU time was used by:

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

The following is an example of the top level view:

```
==========================================================================
                           TOP LEVEL VIEW
==========================================================================
                         TYPE       %Totaltime
--------------------------------------------------------------------------
                          PLI          0.00
                          VCD          0.00
                       KERNEL         29.06
                       DESIGN         70.94
--------------------------------------------------------------------------
```

In this example there is no PLI application and VCS is not writing a VCD or VPD file. VCS used 70.94% of the CPU time to simulate the design, and 29.06% for internal operations, such as scheduling, that can't be attributed to any part of the design. The designation KERNEL, is for these internal operations.

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

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

The Module View

This view shows you the module definitions whose instances use the most CPU time. Module definitions whose module instances collectively used less than 0.5% of the CPU time are not listed.

The following is an example of the module view:

```
==========================================================================
                            MODULE VIEW
==========================================================================
Module(index)               %Totaltime    No of Instances   Definition
--------------------------------------------------------------------------
FD2                 (1)        62.17          10000         design.v:142.
```

```
EN                  (2)         8.73          1000          design.v:131.
--------------------------------------------------------------------------
```

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

The profile data for module FD2 is as follows:

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

The Instance View

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

The following is an example of the instance view:

```
==========================================================================
                          INSTANCE VIEW
==========================================================================
```

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

-------------------------------------------------------------------------
```

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

This instance's hierarchical name is test.lfsr1000_1.lfsr100_1.lfsr10_1.lfsr_1.en_1. Long hierarchical names wrap to the next line.

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

This instance used 0.73% of the CPU time.

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

Note:

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

The Module to Construct Mapping View

This view shows you the CPU time used by different types of Verilog constructs in each module definition in the module view. There are the following types of Verilog constructs:

- always constructs (commonly called always blocks)
- initial constructs (commonly called initial blocks)

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

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

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

The following is an example of a module to construct mapping view:

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


________________________________________________________________________________
                              1. FD2
--------------------------------------------------------------------------------
Construct type    %Totaltime    %Moduletime     LineNo
--------------------------------------------------------------------------------
Always               27.44         44.14        design.v : 150-160.
Module Path          23.17         37.26        design.v : 165-166.
Timing Check         11.56         18.60        design.v : 167-168.
________________________________________________________________________________
```

```
_____________________________________________________________________________
                              2. EN
-----------------------------------------------------------------------------
Construct type    %Totaltime    %Moduletime     LineNo
-----------------------------------------------------------------------------
Combinational        8.73          100.00       design.v:  137.
_____________________________________________________________________________
```

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

`%Totaltime`
: The percentage of the total CPU time that was used by this construct.

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

In the section for module FD2:

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

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

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

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

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

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

The Top Level Construct View

This view show you the CPU time used by different types of Verilog constructs throughout the design. The following is an example of the Top Level Construct View:

```
=============================================================================
                        TOP-LEVEL CONSTRUCT VIEW
-----------------------------------------------------------------------------
                  Verilog Construct          %Totaltime
-----------------------------------------------------------------------------
                             Always            27.47
                        Module Path            23.17
                       Timing Check            11.56
                      Combinational             8.73
                            Initial             0.00
                           Function             0.00
                               Task             0.00
                               Port             0.00
                                Udp             0.00
                          Protected             0.00


_____________________________________________________________________________
```

This view contains a line for each type of construct, for example, all the always blocks in the design used 27.47% of the CPU time.

The Construct View Across Design

This view shows you the module definitions that contains a type of construct that used more that 0.5% of the CPU time. There are separate sections for each type of construct and a listing in each section of the modules that contain that type of construct.

The following is an example of the construct view across design:

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


________________________________________________________________________________
                             1.Always
--------------------------------------------------------------------------------
                         Module          %TotalTime
--------------------------------------------------------------------------------
                            FD2            27.44
________________________________________________________________________________


________________________________________________________________________________
                             2.Module Path
--------------------------------------------------------------------------------
                         Module          %TotalTime
--------------------------------------------------------------------------------
                            FD2            23.17
________________________________________________________________________________


________________________________________________________________________________
                             3.Timing Check
--------------------------------------------------------------------------------
                         Module          %TotalTime
--------------------------------------------------------------------------------
                            FD2            11.56
________________________________________________________________________________
```

```
_____________________________________________________________________
                          4.Combinational
---------------------------------------------------------------------
                    Module          %TotalTime
---------------------------------------------------------------------
                          EN             8.73
_____________________________________________________________________
```

How VCS Prevents Time 0 Race Conditions

At simulation time 0, VCS always executes the always blocks in which any of the signals in the event control expression that follows the `always` keyword (the sensitivity list) initialize 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 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 rst initialized.
- The simulator evaluates the event control sensitivity list for the always block, and so far reg rst has not changed value during this time step so the simulator does not execute the always block, then the simulator executes the initial block and initializes rst. When this happens the simulator does not re-evaluate the event control sensitivity list for the always block.

Protected and Portable Verilog Model

After a design is successfully verified using VCS, there is a separate tool available for those who want to make the design portable and protected. This tool, Verilog Model Compiler (VMC), enables you to secure your design and distribute it to your partners and internal or external customers without an NDA.

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

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

For more information about VMC, refer to the IP Modeling product or the following web site: http://www.synopsys.com/products/lm/ip/ip.html

For information on instantiating and simulating VMC models, see the *VMC Users Manual* and ‘SWIFT VMC Models and SmartModels Introduction” on page 10-2.

5

Debugging and Race Detection

VCS provides the following features for debugging your design and detecting race conditions:

- The non-graphical debugger or CLI (Command Line Interface). VCS has a command language for debugging with commands for, among other things, forcing values and displaying values. You can tell VCS to halt before the first simulation event and display a command prompt for you to enter these commands. See “Non-Graphical Debugging With The CLI” on page 5-2.

Note:

There now is a Unified Command Line Interface for debugging commands. It is unified in that the same command line interface works for VCS, VCS MX, and Vera. It has more commands that the CLI. See the *Unified Command Line Interface User Guide*.

- The DVE or VirSim graphical user interfaces. These debugging interfaces have windows for entering interactive commands and

seeing messages, displaying the design hierarchy, showing module definitions and the order that VCS executes statements in these module definitions, displaying the changing values of selected nets and registers, and displaying waveforms.
You can use DVE or VirSim interactively during simulation or have DVE or VirSim write a simulation history file (VCD+ file) during simulation, and then have DVE or VirSim display the simulation results after simulation. See Chapter 12, “Using VirSim to Debug Your Design.”

- The dynamic race detection tool that finds race conditions during simulation. See “The Dynamic Race Detection Tool” on page 5-10.
- The static race detection too that finds race conditions by analyzing your source code during compilation. See “The Static Race Detection Tool” on page 5-22.

Non-Graphical Debugging With The CLI

Note:
There now is a Unified Command Line Interface for debugging commands. It is unified in that the same command line interface works for VCS, VCS MX, and Vera. It has more commands that the CLI. See the *Unified Command Line Interface User Guide*.

This sections describes use of the VCS interactive non-graphical debugging capability similar in concept to UNIX debuggers such as dbx and gdb. A model compiled with interactive debugging enabled can enter a CLI (Command Line Interface) command at runtime. A command language allows you to set breakpoints, examine the values of registers and wires, and change register values.

In order to use the CLI you have to enable it at compile-time. For most CLI commands there are two ways to do it:

- Use a PLI table file, see “Specifying ACC Capabilities for VCS Debugging Features” on page 7-17.
  This is the method Synopsys recommends. It enables CLI commands to work on only the parts of a design you need.
- Use the `+cli` compile-time option, see “Debugging” on page 3-33. This enables CLI commands to work throughout the design but in doing so, in most cases, slows down simulation significantly more than using the PLI table file.

Some CLI commands: `line`, `next`, and `trace` are enabled with the `-line` compile-time option.

After enabling the CLI you start the CLI interface by using one of the following:

- Include the `-s` on the runtime command line (e.g. `simv -s`).
- Include a `$stop` system task within one of the source files.
- Interrupt the simulation with `<ctrl>-C`.

Interactive Command Language

The following is a summary of the CLI interactive command language:

`.` (period)
Continue simulation.

`?`
Displays a list of the CLI commands and briefly describes what they do.

`alias [alias_name [existing_command]]`
Create or list command alias(es).

`always [#relative_time|@posedge|@negedge] net_or_reg`
Set a repeating breakpoint. A breakpoint halts simulation when the net or register changes value.

`break [#relative_time|@posedge|@negedge] net_or_reg`
Set a repeating breakpoint.

`continue`
Continue simulation.

`define [name [definition]]`
Create or list macro definition(s). Macro usage is `name.

`delete breakpoint_number`
Delete a breakpoint.

`finish`
Exit simulation.

`force net_or_reg = value`
Force a net or a register to a value. Simulation events in your design do not override this value.

`help`
Display this list.

`info`
Display time and scope information.

`line`
Toggle line tracking (requires the use of the `-line` compile-time option).

`list [-n | n]`
Lists 10 lines starting with the current line.

`-n`
Lists `n` lines above the current position.

`n`
Lists `n` lines forward starting with the current position.

`next`
Next line (requires the use of the `-line` compile-time option).

`oformat %[b|c|t|f|e|g|d|h|x|m|o|s|v]`
Set output format as described for format specifications in IEEE Std 1364-1995 pages 174-175. x is an alternative for hexadecimal.

`once [#`*`relative_time`*`|##`*`absolute_time`*`|@posedge|@negedge]` *`net_or_reg`*
Set a one shot breakpoint.

`print %[b|c|t|f|e|g|d|h|x|m|o|s|v]` *`net_or_reg`*
Shows the current value of net or register in the specified format.

`release` *`net_or_reg`*
Releases a net or register from its forced value.

`scope [`*`module_instance_hierarchical_name`*`]`
Set or show scope. A scope is a location in the module hierarchy from which you can see and change values with the CLI.

`set` *`reg_or_memory_address`* `[=]`*`value`* `[,`*`reg_or_memory_address`*`[=]`*`value`*`]`
Deposits a value on a register or a memory address. Simulation events in your design can override this value. This command is not valid for nets. The equal sign (=) operator is optional.

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

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

`drivers` *`net_or_reg`*
Shows the value and strength of the net or register and for nets shows the line number in the source code of the statement that is the source of the value that propagated to this net.

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

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

`variables`
Shows the nets and registers declared in the current scope.

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

`source` *`filename`*
Executes CLI commands from a file.

`tbreak [#`*`relative_time`*`| ##`*`absolute_time`*`| @posedge| @negedge]` *`net_or_reg`*
Set a one shot breakpoint. This command is identical to the `once` command.

`trace`
Toggle on event trace mode (requires the use of the `-line` compile-time option).

`unalias` *`alias_name`*
Removes the alias.

`undefine name`
Removes macro definition.

`upscope`

Sets scope one level higher in the module hierarchy.

The commands `always`, `break`, `once` and `tbreak` accept a relative or absolute time argument. Format of the relative/absolute time argument is as follows:

`digits``[.`*`digits`*`][`*`units`*`]`

Where *`digits`* is a string of decimal digits and underscores, and *`units`* is one of fs, ps, ns, us, ms or s. Examples:

```
always #10ps
tbreak ##0.01ns
once #3_123.3fs
```

Note:

There is a limit to the largest number you can enter for a simulation time number, no matter what time unit you specify. That limit is 2^{32} or 4,294,967,296.

Command Files

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

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

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

Example 5-1 Interactive Debugging Example

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

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

Details of VCS compilation omitted.

```
% simv -s
$stop at time 0
cli_0 > scope
Current scope is top
cli_1 > show var
Reg               a
Reg [31:0]        b
cli_2 > once #1
cli_3 > .
Time break at time 1 breakpoint #1 tbreak ##1
cli_4 > print a
a:   0
cli_5 > set a=1
cli_6 > print a
a:   1
cli_7 > tbreak b
cli_8 > .
Value break time 10 breakpoint #2 tbreak top.b
```

```
cli_9 > print b
b:       00000001
cli_10 > quit
$finish at simulation time 10
             V C S     S i m u l a t i o n  R e p o r t
Time: 10
CPU Time: 0.150 seconds; Data structure size: 0.0Mb
```

Key Files

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

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

You can use the `-k` runtime option to specify a different name or location for the vcs.key file. You can also use this option to tell VCS not to write this file.

The Dynamic Race Detection Tool

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

read - write race condition

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

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

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

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

write - write race condition

When a procedural assignment in one always or initial block, or a continuous assignment assigns a value to a signal (write) at the same time that a procedural assignment in another always or initial block, or another continuous assignment assigns a value to that signal (write). For example:

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

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

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

Finding these race conditions is important because in Verilog simulation you cannot control the order of execution of statements in different always of 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 any simulator, or with different optimizations or performance features of the same version of any simulator.

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

The indications of a race condition are therefore the following:

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

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. VCS can write report files about the race conditions in your design.

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

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

Enabling Race Detection

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

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

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

Specifying The Maximum Size of Signals in Race Conditions

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

`+race_maxvecsize=`*`size`*

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

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

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

The Race Detection Report

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

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

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

Note:

The race.unique.out is automatically created by the PostRace.pl Perl script after simulation. This script needs a perl5 interpreter. The first line of the script points to perl at a specific location, see “Modifying the PostRace.pl Script” on page 5-18. If that location at your site is not a perl5 interpreter, the script fails with syntax errors.

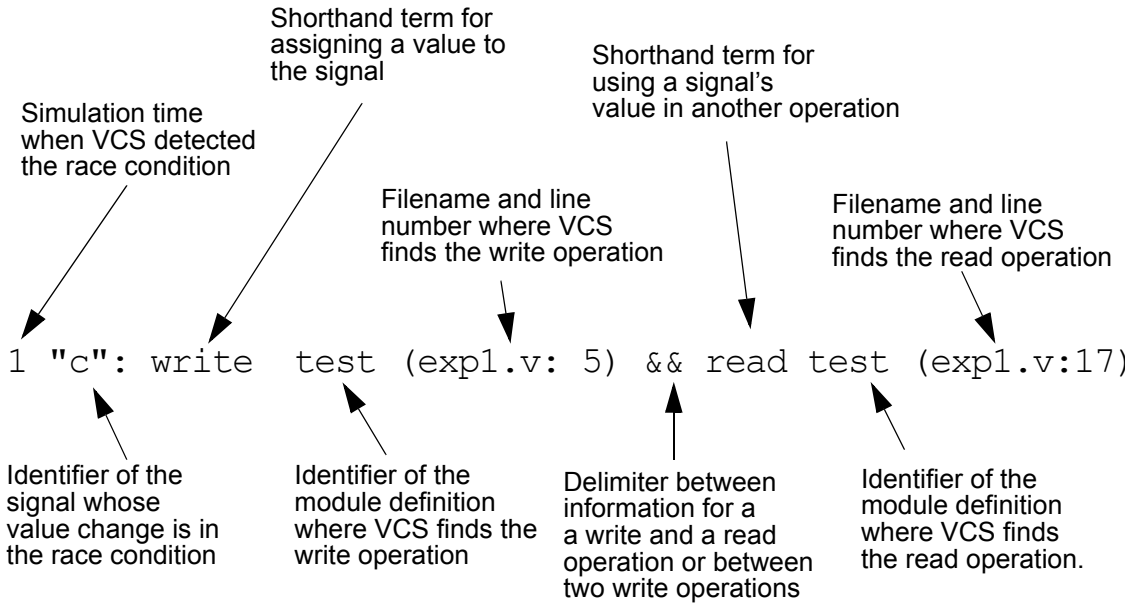
The report describes read - write and write - write race conditions. The following is an example of the contents of a small `race.out` file:

```
                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 a write - write race to a signal at the same simulation time, but the two statements that are assigning to the signal are assigning the same value. This is a race of no consequence sand 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 shows you a subset of the race conditions in the race.out file. You include options on the command line for the PostRace.pl script to specify this subset. These options are as follows:

`-hier` *`module_instance`*

Specifies the hierarchical name of a module instance. The new report lists only the race conditions found in this instance and all module instances hierarchically under this instance.

`-sig` *`signal`*
: Specifies the signal that you want to examine for race conditions. You can only specify one signal and do not include a hierarchical name for the signal. If two signals in different module instances have the same identifier, the report lists race conditions for both signals.

`-minmax` *`min max`*
: Specifies the minimum, or earliest, simulation time and the maximum, or latest, simulation time in the report

`-nozero`
: Omits race conditions that occur at simulation time 0.

`-uniq`
: Omits race conditions that also occurred earlier in the simulation. The output is the same as the contents of the race.unique.out file.

`-f` *`filename`*
: Specifies the name of the input file. Use this option if you changed the name of the race.out file

`-o` *`filename`*
: The default name of the output file is race.out.post. If you want a different name, specify this name with this option.

You can enter on PostRace.pl command line more than one of these options.

If you enter an option more than once, the script uses the last of these multiple entries.

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

The following is an example of the command line:

```
PostRace.pl  -minmax 80 250 -f mydesign.race.out -o
mydesign.race.out.post
```

In this example the output file is named mydesign.race.out.post and reports on the race conditions between 80 and 250 time units. The post-process file is named mydesign.race.out.

Modifying the PostRace.pl Script

The first line of the PostRace.pl Perl script is as follows:

```
#! /usr/local/bin/perl
```

If Perl is installed at a different location at your site you need to modify the first line of this script. This script needs a perl5 interpreter. You will find this script at: *vcs_install_dir*/bin/PostRace.pl

---

Debugging Simulation Mismatches

A design can contain several race conditions where many of them behave the same in different simulations so they are not the cause of a simulation mismatch. For a simulation mismatch you want to find the “critical races,” the race conditions that cause the simulation mismatch. This section describes how to do this.

You will need the to run the simulations that mismatch over again, and this time first recompile with the `+race` or `+racecd` options and modify the source code to add system tasks to generate VCD files.

When you have two VCD files, find their differences with the vcdiff utility. The command line for this utility is as follows:

```
vcdiff vcdfile1.dmp vcdfile2.dmp -options > output_filename
```

If you enter the vcdiff command without arguments, you see usage information including the options. This utility is located in the *vcs_install_dir/*bin directory.

Method 1: If the Number of Unique Race Conditions is Small

A unique race condition is a race condition that can occur several times during simulation but only the first occurrence is reported in the race.unique.out file. If there aren't many lines in the race.unique.out file than the number of unique race conditions is small. If so, for each signal in the race.unique.out file:

1. Look in the output file from the vcdiff utility. If the signal values are different, you have found a critical write - write race condition.
2. If the signal values are not different, look for the signals that are assigned the value of this signal or assigned expressions that include this signal (read operations).
3. If the values of these other signals are different at any point in the two simulations, note the simulation times of these differences on the other signals, and post process the race.out file looking for race conditions in the first signal at around the simulation times of the value differences on the other signals. Specify simulation times just before and just after the time of these differences with the `-minmax` option. Enter:

   ```
   PostRace.pl -sig first_signal -minmax time time2
   ```

   If the race.out.post file contains the first signal, then it is a critical race condition and must be corrected.

Method 2: If the Number of Unique Races is Large

If there are many line in the race.unique.out file than there are a large number of unique race conditions. If so one method of finding the critical race conditions is to do the following:

1. Look in the output file from the vcdiff utility for the simulation time of the first difference in simulation values.
2. Post process the race.out file looking for races at the time of the first simulation value difference.Specify simulation times just before and just after the time of these differences with the `-minmax` option. Enter:

   ```
   PostRace.pl -minmax time time2
   ```

3. For each signal in the resulting race.out.post file:

   a. If the simulation values differ in the two simulations than the race condition in the race.out.post file is a critical race condition.

   b. If the simulation values are not different, check the signals that are assigned the value of this signal or assigned expressions that include this signal. If the values of these other signals are different than the race condition in the race.out.post file is a critical race condition.

Method 3: An Alternative When the Number of Unique Race Conditions is Large

1. Look in the output file from the vcdiff utility for the simulation time of the first difference in simulation values.
2. For each signal that has a difference at this simulation time:

   a. Traverse the signal dependency backwards in the design until you find a signal whose values are the same in both simulations.

b. Look for a race condition on that signal at that time. Enter:

```
PostRace.pl -sig signal -minmax time time2
```

If there is a race condition at that time on that signal, it is a critical race condition.

The Static Race Detection Tool

It is possible for a group of statements to combine to form a loop that is a race condition in that the loop will be executed more than once by other Verilog simulators but only once by VCS.

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

You start the static race detection tool with the `+race=all` compile-time option (not the `+race` compile-time option, which compiles your source code so that VCS runs the dynamic race detection tool during simulation).

After compilation the static race detection tool writes the file named race.out.static that reports on the race conditions it finds.

The following is a excerpt from a line numbered source code example that shows such an `always` block that combines with other statements to form such 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’.
```

6

Using Radiant Technology

VCS's Radiant Technology applies performance optimizations to your design while VCS compiles your source code. These Radiant optimizations improve the simulation performance of all types of designs from behavioral and RTL to gate-level designs. Radiant Technology particularly improves the performance of functional simulations where there are no timing specifications or when delays are distributed to gates and assignment statements.

Compiling For Radiant Technology

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

`+rad`
: Specifies using Radiant Technology

`+optconfigfile`

Specifies applying Radiant Technology optimizations to part of the design using a configuration file, see “Applying Radiant Technology to Parts of The Design” on page 6-3.

Known Limitations

Radiant Technology is not applicable to all simulation situations. Some features of VCS are not supported when you use Radiant Technology. This section describes these limitations.

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.

Signed Arithmetic Extensions

You cannot use Radiant Technology if your design contains signed arithmetic extensions that enable more data types to have positive, zero, or negative values. See “Signed Arithmetic Extensions” on page 2-29.

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

Coverage metrics is supported with Radiant Technology, you can enter both the `+rad` and `-cm` compile-time options, but 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 few modules have changed. However, VCS still only performs the code generation phase on the parts of the design that have actually changed. Therefore the incremental compile times will be longer when you use Radiant Technology but will be shorter than a full recompilation of the design.

Applying Radiant Technology to Parts of The Design

The configuration file enables you to apply Radiant optimizations selectively to different parts of your design. You can enable or disable Radiant optimizations for all instances of a module, specific instances of a module, or specific signals.

You specify the configuration file with the `+optconfigfile` compile-time option, for example:

```
+optconfigfile+filename
```

Note:

The configuration file is a general purpose file that has other purposes, such as two state simulation and 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 to enable or disable 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};
```

Where:

`module`
: Keyword that specifies that the attributes in this statement apply to all instances of the modules in the list, specified by module identifier.

`list_of_module_identifiers`
: A comma separated list of module identifiers enclosed in curly braces: `{ }`

`list_of_attributes`
: A comma separated list of Radiant optimization attributes enclosed in curly braces: `{ }`

`instance`
: Keyword that specifies that the attributes in this statement apply to:

  - All instances of the modules in the list specified by module identifier.
  - All module instances in the list specified by their hierarchical names and all the other instances as well. VCS determines the module definition for each module instance specified and applies the attributes to all instances of the module not just the specified module instance.
  - The individual signals in the list specified by their hierarchical names.

`list_of_module_identifiers_and_hierarchical_names`
: A comma separated list of module identifiers and hierarchical names of module instances and signals enclosed in curly braces: `{ }`

`tree`
: Keyword that specifies that the attributes in this statement apply to all instances of the modules in the list, specified by module identifier, and also apply to all module instances hierarchically under these module instances.

`depth`
: An integer that specifies how far down the module hierarchy, from the specified modules, you want to apply Radiant optimization attributes. You can specify a negative value. A negative value specifies descending to the leaf level and counting up levels of the hierarchy to apply these attributes. This specification is optional. Enclose this specification in parentheses: `()`

The valid Radiant optimization attributes are as follows:

`noOpt`
: Disables Radiant optimizations on the module instance or signal.

`noPortOpt`
: Prevents port optimizations such as optimizing away unused ports on a module instance.

`Opt`
: Enables all possible Radiant optimizations on the module instance or signal.

`PortOpt`
: Enables port optimizations such as optimizing away unused ports on a module instance.

Statements can use more than one line and must end with a semicolon `;`.

The Verilog comment characters `/* comment */` and `// comment` also work in the configuration file.

Configuration File Statement Examples

The following are example of statements in a configuration file.

module statement example

```
module {mod1, mod2, mod3} {noOpt, PortOpt};
```

In this module statement example, Radiant optimizations are disabled for all instances of modules mod1, mod2, and mod3 with the exception of port optimizations.

multiple module statement example

```
module {mod1, mod2} {noOpt};
module {mod1} {Opt};
```

In this example of two module statements, Radiant optimizations are disabled for all instances of modules mod1 and mod2 in the first module statement and then Radiant optimizations are enabled for all instances of module mod1 in the second module statement. Statements are processed in the order that they appear in the configuration file so the disabling of optimizations for instances of module mod1 in the first statement is overridden by the second statement.

instance statement example

```
instance {mod1} {noOpt};
```

In this example mod1 is a module identifier so Radiant optimizations are disabled for all instances of mod1. This statement is the equivalent of:

```
module {mod1} {noOpt};
```

module and instance statement example

```
module {mod1} {noOpt};
instance {mod1.mod2_inst1.mod3_inst1,
mod1.mod2_inst1.rega} {noOpt};
```

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

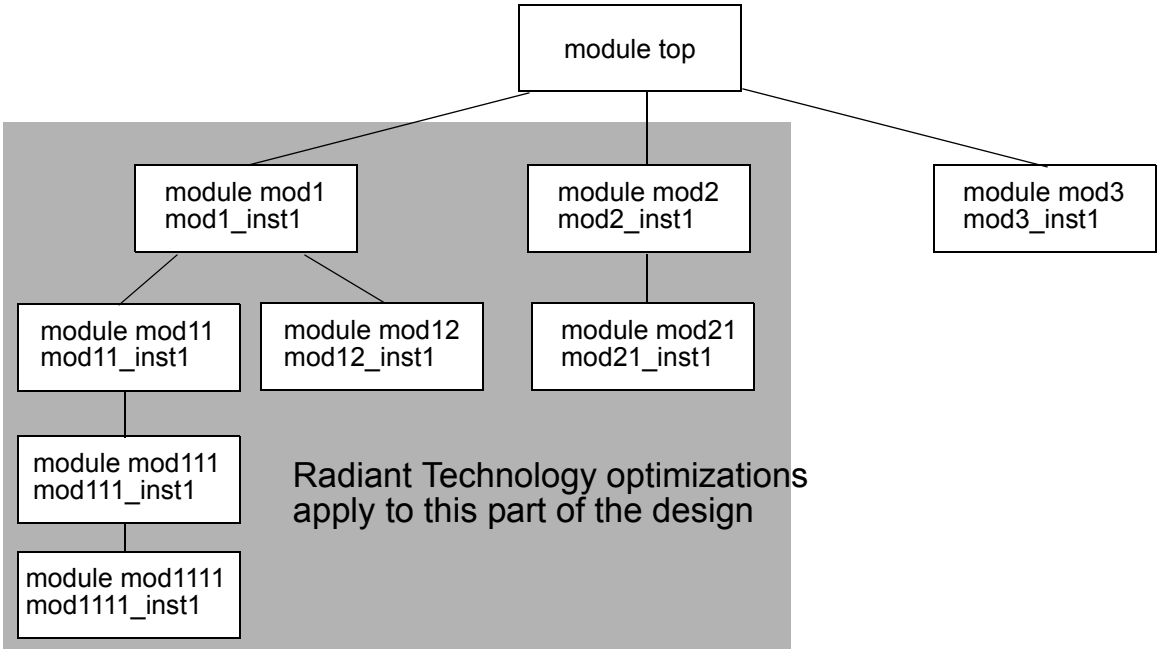
The instance statement disables Radiant optimizations for the following:

- module mod1 (already disabled by the module statement)
- the module instance with the instance identifier mod2_inst1 in mod1
- the module instance with the instance identifier mod3_inst1 under module instance mod2_inst1
- signal rega in module instance mod2_inst1.

first tree statement example

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

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



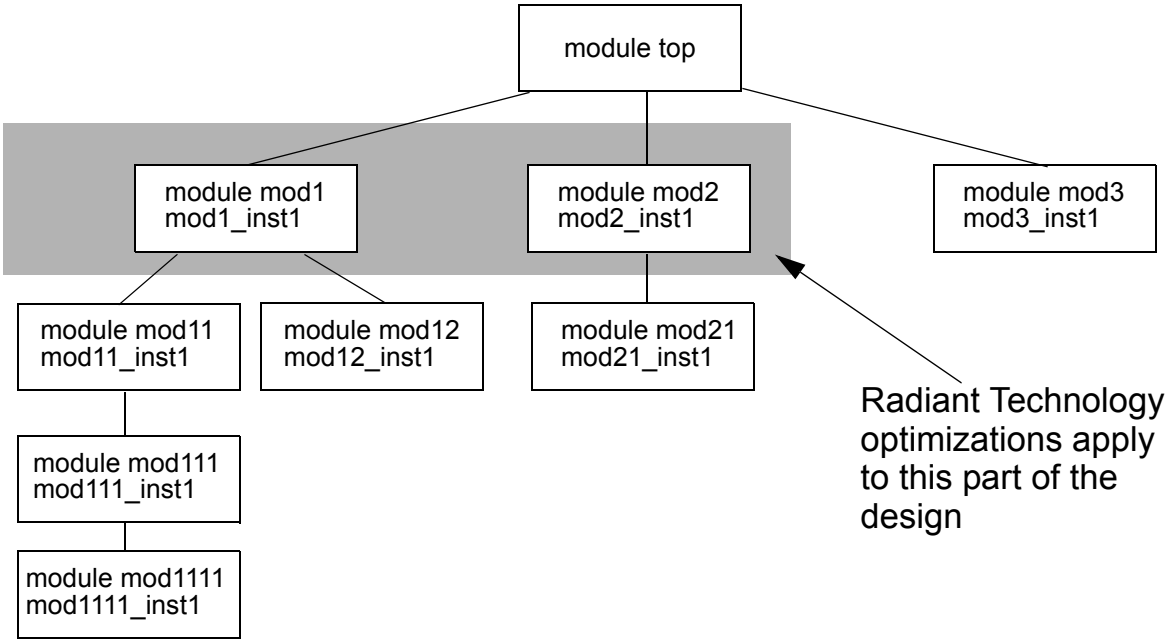
The statement enables Radiant Technology optimizations for the instances of modules mod1 and mod2 and all the for all the module instances hierarchically under these instances.

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 that the instances of the specified modules, mod1 and mod2.



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

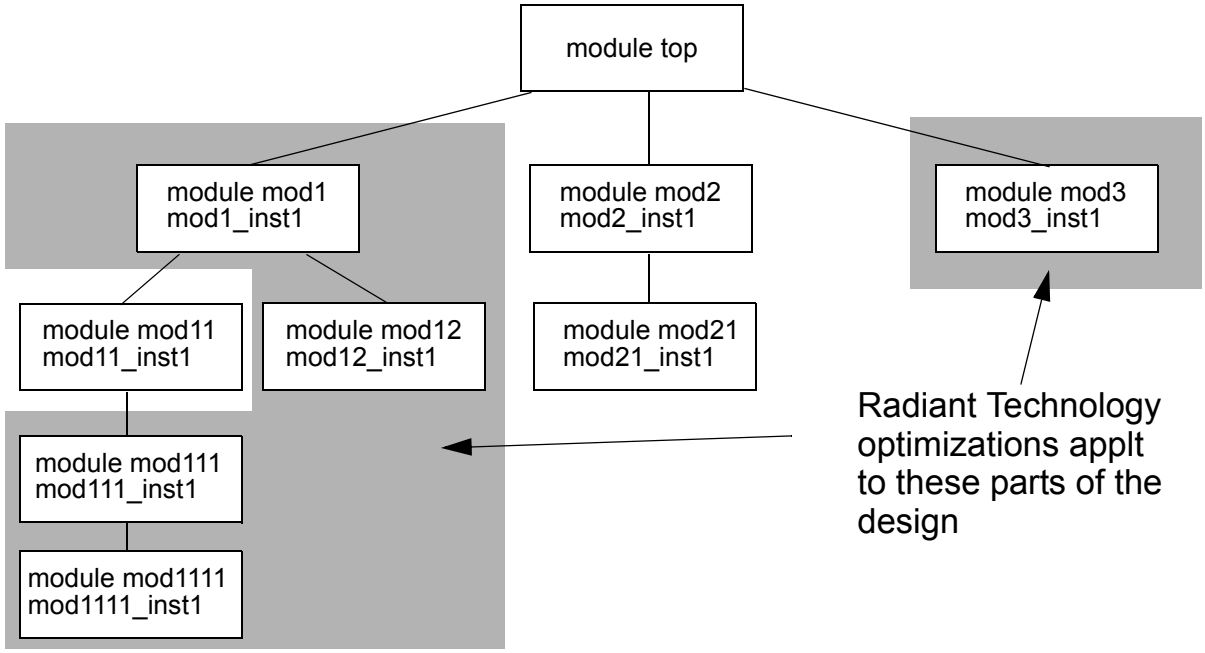
third tree statement example

You can specify a negative value for the depth value. When you do you specify ascending the hierarchy from the leaf level. For example:

```
tree (-2) {mod1, mod3} {Opt};
```

This statement specifies looking down the module hierarchy under the instances of modules mod1 and mod3 to the leaf level and

counting up from there. (Leaf level module instances contain no module instantiation statements.)



In this example the instances of mod1111, mod12, and mod3 are at a depth of -1 and the instances of mod111 and mod1 are at a depth of -2. The attributes do not apply to the instance of mod11 because it is at a depth of -3.

fourth tree statement example

You can disable Radiant optimizations at the leaf level under specified modules. for example:

```
tree(-1) {mod1, mod2} {noOpt};
```

This example disables optimizations at the leaf level, the instances of modules mod1111, mod12, and mod21, under the instances of modules mod1 and mod2.

7

Using the PLI

IN VCS the PLI is a programming Language Interface (PLI) between C/C++ functions and VCS. You can use the PLI as an interface between an application that you want to link with VCS, to execute concurrently with VCS, and in which the functions in the application can access simulation values in VCS to read and write delay and simulation values and VCS can call these functions during simulation.

Also in VCS, many debugging features, such as the CLI and VirSim, are implemented using the PLI, so when you are using these debugging features, you are using the PLI even though you have not written a PLI application.

VCS has implemented the TF and ACC routines for the PLI. These routines are documented in IEEE Std 1364-1995 Sections 18 through 21. Status of VPI procedural interface routine implementation is outlined in “Using VPI Routines” on page 7-29. These routines are documented in IEEE Std 1364-1995 Sections 22 through 23.

The ACC routines, which access the values in a design, changing, for example, the delay values of module paths or the simulation values of individual signals, are more powerful that the TF routines which operate only on data passed to them, but the ACC routines also have a greater performance cost. The ability of ACC routines to traverse the entire design and make extensive value changes requires VCS to omit powerful performance optimizations so that it is possible for the ACC routines to make these changes.

This performance cost of the ACC routines is a major consideration in VCS performance.

There are ways to limit this performance cost and doing so is an important step in developing a PLI application that uses ACC routines.

Many debugging features, including many in the CLI and VirSim, use ACC routines. If you are in the debugging phase of your design and are using the debugging features in VCS and VirSim, limiting the performance cost of the ACC routines is also an important step.

If you are writing a PLI application:

1. You write the C/C++ functions of the application calling the TF and ACC routines that we have implemented to access data inside VCS.

2. You associate user-defined system tasks and system functions with the C/C++ functions in your application. VCS will call these functions when it compiles or executes these system tasks or system functions in your Verilog source code. In VCS you associate your user-defined system tasks and system functions with the C/C++ functions in your application using a PLI table file. In this file you can also limit the scope and operations of the ACC routines for faster performance.

3. You enter your user-defined system tasks and functions in your Verilog source code.

4. You 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, also specify the compiler and linker that generated them with the `-cc` and `-ld` options.)

If you are using the debugging features:

1. You write a PLI table file, limiting the scope and operations of the ACC routines used by the debugging features.

2. You compile and simulate your design, specifying the table file

These procedures are not mutually exclusive, it is, for example, quite possible that you have a PLI application that you wrote and use during the debugging phase of your design. If so you can write a PLI table file that both:

1. Associate user-defined system tasks or system functions with the functions in you application and limits the scope and operations of the ACC routines called by your functions for faster performance

2. Limits scope and operations of the ACC routines called by the debugging features in VCS and VirSim

We have also implemented a number of ACC routines that are not part of the IEEE Std 1364-1995. These routines access:

- Reading and writing to memories
- Multi-Dimensional arrays”
- Probabilistic distribution”

- Returning a string pointer to a parameter value
- Extended VCD files
- Line callbacks
- Source protection
- Signal in a generate block

This chapter describes:

- The functions in a PLI application that VCS expects or requires—The types of functions VCS calls for a PLI application
- The Header Files for PLI Applications—The files containing the definitions for TF and ACC routines including the ACC routines that are exclusive to VCS
- The PLI table file—The purposes and syntaxes of the various specifications in a PLI table file
- PLI Object Compatibility with Verilog-XL—PLI information for users converting from Verilog-XL
- Globally Enabling ACC Capabilities—Compile-time options for specifying ACC capabilities
- Using Only The ACC Capabilities That You Need—Compile-time and runtime options that enable subsequent simulations of your design to use no more than the ACC capabilities used in a previous simulation
- Using VPI Routines—Access routines that are sometimes called PLI 2.0 routines
- Writing Your Own main() Routine—A technique for PLI applications that must do some processing before simulation starts

The Functions in a PLI Application

When you write a PLI application you typically write a number of functions. The following are PLI functions that VCS expects with a user-defined system task or system function:

- The function that VCS calls when it executes the user-defined system task. (The IEEE Std 1364-1995 calls this function the calltf application, see page 230.) Other functions are not necessary but there must be this call function.
  It's not unusual for you to need more than one call function. You'll need a separate user-defined system task for each call function. If the function returns a value then you write a user-defined system function for it instead of a user-defined system task.
- The function that VCS calls when it is compiling the design to check to see if the user-defined system task has the correct syntax.(The IEEE Std 1364-1995 calls this function the checktf application, see page 230.) You can omit this check function.
- The function that VCS calls for miscellaneous reasons such as the execution of `$stop` or `$finish` but it can be other reasons such a value change. When VCS calls this function it passes to the function a reason argument to tell the function why VCS is calling it. (The IEEE Std 1364-1995 calls this function the misctf application, see page 230.) You can omit this misc function.

PLI applications can have, of course, several more functions that are called by other functions, these functions listed above are the ones you tell VCS about in the PLI table file.

You don’t specify a function to determine the return value size of a user-defined system function (The IEEE Std 1364-1995 calls these functions sizetf applications, see page 230) instead you specify the size directly in the PLI table file.

Header Files for PLI Applications

For PLI applications you will need to include one or more of the following header files:

acc_user.h
: For PLI Applications whose functions call IEEE Standard ACC routines as documented in IEEE Std 1364-1995

vcsuser.h
: For PLI applications whose functions call IEEE Standard TF routines as documented in IEEE Std 1364-1995

vcs_acc_user.h
: For PLI applications whose functions call the special ACC routines implemented exclusively for VCS.

You will find these header files at $VCS_HOME/*platform*/lib, where *platform* is the platform you are using, such as sun_sparc_solaris_5.7.

The header file for the VPI routines is $VCS_HOME/include/vpi_user.h, see “Using VPI Routines” on page 7-29.

The PLI Table File

The PLI table file (commonly called the pli.tab file) has two purposes:

- To associate user-defined system tasks and system functions with functions in a PLI application, so VCS calls these functions when it compiles or executes the system task or function, and to limit the scope and operations of the ACC routines called by these functions.
- To limit the scope and operation of the ACC routines called by the debugging features. If this is all that you need to do in the PLI table file, see “Specifying ACC Capabilities” on page 7-12 and “Specifying ACC Capabilities for VCS Debugging Features” on page 7-17.

For a PLI application, after you write the functions for your PLI application, you write a PLI table file. In this file is a line for each user-defined system task or system function your application needs. In each line you list the following information:

- The name of the user-defined system task or system function. This name, of course, begins with the dollar sign `$`.
- The PLI specifications for the user-defined system task or system function. These specifications must include the call function. For user-defined system functions, they must also include the size of the return value.

  These specifications can also include the check and misc functions. There are a number of other PLI specifications, like a specification for the ability to enter the user-defined system task on the command line, that you can also enter on a line in the PLI table file.

- The ACC capabilities that you want to enable for the functions, particularly the call function, for the user-defined system task or system function.
  When you specify ACC capabilities you specify the types of ACC operations that the ACC routines will perform and where in the design they can perform these operations.
  Entering ACC capabilities to limit ACC routine scope and operations is optional but recommended to enhance performance.

The syntax for a line in a PLI table file is as follows:

```
$name PLI_specifications [ACC_capabilities]
```

Where:

$name
: Is the name of your user-defined system task or system function

PLI_specifications
: Is one or more specifications such as the name of the C function VCS calls when it executes the user defined system task or system function.

ACC_capabilities
: Specifications for ACC capabilities to be added, removed, or changed from various parts of the design hierarchy.

The PLI Specifications

The valid PLI specifications are as follows:

`call=`*`function`*
: Specifies the name of call function. This specification is required.

`check=`*`function`*
: Specifies the name of check function.

`misc=`*`function`*
: Specifies the name of misc function.

`data=`*`integer`*
: Specifies the data value passed as first argument to call, check, and misc routines. The default is 0. You use this argument if you want more than one user-defined system task or system function to use the same call, check, or misc function. You specify a different integer for each user-defined system task or system function in the PLI table file that uses the same call, check, or misc function. See The IEEE Std 1364-1995 Section 17.6.1 for more information.

`size=`*`number`*
: Specifies the size of returned value in bits. This specification is required for user-defined system functions. You can omit this specification or specify 0 in the PLI specifications for a user-defined system function.
  For user-defined system functions, specify a decimal value for the number or bits, for example:
  `size=64`
  If the user-defined system function returns a real value, specify `r`, for example:
  `size=r`

`args=`*`number`*
: Specifies the number of arguments to the user-defined system task or system function.

`minargs=`*`number`*
: Specifies the minimum number or arguments.

`maxargs=`*`number`*
: Specifies the maximum number or arguments.

`nocelldefinepli`
: Disables the dumping of value change and simulation time data, from modules defined under the `` `celldefine `` compiler directive, into the VPD file created by the `$vcdpluson` system task.
  You make this change in the line for the `$xvcs` system task in the virsims.tab, virsims_dki.tab, and virsims_pp.tab files in the $VCS_HOME/virsims_support/vcdplus directory. It doesn't make sense for you to use this specification in a PLI table file for any other PLI application.
  This capability is intended for batch-simulation only and not for interactive debugging with VirSim.

`persistent`
: Enables you to enter the user-defined system task on the CLI command line (or the Command field of the VirSim Interactive window) without including any of the `+cli` compile time options.

The following are example lines in PLI table files that contain PLI specifications (the ACC capabilities parts are omitted):

```
$val_proc call=val_proc check=check_proc misc=misc_proc
```

In these PLI specifications, 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 it the user-defined system task has the correct syntax, and calls the misc_proc function in special circumstances like interrupts.

```
$value_passer size=0 args=2 call=value_paser persistent
```

In these PLI specifications there is an associated user-defined system task (because it has a return size of 0), the system task takes two arguments, when VCS executes the `$value_passer` system task it calls the function named value_passer, and you can enter the system task on the CLI command line without including the `+cli` compile-time option.

```
$set_true size=16 call=set_true
```

In these PLI specifications there is an associated user-defined system function that returns a 15 bit return value. VCS calls the function named set_true when it executes this system function.

Note:

Do not enter blank spaces inside a PLI specification. The following copy of the last example of PLI specifications does not work:

```
$set_true size = 16 call = set_true
```

Specifying ACC Capabilities

In a PLI table file you specify ACC capabilities for the following two reasons:

- To specify the ACC capabilities for the PLI functions associated with your user-defined system task or system function.
  To do this you specify them, on a line in a PLI table file, after the name of the user-defined system task or system function and its PLI specifications.
- To specify the ACC capabilities that the debugging features of VCS and VirSim can use. To do this you just specify ACC capabilities alone on a line in a PLI table file, without an associated user-defined system task or system function.

When you specify ACC capabilities, you specify both of the following:

- The ACC capabilities you want to enable or disable.
- The part or parts of the design that you want this enabling or disabling to occur.

In many ways specifying ACC capabilities for your PLI functions, and specifying them for VCS debugging features, is the same, but the capabilities that you enable, and the parts of the design to which you can apply them are different. Therefore we are presenting specifying ACC capabilities in separate sections. See “Specifying ACC Capabilities for PLI Functions” immediately following and “Specifying ACC Capabilities for VCS Debugging Features” on page 7-17.

Specifying ACC Capabilities for PLI Functions

The format for specifying ACC capabilities is as follows:

`acc=|+=|-=|:=`*`capabilities`*`:`*`module_names`*`[+]|%CELL|%TASK|*`

Where:

`acc`
: Is a keyword that begins a line for specifying ACC capabilities

`=|+=|-=|:=`
: Are operators for adding, removing, or changing ACC capabilities

`capabilities`
: Is a comma separated list of ACC capabilities

`module_names`
: Is a comma separated list of module identifiers (or names). Specifying modules enables, disables, or changes (depending on the operator) the ability of the PLI function to use the ACC capability in all instances of the specified module.

`+`
: Specifies adding, removing, or changing the ACC capabilities for not only the instances of the specified modules but also the instances hierarchically under the instances of the specified modules.

`%CELL`
: Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the ACC capability in all instances of module definitions compiled under the `` `celldefine `` compiler directive and all module definitions in Verilog library directories and library files (as specified with the `-y` and `-v` compile-time options.)

`%TASK`

Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the ACC capability in all instances of module definitions that contain the user-defined system task or system function associated with the PLI functions.

`*`

Enables, disables, or changes (depending on the operator) the ability of the PLI function to use the ACC capability throughout the entire design. Using wildcard characters could seriously impede the performance of VCS.

Note:

There are no blank spaces when specifying ACC capabilities.

The operators in this syntax are as follows:

`=` A shorthand for `+=`

`+=` Specifies adding the listed capabilities that follow to the parts of the design that follow, as specified by module name, `%CELL`,`%TASK`, or `*` wildcard character.

`-=` Specifies removing the listed capabilities that follow from the parts of the design that follow, as specified by module name, `%CELL`,`%TASK`, or `*` wildcard character.

`:=` Specifies changing the ACC capabilities of the parts of the design that follow, as specified by module name, `%CELL`,`%TASK`, or `*` wildcard character, to only those in the list of capabilities on this specification. A specification with this operator can change the capabilities specified in a previous specification.

The ACC capabilities that you can specify for the functions in your PLI specifications are as follows:

`r` or `read`

To read the values of nets and registers in your design.

`rw` or `read_write`
: To both read from and write to the values of registers or variables (but not nets) in your design.

`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`
: To force values on nets and registers.

`prx` or `pulserx_backannotation`
: To set pulse error and pulse rejection percentages for module path delays.

`s` or `static_info`
: The ability to access static information, such as instance or signal names and connectivity information. Signal values are not static information.

`tchk` or `timing_check_backannotation`
: To back annotate timing check delay values.

`gate` or `gate_backannotation`
: To back annotate delay values on gates.

`mp` or `module_path_backannotation`
: To back annotate module path delays.

`mip` or `module_input_port_backannotation`
: To back annotate delays on module input ports.

`mipb` or `module_input_port_bit_backannotation`
: To back annotate delays on individual bits of module input ports.

The following examples are the PLI specification examples from the previous section with ACC capabilities added to them (They wrap to more than one line, but when you edit your PLI table file, be sure there are no line breaks in these lines):

```
$val_proc call=val_proc check=check_proc misc=misc_proc
acc+= rw,tchk:top,bot acc-=tchk:top
```

This example adds the ACC capabilities for reading and writing to nets and registers, and to back annotate timing check delays, to these PLI functions, and enables them to do these things in all instances of modules top and bot, then removes the ACC capability for back annotating timing check delay values from these PLI functions in all instances of module top.

```
$value_passer size=0 args=2 call=value_passer persistent
acc+=rw:%TASK acc-=rw:%CELL
```

This example adds the ACC capability to read from and write to the values of nets and registers, to these PLI functions, and enables them to do these things in all instances of modules declared in module definitions that contain the $value_passer user-defined system task. The example then removes the ACC capability to read from and write to the values of nets and registers, from these PLI functions, in module definitions compiled under the `` `celldefine `` compiler directive and all module definitions in Verilog library directories and library files.

```
$set_true size=16 call=set_true acc+=rw:*
```

This example adds the ACC capability to read from and write to the values of nets and registers to the PLI functions and enables them to do this throughout the entire design.

Specifying ACC Capabilities for VCS Debugging Features

The format for specifying ACC capabilities for VCS debugging features is as follows:

`acc=|+=|-=|:=`*`capabilities`*`:`*`module_names`*`[+]|%CELL|*`

Where:

`acc`
: Is a keyword that begins a line for specifying ACC capabilities

`=|+=|-=|:=`
: Are operators for adding, removing, or changing ACC capabilities

`capabilities`
: Is a comma separated list of ACC capabilities

`module_names`
: Is a comma separated list of module identifiers. The specified ACC capabilities will be added, removed, or changed for all instances of these modules.

`+`
: Specifies adding, removing, or changing the ACC capabilities for not only the instances of the specified modules but also the instances hierarchically under the instances of the specified modules.

`%CELL`
: Specifies all modules compiled under the `` `celldefine `` compiler directive and all modules in Verilog library directories and library files (as specified with the `-y` and `-v` compile-time options.)

`*`
: Specifies all modules in the design. Using a wildcard character is no more efficient than using the `+cli` compile-time option.

The ACC capabilities and the interactive commands they enable are as follows:

ACC Capability	What it enables your PLI functions to do
`r` or `read`	For specifying “reads” in your design, it enables commands for doing the following, (the actual CLI commands described are in parentheses): • Creating an alias for another CLI command (`alias`) • Displaying CLI help (`?` and `help`) • Specifying the radix of displayed simulation values (`oformat`) • Displaying simulation values (`print`) • Descending and ascending the module hierarchy (`scope`) • Depositing values on registers (`set`) • Displaying the set breakpoints on signals (`show break`) • Displaying the port names of the current location, the current module instance or scope, in the module hierarchy (`show ports`) • Displaying the names of instances in the current module instance or scope (`show scopes`) • Displaying the nets and registers in the current scope (`show variables`) • Moving up the module hierarchy (`upscope`) • Deleting an alias for another CLI command (`unalias`) • Ending the simulation (`finish`)
`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.

ACC Capability	What it enables your PLI functions to do
`cbk` or `callback`	Commands for doing the following, (the actual CLI commands described are in parentheses): • Setting a repeating breakpoint, in other words always halting simulation, when a specified signal changes value (`always` or `break`) • Setting a one shot breakpoint, in other words halting simulation the next time the signal changes value but not the subsequent times it changes value (`once` or `tbreak`) • Remove a breakpoint from a signal (`delete`) • Show the line number or number in the source code of the statement or statements that causes the current value of a net (`show drivers`) A longer way to specify this capability is with the `callback` keyword.
`frc` or `force`	Commands for doing the following, (the actual CLI commands described are in parentheses): • Forcing a net or a register to a specified value so that this value cannot be changed by subsequent simulation events in the design (`force`) • Releasing a net or register from its forced value (`release`) A longer way to specify this capability is with the `force` keyword.

The following specification enables many interactive commands including those for displaying the values of signals in specified modules and depositing values to the signals that are registers:

```
acc+=r:top,mid,bot
```

Notice that there are no blank spaces in this specification. Blank spaces cause a syntax error.

The following specifications enable most interactive commands for most of the modules in a design, then change the ACC capabilities preventing breakpoint and force commands in instances of modules in Verilog libraries and modules designated as cells with the `` `celldefine `` compiler directive.

```
acc+=rw,cbk,frc:top+ acc:=rw:%CELL
```

Here the first specification enables the interactive commands that are enabled by the `rw`, `cbk`, and `frc` capabilities for module top, which in this example is the top-level module of the design, and all module instances under it. The second specification limits the interactive commands for the specified modules to only those enabled by the `rw` (same as `r`) capability.

Specifying the PLI Table File

You specify the PLI table file with the `-P` compile-time option, followed by the name of the PLI table file (by convention, the PLI table file has a .tab extension), for example:

```
-P pli.tab
```

When you enter this option on the vcs command line, if you have a PLI application, you can then also enter C source files, or compiled .o object files or .a libraries on the vcs command line, to specify the PLI application that you want to link with VCS. For example:

```
vcs -P pli.tab pli.c my_design.v
```

One advantage to entering .o object files and .a libraries is that you do not recompile the PLI application every time you compile your design.

PLI Object Compatibility with Verilog-XL

VCS is object-compatible with Verilog-XL PLI applications that call TF and ACC routines. This means that you need not recompile the object files and libraries for an old PLI application when you switch to VCS.

You can convert your veriuser.c file to a VCS PLI table file using the perl script named veriuser_to_pli_tab. This script is in the $VCS_HOME/bin directory.

To create, for example, a PLI table file named pli.tab, enter the following at the system prompt:

```
veriuser_to_pli_tab veriuser.c > pli.tab
```

If the output contains `??` it means that it's a function that returns data. If you don't know the bit width of the returned data try replacing `??` with `32`.

Globally Enabling ACC Capabilities

You can enter the `+acc+`*`level_number`* compile-time option to globally enable ACC capabilities throughout your design.

Note:

Doing so significantly impedes the simulation performance of your design. We recommend using a PLI table file instead to enable ACC capabilities for only the parts of your design that you need.

The *level_number* in this option specifies more and more ACC capabilities as follows:

`+acc+1` or `+acc`
: Enables all capabilities except value change callbacks and delay annotation.

`+acc+2`
: Above, plus value change callbacks.

`+acc+3`
: Above, plus module path delay annotation.

`+acc+4`
: Above, plus gate delay annotation.

Enabling ACC Write Capabilities Using The Configuration File

You specify the configuration file with the `+optconfigfile` compile-time option, for example:

`+optconfigfile+`*`filename`*

The VCS configuration file enables you to enter statements that specify:

- Using the optimizations of Radiant technology on part of a design
- Enabling PLI ACC write capabilities for all memories in the design, disabling them for the entire design, or enabling them for part or parts of the design hierarchy
- Four state simulation for part of a design

The entries in the configuration file override the ACC write enabling entries in the PLI table file.

The syntax of each type of statement in the configuration file to enable ACC write capabilities is as follows:

```
set writeOnMem;
```

or

```
set noAccWrite;
```

or

```
module {list_of_module_identifiers} {accWrite};
```

or

```
instance {list_of_module_instance_hierarchical_names}
{accWrite};
```

or

```
tree [(depth)] {list_of_module_identifiers}
{accWrite};
```

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`
: Attribute specifying the enabling of ACC write capabilities

`module`
: Keyword that specifies 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 that specifies that the accWrite attribute in this statement applies to all instances in the list.

`list_of_module_instance_hierarchical_names`
: Comma separated list on module instance hierarchical names.

`tree`
: Keyword that specifies that the `accWrite` attribute in this statement applies to all instances of the modules in the list, specified by module identifier, and also applies to all module instances hierarchically under these module instances.

`depth`
: An integer that specifies how far down the module hierarchy, from the specified modules, you want to apply the `accWrite` attribute. You can specify a negative value. A negative value specifies descending to the leaf level and counting up levels of the hierarchy to apply these attributes. This specification is optional. Enclose this specification in parentheses: `()`

Using Only The ACC Capabilities That You Need

There are compile-time and runtime options that enable the debugging features Simulation and PLI applications to use only the ACC capabilities they need and no more than they need. The procedure to use these options is as follows:

1. Use the `+vcs+learn+pli` runtime option to tell VCS to keep track of, or learn, the ACC capabilities that are used by different modules in your design. VCS uses this information to create a secondary PLI table file, named pli_learn.tab, that you can use to recompile your design so that subsequent simulations only use the ACC capabilities that they need.
2. You tell VCS to apply what it has learned in the next compilation of your design, and specify the secondary PLI table file, with the `+applylearn+`*`filename`* compile-time option (you can omit *+filename* from the `+applylearn` compile-time option and VCS uses the pli_learn.tab secondary PLI table file).
3. Simulate again with a simv executable in which only the ACC capabilities you need are enabled.

Learning What ACC Capabilities Are Used

You include the `+vcs+learn+pli` runtime option to tell VCS to learn and write into a secondary PLI table file named pli_learn.tab, the ACC capabilities that were used by the modules in your design.

We call this file a secondary PLI table file because it does not replace the first PLI table file that you used (if you used one). This file does however modify what ever ACC capabilities are specified in a first PLI table file, or other means of specifying ACC capabilities, so that you enable only the ACC capabilities you need in subsequent simulations.

You should look at the contents of the pli_learn.tab file that VCS writes to see what ACC capabilities were actually used during simulation. The following is an example of this file:

```
////////////////// SYNOPSYS INC ///////////////
//                PLI LEARN FILE
// AUTOMATICALLY GENERATED BY VCS(TM) LEARN MODE
//////////////////////////////////////////////
acc=r:testfixture
     //SIGNAL STIM_SRLS:r
acc=rw:SDFFR
     //SIGNAL S1:rw
```

in this file:

```
acc=r:testfixture
     //SIGNAL STIM_SRLS:r
```

Specifies that during simulation the ACC read capability was needed for signals in the module named testfixture. The comment lets you know that the only signal for which this capability was needed was the signal named STIM_SRLS. This line is in the form of a comment because the syntax of the PLI table file does not permit specifying ACC capabilities on a signal by signal basis.

```
acc=rw:SDFFR
     //SIGNAL S1:rw
```

Specifies that during simulation the ACC read and write capabilities were needed for signals in the module named SDFFR, specifically for the signal named S1.

Signs Of A Potentially Significant Performance Gain

You might see one of following comments in the pli_learn.tab file:

`//!VCS_LEARNED: NO_ACCESS_PERFORMED`	This indicates that none of the enabled ACC capabilities were used during the simulation.
`//!VCS_LEARNED: NO_DYNAMIC_ACCESS_PERFORMED`	This indicates that only static information was accesses through ACC capabilities and not value change information during simulation.

These comments mean there is a potentially significant performance gain when you apply the acc capabilities in the pli_learn.tab file.

Compiling to Enable Only The ACC Capabilities You Need

After you have run the simulation to learn what ACC capabilities were actually used by your design, you then recompile the design with the information you have learned so the resulting simv executable only uses the ACC capabilities that you need.

When you recompile your design include the `+applylearn` compile-time option.

If for some reason you renamed the pli_learn.tab file that VCS writes when you include the `+vcs+learn+pli` runtime option, specify the new filename in the compile-time option by appending it to the option with the following syntax:

`+applylearn+`*`filename`*

When you recompile your design with the `+applylearn` compile time option, it is important that you also re-enter all the compile-time options that you used for the previous compilation, so for example, If in a previous compilation you specified a PLI table file with the `-P` compile-time option, specify this PLI table file again, using the `-P` option, along with the `+applylearn` option.

Note:

If you change your design after VCS writes the pli_lean.tab file, and you want to make sure that you are using only the ACC capabilities you need, you will need to have VCS to write another one, by including the `+vcs+learn+pli` runtime option and then compile your design again with the `+applylearn` option.

Limitations

VCS is not able to keep track of all ACC capabilities. The capabilities it can keep track of, and specify in the pli_learned.tab file, are as follows:

`r` - read

`rw` - read and write

`cbk` - callbacks

`cbka` - callback all including unnamed objects

`frc` - forcing values on signals

The `+applylearn` compile-time option does not work if you also use any of the following compile-time options:

`+cli` - because this option does not specify what information will be accessed through CLI commands.

`+multisource_int_delays` or `+transport_int_delays` because interconnect delays need global ACC capabilities.

If you enter the `+applylearn` compile-time option more than once on the vcs command line, VCS ignore all but the first one.

Using VPI Routines

To enable VPI capabilities in VCS, you must include the `+vpi` compile-time option. For example:

```
vcs +vpi test.v -P test.tab test.c
```

The header file for the VPI routines is $VCS_HOME/include/vpi_user.h.

You can register your user defined system tasks/function-related callbacks using the vpi_register_systf VPI routine, see “Support for the vpi_register_systf Routine” on page 7-31.

You can also use a PLI .tab file to associate your user defined system tasks with your VPI routines, see “PLI Table File For VPI Routines” on page 7-32.

VCS has not implemented everything specified for VPI routines in the IEEE Std 1364-2001 because some routines would be rarely used and some of the data access operations of other routines would be rarely used. The unimplemented routines are as follows:

```
vpi_get_data  vpi_put_data  vpi_sim_control
```

Object data model diagrams in the IEEE Std 1364-2001 standard specify that some VPI routines should be able to access data that is rarely needed. These routines and the data they can't access are as follows:

`vpi_get_value`

Cannot retrieve the value of var select objects (diagram 26.6.8 Variables) and func call objects (diagram 26.6.18 Task, function declaration).

Cannot retrieve the value of VPI operators (expressions) unless they are arguments to system tasks or system functions.

Cannot retrieve the value of UDP table entries (vpiVectorVal not implemented).

`vpi_put_value`

Cannot set the value of var select objects (diagram 26.6.8 Variables) and primitive objects (diagram 26.6.13 Primitive, prim term).

`vpi_get_delays`

Cannot retrieve the values of continuous assign objects (diagram 26.6.24 Continuous assignment) or procedurally assigned objects.

`vpi_put_delays`

Cannot put values on continuous assign objects (diagram 26.6.24 Continuous assignment) or procedurally assigned objects.

`vpi_register_cb`

Cannot register the following types of callbacks that are defined for this routine:

`cbEndOfSimulation`	`cbError`	`cbPliError`
`cbTchkViolation`	`cbSignal`	`cbForce`
`cbRelease`	`cbAssign`	`cbDeassign`

Also for the `cbValueChange` callback is not supported for the following objects:

- a memory or a memory word (index or element)
- VarArray or VarSelect

Support for the vpi_register_systf Routine

VCS supports the vpi_register_systf VPI access routine. To use it you need to make an entry in the vpi_user.c file. You can copy this file from $VCS_HOME/etc/vpi. The following is an example:

```
/*=====================================================
          Copyright (c) 2003 Synopsys Inc
=====================================================*/

/* Fill your start up routines in this array, Last entry
should be
zero, use -use_vpiobj to pick up this file */
extern void register_me();
void (*vlog_startup_routines[])() = {
     register_me,
     0 /* Last Entry */
};
```

entry here

In this example:

1. The routine named register_me is externally declared
2. It is also included in the array named vlog_startup_routines
3. The last entry in the array is zero

You specify this file with the `-use_vpiobj` compile-time option, for example:

```
vcs any.c any.v -use_vpiobj vpi_user.c +cli+3 +vpi
```

PLI Table File For VPI Routines

The PLI table file for VPI routines works the same way, and with the same syntax as a PLI table file for user-defined system tasks that execute C functions that call PLI ACC routines. The following is an example of such a PLI table file:

```
$set_mipd_delays call=PLIbook_SetMipd_calltf
check=PLIbook_SetMipd_compiletf
acc=mip,mp,gate,tchk,rw:test+
```

Notice that this entry includes `acc=` even though the C functions in the PLI specification call VPI routines instead of ACC routines. The syntax has not changed, you use the same syntax for enabling ACC and VPI routines.

This PLI table file is for a example file named set_mipd_delays_vpi.c that is available with *The Verilog PLI Handbook* by Stuart Sutherland, Kluwer Academic Publishers, Boston, Dordrect, and London.

Integrating a VPI Application With VCS

If you create one or more shared libraries for a VPI application, the application should not contain the array named vlog_startup_routines.

Instead you enter the `-load` compile-time option to specify the registration routine. The syntax is as follows:

```
-load shared_library:registration_routine
```

You don’t have to specify the pathname of the shared library if that path is part of your LD_LIBRARY_PATH environment variable.

The following are some examples of using this option:

```
-load lib1:my_register
```

The my_register() routine is in lib1.so. The location of lib1.so is in the LD_LIBRARY_PATH environment variable.

```
-load lib1:my_register,new_register
```

The registration routines my_register() and new_register() are in lib1.so.The location of lib1.so is in the LD_LIBRARY_PATH environment variable.

```
-load lib1:my_register -load lib2:new_register
```

The registration routine my_register() is in lib1.so and the second registration routine new_register() is in lib2.so. The path to both of these libraries are in the LD_LIBRARY_PATH environment variable. You can enter more than one `-load` option to specify multiple shared libraries and their registration routines.

```
-load lib1.so:my_register
```

The registration routine my_register() is in lib1.so. The location of lib1.so is in the LD_LIBRARY_PATH environment variable.

```
-load /usr/lib/mylib.so:my_register
```

The registration routine my_register() is in lib1.so, which is in /usr/lib/mylib.so, and not in the LD_LIBRARY_PATH environment variable.

Writing Your Own main() Routine

You write your own main() routine if you wrote your PLI application in C++ code or if your standard C code application does some processing before starting the simv executable.

When you do you must include the `-e` compile-time option on the vcs command line. The syntax is as follows:

```
-e new_name_for_main
```

For example:

```
vcs -P my_main.tab my_main.cc -e SimvMain source.v
```

The contents of my_main.cc is as follows:

```
#if defined(__cplusplus)
extern "C" {
#endif

extern int SimvMain(int argc, char *argv[]);
extern void vcs_atexit (void(*pfun)(int code));

#if defined(__cplusplus)
}
#endif

static void do_cleanup(int code)
{
```

```
    /* do simv post-processing work */
}

  int main(int argc, char *argv[])
   {
       /* Startup code (if any) goes here. */

       vcs_atexit(do_cleanup); /* Register callback */

       SimvMain(argc, argv);   /* Run simv */

       return 0;               /* Never gets here */
 }
```

Note that SimvMain does not return, it calls exit() directly. If you need to do any post-processing, you can register at-exit functions using vcs_atexit().

The function you pass to vcs_atexit() will be called with the exit status. If you make multiple calls to vcs_atexit(), your functions will be called in the reverse order of registration.

Note:

You cannot use this feature when using the VCS/SystemC cosimulation interface.

Reading and Writing to Memories

VCS comes with the a number of access routines, in addition to the acc and tf routines defined in the IEEE Std 1364-1995, for reading and writing to a memory.

These access routines are as follows:

`acc_setmem_int`
: For writing an integer value to specific bits in a Verilog memory word.

`acc_getmem_int`
: For reading an integer value for specific bits in a Verilog memory word.

`acc_clearmem_int`
: For clearing a memory. Writing zeros to all bits.

`acc_setmem_hexstr`
: For writing a hexadecimal string value to specific bits in a Verilog memory word.

`acc_getmem_hexstr`
: For reading a hexadecimal string value to specific bits in a Verilog memory word.

`acc_setmem_bitstr`
: For writing a string of binary bits (including X and z) to a Verilog memory word

`acc_getmem_bitstr`
: For reading a bit string for specific bits in a Verilog memory word.

`acc_handle_mem_by_fullname`
: Returns the handle used by `acc_readmem`.

`acc_readmem`
: Reads a data file and writes the contents to a memory.

`acc_getmem_range`
: Returns the upper and lower limits of a memory.

`acc_getmem_size`
: Returns the number of elements (or words or addresses) in a memory.

`acc_getmem_word_int`
: Returns the integer of a memory element.

`acc_getmem_word_range`
: Returns the least significant bit of a memory element and the length of the element.

acc_setmem_int

You use the `acc_setmem_int` access routine to write an integer value to specific bit in an Verilog memory word.

<table>
<tr><th colspan="4">acc_setmem_int</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Writes an integer value to specific bits in a memory word.</td></tr>
<tr><td>Syntax:</td><td colspan="3"><code>acc_setmem_int (memhand, value, row, start, length)</code></td></tr>
<tr><td rowspan="2">Returns:</td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>void</td><td colspan="2"></td></tr>
<tr><td rowspan="6">Arguments:</td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>handle</td><td>memhand</td><td>Handle to memory</td></tr>
<tr><td>int</td><td>value</td><td>The integer value written in binary format to the bits in the word</td></tr>
<tr><td>int</td><td>row</td><td>The memory array index</td></tr>
<tr><td>int</td><td>start</td><td>Bit number of the left-most bit in the memory word where this routine starts writing the value.</td></tr>
<tr><td>int</td><td>length</td><td>Specifies the total number of bits this routine writes to starting with the start bit.</td></tr>
<tr><td>Related routines:</td><td colspan="3">acc_getmem_int
acc_setmem_hexstr
acc_getmem_hexstr
acc_setmem_bitstr
acc_getmem_bitstr
acc_clearmem_int
acc_handle_mem_by_fullname
acc_readmem
acc_getmem_range
acc_getmem_size
acc_getmem_word_int
acc_getmem_word_range</td></tr>
</table>

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>
<tr><th colspan="4">acc_getmem_int</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Returns an integer value for specific bits in a memory word.</td></tr>
<tr><td>Syntax:</td><td colspan="3"><code>acc_getmem_int (memhand, row, start, length)</code></td></tr>
<tr><td rowspan="2">Returns:</td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>int</td><td colspan="2">Integer value of the bits in the memory word</td></tr>
<tr><td rowspan="5">Arguments:</td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>handle</td><td>memhand</td><td>Handle to memory</td></tr>
<tr><td>int</td><td>row</td><td>The memory array index</td></tr>
<tr><td>int</td><td>start</td><td>Bit number of the left-most bit in the memory word where this routine starts reading the value.</td></tr>
<tr><td>int</td><td>length</td><td>Specifies the total number of bits this routine reads starting with the start bit.</td></tr>
<tr><td>Related routines:</td><td colspan="3">acc_setmem_int
acc_setmem_hexstr
acc_getmem_hexstr
acc_setmem_bitstr
acc_getmem_bitstr
acc_clearmem_int
acc_handle_mem_by_fullname
acc_readmem
acc_getmem_range
acc_getmem_size
acc_getmem_word_int
acc_getmem_word_range</td></tr>
</table>

acc_clearmem_int

You use the `acc_clearmem_int` access routine to write zeros to all bits in a memory.

acc_clearmem_int			
Synopsis:	Clears a memory word.		
Syntax:	`acc_clearmem_int (memhand)`		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	memhand	Handle to memory
Related routines:	acc_setmem_int acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

Examples

The following code examples illustrate the use of Using `acc_getmem_int`, `acc_setmem_int`, and `acc_clearmem_int`:

- Example 7-1 shows C code that includes a number of functions to be associated with user defined system tasks.
- Example 7-2 shows the PLI table for associating these functions with these system tasks.

- Example 7-3 shows the Verilog source code containing these system tasks.

Example 7-1 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 and assigns them to local variables in the function. This function then calls `acc_setmem_int` to write to the specified memory in the specified word, start bit, for the specified length.

The function with the get_mem identifier similarly 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 7-2 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 7-3 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 `$clear_mem` system task begins by clears the memory.
2. The the `$set_mem` system task deposits values in specified words, and in specified bits in the memory named mymem.
3. In a for loop, the `$get_mem` system task reads values from the memory and displays those values.

acc_setmem_hexstr

You use the `acc_setmem_hexstr` access routine for writing the corresponding binary representation of a hexadecimal string to a Verilog memory.

acc_setmem_hexstr			
Synopsis:	Writes a hexadecimal string to a word in a Verilog memory.		
Syntax:	`acc_setmem_hexstr (memhand, hexStrValue, row, start)`		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	memhand	Handle to memory
	char *	hexStrValue	Hex string
	int	row	The memory array index
	int	start	Bit number of the left-most bit in the memory word where this routine starts writing the string.
Related routines:	acc_setmem_int acc_getmem_int acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_clearmem_int acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

This routine takes a value argument which is a hexadecimal string of any size and puts its corresponding binary representation into the memory word indexed by *`row`*, starting at the bit number *`start`*.

Examples

The following code examples illustrates the use of `acc_setmem_hexstr`:

- Example 7-4 shows the C source code for an application that calls `acc_setmem_hexstr`.
- Example 7-5 shows the contents of a data file read by the application.
- Example 7-6 shows the PLI table file that associates the user-defined system task in the Verilog code with the application.
- Example 7-7 shows the Verilog source that calls the application.

Example 7-4 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 7-4 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 hex 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 7-5 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 7-6 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 7-7 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 7-7 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>
<tr><th colspan="5">acc_getmem_hexstr</th></tr>
<tr><td>Synopsis:</td><td colspan="4">Returns a hexadecimal string from a Verilog memory.</td></tr>
<tr><td>Syntax:</td><td colspan="4">acc_getmem_hexstr (memhand,hexStrValue,row,start,len)</td></tr>
<tr><td rowspan="2">Returns:</td><td>Type</td><td colspan="3">Description</td></tr>
<tr><td>void</td><td colspan="3"></td></tr>
<tr><td rowspan="6">Arguments:</td><td>Type</td><td>Name</td><td colspan="2">Description</td></tr>
<tr><td>handle</td><td>memhand</td><td colspan="2">Handle to memory</td></tr>
<tr><td>char *</td><td>hexStrValue</td><td colspan="2">Pointer to a character array into which the string is written</td></tr>
<tr><td>int</td><td>row</td><td colspan="2">The memory array index</td></tr>
<tr><td>int</td><td>start</td><td colspan="2">Bit number of the left-most bit in the memory word where this routine starts reading the string.</td></tr>
<tr><td>int</td><td>length</td><td colspan="2">Specifies the total number of bits this routine reads starting with the start bit.</td></tr>
<tr><td>Related routines:</td><td colspan="4">acc_setmem_int
acc_getmem_int
acc_setmem_hexstr
acc_setmem_bitstr
acc_getmem_bitstr
acc_clearmem_int
acc_handle_mem_by_fullname
acc_readmem
acc_getmem_range
acc_getmem_size
acc_getmem_word_int
acc_getmem_word_range</td></tr>
</table>

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>
<tr><th colspan="5">acc_setmem_bitstr</th></tr>
<tr><td>Synopsis:</td><td colspan="4">Writes a string of binary bits string to a word in a Verilog memory.</td></tr>
<tr><td>Syntax:</td><td colspan="4"><code>acc_setmem_bitstr (memhand, bitStrValue, row, start)</code></td></tr>
<tr><td rowspan="2">Returns:</td><td>Type</td><td colspan="3">Description</td></tr>
<tr><td>void</td><td colspan="3"></td></tr>
<tr><td rowspan="5">Arguments:</td><td>Type</td><td>Name</td><td colspan="2">Description</td></tr>
<tr><td>handle</td><td>memhand</td><td colspan="2">Handle to memory</td></tr>
<tr><td>char *</td><td>bitStrValue</td><td colspan="2">Bit string</td></tr>
<tr><td>int</td><td>row</td><td colspan="2">The memory array index</td></tr>
<tr><td>int</td><td>start</td><td colspan="2">Bit number of the left-most bit in the memory word where this routine starts writing the string.</td></tr>
<tr><td>Related routines:</td><td colspan="4">acc_setmem_int
acc_getmem_int
acc_setmem_hexstr
acc_getmem_hexstr
acc_getmem_bitstr
acc_clearmem_int
acc_handle_mem_by_fullname
acc_readmem
acc_getmem_range
acc_getmem_size
acc_getmem_word_int
acc_getmem_word_range</td></tr>
</table>

This routine takes a value argument which is a bit string of any size, that can include the x and z values, and puts its corresponding binary representation into the memory word indexed by *`row`*, starting at the bit number *`start`*.

acc_getmem_bitstr

You use the `acc_getmem_bitstr` access routine to get a bit string, including x and z values, from a Verilog memory.

acc_getmem_bitstr			
Synopsis:	Returns a hexadecimal string from a Verilog memory.		
Syntax:	`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 left-most bit in the memory word where this routine starts reading the string.
	int	length	Specifies the total number of bits this routine reads starting with the start bit.
Related routines:	acc_setmem_int acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_clearmem_int acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

acc_handle_mem_by_fullname

Returns a handle to a memory that can only be used as a parameter to `acc_readmem`.

acc_handle_mem_by_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 acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

acc_readmem

You use the `acc_readmem` access routine to read a data file into a memory like the `$readmemb` or `$readmemh` system tasks.

The memhandle argument must be the handle returned by `acc_handle_mem_by_fullname`.

acc_readmem			
Synopsis:	Reads a data file into a memory		
Syntax:	`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 acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

Examples

The following code examples illustrate the use of `acc_readmem` and `acc_handle_mem_by_fullname`.

Example 7-8 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 7-9 The PLI Table File

```
$test_acc_readmem call=test_acc_readmem
```

Example 7-10 The Verilog Source Code

```
module top;
reg [7:0] CORE[7:0];
initial $acc_readmem(CORE, "CORE");
initial $test_acc_readmem("top.CORE", "test_mem_file");
endmodule
```

acc_getmem_range

You use the `acc_getmem_range` access routine to access the upper and lower limits of a memory..

acc_getmem_range			
Synopsis:	Returns the upper and lower limits of a memory		
Syntax:	`acc_getmem_range (memhandle, p_left_index,p_right_index)`		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	handle	memhandle	Handle to a memory
	int*	p_left_index	Pointer to int
	int	p_right_index	Pointer to int
Related routines:	acc_setmem_int acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_size acc_getmem_word_int acc_getmem_word_range		

acc_getmem_size

You use the `acc_getmem_size` access routine to access the number of elements in a memory..

acc_getmem_size			
Synopsis:	Returns the number of elements in a memory		
Syntax:	`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 acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_word_int acc_getmem_word_range		

acc_getmem_word_int

You use the `acc_getmem_word_int` access routine to access the integer value of an element (or word, address, or row)..

acc_getmem_word_int			
Synopsis:	Returns the integer value of an element		
Syntax:	`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 acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_range		

acc_getmem_word_range

You use the `acc_getmem_word_range` access routine to access the least significant bit of an element (or word, address, or row) and the length of the element..

acc_getmem_word_range			
Synopsis:	Returns the least significant bit of an element and the length of the element		
Syntax:	`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 acc_getmem_int acc_setmem_hexstr acc_getmem_hexstr acc_setmem_bitstr acc_getmem_bitstr acc_handle_mem_by_fullname acc_readmem acc_getmem_range acc_getmem_size acc_getmem_word_int		

Multi-Dimensional Arrays

We have a new type for multi-dimensional arrays defined in the vcs_acc_user.h file. Its name is accMda.

We also have the following new 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

`acc_get_mda_range`
: Returns all the ranges of the multi-dimensional array.

`acc_get_mda_word_range`
: Returns the range of an element in a multi-dimensional array.

`acc_getmda_bitstr`
: Reads a bit string, including X and Z values, from an element in a multi-dimensional array.

`acc_setmda_bitstr`
: Writes a bit string, including X and Z values, from an element in a multi-dimensional array.

tf_mdanodeinfo and tf_imdanodeinfo

You use these routines to access parameter node information from a multi-dimensional array.

tf_mdanodeinfo(), tf_imdanodeinfo()			
Synopsis:	Returns access parameter node information from a multi-dimensional array.		
Syntax:	`tf_mdanodeinfo(nparam, mdanodeinfo_p)` `tf_imdanodeinfo(nparam, mdanodeinfo_p, instance_p)`		
Returns:	Type	Description	
	mdanodeinfo_p *	The value of the second argument if successful; 0 if an error occurred	
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 acc_get_mda_word_range acc_getmda_bitstr 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.

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
	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 acc_get_mda_word_range acc_getmda_bitstr 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];
```

Calling the routine, such as the following:

```
handle hN = acc_handle_by_name(my_mem);
acc_get_mda_range(hN, &size, &msb, &lsb, &dim, &plndx,
&prndx);
```

Yields the following:

```
size = 8;
msb = 7, lsb = 0;
dim = 4;
plndx[] = {255, 255, 31}
prndx[] = {0, 0, 0}
```

acc_get_mda_word_range()

The `acc_get_mda_word_range` routine returns the range of an element in a multi-dimensional array.

acc_get_mda_word_range()			
Synopsis:	Gets the range of an element in a multi-dimensional array.		
Syntax:	`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 acc_get_mda_range acc_getmda_bitstr 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];
```

Calling the routine, such as the following:

```
handle hN = acc_handle_by_name(my_mem);
acc_get_mda_word_range(hN, &left, &right);
```

Yields:

```
left = 7;
right = 0;
```

acc_getmda_bitstr()

You use the `acc_getmda_bitstr` access routine to read a bit string, including x and z values, from a multi-dimensional array.

acc_getmda_bitstr()			
Synopsis:	Gets a bit string from a multi-dimensional array.		
Syntax:	`acc_getmda_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 acc_get_mda_range acc_get_mda_word_range 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];
```

Calling the 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);
```

Yields the string from `my_mem[5][5][10][3:5]`.

acc_setmda_bitstr()

You use the `acc_setmda_bitstr` access routine to write a bit string, including x and z values, into a multi-dimensional array.

acc_setmda_bitstr()			
Synopsis:	Sets a bit string in a multi-dimensional array.		
Syntax:	`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 acc_get_mda_range acc_get_mda_word_range 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];
```

Calling the 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);
```

Writes 111 in `my_mem[5][5][10][3:5]`.

Probabilistic Distribution

VCS comes with the following API routines that duplicate the behavior of the Verilog system functions for probabilistic distribution:

`vcs_random`
: Returns a random number and takes no argument.

`vcs_random_const_seed`
: Returns a random number and takes an integer argument.

`vcs_random_seed`
: Returns a random number and takes a pointer to integer argument.

`vcs_dist_uniform`
: Returns random numbers uniformly distributed between parameters.

`vcs_dist_normal`
: Returns random numbers with a specified mean and standard deviation.

`vcs_dist_exponential`
: Returns random numbers where the distribution function is exponential.

vcs_dist_poisson
: Returns random numbers with a specified mean.

These routines are declared in the vcs_acc_user.h file in $VCS_HOME/lib.

vcs_random

You use this routine to obtain a random number.

vcs_random()			
Synopsis:	Returns a random number.		
Syntax:	`vcs_random()`		
Returns:	Type	Description	
	int	Random number	
Arguments:	Type	Name	Description
	None		
Related routines:	vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_normal vcs_dist_exponential vcs_dist_poisson		

vcs_random_const_seed

You also use this routine to return a random number and you supply an integer constant argument as the seed for the random number.

vcs_randon_const_seed			
Synopsis:	Returns a random number.		
Syntax:	`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 also use this routine to return a random number and you supply a pointer argument

vcs_random_seed()			
Synopsis:	Returns a random number.		
Syntax:	`vcs_random_seed(seed)`		
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.

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.

vcs_dist_normal			
Synopsis:	Returns random numbers with a specified mean and standard deviation.		
Syntax:	`vcs_dist_normal(seed, mean, standard_deviation)`		
Returns:	Type	Description	
	int	Random number	
Arguments:	Type	Name	Description
	int *	seed	Pointer to a seed integer value.
	int	mean	An integer that is the average value of the possible returned random numbers.
	int	standard_ deviation	An integer that is the standard deviation from the mean for the normal distribution.
Related routines:	vcs_random vcs_random_const_seed vcs_random_seed vcs_dist_uniform vcs_dist_exponential vcs_dist_poisson		

vcs_dist_exponential

You use this routine to return a random number where the distribution function is exponential.

vcs_dist_exponential			
Synopsis:	Returns random numbers where the distribution function is exponential.		
Syntax:	`vcs_dist_exponential(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_poisson		

vcs_dist_poisson

You use this routine to return a random number with a specified mean.

vcs_dist_poisson			
Synopsis:	Returns random numbers with a specified mean.		
Syntax:	`vcs_dist_poisson(seed, mean)`		
Returns:	Type	Description	
	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		

Returning a String Pointer to a Parameter Value

The 1364 Verilog standard states that for access routine acc_fetch_paramval you can cast the return value to a character pointer using the C language cast operators `(char*)(int)` for example:

```
str_ptr=(char*)(int)acc_fetch_paramval(...);
```

In 64-bit simulation you should use `long` instead of `int`:

```
str_ptr=(char*)(long)acc_fetch_paramval(...);
```

For your convenience VCS provides the acc_fetch_paramval_str routine to directly return a string pointer.

acc_fetch_paramval_str

Returns the value of a string parameter directly as `char*`.

acc_fetch_paramval_str			
Synopsis:	Returns the value of a string parameter directly as `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		

Extended VCD Files

Routines to monitor the port activity of a device.

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

```
#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 filename also being used by a `$dumpports` or `$lsi_dumpports` system task.

Example

```
#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 filename must be unique for each call.

For proper dumpports operation, your task's misc function must call acc_lsi_dumpports_misc() with every call it gets. This ensures that the dumpports routines will see all of the simulation events needed for proper update and closure of the dumpports (extended VCD) files. For example, your misc 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 walks 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

```
#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

```
#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

```
#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 misc events.

This is a companion routine for acc_lsi_dumpports_call().

For proper dumpports operation, your task's misc function must call this routine for each call it gets.

Returns

No return value.

Example

```
#include "acc_user.h"
#include "vcs_acc_user.h"

void my_task_misc(int data, int reason)
{
   acc_lsi_dumpports_misc(data, reason);
    ...
}
```

---

acc_lsi_dumpports_off

Syntax

```
int acc_lsi_dumpports_off(char *filename)
```

Synopsis

Suspends VCDE file dumping.

This is a PLI interface to the `$dumpportsoff` system task.

If the filename is NULL dumping is suspended on all open files.

Returns

The number of files that matched.

Example

```
#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

```
#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_dumports_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

```
#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

```
#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.

Line Callbacks

Routines to monitor code execution.

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 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

```
#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

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"

/* VCS 4.x callback rtn */
void line_call_back(filename, lineno, userdata, tag)
char *filename;
int lineno;
char *userdata;
int tag;
{
   handle handle_mod = (handle)userdata;

   io_printf("Tag %2d, file %s, line %2d, module %s\n",
             tag, filename, lineno,
```

```
              acc_fetch_fullname(handle_mod));
}

/* unregister all line callbacks (recursive) */
void unregister_lcb (parent_mod)
handle parent_mod;
{
    handle child = NULL;

   if (! acc_object_of_type(parent_mod, accModule)) return;

   io_printf("Unregistering %s\n",
             acc_fetch_fullname (parent_mod));

   acc_mod_lcb_del (parent_mod, line_call_back, parent_mod);

   while ((child = acc_next_child (parent_mod, child))) {
        register_lcb (child);
    }
}
```

Caution

The module handle, consumer routine and user_data arguments must match those supplied to the acc_mod_lcb_add() routine for a successful delete.

For example, using the result of a call such as acc_fetch_name() as the user_data will fail, because that routine returns a different pointer each time it's called.

---

acc_mod_lcb_enabled

Syntax

```
int acc_mod_lcb_enabled()
```

Synopsis

Test to see if line callbacks is enabled.

By default the extra code required to support line callbacks is not added to a simulation executable. You can use this routine to determine if line callbacks have been enabled.

Returns

Non-zero if line callbacks are enabled, 0 if not enabled.

Example

```
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, filename pairs. Termination of the array is indicated by a NULL filename 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

```
#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.)

```
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

```
#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 a file name with line numbers, or if a module definition is in more than one file, an aray of vcs_srcfile_info_s struct entries specifying all the filenames 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

```
#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);
    }
}
```

Source Protection

The enclib.o file provides a set of access routines which can be used to create applications which directly produce encrypted verilog source code. There is no user accessible decode routine. Encrypted code can only be decoded by the VCS compiler.

Note that both verilog and SDF code can be protected. VCS knows how to automatically decrypt both.

The following code fragment outlines the basic use of 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 which contain include directives, you must switch off encryption (vcsSpEncodeOff), output the include directive and then switch encryption back on. This insures that when the parser begins reading the included file, it's in a known (non-decode) state.

If the file being included has proprietary data it can be encrypted separately. (Don't forget to change the `` `include `` compiler directive to point to the new encrypted name.)

vcsSpClose

Syntax

```
void vcsSpClose(vcsSpStateID esp)
```

Synopsis

This routine frees the memory allocated by vcsSpInitialize(). Call it when source encryption is finished on the specified stream.

Returns

No return value.

Example

```
vcsSpStateID esp = vcsSpInitialize();
...

vcsSpClose(esp);
```

vcsSpEncodeOff

Syntax

```
int vcsSpEncodeOff(vcsSpStateID esp)
```

Synopsis

This function performs two operations:

1. It inserts a trailer section into the output file which contains some closing information used by the decryption algorithm plus the `` `endprotected `` compiler directive.
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

```
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 into the output file which contains the `'protected` compiler directive plus 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

```
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() turn encoding on and off. This function can be used to get the current state of encoding.

Returns

1 for on, 0 for off.

Example

```
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

```
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

Allocate 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

vcsSpStateID pointer or NULL if memory could not be malloc'd.

Example

```
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

Decrypt 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

```
#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

Switch to regular encryption.

Tells VCS' encrypter to use the standard algorithm. This is the default mode.

Returns

No return value.

Example

```
#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' encrypter to use the OVA IP algorithm.

Returns

No return value.

Example

```
#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

Sets the DisplayMsg flag.

By default the VCS compiler will not display decrypted source code in it’s error or warning messages. Use this routine to enable this display.

Returns

No return value.

Example

```
vcsSpStateID esp = vcsSpInitialize();
vcsSpSetDisplayMsgFlag(esp, 0);
```

---

vcsSpSetFilePtr

Syntax

```
void vcsSpSetFilePtr(vcsSpStateID esp, FILE *fp)
```

Synopsis

Specifies the output file stream.

Before using the vcsSpWriteChar() or vcsSpWriteString() routines you must specify the output file stream.

Returns

No return value.

Example

```
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

Sets the OEM license code.

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 will verify that the end user has the specified license. If the license does not exist or has expired, VCS will abort.

Returns

No return value.

Example

```
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 above example code segment shows how to enable and disable the addition of the license code to the protected regions. Normally you would call this routine once, after calling vcsSpInitialize() and before the first call to vcsSpEncodeOn().

---

vcsSpSetPliProtectionFlag

Syntax

```
void vcsSpSetPliProtectionFlag(vcsSpStateID esp, int enable)
```

Synopsis

Sets the PLI protection flag.

This routine can be used 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 effects encrypted Verilog files. Encrypted SDF files, for example, are not effected.

Returns

No return value.

Example

```
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

Writes one character to the protected file.

If encoding is enabled (see vcsSpEncodeOn()) the specified character will be encrypted as it is written to the output file.

If encoding is disabled (see vcsSpEncodeOff()) the specified character 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

```
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

Writes a character string to the protected file.

If encoding is enabled (see vcsSpEncodeOn()) the specified string will be 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

```
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

acc_object_of_type

Syntax

```
bool acc_object_of_type(accGenerated, sigHandle)
```

Synopsis

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 this function is called to elaborate the design and to initialize various data structures, scheduling queues, etc. that are used by VCS. After this routine executes, all the initial time 0 events, such as the execution of initial blocks, are scheduled.

Call `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 word. The first [0] for simulation times from 0 to 2^{32} -1, the second 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.

8

Delays, Timing and SDF Files

This chapter describes:

- Transport and inertial delays and their various implementations. See “Transport and Inertial Delays” on page 8-2
- Pulse Control for specifying how VCS filters out narrow pulses. See “Pulse Control” on page 8-7.
- Delay modes, using module path delays or the delay specifications in primitive instantiation and continuous assignment statement along the path. See “Specifying The Delay Mode” on page 8-21.
- SDF files including using SDF configuration files. See “Using SDF Files” on page 8-23.
- INTERCONNECT delays from INTERCONNECT entries in SDF files. See “INTERCONNECT Delays” on page 8-50.

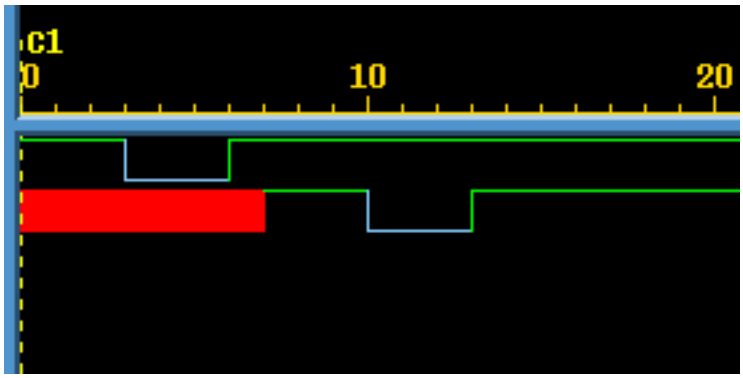
- MIN:TYP:MAX delay value triplets and how to specify which one to use, see “Min:Typ:Max Delays” on page 8-55.
- The Timopt timing optimizer, see “Using The Timopt Timing Optimizer” on page 8-59.

Transport and Inertial Delays

Delays can be categorized into transport and inertial delays.

Transport delays allow all pulses that are narrower than the delay, to propagate through. For example, Figure 8-1 shows the waveforms for an input and output port of a module that models a buffer with a module path delay of seven time units between these ports. The waveform on top is that of the input port and the waveform underneath is that of the output port. In this example you have enabled transport delays for module path delays and specified that a pulse three time units wide can propagate through (how this is done is explained in “Enabling Transport Delays” on page 8-7 and “Pulse Control” on page 8-7).

Figure 8-1 Transport Delay Waveforms

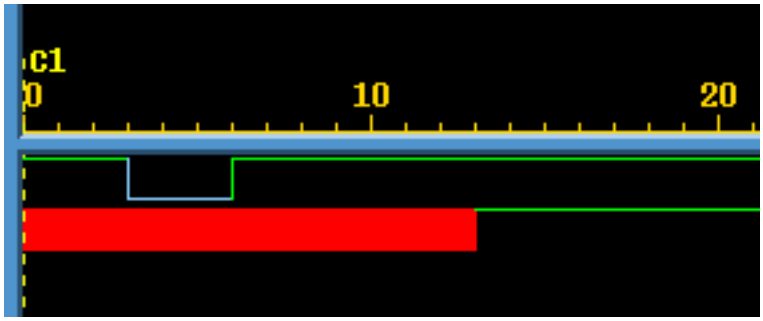


At time 0 a pulse three time units wide begins on the input port. This pulse is narrower than the module path delay of seven time units, but this pulse propagates through the module and appears on the output port after seven time units. Similarly another narrow pulse begins on the input port at time 3 and it also appears on the output port seven time units later.

You can apply transport delays on all module path delays and all SDF INTERCONNECT delays back annotated to a net from and SDF file.

Inertial delays, in contrast, filter out all pulses that are narrower than the delay. Figure 8-2 shows the waveforms for the same input and output ports but you have not enabled transport delays for module path delays.

Figure 8-2 Inertial Delay Waveforms



The pulse that begins at time 0 that is three time units wide does not propagate to the output port because it is narrower than the seven time unit module path delay, neither does the narrow pulse that begins at time 3. Note that the wide pulse that begins at time 6 does propagate to the output port.

Gates, switches, MIPDs, and continuous assignments only have inertial delays and inertial delays are the default type of delay for module path delays and INTERCONNECT delays back annotated from an SDF file to a net.

Different Inertial Delay Implementations

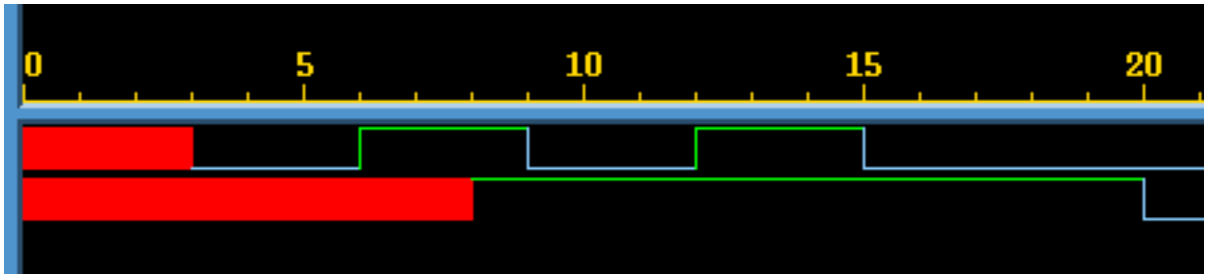
For compatibility with the earlier generation of Verilog simulators, inertial delays have two different implementations, one for primitives (gates, switches and UDPs), continuous assignments, and MIPDs (Module Input Port Delays) and the other for module path delays and INTERCONNECT delays back annotated from an SDF file to a net. (There is also a third implementation that is for module path and INTERCONNECT delays and pulse control, see “Pulse Control” on page 8-7.)

Inertial Delays for Primitives, Continuous Assignments, and MIPDs

Both implementations were devised to filter out narrow pulses but the one for primitives, continuous assignments, and MIPDs can produce unexpected results. For example Figure 8-3 shows the waveforms for nets connected to the input and output terminals of a `buf` gate with a delay of five time units.

In this implementation there can never be more than one scheduled event on an output terminal and, to filter out narrow pulses, the trailing edge of a pulse can alter the value change but not the transition time of the event scheduled by the leading edge of the pulse if the event has not yet occurred.

Figure 8-3 Gate Terminal Waveforms



In the example illustrated in Figure 8-3, the following occurs:

1. At time 3 the input terminal changes to 0. This is the leading edge of a three time unit wide pulse. This event schedules a value change to 0 on the output terminal at time 8 because there is a `#5` delay specification for the gate.
2. At time 6 the input terminal toggles to 1. This implementation keeps the scheduled transition on the output terminal at time 8 but alters the value change to a value of 1.
3. At time 8 the output terminal transitions to 1.
   This transition might be unexpected because all pulses on the input have been narrower than the delay but this is how this implementation works.
   There is now no event scheduled on the output and a new event can now be scheduled.
4. At time 9 the input terminal toggles to 0 and the implementation schedules a transition of the output to 0 at time 14.
5. At time 12 the input terminal toggles to 1 and the value change scheduled on the output at time 14 changes to a 1.

6. At time 14 the output is already 1 so there is no value change. The narrow pulse on the input between time 9 and 12 is filtered out. This implementation was devised for these narrow pulses. There is now no event scheduled for the output.

7. At time 15 the input toggles to 0 and this schedules the output to toggle to 0 at time 20.

Inertial Delays for Module Path Delays and INTERCONNECT delays

The implementation of inertial delays for module path delays and SDF INTERCONNECT delays is that if the event scheduled by the leading edge of a pulse is scheduled for a later simulation time or, in other words, has not yet occurred, the event scheduled by the trailing edge, at the end of the specified delay and at a new simulation time, replaces the event scheduled by the leading edge. All narrow pulses are filtered out.

Note:

SDF INTERCONNECT delays follow this implementation if you include the `+multisource_int_delays` compile-time option. If you don't include this option, VCS used a MIPD to model the SDF INTERCONNECT delay and the delay uses the inertial delay implementation for MIPDs. See "INTERCONNECT Delays" on page 8-50.

If you use pulse control, as specified by the `+pulse_e/`*`number`* and `+pulse_r/`*`number`* options (See "Pulse Control" on page 8-7); VCS does not use this implementation but uses a third implementation of inertial delays for module path and INTERCONNECT delays what works with the pulse control.

Enabling Transport Delays

Transport delays are never the default delay.

You can specify transport delays on module path delays with the `+transport_path_delays` compile-time option. For this option to work you must also include the `+pulse_e/`*`number`* and `+pulse_r/`*`number`* compile-time options. See “Pulse Control” on page 8-7.

You can specify transport delays on a net to which you back annotate SDF INTERCONNECT delays with the `+transport_int_delays` compile-time option. For this option to work you must also include the `+pulse_int_e/`*`number`* and `+pulse_int_r/`*`number`* compile-time options. See “Pulse Control” on page 8-7.

The `+transport_path_delays` and `+transport_int_delays` options enable transport delays. The `+pulse_e/`*`number`*, `+pulse_r/`*`number`*, `+pulse_int_e/`*`number`*, and `+pulse_int_r/`*`number`* options enable you to tell VCS to substitute an X value and display an error message for particularly narrow pulses filter out or reject even narrower pulses.

Pulse Control

So far we’ve seen that with pulses narrower that a module path or INTERCONNECT delay you have the option of filtering all of them out by using the default inertial delay or allowing all of them to propagate through, by specifying transport delays. However, VCS does not stick you with an all or nothing choice, that either filters out pulses that are just slightly narrower than the delay, or allows mere glitch-like pulses that are far narrower than the delay to propagate through. Instead VCS has pulse control.

With pulse control you can:

- Allow pulses that are slightly narrower than the delay to propagate through.
- Have VCS replace even narrower pulses with an X value pulse on the output and display a warning message.
- Have VCS then filter out and ignore pulses that are even narrower that the ones for which it propagates an X value pulse and displays an error message.

You specify pulse control with the `+pulse_e/`*`number`* and `+pulse_r/`*`number`* compile-time options for module path delays and the `+pulse_int_e/`*`number`* and `+pulse_int_r/`*`number`* compile-time options for INTERCONNECT delays.

The `+pulse_e/`*`number`* option's *`number`* argument specifies a percentage of the module path delay. VCS replaces pulses whose widths that are narrower than the specified percentage of the delay with an X value pulse on the output or inout port and displays a warning message.

Similarly, the `+pulse_int_e/`*`number`* option's *`number`* argument specifies a percentage of the INTERCONNECT delay. VCS replaces pulses whose widths are narrower than the specified percentage of the delay with an X value pulse on the inout or output port instance that is the load of the net to which you back annotated the INTERCONNECT delay and displays a warning message.

The `+pulse_r/`*`number`* option's *`number`* argument also specifies a percentage of the module path delay. VCS filters out the pulses whose widths are narrower than the specified percentage of the delay. With these pulses there is no warning message, VCS simply ignores these pulses.

Similarly, the +pulse_int_r/*number* option's *number* argument specifies a percentage of the INTERCONNECT delay. VCS filters out pulses whose widths are narrower than the specified percentage of the delay. There is also no warning message with these pulses.

You can use pulse control with transport delays (see "Pulse Control with Transport Delays" on page 8-9) or inertial delays (see "Pulse Control with Inertial Delays" on page 8-12).

When a pulse is narrow enough for VCS to display a warning message and propagate an X value pulse, you can have VCS 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, or place that starting edge on the output at the time when the rising or falling edge of the narrow pulse would have propagated to the output, which is the default behavior. See "Specifying Pulse on Event or Pulse on Detect Behavior" on page 8-16.

Also when a pulse is sufficiently narrow to display a warning message and propagate an X value pulse, you can have VCS propagate the X value pulse but disable the display of the warning message with the +no_pulse_msg runtime option.

Pulse Control with Transport Delays

You specify transport delays for module path delays with the +transport_path_delays, +pulse_e/*number*, and +pulse_r/*number* options. You must include all three of these options.

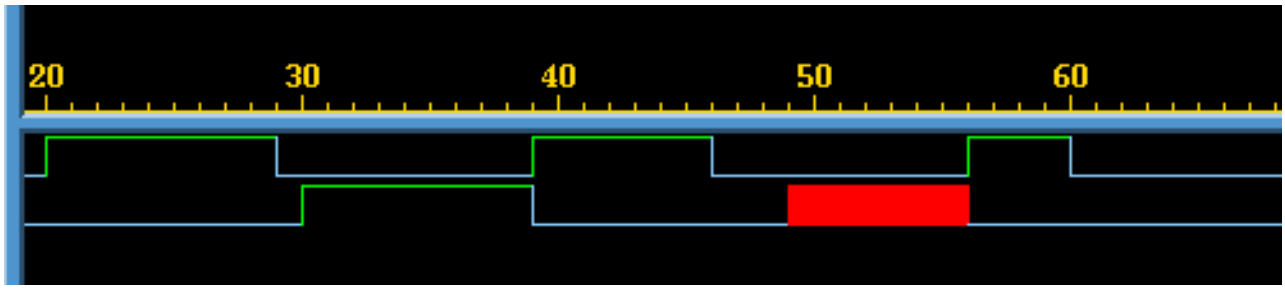
You specify transport delays for INTERCONNECT delays on nets with the +`transport_int_delays`, `+pulse_int_e/`*`number`*, and `+pulse_int_r/`*`number`* options. You must include all three of these options.

If you want VCS to propagate all pulses, no matter how narrow, specify a 0 percentage. If you want VCS to, for example, replace pulses that are narrower than 80% of the delay with a X value pulse (and display a warning message) and filter out pulses that are narrower than 50% of the delay, enter the `+pulse_e/80` and `+pulse_r/50` or `+pulse_int_e/80` and `+pulse_int_r/50` compile-time options.

Figure 8-4 shows the waveforms for the input and output ports for an instance of a module that models a buffer with a ten time unit module path delay. The `vcs` command line contains the following compile-time options:

```
+transport_path_delays +pulse_e/80 +pulse_r/50
```

Figure 8-4 Pulse Control with Transport Delays



In the example illustrated in Figure 8-4 the following occurs:

1. At time 20 the input port toggles to 1.

2. At time 29 the input port toggles to 0 ending a nine time unit wide value 1 pulse on the input port.

3. At time 30 the output port toggles to 1. The nine time unit wide value 1 pulse that began at time 20 on the input port is propagating to the output port because we have enabled transport delays and nine time units is more than 80% of the ten time unit module path delay.

4. At time 39 the input port toggles to 1 ending a ten time unit wide value 0 pulse.
   Also at time 39 the output port toggles to 0. The ten time unit wide value 0 pulse that began at time 29 on the input port is propagating to the output port.

5. At time 46 the input port toggles to 0 ending a seven time unit wide value 1 pulse.

6. At time 49 the output port transitions to X. The seven time unit wide value 1 pulse that began at time 39 on the input port has propagated to the output port but VCS has replaced it with an X value pulse because seven time unit is less than 80% of the module path delay.
   You also see at this time the following warning message:

   ```
   Warning : Time = 49; Pulse flagged as an error in
   module_instance_name, value = StE.
   Path: input_port --->output_port = 10;
   ```

7. At time 56 the input port toggles to 1 ending a ten time unit wide value 0 pulse. Also at time 56 the output port toggles to 0. The ten time unit wide value 0 pulse that began at time 46 on the input port is propagating to the output port.

8. At time 60 the input port toggles to 0 ending a four time unit wide value 1 pulse. Four time units is less than 50% of the module path delay so VCS filters out this pulse and no indication of it appears on the output port.

Pulse Control with Inertial Delays

You can enter the `+pulse_e/number` and `+pulse_r/number` or `+pulse_int_e/`*`number`* and `+pulse_int_r/`*`number`* options without the `+transport_path_delays` or `+transport_int_delays` options. When you do you are specifying pulse control for inertial delays on module path delays and INTERCONNECT delays.

There is a special implementation of inertial delays with pulse control for module path delays and INTERCONNECT delays. In this implementation value changes on the input can schedule two events on the output.

The first of these two scheduled events always causes a change on the output. The type of value change on the output is determined by the following:

- If the first event is scheduled by the leading edge of a pulse whose width is equal to or wider than the percentage specified by the `+pulse_e/`*`number`* number option, the value change on the input propagates to the output.
- If the pulse is not wider than percentage specified by the `+pulse_e/`*`number`* number option, but is wider that the percentage specified by the `+pulse_r/`*`number`* option, the value change is replaced by an X value.

- If the pulse is not wider than percentage specified by the `+pulse_r/`*`number`* option, the pulse is filtered out.

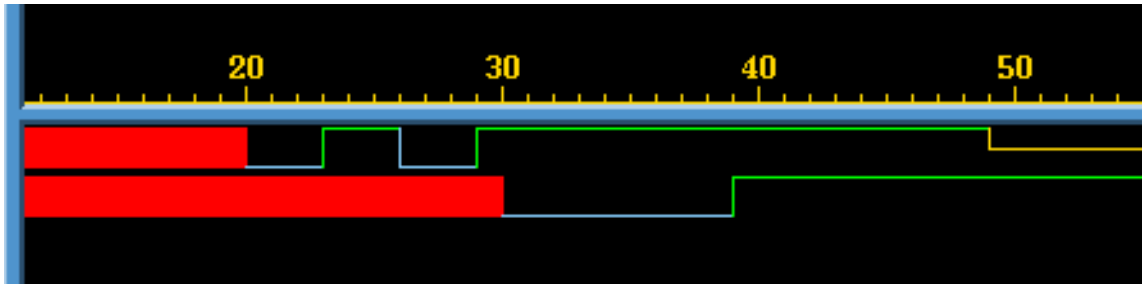
The second scheduled event is always tentative. If another event occurs on the input before the first event occurs on the output, that additional event on the input cancels the second scheduled event and schedules a new second event.

Figure 8-5 shows the waveforms for the input and output ports for an instance of a module that models a buffer with a ten time unit module path delay. The `vcs` command line contains the following compile-time options:

```
+pulse_e/0 +pulse_r/0
```

Specifying 0 percentages here means that the trailing edge of all pulses can change the second scheduled event on the output. Specifying 0 does not mean that all pulses propagate to the output because this implementation has its own way of filtering out short pulses.

Figure 8-5 Pulse Control with Inertial Delays



In the example illustrated in Figure 8-5 the following occurs:

1. At time 20 the input port transitions to 0. This schedules a transition to 0 on the output port at time 30, ten time units later as specified by the module path delay. This is the first scheduled event on the output port. This event is not tentative, it will occur.

2. At time 23 the input port toggles to 1. This schedules a transition to 1 on the output port at time 33. This is the second scheduled event on the output port. This event is tentative.

3. At time 26 the input port toggles to 0. This cancels the current scheduled second event and replaces it by scheduling a transition to 0 at time 36. The first scheduled event is a transition to 0 at time 30 so the new second scheduled event isn't really a transition on the output port.
   This is how this implementation filters out narrow pulses.

4. At time 29 the input port toggles to 1. This cancels the current scheduled second event and replaces it by scheduling a transition to 1 at time 39.

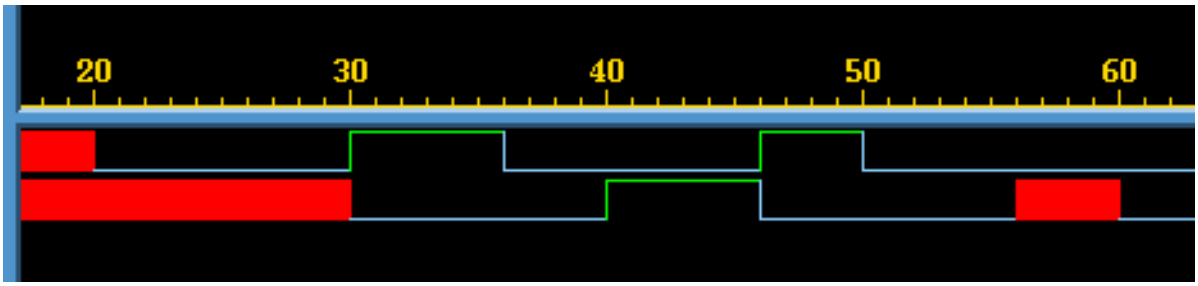
5. At time 30 the output port transitions to 0. The second scheduled event on the output becomes the first scheduled event and is therefore no longer tentative.

6. At time 39 the output port toggles to 1.

Typically, however, you will want to specify that VCS replace or reject certain narrow pulses. Figure 8-6 shows the waveforms for the input and output ports for an instance of the same module with a ten time unit module path delay. The `vcs` command line contains the following compile-time options:

```
+pulse_e/60 +pulse_r/40
```

Figure 8-6 Pulse Control with Inertial Delays and a Narrow Pulses



In the example illustrated in Figure 8-6 the following occurs:

1. At simulation time 20 the input port transitions to 0. This schedules the first event on the output port, a transition to 0 at time 30.

2. At simulation time 30 the input port toggles to 1. This schedules the output port to toggle to 1 at time 40.
   Also at simulation time 30 the output port transitions to 0.
   It doesn't matter which of these events happened first. At the end of this time there is only one scheduled event on the output.

3. At simulation time 36 the input port toggles to 0. This is the trailing edge of a six time unit wide value 1 pulse. The pulse is equal to the width specified with the `+pulse_e/60` option so VCS schedules a second event on the output, a value change to 0 on the output at time 46.

4. At simulation time 40 the output toggles to 1 so now there is only one event scheduled on the output, the value change to 0 at time 46.

5. At simulation time 46 the input toggles to 1 scheduling a transition to1 at time 56 on the output.
   Also at time 46 the output toggles to 0.
   There is now only one event scheduled on the output.

6. At time 50 input port toggles to 0. This is the trailing edge of a four time unit wide value 1 pulse. The pulse is not equal to the width specified with the `+pulse_e/60` option but is equal to the width specified with the `+pulse_r/40` option so VCS changes the first scheduled event from a change to 1 to a change to X at time 56 and schedules a second event on the output, a transition to 0 at time 60.

7. At time 56 the output transitions to X and VCS displays the error message:

   ```
   Warning: time = 56; Pulse Flagged as error in
   module_instance_name, Value = StE
   port_name ---> port_name = 10;
   ```

8. At time 60 the output transitions to 0.

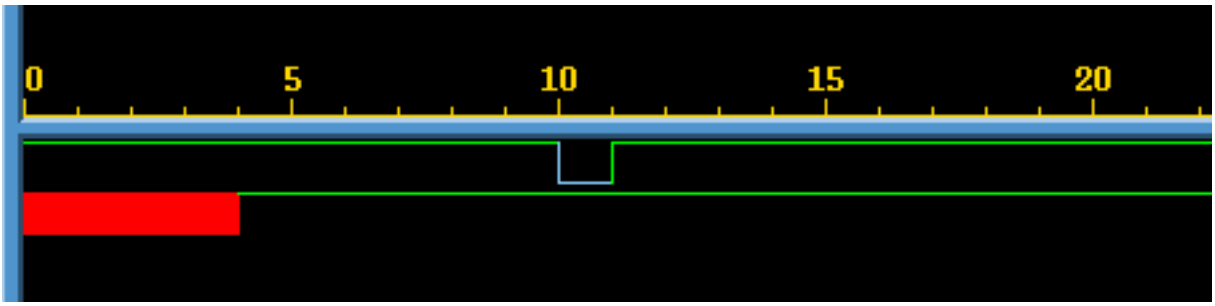
Pulse control sometimes blurs the distinction between inertial and transport delays. In this example the results would have been the same if you also included the `+transport_path_delays` option.

Specifying Pulse on Event or Pulse on Detect Behavior

Asymmetric delays, such as different rise and fall times for a module path delay, can cause schedule cancellation problems for pulses. These problems persist when you specify transport delay and can persist for a wide range of percentages that you specify for the pulse control options.

For example for a module that models a buffer, if you specify a rise time of 4 and a fall time of 6 for a module path delay a narrow value 0 pulse can cause scheduling problems, as illustrated in Figure 8-7.

Figure 8-7 Asymmetric Delays and Scheduling Problems



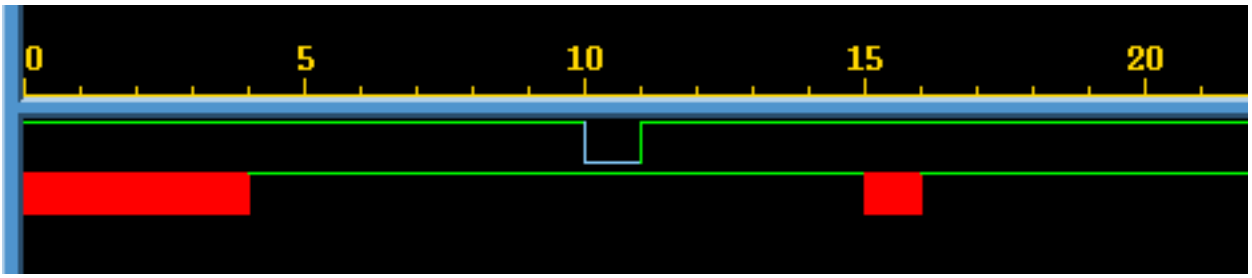
Here you include the `+pulse_e/100` and `+pulse_r/0` options. The scheduling problem is that the leading edge of the pulse on the input, at time 10, schedules a transition to 0 on the output at time 16; but the trailing edge, at time 11, schedules a transition to 1 on the output at time 15.

Obviously the output has to end up with a value of 1 so VCS can't allow the events scheduled at time 15 and 16 to occur in sequence; if it did the output would end up with a value of 0. This problem persists when you enable transport delays and whenever the percentage specified in the `+pulse_r/`*`number`* option is low enough to enable the pulse to propagate through the module.

To circumvent this problem, when a later event on the input schedules an event on the output that is earlier than the event scheduled by the previous event on the input, VCS cancels both events on the output.

This ensures that the output ends up with the proper value but what it doesn't do is indicate that something happened on the output between times 15 and 16. You might want to see an error message and an X value pulse on the output indicating there was an undefined event on the output between these simulation times. You see this message and the X value pulse if you include the `+pulse_on_event` compile-time option, specifying pulse on event behavior, as illustrated in Figure 8-8. Pulse on event behavior calls for an X value pulse on the output after the delay and when there are asymmetrical delays scheduling events on the output that would be canceled by VCS, to output an X value pulse between those events instead.

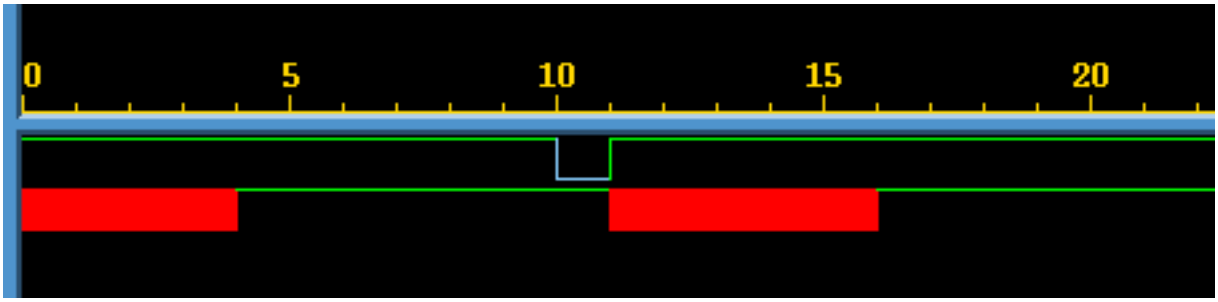
Figure 8-8 Using +pulse_on_event



In most cases where the `+pulse_e/`*`number`* and `+pulse_r/`*`number`* options already create X value pulses on the output, also including the `+pulse_on_event` option to specify pulse on event behavior will make no change on the output.

Pulse on detect behavior, specified by the `+pulse_on_detect` compile-time option, displays the leading edge of the X value pulse on the output as soon as events on the input, controlled by the `+pulse_e/`*`number`* and `+pulse_r/`*`number`* options, schedule an X value pulse to appear on the output. Pulse on detect behavior differs from pulse on event behavior in that it calls for the X value pulse to begin before the delay elapses. Figure 8-9 illustrates pulse on detect behavior.

Figure 8-9 Using +pulse_on_detect

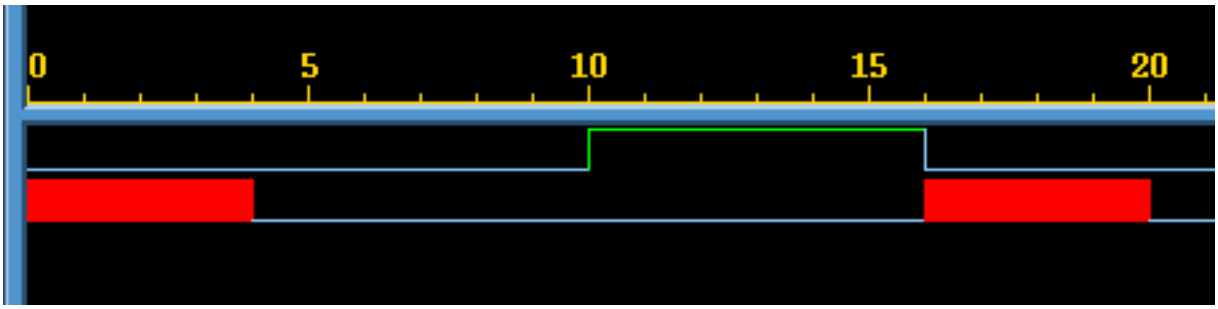


In this example, by including the `+pulse_on_detect` option, when VCS detects the unusual events on the output between times 15 and 16 at simulation time 11 so the leading edge of the X value pulse on the output begins at time 11.

Using pulse on detect behavior can also show you when VCS has scheduled multiple events for the same simulation time on the output but starting the leading edge of an X value pulse on the output as soon as VCS has scheduled the second event at the same simulation time on the output.

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 8-10 shows the waveforms for the input and output port when you include the `+pulse_on_detect` option.

Figure 8-10 Pulse on Detect Behavior Showing Multiple Transitions



In the example illustrated in Figure 8-10 the following occurs:

1. At simulation time 0 the input port transitions to 0 scheduling the first event on the output, a transition to 0 at time 4.
2. At time 4 the output transitions to 0.
3. At time 10 the input transitions to 1 scheduling a transition to 1 on the output at time 20.
4. At time 16 the input toggles to 0 scheduling a second event on the output at time 20, a transition to 0. This event also is the trailing edge of a six time unit wide value 1 pulse so the first event changes to a transition to X. There are more than one event, for different value changes on the output at time 20 so VCS begins the leading edge of the X value pulse on the output at this time.
5. At time 20 the output toggles to 0, the second scheduled event at this time.

If you did not include the `+pulse_on_detect` option, or substituted the `+pulse_on_event` option, you would not see the X value pulse on the output between times 16 and 20.

Pulse on detect behavior does not just show you when asymmetrical delays schedule multiple events on the output. Other kinds of events can cause multiple events on the output at the same simulation time, such as different transition times on two input ports and different module path delays from these input ports to the output port could schedule multiple events on the output at the same simulation time. Pulse on detect behavior would show you an X value pulse on the output starting when the second event was scheduled on the output port.

Specifying The Delay Mode

It is possible for a module definition to include module path delay that does no equal the cumulative delay specifications in primitive instances and continuous assignment statements in that path. Example 8-11 shows just such a conflict.

Example 8-11 Conflicting Delay Modes

```
‘timescale 1 ns / 1 ns
module design (out,in);
output out;
input in;
wire int1,int2;

assign #4 out=int2;

buf #3 buf2 (int2,int1),
       buf1 (int1,in);

specify
(in => out) = 7;
endspecify
endmodule
```

In Example 8-11, the module path delay is seven time units but the delay specifications distributed along that path add up to ten time units.

If you include the `+delay_mode_path` compile-time option, VCS ignores the delay specifications in the primitive instantiation and continuous assignment statements and uses only the module path delay. In Example 8-11, it would use the seven time unit delay for propagating signal values through the module.

If you include the `+delay_mode_distributed` compile-time option, VCS ignores the module path delays and uses the delay in the delay specifications in the primitive instantiation and continuous assignment statements. In Example 8-11, it would use 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 8-11 the `` `timescale `` compiler directive has a precision argument of 1 ns. VCS might use this 1 ns as the delay, but if the module definition is used in a larger design and there is another `` `timescale `` compiler directive in the source code with a finer precision argument, then VCS would use that 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 8-11, the module path delay does not equal the distributed delays along the path, VCS uses the longer of the two.

Using SDF Files

The OVI Standard Delay File (SDF) specification provides a standard ASCII file format for representing and applying delay information. VCS supports the OVI versions 1.0, 1.1, 2.0, 2.1, and 3.0 of this specification.

In the SDF format a tool can specify intrinsic delays, interconnect delays, port delays, timing checks, timing constraints, and pulse control (PATHPULSE).

When VCS reads an SDF file it “back annotates” delay values to the design; it adds delay values or changes the delay values specified in the source files. You tell VCS to back annotate delay values with the `$sdf_annotate` system task.

There are two methods that you can use to back annotate delay values from an SDF file:

- Compiling the SDF file at compile-time
  This method is preferable in almost all cases and VCS does this by default.
- Reading the ASCII SDF file at run-time
  This method remains chiefly for compatibility purposes.

Compiling the SDF file, when you compile your Verilog source files, creates binary data files that VCS reads much faster than ASCII SDF file when VCS executes a `$sdf_annotate` system task. The additional compile time will always be less than the time saved at run time. There are, however, limitations on your design when you compile an SDF file. If you cannot circumvent these limitations you can use the method of telling VCS to read the ASCII SDF file when it executes a `$sdf_annotate` system task.

When you use an SDF file to back annotate delay values you can also use an SDF configuration file. In this file you can specify, among other things, the selection of minimal, typical, or maximal delay values in min:typ:max delay value triplets, the scaling of these delay values, and specify these delay value operations for your entire design and on a module by module basis.

This section includes:

- The `$sdf_annotate` system task syntax
- Compiling the SDF file at compile-time
- Reading the ASCII SDF file during run-time
- Using an SDF configuration file

The $sdf_annotate System Task

You use the `$sdf_annotate` system task to tell VCS to back annotate delay values from and SDF file to your Verilog design.

The syntax for the `$sdf_annotate` system task is as follows:

```
$sdf_annotate ("sdf_file"[, module_instance]
[,"sdf_configfile"][,"sdf_logfile"][,"mtm_spec"]
```

[,"*scale_factors*"][,"*scale_type*"]);

Where:

"*sdf_file*"
Specifies the path to the SDF file.

module_instance
Specifies the scope where back annotation starts. The default is the scope of the module instance calling `$sdf_annotate`.

"*sdf_configfile*"
Specifies the SDF configuration file

"*sdf_logfile*"
Specifies the SDF log file to which VCS sends error messages and warnings. By default VCS displays no more than ten warning and 10 error messages about back annotation and writes no more than that in the log file you specify with the `-l` option, but if you specify an SDF log 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. Specify `MINIMUM`, `TYPICAL`, `MAXIMUM` or `TOOL_CONTROL` (default).

"*scale_factors*"
Colon separated string of three positive, real numbers - `"1.0:1.0:1.0"` by default. Specifies the multiplier for the minimum, typical and maximum components of delay triplets.

"*scale_type*"

Specifies the delay value from each triplet in the SDF file for use before scaling. Possible values: "FROM_TYPICAL", "FROM_MIMINUM", "FROM_MAXIMUM", "FROM_MTM" (default).

Compiling The ASCII SDF File at Compile-Time

VCS automatically compiles SDF files when you compile your design. VCS compiles the SDF file you specify as the first argument to the $sdf_annotate system tasks in your design.

This method saves you simulation time, however in some cases you must disable the automatic compilation of sdf files with the +oldsdf compile-time option.

Limitations on Compiling The SDF File

Work is progressing on eliminating the limitations on compiling the SDF file, however currently you cannot have VCS compile your SDF file if:

- You do not use a string literal to specify the SDF file in the $sdf_annotate system task, for example, you assign the SDF file name to a register and enter the register as the first argument to the $sdf_annotate system task.
- You include the sdf_configfile, scale_type, or scale_factor arguments in the $sdf_annotate system task.

If your design exceeds these limitations you must use the method of reading the ASCII SDF file during runtime and include the +oldsdf compile-time option.

Example 8-12 Compiling the SDF File Example

The following small Verilog model, in source file `ex.v`, does not exceed the limitations of compiling the SDF file:

```
`timescale 1ns / 1ns

module test();
wire in, out, clk, out1;
        initial $sdf_annotate( "./ex.sdf");
    leafA leaf1(out, in, clk);
    leafB leaf2(out1, out, clk);
endmodule

module leafA(out,D,CK);
output out;
input CK, D;

specify
 (D *> out) = (1,2);
 (CK *> out) = (3);
endspecify
endmodule

module leafB(out,D,CK);
output out;
input D;
input CK;

buf(out,D);

endmodule
```

The following is the SDF file, `ex.sdf`, for the Verilog model

```
(DELAYFILE
  (DESIGN              "test")
  (VENDOR              "")
  (DIVIDER             .)
  (VOLTAGE             :1:)
  (PROCESS             "typical")
  (TEMPERATURE:1:)
```

```
  (TIMESCALE1ns)
  (CELL
    (CELLTYPE          "leafB")
    (INSTANCE   leaf2)
    (DELAY
       (ABSOLUTE
          (PORT D (1:2:3)))
      )
   )
  (CELL
    (CELLTYPE          "leafA")
    (INSTANCE   leaf1)
    (DELAY
       (ABSOLUTE
          (IOPATH  D out (7)))
      )
  )
)
```

The following is the vcs command line that compiles both the verilog source file and the SDF file:

```
vcs +compsdf ex.v
```

You do not have to specify the `ex.sdf` file, or any SDF table file, on the vcs command line. When VCS compiles the SDF file it creates binary data files in the `simv.daidir` directory. VCS reads these binary files when it executes the `$sdf_annotate` system task.

Precompiling An SDF File

Whenever you compile your design, if you 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, and even when you were using incremental compilation and the parts of the design back annotated by the SDF file were unchanged.

VCS can parse the precompiled SDF file much faster than it can parse the ASCII text sdf file so for large SDF files it's a good idea to have VCS create a precompiled version on 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 the ASCII text SDF file's extension. For example, if the /u/design/sdf directory contains a design1.sdf file, using the `+csdf+precompile` option creates the precompiled version of the file named design1.sdf_c in the /u/design/sdf directory.

After you have created the precompiled version of the SDF file you no longer need to include the `+csdf+precompile` option on the `vcs` command line unless there is a change in the SDF file. Continuing to include it, however, such as in a script that you run every time you compile your design, would have no effect when the precompiled version was newer than the ASCII text SDF file, but would create a new precompiled version of the SDF file whenever the ASCII test SDF file changes. Therefore this option is intended to be used in scripts for compiling your design.

When you recompile your design, VCS will find the precompiled SDF file is the same directory as the SDF file specified in the `$sdf_annotate` system task. You can also specify the precompiled SDF file in the `$sdf_annotate` system task.

Specifying an Alternative Name and Location

You can use the `+csdf+precomp+dir+`*`directory`* option to specify the directory path where you want VCS to write the precompiled SDF file. When you do, make sure that all directories in the path have already been created. VCS does not create directories that are specified with this option.

You can use the `+csdf+precomp+ext+`*`ext`* option to specify an alternative to the "_c" character string addition to the filename extension of the precompiled SDF file.

For example, in the /u/designs/mydesign directory are the design.v and design.sdf files and the sdfhome directory. If you enter the following command line:

```
vcs design.v +csdf+precompile +csdf+precomp+dir+sdfhome
+csdf+precomp+ext+_precompiled
```

VCS creates, in the sdfhome directory, the design.sdf_precompiled file.

Now that the precompiled file is not in the default location and does not have the default filename extension, you must tell VCS where it is and what's its name in subsequent compilations. There are two ways to do this:

1. Continue to include these options on the `vcs` command line in subsequent compilations. In this example you would always include `+csdf+precomp+dir+sdfhome` and `+csdf+precomp+ext+_precompiled`.

   This method does not require you to make a change in the source code. You can just add these options to a script you run whenever you compile your design.

2. Change the filename argument in the `$sdf_annotate` system task to the precompiled file. In this example you would change:

   ```
   $sdf_annotate("design.sdf");
   ```

   to:

   ```
   $sdf_annotate("sdfhome/design.sdf_precompiled");
   ```

Reading The ASCII SDF File During Runtime

You can use the ACC capability support that is part of the PLI interface to tell VCS to read the ASCII SDF file when it executes the `$sdf_annotate` system task.

To do this you include the `+oldsdf` compile-time option and create a PLI table file that maps the `$sdf_annotate` system task to the C function `sdf_annotate_call` (automatically provided with VCS) and indicates which modules will be annotated and what types of constructs will be annotated.

For faster simulation enable, in the PLI table, only the ACC capabilities you need. These capabilities are as follows:

`tchk:`
: Annotate to timing checks (sdf SETUP, HOLD, etc.)

`gate:`
: Annotate to gate primitives (sdf DEVICE)

`mp:`
: Annotate propagation delays to module paths (sdf IOPATH)

`mip:`
: Annotate propagation delays to module input port delays (sdf PORT and INTERCONNECT)

`mipb:`
: Annotate to module input port bit delays (sdf PORT and INTERCONNECT)

`prx:`
: Annotated pulse rejection and error delays to module paths (sdf PATHPULSE)

Given module myasic, SDF annotation of module paths contained within the hierarchy of myasic requires a PLI table such as this:

```
$sdf_annotate call=sdf_annotate_call acc+=mp,prx:myasic+
```

If possible, take advantage of the %CELL wildcard scope to add the needed ACC capabilities to library cells only, often the only cells requiring SDF annotation:

```
$sdf_annotate call=sdf_annotate_call acc+=mp, prx:%CELL
```

For methodologies requiring delay annotation in the sub-hierarchy of cells use:

```
$sdf_annotate call=sdf_annotate_call acc+=mp, prx:%CELL+
```

When running vcs use the -P option to specify this PLI table, as you would to specify any user task or function implemented in a custom PLI application. You do not need to provide the function sdf_annotate_call as it is part of the VCS product by default.

```
% vcs -P sdf.myasic.tab myasic.v +oldsdf
```

Example 8-13 Reading the ASCII SDF File Example

The following small Verilog model, in source file ex2.v, contains a specparam in its specify block:

```
‘timescale 1 ns / 1 ns
module top;
reg in;
leaf leaf1(in,out);
initial begin
     $sdf_annotate(“ex2.sdf”,top);
     $monitor($time,,in,,out);
          in = 0;
     #100 $finish;
end
endmodule
module leaf(in,out);
input in;
output out;
buf(out,in);
specify
     specparam mpath_d=1.0;
```

```
     (in => out) = (mpath_d);
endspecify
endmodule
```

The following is the SDF file, `ex2.sdf`, for the Verilog model:

```
(DELAYFILE
     (TIMESCALE 1 ns)
          (CELL
          (CELLTYPE “leaf”)
          (INSTANCE leaf1)
          (DELAY
          (ABSOLUTE
          (IOPATH in out (5))))))
```

In this file the SDF construct `IOPATH` corresponds to a Verilog module path delay. The delay value specified is 5. The time unit, specified by the `TIMESCALE` construct makes the annotated delay value to the module path delay 5 ns.

The following is the contents of the PLI table file, `ex2.tab`:

```
$sdf_annotate call=sdf_annotate_call acc=mp:top+
```

We specify the PLI table file on the vcs command line:

```
vcs -P ex2.tab ex2.v
```

We see the successful back annotation of the delay value when we execute the simv executable and see transition times and values from the `$monitor` system task:

```
0 0 x
5 0 0
$finish at simulation time 100
V C S        S i m u l a t i o n              R e p o r t
Time: 100 ns
CPU Time: 0.100 seconds; Data structure size: 0.0Mb
```

Performance Considerations

Because the compiler must make large quantities of information about the structure of the design available at run time in order to allow annotation, you must consider simulation efficiency when using SDF annotation. Keep in mind the following:

- For annotation capabilities `gate`, `mp`, and `tchk`, there is overhead only if the modules actually contain the related constructs.
- For module input port delays, significant compile and run time overhead can be incurred if annotation is enabled on ports that will not be annotated.
- Enabling port bit annotation increases the overhead.

Use the `%CELL` wildcard scope as shown above (or `%CELL+` if all the annotated cells are below the `$sdf_annotate` call) to annotate only those modules that require SDF annotation ACC capabilities.

Replacing Negative Module Path Delays in SDF Files

VCS does not backannotate negative module path delays from IOPATH entries in SDF files. By default VCS substitutes a 0 delay for these negative delays.You can tell VCS to instead use the module path delay specified in a module's specify block by including the `+old_iopath` compile-time option.

Using The Shorter Delay in IOPATH Entries

It is valid syntax in an SDF file to have two IOPATH entries for the same pair of ports but one entry for a rising edge on the input or inout port and another entry for a falling edge on the input or inout port, for example:

```
(IOPATH (posedge inport) outport (3) (6))
(IOPATH (negedge inport) outport (5) (4))
```

These entries specify back annotating the following delay values to the module path delay between the port named inport and the port named outport:

- If a rising edge on inport results in a rising edge on outport, then the module path delay is three.
- If a rising edge on inport results in a falling edge on outport, then the module path delay is six.
- If a falling edge on inport results in a rising edge on outport, then the module path delay is five.
- If a falling edge on inport results in a falling edge on outport, then the module path delay is four.

VCS does not however back annotate the module path delay values as specified. Instead VCS back annotates the shorter of the corresponding delays in the two IOPATH entries. Therefore the following behavior occurs during simulation when a value propagates from inport to outport:

- When the value propagation results in a rising edge on outport, the module path delay from inport to outport is three, no matter whether the value propagation began with a rising or falling edge on inport, because three is shorter than five.
- When the value propagation results in a falling edge on outport, the module path delay from inport to outport is four, no matter whether the value propagation began with a rising or falling edge on inport, because four is shorter than six.

In this example there are two delay values in the two IOPATH entries for the same pair of ports, for a rising and falling edge on the output or inout port, but VCS would also back annotate the shorter of the corresponding delay values if the IOPATH entries each had only one delay value or each had three or six delay values (delay value lists with 12 values are not implemented).

VCS does this, back annotates the shorter of the corresponding delay values, to be compatible with the earlier generation of Verilog simulators, Cadence's Verilog-XL.

Disabling CELLTYPE Checking in SDF Files

Sometimes when you merge smaller designs into a larger design you discover that you have more than one module in the larger design with the same identifier or name. When this happens you must resolve the conflict by editing your Verilog source code to rename one of the modules that shares the same identifier. (Some users developed their own tool to change the module identifiers in their source code.)

If an SDF file backannotates delay values to an instance of a module that must be renamed, the CELLTYPE entry in the SDF file for the instance will contain the old identifier of the module, and this is ordinarily would cause VCS to display the following error message:

```
SDF Error: Instance instance_name is CELLTYPE of
"new_module_identifier" not "old_module_identifier",
ignoring
```

In this situation, to avoid editing the SDF file, include the `+sdf_nocheck_celltype` compile-time option on the `vcs` command line.

The SDF Configuration File

The configuration file syntax shown below allows the user on a module type as well as a global basis, to control:

- min:typ:max selection
- scaling
- turnoff delay determination
- mipd (module-input-delay) approximation policy for cases of ‘overlapping’ annotations to the same input port.

Additionally, there is a mapping command to redirect the target of `IOPATH` and `TIMINGCHECK` statements for all instances of a specified module type, from the scope of the `INSTANCE` instead to a specific `IOPATH` or `TIMINGCHECK` in its subhierarchy as specified by the user.

Delay Objects and Constructs

The mapping from SDF statements to simulation objects in VCS is in fact fixed, as shown in Table 8-14.

Table 8-14 VCS Simulation Delay Objects/Constructs

SDF Constructs	VCS Simulation Object
Delays	

Table 8-14 VCS Simulation Delay Objects/Constructs

SDF Constructs	VCS Simulation Object
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` 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
SKEWCONSTRAINT	ignored

SDF Configuration File Commands

This section explains the Commands used in SDF configuration file, with syntax and examples. The following is the list of SDF configuration file commands:

- approx_command
- map_command
- mtm_command
- scale_command
- turnoff_command
- modulemap_command

approx_command:

The `INTERCONNECT_MPID` keyword selects the INTERCONNECT delays in the SDF file that are mapped to MIPDs in VCS.

Specifies one of the following to VCS:

`MINIMUM`
: Annotates, to the MIPD for the input or inout port instance, the shortest delay of all the INTERCONNECT delay value entries in the SDF file that specify a connection to the input or inout port.

`MAXIMUM`
: Annotates, to the MIPD for the input or inout port instance, the longest delay of all the INTERCONNECT delay value entries in the SDF file that specify a connection to the input or inout port.

AVERAGE
: Annotates, to the MIPD for the input or inout port instance, the average delay of all the INTERCONNECT delay value entries in the SDF file that specify a connection to the input or inout port.

LAST
: Annotates, to the MIPD for the input or inout port instance, the delays in the last INTERCONNECT entry in the SDF file that specifies a connection to the input or inout port.

The default approximation is MAXIMUM

Syntax:

```
INTERCONNECT_MIPD = MINIMUM | MAXIMUM | AVERAGE | LAST;
```

Example:

```
INTERCONNECT_MIPD=LAST;
```

map_command:

This command maps the sdf constructs to VCS simulation Delay Objects.

Note: Refer to Table 8-14, VCS Simulation Delay Objects/Constructs.

Syntax:

```
sdf_construct = veritool_map ;
sdf_construct : IOPATH | PORT | INTERCONNECT | NETDELAY |
DEVICE | SETUP | HOLD | SETUPHOLD | RECOVERY | SKEW | WIDTH
| PERIOD | NOCHANGE | PATHPULSE | GLOBALPATHPULSE
veritool_map : IGNORE | INTERMOD_PATH | MIPD | CELL | USE
```

Example:

```
INTERCONNECT=MIPD;
```

```
PATHPULSE=IGNORE;
```

mtm_command:

Annotates the minimum, typical, or maximum delay value. Specifies one of the following keywords:

`MINIMUM`
: Annotates the minimum delay value

`TYPICAL`
: Annotates the typical delay value

`MAXIMUM`
: Annotates the maximum delay value

The default for min_typ_max is `TOOL_CONTROL`.

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

Syntax:

```
MTM = MINIMUM | TYPICAL | MAXIMUM | TOOL_CONTROL;
```

Example:

```
MTM=MAXIMUM;
```

scale_command:

SCALE_FACTORS - Set of three real number multipliers that scales the timing information in the SDF file to the minimum, typical, and maximum timing information that is back annotated to the verilog tool. The multipliers each represent a positive real number, i.e. 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.

Syntax:

```
SCALE_FACTORS =  number : number : number ;
SCALE_TYPE = FROM_MINIMUM | FROM_TYPICAL | FROM_MAXIMUM |
FROM_MTM ;
```

Example:

```
SCALE_FACTORS=100:0:9;
SCALE_TYPE=FROM_MTM;
SCALE_FACTORS=1.1:2.1:3.1;
SCALE_TYPE=FROM_MINIMUM;
```

turnoff_command:

The turnoff_command keyword globally specifies which delays the SDF annotator uses.

FROM_FILE - The SDF annotator uses the turn-off delays in the SDF file. If you do not specify FROM_FILE, the default is min (rise,fall).

If you specify FROM_FILE but the SDF files does not contain the turn-off delay, the turn-off delay is set to min (rise,fall).

SDF annotator uses the min, max, or average of the values specified in the SDF file (as with the min:typ:max triplet). SDF annotator uses one of the following for a given object:

`MINIMUM`
: Minimum of the rise and fall delay value.

`MAXIMUM`
: MAXimum of the rise and fall delay value.

`AVERAGE`
: Average of the rise and fall delay value.

Syntax:

```
TURNOFF_DELAY = FROM_FILE | MINIMUM | MAXIMUM | AVERAGE ;
```

Example:

```
TURNOFF_DELAY=FROM_FILE;
```

modulemap_command:

Redirects the delay object of `IOPATH` or `TIMINGCHECK` sdf statements for all instances of a specified module type to a different module path or timing check object in the same or lower scope.

Syntax:

```
MODULE verilog_id { list_of_module_mappings } ;
```

`verilog_id` is the name of a specific type of module (not instance name) specified in the corresponding verilog HDL description.

list_of_module_mappings:

`mtm_command | scale_command | map_inner_command`

mtm and scale commands are as defined above. Note that using these commands as arguments for the module_map_command affects only the `IOPATH`, `DEVICE`, and `TIMINGCHECK` information annotated to a specific type of module.

Syntax:

```
MAP_INNER = verilog_id ;
| systchk = ADD { list_of_systchk }
| systchk = ADD { list_of_systchk }
| systchk = OVERRIDE { list_of_systchk }
| systchk = IGNORE ;
| path_declaration = ADD { list_of_path_declaration }
| path_declaration = OVERRIDE { list_of_path_declaration }
| path_declaration = IGNORE ;
```

SDF annotator uses hierarchical_path as the Verilog hierarchical path name of a submodule within module_type. The paths specified in the SDF file are mapped to module_type. This path applies to all path delays and timing checks specified for this module in the SDF file including those mapped with `ADD` and `OVERRIDE`.

`ADD`

Adds to the mapping specifications of the SDF file. The original_ timing specification is mapped to new_timing, the verilog HDL syntax of a path delay or timing check.

`OVERRIDE`

Replaces the mapping specifications of the SDF file. The original_timing specification is mapped to new_timing, the verilog HDL syntax of a path delay or timing check.

IGNORE

Ignores the mapping specifications in the SDF file. In all cases, the hierarchical_path name is applied to any new_timing specification before they are annotated to VCS.

```
list_of_systchk : systchk ';' | list_of_systchk systchk ';'
systchk: '$setup' '(' systchk_arg ',' systchk_arg ',' expression opt_notifier ')'
              |'$hold' '(' systchk_arg ',' systchk_arg ',' expression
                   opt_notifier ')'
              |'$setuphold' '(' systchk_arg ',' systchk_arg ',' expression ','
                   expression opt_notifier ')'
              |'$recovery' '(' systchk_arg ',' systchk_arg ',' expression
                   opt_notifier ')'
              |'$period' '(' systchk_arg ',' expression opt_notifier ')'
              |'$width' '(' systchk_arg ',' expression ',' expression
                   opt_notifier ')'
              |'$skew' '(' systchk_arg ',' systchk_arg ',' expression
                   opt_notifier ')'
              |'$nochange' '(' systchk_arg ',' systchk_arg ',' expression ','
                   expression opt_notifier ')'
opt_notifier: ',' expression | ',' | ;
systchk_arg: expression
              | expression '&&&' timing_check_condition
              | timing_check_event_control specify_terminal_descriptor
              | timing_check_event_control specify_terminal_descriptor
                   '&&&' timing_check_condition
timing_check_condition: expression
list_of_path_declaration: path_declaration ';'
              | list_of_path_declaration path_declaration ';'
path_declaration: opt_if '(' list_of_path_in_td path_type list_of_path_out_td
')'
              | opt_if '(' list_of_path_in_td path_type '('
list_of_path_out_td ')' ')'
opt_if: 'if' '(' expression ')' | ;
opt_edge: timing_check_event_control | ;
timing_check_event_control': 'posedge' | 'negedge' | 'edge'
        '[' edge_descriptor_list ']'
edge_descriptor_list: edge_descriptor
        | edge_descriptor_list ',' edge_descriptor
edge_descriptor : '01' | '10' | '0x' | 'x1' | '1x' | 'x0'
path_type: '=>' | '-' '=>' | '+' '=>' | '*>' | '-' '*>' | '+' '*>'
list_of_path_out_td: list_of_path_out_td ','
        specify_out_terminal_descriptor | specify_out_terminal_descriptor
specify_out_terminal_descriptor: '(' specify_terminal_descriptor data_op
expression ')'
              | specify_terminal_descriptor
              data_op : ':' | '-' ':' | '+' ':'
list_of_path_in_td: list_of_path_in_td ','
        opt_edge_specify_terminal_descriptor
              | opt_edge_specify_terminal_descriptor ;
```

```
opt_edge_specify_terminal_descriptor : opt_edge
             specify_terminal_descriptor ;
specify_terminal_descriptor: verilog_id | verilog_id '[' expression ']'
        | verilog_id '[' expression ':' expression ']' ;
expression : primary | unary_op primary
        | expression '+' expression | expression '-' expression
        | expression '*' expression | expression '/' expression
        | expression '%' expression | expression '==' expression
        | expression '!=' expression | expression '===' expression
        | expression '!==' expression | expression '&&' expression
        | expression '||' expression | expression '<' expression
        | expression '<=' expression | expression '>' expression
        | expression '>=' expression | expression '&' expression
        | expression '|' expression | expression '^' expression
        | expression '^~' expression | expression '~^' expression
        | expression '>>' expression | expression '<<' expression
        | expression '?' expression ':' expression
unary_op : '!' | '~' | '+' | '-' | '&' | '~&' | '|' | '~|' | '^' | '~^' | '^~'

primary : number | lident | lident '[' number ']'
        | lident '[' number ':' number ']' | '{' cat_expr_list '}'
        | '{' expression '{' cat_expr_list '}' '}' | '(' expression ')'
cat_expr_list: cat_expr_list ',' expression | expression

lident: identifier

identifier: verilog_id | identifier_head verilog_id

identifier_head : verilog_id '.' | identifier_head verilog_id '.'

number : "Any sized or unsized literal decimal, octal, binary, hex, or real number"

verilog_id : "Any legal escaped or non-escaped Verilog identifier (excluding
range selection portion in square brackets)."
```

Example:

```
MODULE sub       {
     // scale_commads
     SCALE_TYPE=FROM_MTM;
     SCALE_FACTORS=1:2:3;
     // mtm_commads
     MTM=MINIMUM;
     // map_inner_commands
     MAP_INNER = X;
     (i1 *> o1) = IGNORE;
     (i *> o1) = ADD { (ib *> oa); }
```

```
    (i1 *> o1) = ADD { (ia *> oa); }
    (i1 *> o1) = ADD { (ia *> oa); }
    (i1 *> o1) = ADD { (ib *> ob); }
    if (i2==1) (i2 *> o2) = ADD { (ib *> ob); }
}
```

SDF Example with Configuration File

The following example uses 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
INTERCONNECT=MIPD;
PATHPULSE=IGNORE;
INTERCONNECT_MIPD=MAXIMUM;
MTM=TOOL_CONTROL;
SCALE_FACTORS=100:0:9;
SCALE_TYPE=FROM_MTM;
TURNOFF_DELAY=FROM_FILE;
MTM = TYPICAL;
SCALE_TYPE=FROM_MINIMUM;
SCALE_FACTORS=1.1:2.1:3.1;
MODULE sub {
     SCALE_TYPE=FROM_MTM;
     SCALE_FACTORS=1:2:3;
     MTM=MINIMUM;
     MAP_INNER = X;
     (i1 *> o1) = IGNORE;
     (i1 *> o1) = ADD { (ia *> oa); }
     (i1 *> o1) = ADD { (ib *> ob); }
     if (i2==1) (i2 *> o2) = ADD { (ib *> ob); }
     }
```

INTERCONNECT Delays

INTERCONNECT entries in an SDF file are for backannotating delays to a net that connects two or more module instance ports.

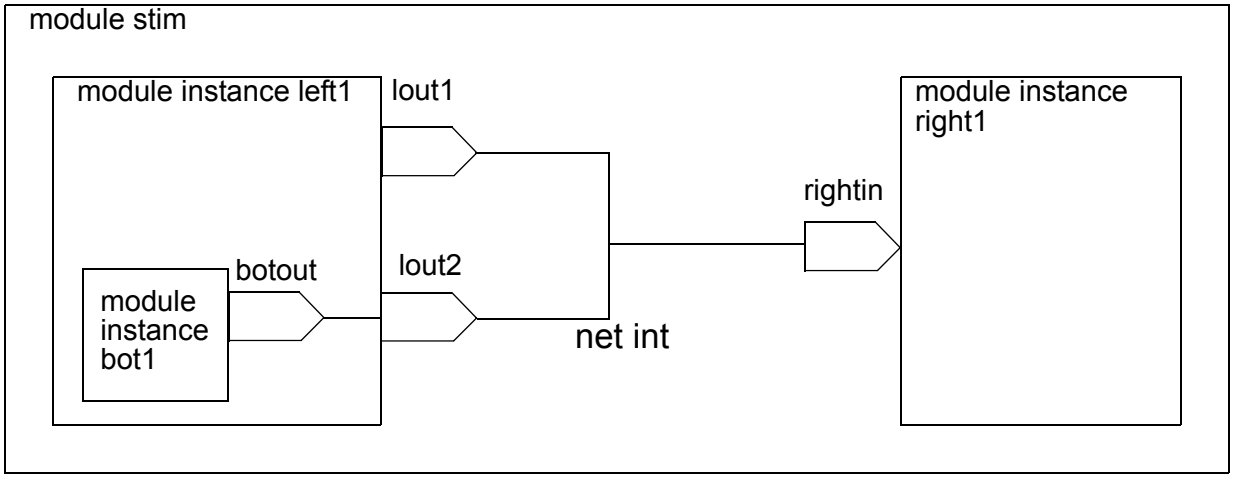
If that net has more than one driver, for example, such as when a net connects more than one output port to an input port, you can use an SDF file to backannotate different delays between the input port and each of the output ports. Doing so is backannotating multisource INTERCONNECT delays.

If that net has only one driver, you are back annotating single source INTERCONNECT delays. There is an option developed for multisource INTERCONNECT delays that can be handy for single source INTERCONNECT delays. See “Single Source INTERCONNECT Delays” on page 8-55.

Multisource INTERCONNECT Delays

Figure 8-15 shows a multisource net, a net with more than one driver.

Figure 8-15 Net with Multiple Drivers



The SDF file could specify different delays between output port lout1 and input port rightin and output port botout and input port rightin. The ports do not have to be on the same hierarchical level.

An example for the entries that specify these different delays are as follows:

```
(CELL
  (CELLTYPE "stim")
  (INSTANCE stim)
  (DELAY
    (ABSOLUTE
     (INTERCONNECT stim.left1.bot1.botout stim.right1.rightin (2))
        (INTERCONNECT stim.left1.lout1 stim.right1.rightin(1))
    )
  )
)
```

These entries specify a 2 time unit delay between output port botout and input port rightin, and a 1 time unit delay between output port lout1 and input port rightin.

Note:

In delay value lists in delay definition entries in an SDF file, you can list one, two, three, or six delay values. Lists of twelve values, that also specify delays for transitions to and from X are not yet implemented.

The `+multisource_int_delays` compile-time option tells VCS to use a special algorithm for multisource INTERCONNECT delays. This algorithm simulates different delays from each of the multisource output or inout ports to the load input or inout port, or in other words, true pin to pin delays.

Omitting The +multisource_int_delays Option

If you omit the `+multisource_int_delays` option, VCS uses an older algorithm that creates a MIPD (Module Input Port Delay) to model the INTERCONNECT delay. Omitting the option, therefore, has the following drawbacks:

- If you specify multiple sources with multiple INTERCONNECT entries for connecting more than one output or inout port instances to a particular input or inout port instance, this algorithm uses the longest delays in these INTERCONNECT entries for the delays in the MIPD so the delay propagating from all sources will be the same.
- MIPDs only take three delay values for transitions to 1, to 0, and to Z. If your INTERCONNECT entry has six delay values, the MIPD only uses the first three so the Z to 1 delay is the same as the 0 to 1 delay, the 1 to Z delay is the same as the 0 t0 Z delay, and the Z to 0 delay is the same as the 1 to 0 delay.

Simultaneous Multiple Source Transitions

When there are simultaneous transitions on more than one source port instance, the algorithm for the `+multisource_int_delays` option applies the shortest delay to all of these transitions instead of the one specified in the SDF file. For example, if the SDF file specifies the following:

```
(CELL
  (CELLTYPE "stim")
  (INSTANCE stim)
  (DELAY
    (ABSOLUTE
     (INTERCONNECT stim.left1.bot1.botout stim.right1.rightin (2))
        (INTERCONNECT stim.left1.lout1 stim.right1.rightin(1))
    )
  )
)
```

Here the delay values are reversed from the previous SDF example. The delay from output port botout to rightin is 2 and the delay from output port lout1 to rightin is 1.

Figure 8-16 shows the waveforms for these ports and net int that connects these ports and to which you back annotate the INTERCONNECT delay.

Figure 8-16 Simultaneous Source Transitions

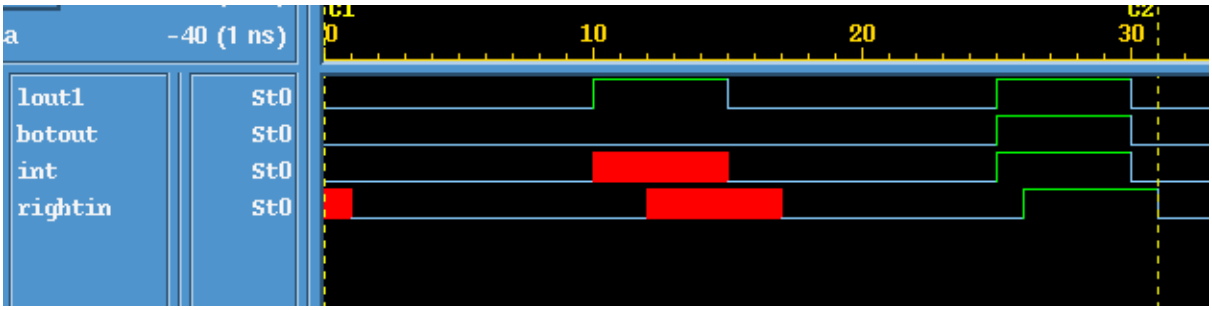


Figure 8-16 illustrates the following series of events:

1. At time 10 output port lout1 transitions to 1. Net int transitions therefore to X and there is a two time unit delay before this X value propagates through int to port rightin.
2. At time 15 port lout1 transitions to 0, Net int transitions to 0 and port right in transactions to 0 two time units later.
3. At time 25 both output ports lout1 and botout transitions to 1. So does net int. Input port rightin transitions to 1 only one time unit later even though there is a two time unit delay between lout1 and rightin. The algorithm applied the shortest delay, one time unit, to the simultaneous transitions at time 25 on lout1 and botout.

Single Source INTERCONNECT Delays

If the INTERCONNECT entries in your SDF file connect no more than one output or inout port to each of the input or inout ports in your design you should consider including the `+multisource_int_delays` compile-time option that is used for multisource INTERCONNECT delays.

If you omit the `+multisource_int_delays` option, VCS uses an older algorithm that creates a MIPD (Module Input Port Delay) to model the INTERCONNECT delay. MIPDs only take three delay values for transitions to 1, to 0, and to Z. If your INTERCONNECT entry has six delay values, the MIPD only uses the first three so the Z to 1 delay is the same as the 0 to 1 delay, the 1 to Z delay is the same as the 0 t0 Z delay, and the Z to 0 delay is the same as the 1 to 0 delay.

Note:

In delay value lists in delay definition entries in an SDF file, you can list one, two, three, or six delay values. Lists of twelve values, that also specify delays for transitions to and from X are not yet implemented.

Min:Typ:Max Delays

You can enter min:typ:max delay value triplets where ever you can enter a delay specification, for example:

```
assign #(4:10:14) w1=r1;
```

Here the minimum delay value is 4, the typical delay value is 10, and the maximum delay value is 14.

You can also enter min:typ:max delay value triplets in delay value in entries for different kinds of delays in SDF files. For example, if an SDF file specifies the following:

```
(CELL
  (CELLTYPE "stim")
  (INSTANCE stim)
  (DELAY
    (ABSOLUTE
              (INTERCONNECT stim.left1.lout1
    stim.right1.rightin(6:10:14))
    )
  )
)
```

The INTERCONNECT delay on the net that connects the specified port instances has a minimum delay of 6, a typical delay of 10, and a maximum delay of 14. You can enter min:typ:max delays for delay values in other kinds of delays such as IOPATH or PORT.

Include the `+mindelays` compile-time option to specify using the minimum delay of the min:typ:max delay value triplet either in delay specification or in the delay value in entries in an SDF file.

Include the `+maxdelays` compile-time option to specify using the maximum delay.

By default VCS uses the typical delays. You can specify using the typical delays with the `+typdelays` compile-time option.

In the case of SDF files, the *`mtm_spec`* argument to the `$sdf_annotate` system task overrides the `+mindelays`, `+typdelays`, or `+maxdelays` options.

Specifying Min:Typ:Max Delays at Runtime

If you have either of the following:

- An SDF file to backannotate delays to your design when simulation starts and the SDF file contains delay values that are min:typ:max delay value triplets
- Module path delays or timing check delays in your Verilog source code are min:typ:max delay value triplets

There is a method that enables you to specify using minimum, typical, or maximum delays at runtime instead of at compile-time.

This method is to use the `+allmtm` compile-time option and use the `+mindelays`, `+typdelays`, and `+maxdelays` options at runtime instead of at compile-time.

Using the `+allmtm` compile-time option tells VCS to write three different compiled SDF files when VCS compiles the ASCII text SDF file. One has the minimum delays from the min:typ:max delay value triplets. Another has the typical delays. The third has the maximum delays. You specify which one to back annotate delays from at runtime with the `+mindelays`, `+typdelays`, or `+maxdelays` runtime options.

Using the `+allmtm` compile-time option also tells VCS to prepare the executable so that you can use the `+mindelays`, `+typdelays`, or `+maxdelays` options at runtime to specify using the minimum, typical, or maximum delay values in the min:typ:max delay value triplets in the module path and timing check delays in your source code.

When you use the `+allmtm` compile-time option, the `+typdelays` option is the default at runtime.

This method does not apply to other delay specifications in your source code that might contain min:typ:max delay value triplets, such as gate delays or delays in continuous assignments.

Using The Configuration File To Disable Timing

You can use the VCS configuration file to disable module path delays, specify blocks, and timing checks for module instances you specify or all instances of module definitions you specify. You use the instance, module, and tree statements to do this just like you do for applying RadiantTechnology, See “The Configuration File Syntax” on page 6-4. The attributes for timing are as follows:

`noIopath`
: Attribute keyword that specifies disabling the module path delays in the specified module instances.

`noSpecify`
: Attribute keyword that specifies disabling the specify blocks in the specified module instances.

`noTiming`
: Attribute keyword that specifies disabling the timing checks in the specified module instances.

This feature does not work with negative timing checks so omit the `+neg_tchk` compile-time option.

Using The Timopt Timing Optimizer

The Timopt timing optimizer can yield large speedups for full-timing gate-level designs. Timopt makes its optimizations based on the clock signals and sequential devices that it identifies in the design. Timopt is particularly useful when you use SDF files because SDF files can't be used with RadiantTechnology (`+rad`).

You enable Timopt with the `+timopt+`*`clock_period`* compile-time option, where the argument is the shortest clock period (or clock cycle) of the clock signals in your design, for example:

```
+timopt+100ns
```

This options specifies that the shortest clock period is 100ns.

Timopt first displays the number of sequential devices that it finds in the design and the number of these sequential devices to which it might be able to apply optimizations, for example:

```
Total Sequential Elements : 2001
Total Sequential Elements 2001, Optimizable 2001
```

Timopt then tells you the percentage of identified sequential devices that it actually can apply optimizations to followed by messages about the optimization process.

```
TIMOPT optimized 75 percent of the design
Starting TIMOPT Delay optimizations
Done TIMOPT Delay Optimizations
DONE TIMOPT
```

The next step is to simulate the design and see if the optimizations applied by Timopt produce a satisfactory increase in performance. If you are not satisfied there are additional steps that you can take to get more optimizations from Timopt.

If Timopt was able to identify all the clock signals and all the sequential devices with an absolute certainty it simply applies its optimizations. If Timopt is uncertain about a number of clock signals and sequential devices then Timopt and you follow the following process to maximize the Timopt optimizations:

1. Timopt writes a configuration file named timopt.cfg in the current directory that lists the signals and sequential devices it's not sure of.

2. You review and edit this file, validating that the signals in the file are or are not clock signals and that the module definitions in it are or are not sequential devices. If you don't need to make any changes in the file, go to step 5. If you do make changes, go to step 3.

3. Compile your design again with the `+timopt+`*`clock_period`* compile-time option.

   Timopt will make the additional optimizations that it did not make because it was unsure of the signals and sequential devices in the timopt.cfg file that it wrote during the first compilation.

4. Look at the timopt.cfg file again:

   If Timopt wrote no new entries for potential clock signals or sequential devices, got to step 5.

   If Timopt wrote new entries but you make no changes to the new entries, got to step 5.

If you make modifications to the new entries, return to step 3.

5. Timopt does not need to look for any more clock signals and it can assume that the timopt.cfg file correctly specifies clock signal and sequential devices. Now it just needs to apply the latest optimizations.
   Compile your design one more time including the `+timopt` compile-time option but without its *`+clock_period`* argument.

6. You now simulate your design using Timopt optimizations. Timopt monitors the simulation. Timopt makes its optimizations based on its analysis of the design and information in the timopt.cfg file. If during simulation it finds that its assumptions are incorrect, for example the clock period for a clock signal is incorrect or there in fact is a port for asynchronous control on a module for a sequential device, Timopt displays a warning message like the following:

```
+ Timopt Warning: for clock testbench.clockgen..clk:
TimePeriod 50ns    Expected 100ns
```

Editing The timopt.cfg File

When editing the timopt.cfg file, first edit the potential sequential device entries. Edit the potential clock signal only when you have made no changes to the entries for sequential devices.

Editing Potential Sequential Device Entries

The following is an example of sequential devices that Timopt was not sure of:

```
// POTENTIAL SEQUENTIAL CELLS
// flop {jknpn} {,};
// flop {jknpc} {,};
// flop {tfnpc} {,};
```

You uncomment the module definitions that in fact model sequential devices and 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, the keyword `true` indicates that the sequential device is triggered on the posedge of the clock port and CLN is an asynchronous port.

If you uncomment any of these module definitions, then Timopt might identify additional clock signals that drive these sequential devices. To enable Timopt to do this:

1. Remove the clock signal entries from the timopt.cfg file
2. Recompile the design with the same `+timopt+`*`clock_period`* compile-time option.

Timopt will write new clock signal entries in the timopt.cfg file.

Editing Clock Signal Entries

The following is an example of the clock signal entries:

```
clock {
    // test.badClock , // 1
    test.goodClock // 2000
} {100ns};
```

The these clock signals have a period of 100 ns or longer. This time value comes from the `+clock_period` argument you added to the `+timopt` compile-time option when you first compiled the design. The entry for the signal test.badClock is commented out because it connects to a small percentage of the sequential devices in the design, in this case only 1 of the 2001 sequential devices that it identified in the design. The entry for the signal test.goodClock is not commented out because it connects to a large percentage of the sequential devices, in this case 2000 of the 2001 sequential devices in the design.

If a commented out clock signal is in fact a clock signal that you want Timopt to use when it optimizes the design in a subsequent compilation, then remove the comment characters from in front of the signal's hierarchical name.

9

Negative Timing Checks

Negative timing checks are `$setuphold` timing checks with negative setup or hold limits or `$recrem` timing checks with negative recovery or removal limits. Their purpose, how they work, and how to use them are described in the following sections:

- The Need For Negative Value Timing Checks
- The $setuphold Timing Check Extended Syntax
- The $recrem Timing Check Syntax
- Enabling Negative Timing Checks
- Checking Conditions
- Toggling The Notifier Register
- SDF Backannotating to Negative Timing Checks
- How VCS Calculates Delays

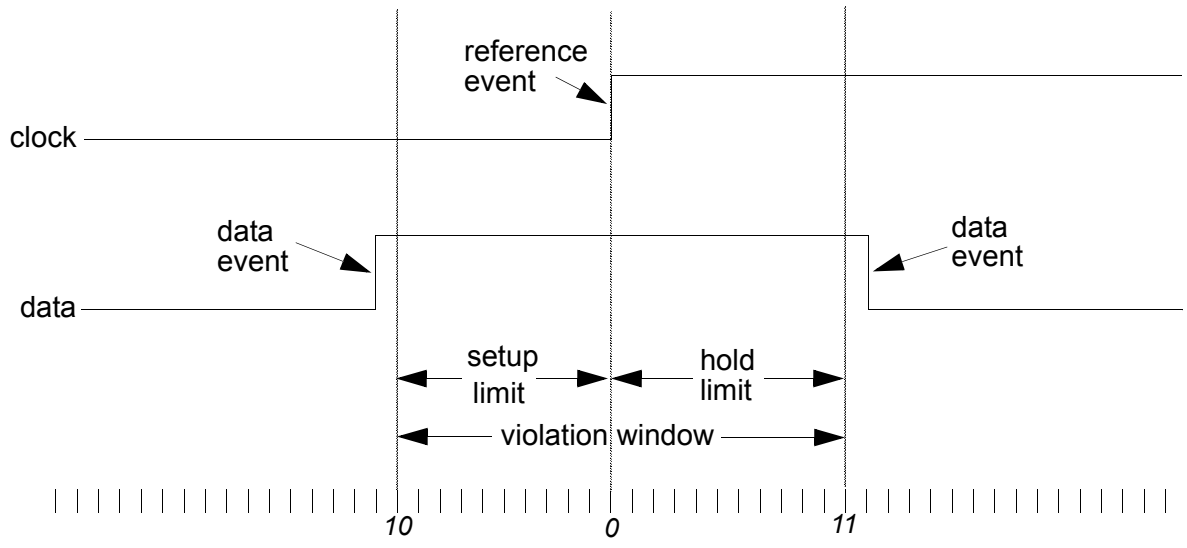
The Need For Negative Value Timing Checks

The `$setuphold` timing check defines a timing violation window of a specified amount of simulation time before and after a reference event, such as a transition on some other signal like a clock signal, in which a data signal must remain constant. A transition on the data signal, called a data event, during the specified window is a timing violation. For example:

```
$setuphold (posedge clock, data, 10, 11, notifyreg);
```

Here VCS will report 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 notifier register, in this example reg notifyreg. You could use this toggling of a notifier register to output an X value from a device, such as a sequential flop, when there is a timing violation.

Figure 9-1 Positive Setup and Hold Limits

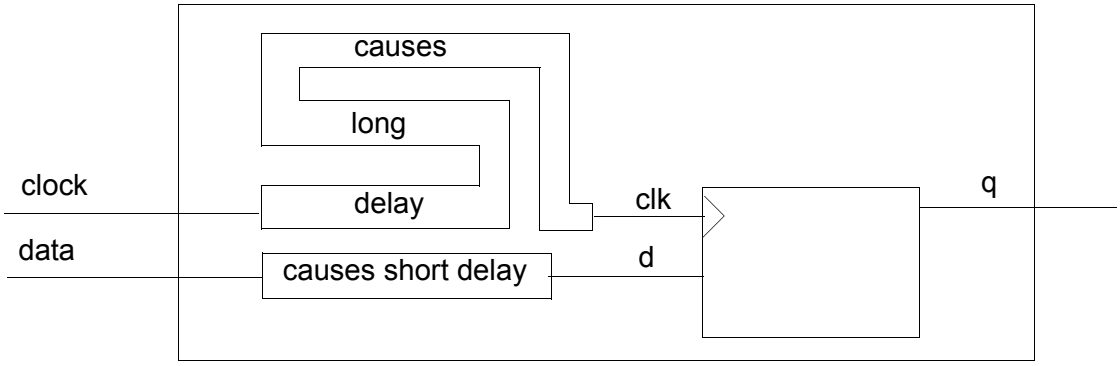


Here both the setup and hold limits have positive values. When this happens the violation window straddles the reference event.

Now 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, where timing must be accurate at this sequential device, and where you need to check for timing violations at the cell boundary. It also occurs when the opposite is true, when the reference event takes longer than the data event to propagate to that sequential device.

When this happens, you use the `$setuphold` timing check in the top-level module of the cell to look for timing violations when signal values propagate to that sequential device, and you need to use negative setup or hold limits in the `$setuphold` timing check.

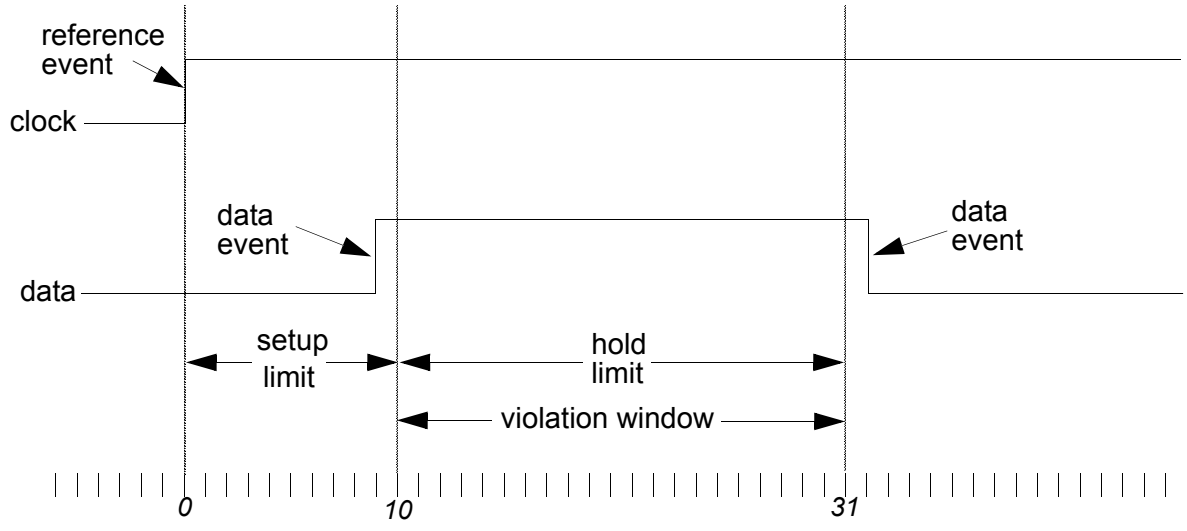
Figure 9-2 ASIC Cell with Long Propagation Delays on Reference Events



When this happens the violation window shifts at the cell boundary so that it no longer straddles the reference event. It shifts to the right when there are longer propagation delays on the reference event. This right shift requires a negative setup limit:

```
$setuphold (posedge clock, data, -10, 31, notifyreg);
```

Figure 9-3 Negative Setup Limit

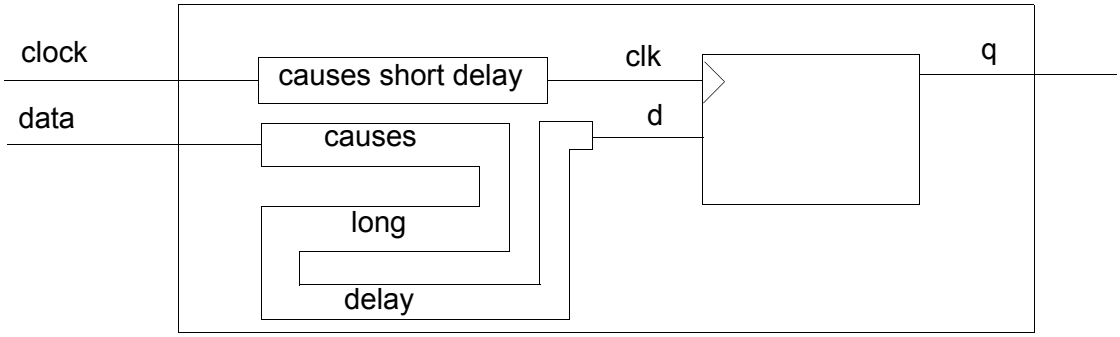


Here the `$setuphold` timing check is in the specify block of the top-level module of the cell. It specifies that there is a timing violation if there is a data event between 10 and 31 time units after the reference event on the cell boundary.

This is giving the reference event a “head start” at the cell boundary, anticipating that the delays on the reference event will allow the data events to “catch up” at the sequential device inside the cell.

Note: When you specify a negative setup limit, its value must be less than the hold limit.

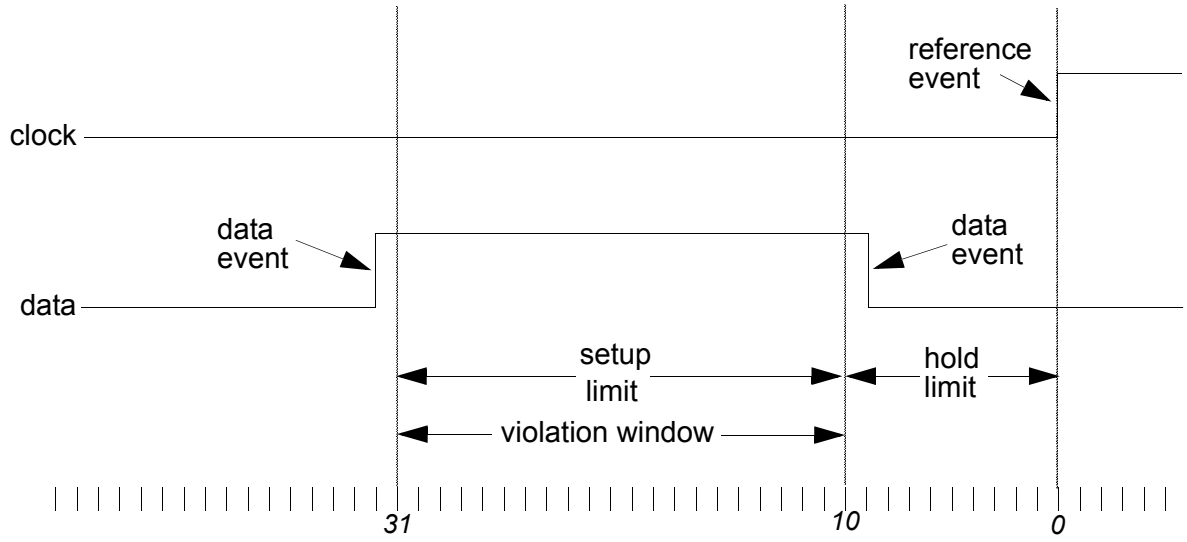
Figure 9-4 ASIC Cell with Long Propagation Delays on Data Events



The violation window shifts to the left when there are longer propagation delays on the data event. This left shift requires a negative hold limit:

```
$setuphold (posedge clock, data, 31, -10, notifyreg);
```

Figure 9-5 Negative Hold Limit



Here the `$setuphold` timing check is also in the specify block of the top-level module of the cell. It specifies that there is a timing violation if there is a data event between 31 and 10 time units before the reference event on the cell boundary.

This is giving the data events a “head start” at the cell boundary, anticipating that the delays on the data events will allow the reference event to “catch up” at the sequential device inside the cell.

Note: When you specify a negative hold limit, its value must be less that the setup limit.

To implement negative timing checks VCS creates delayed versions of the signals that carry the reference and data events and an alternative violation window where the window straddles the delayed reference event.

You can specify the names of the delayed versions using the extended syntax of the `$setuphold` system task or let VCS name them internally.

The extended syntax also allows you to specify expressions for additional conditions that must be true for a timing violation to occur.

Negative Timing Checks for Asynchronous Controls

The `$recrem` timing check is for checking how close asynchronous control signal transitions are to clock signals. Like the `$setuphold` timing checks setup and hold limits, the `$recrem` timing check has recovery and removal limits. The recovery limit specifies how much time must elapse after a control signal, like a clear 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.

Similarly to how a reference signal like a clock signal and data signal can have different propagation delays from the cell boundary to a sequential device inside the cell, so can there be different propagation delays between the clock signal and the control signal. For this reason there can also be negative recovery and removal limits in the `$recrem` timing check.

The $setuphold Timing Check Extended Syntax

The `$setuphold` timing check has an extended syntax that is not part of the IEEE Std 1364-1995 standard but is in the 1364-2001 standard. It has the following syntax:

```
$setuphold(reference_event, data_event, setup_limit,
hold_limit, notifier, timestamp_cond, timecheck_cond,
delayed_reference_signal, delayed_data_signal);
```

The additional arguments are optional.

timestamp_cond
: In the setup phase of a `$setuphold` timing check, VCS records or “stamps” the time of a data event internally so that when a reference event occurs it can compare the times of these events to see if there is a setup timing violation. If the condition specified by this argument is false, VCS does not record or “stamp” the data event so there cannot be a setup timing violation.
Similarly, in the hold phase of a `$setuphold` timing check, VCS records or “stamps” the time of a reference event internally so that when a data event occurs it can compare the times of these events to see if there is a hold timing violation. If the condition specified by this argument is false, VCS does not record or “stamp” the reference event so there cannot be a hold timing violation.

timecheck_cond
: In the setup phase of a `$setuphold` timing check, VCS compares or “checks” the time of the reference event with time of the data event to see if there is a setup timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no setup timing violation.
Similarly, in the hold phase of a `$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 is an example of using the extended syntax:

```
$setuphold(ref,data,-4,10,notifr1,stampreg===1,,d_ref,
           d_data);
```

In this example the *timestamp_cond* argument specifies reg stampreg must equal 1 for VCS to “stamp” or record the times of data events in the setup phase or “stamp” the times of reference events in the hold phase. If this condition is not met, and stamping does not occur, VCS will not find timing violations no matter what the time of 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 delayed signals dClk, dD, dRST, and the notifier reg.

The $recrem Timing Check Syntax

The `$recrem` timing check is not part if the IEEE std 1364-1995 standard and It originated after the need for negative timing checks arose. It is in the IEEE Std 1364-2001 standard. Its syntax is very similar to the extended syntax for `$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 a a clear signal, and the active edge is specified with the `posedge` or `negedge` keyword.

data_event
: Also typically, the data event occurs on a clock signal and the active edge on this signal is also specified with the `posedge` or `negedge` keyword.

recovery_limit
: Specifies how much time must elapse after a control signal, like a clear signal, toggles from its active state, the reference event, before there is an active clock edge, the data event.

removal_limit
: Specifies how much time must elapse after an active clock edge, the data event, before the control signal can toggle from its active state, the reference event.

notifier
: A register whose value VCS toggles when there is a timing violation.

timestamp_cond
: In the recovery phase of a `$recrem` timing check, VCS records or “stamps” the time of a reference event internally so that when a data event occurs it can compare the times of these events to see if there is a recovery timing violation. If the condition specified by this argument is false, VCS does not record or “stamp” the reference event so there cannot be a recovery timing violation. Similarly, in the removal phase of a `$recrem` timing check, VCS records or “stamps” the time of a data event internally so that when a reference event occurs it can compare the times of these events to see if there is a removal timing violation. If the condition specified by this argument is false, VCS does not record or “stamp” the data event so there cannot be a removal timing violation.

timecheck_cond
In the recovery phase of a `$recrem` timing check, VCS compares or “checks” the time of the data event with time of the reference event to see if there is a recovery timing violation. If the condition specified by this argument is false, VCS does not make this comparison and so there is no recovery timing violation. Similarly, in the removal phase of a `$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 including the +notimingcheck or +nospecify compile-time options, VCS does not compile into the simv executable the $setuphold and $recrem timing checks but creates the “delayed” versions of the signals to drive sequential devices in the cell. 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 there is no delay on them and they have the same transition times as the signals specified in the *reference_event* and *data_event* arguments.

Similarly If you include +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 Use 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, however, 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 signal is particularly useful when the other timing checks use a notifier register to change the output of sequential element to X. Consider Example 9-6:

Example 9-6 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
```

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 occur:

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:

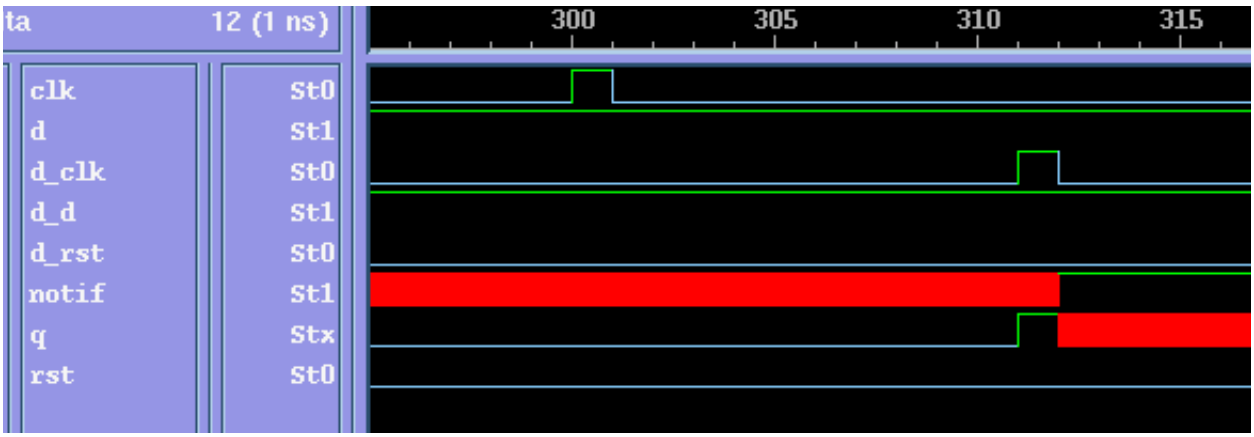
```
"test1.v", 31: Timing violation in top.dff1

$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 9-7 Other Timing Checks Using The Delayed Versions



If you include the `+neg_tchk` compile-time option and also the `+old_ntc` compile time option, the `$width` timing check does not use the delayed version of signal clk and the following sequence of events occur:

1. At time 301 the narrow pulse on signal clk causes a width violation:

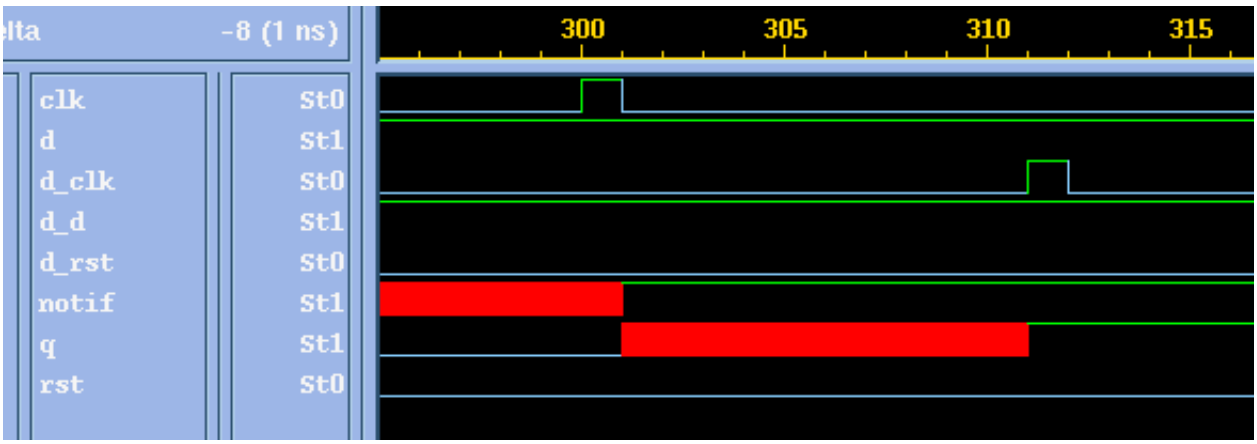
```
"test1.v", 31: Timing violation in top.dff1

$width( posedge clk:300,    : 301, limit: 5);
```

2. Also at time 301 the notifier reg named notif toggles from X to 1. This in turn changes the output q of the user-defined primitive DFF_UDP and module instance dff1 from 0 to X.

3. At time 311 the delayed version of signal clk, named d_clk, reaches the user-defined primitive DFF_UDP changing the output q to 1 erasing the X value on this output.

Figure 9-8 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 therefore we do not recommend using the `+old_ntc` compile-time option. We provided it only for unforeseen circumstances.

Checking Conditions

When should VCS evaluate the expressions in the *timestamp_cond* and *timecheck_cond* arguments? Should VCS evaluate these expressions when there is a value change on the original reference and data signals at the cell boundary or should it wait and evaluate these expressions when the value changes propagate from the delayed versions of these signals at the sequential device inside the cell? It depends on what signals are the operands in these expressions. Note the following:

- If the operands in these expressions are neither the original or delayed versions of the reference or data signals, and if these operands are signals that do not change value between value changes on the original reference and data signals and their delayed versions, then it does not matter when VCS evaluates these expressions.
- If the operands in these expressions are delayed versions of the original reference and data signals, then you want VCS to evaluate these expressions when there are value changes on the delayed versions of the reference and data signals. VCS does this by default.
- If the operands in these expressions are the original reference and data signals and not the delayed versions, then you want VCS to evaluate these expressions when there are value changes on the original reference and data signals.
  To specify evaluating these expressions when the original reference and data signals change value, include the `+NTC2` compile-time option.

Toggling The Notifier Register

VCS waits for the 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 Backannotating to Negative Timing Checks

You can backannotate negative setup and hold limits from SDF files to `$setuphold` timing checks and negative recovery and removal limits from SDF files to `$recrem` timing checks, if the following conditions are met:

- You included the arguments for the names of the delayed reference and data signals in the timing checks.
- You compiled your design with the `+neg_tchk` compile-time option.
- For all `$setuphold` timing checks the positive setup or hold limit must be greater than the negative setup or hold limit.
- For all `$recrem` timing checks the positive recovery or removal limit must be 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, instead 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, if you enter the following:

```
$setuphold(posedge clock,data,-10,20, , , , del_clock,
           del_data);
```

You are specifying 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 above.

Multiple timing checks, that share reference or data events and specified delayed signal names, can define a set of delay relationships. For example:

```
$setuphold(posedge CP,D,-10,20, notifier, , ,
           del_CP, del_D);
$setuphold(posedge CP,TI,20,-10, notifier, , ,
           del_CP, del_TI);
$setuphold(posedge CP,TE,-4,8, notifier, , ,
           del_CP, del_TE);
```

Here:

- The first `$setuphold` timing check specifies the delay on del_CP is more than 10 but less than 20 time units more than the delay on del_D.
- The second `$setuphold` timing check specifies the delay on del_TI is more than 10 but less than 20 time units more than the delay on del_CP.
- The third `$setuphold` timing check specifies the delay on del_CP is more than 4 but less than 8 time units more than the delay on del_TE

Therefore:

- The delay on del_D is 0 because its delay does not have to be more than any other delayed signal.
- The delay on del_CP is 11 because it must be more than 10 time units more than the 0 delay on del_D
- The delay on del_TE is 4 because if the delay on del_CP is 11. That 11 makes the possible delay on del_TE is larger than 3 but less that 7. A delay of 3 or less cannot be 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 these signals are no longer mutually exclusive:

1. Sets one negative limit to 0
2. recalculates the delays of the delayed signals

Using Multiple Non-Overlapping Violation Windows

You include the `+overlap` compile-time option to enable accurate simulation of multiple violation windows for the same two signals when the following conditions occur:

- The violation windows are specified with negative delay values that are back annotated from an SDF file.
- The violation windows do not converge or overlap

The default behavior of VCS when these conditions are met, because it is the customary behavior of Verilog simulators, is to replace the negative delay values with zeros so that the violation windows overlap. Consider the following code example:

```
‘timescale 1ns/1ns
module top;
reg in1, clk;
wire out1;

FD1  fd1_1 ( .d(in1), .cp(clk), .q(out1) );

initial
begin
   $sdf_annotate("overlap1.sdf");
in1 = 0;
   #45 in1=1;
end

initial
begin
   clk=0;
   #50 clk = 1;
   #50 clk = 0;
end
endmodule

module FD1 (d, cp, q);
input d, cp;
output q;
wire q;
reg notifier;
reg q_reg;

always @(posedge cp)
```

```
q_reg = d;

assign q = q_reg;

specify
   $setuphold( posedge cp, negedge d, 40, 30, notifier);
   $setuphold( posedge cp, posedge d, 20, 10, notifier);
endspecify
endmodule
```

THe SDF file contains the following to back annotate negative delay values:

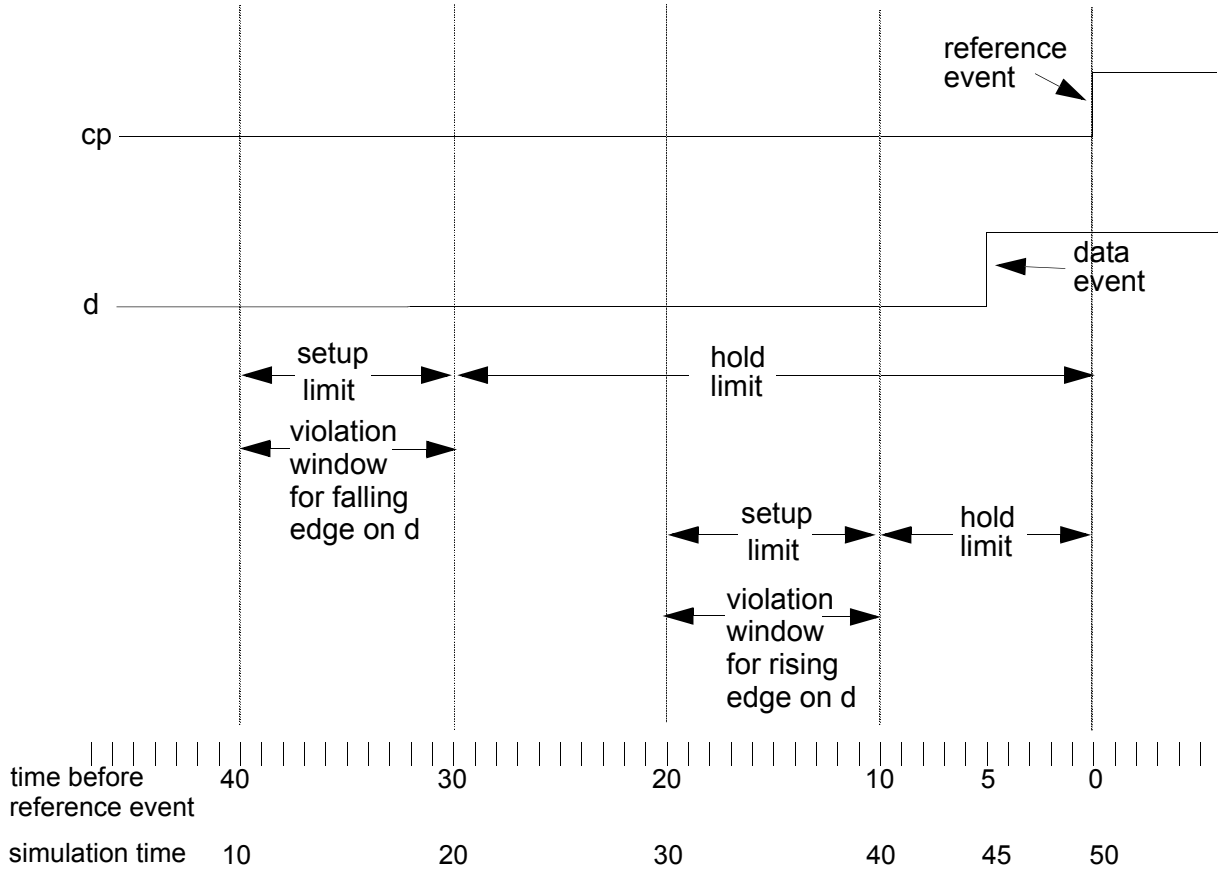
```
(CELL
  (CELLTYPE "FD1")
  (INSTANCE top.fd1_1)
  (TIMINGCHECK
     (SETUPHOLD (negedge d) (posedge cp) (40) (-30))
     (SETUPHOLD (posedge d) (posedge cp) (20) (-10))
  )
)
```

So the timing checks are now:

```
   $setuphold( posedge cp, negedge d, 40, -30, notifier);
   $setuphold( posedge cp, posedge d, 20, -10, notifier);
```

The violation windows and the transitions that occur on signals top.fd1_1.cp and top.fd1_1.d are shown in Figure 9-9.

Figure 9-9 Non-Overlapping Violation Windows



The `$setuphold` timing checks now specify:

- A violation window for a falling edge on signal d between 40 and 30 time units before a rising edge on signal cp
- A violation window for a rising edge on signal d between 20 and 10 time units before a rising edge on signal cp

The test bench module top applies stimulus so that the following transitions occur:

1. A rising edge on signal d at time 45
2. A rising edge on signal cp at time 50

The rising edge on signal d at time 45 is not inside the violation window for a rising edge on signal d. If you include the `+overlap` compile-time option you will not see a timing violation. This behavior is what you want because there is no transition in the violation windows so VCS should not display a timing violation.

The `+overlap` option tells VCS not to change the violation windows, just like it would if the windows overlapped.

If you omit the `+overlap` option VCS does what Verilog simulators traditionally do, which is both pessimistic and inaccurate:

1. During compilation VCS replaces the -30 and -10 negative delay values is the `$setuphold` timing checks with 0 values. It displays the following warning:

```
Warning: Negative Timing Check delays did not converge,
Setting minimum constraint to zero and using approximation
solution (
"sourcefile",line_number_of__second_timing_check)
```

VCS alters the violation windows:

- For the falling edge the window starts 40 time units before the reference event and ends at the reference event.
- For the rising edge the window starts 20 time units before the reference event and also ends at the reference event.

VCS alters the windows so that they overlap or converge.

2. During simulation, at time 50, the reference event, VCS displays the timing violation:

```
"sourcefile.v", line_number_of_second_timing_check: Timing
```

```
violation in top.fd1_1
   $setuphold( posedge cp:50 posedge d:45, limits (20,0)  );
```

The rising edge on signal d is in the altered violation window for a rising edge on d, that starts 20 time units before the reference event and now ends at the reference event. The rising edge on signal d occurs five time units before the reference event.

10

Using Synopsys Models

The models from Synopsys that you can instantiate in your design include:

- SWIFT VMC models and SmartModels
- hardware models

SmartModels are models from Synopsys that model devices from various vendors. VMC models are secure, protected, and portable models of Verilog designs that contain no Verilog code. SWIFT is the interface for both of these kinds of models.

See “SWIFT VMC Models and SmartModels Introduction” on page 10-2.

You can also instantiate an actual device in your Verilog design using a Synopsys hardware modeler. See “Synopsys Hardware Models Introduction” on page 10-17.

Note:

The information in this chapter is provided as a convenience to the Synopsys model user. It includes basic information about simulating Synopsys models. This chapter is not, however, the authoritative source on this subject. More complete and sometimes more up to date information can be found in the *Simulator Configuration Guide for Synopsys Models*. This guide is available on-line in PDF format at http://www.synopsys.com/products/lm/doc/smartmodel/manuals/simcfg.pdf or http://www.synopsys.com/products/lm/doc/hardwaremodel/manuals/simcfg.pdf

SWIFT VMC Models and SmartModels Introduction

VCS enables you to instantiate both of these kinds of models in your design and simulate these models as part of your design. The steps you take are as follows:

1. Set the SWIFT environment variables
2. Generate a Verilog template file for the model. You use this template file to instantiate the model.
3. Enable the monitoring of signals inside the model through the model window.
4. Enter commands for the model in your source code using the LMTV window commands or the SWIFT command channel.
5. Compile the design with compile-time options for the SWIFT interface and then simulate your design.

SWIFT Environment Variables

You set some environment variables on all platforms. Others you set only on certain platforms.

All Platforms

You must set the LMC_HOME environment variable to the directory where you installed the models, for example:

```
setenv LMC_HOME /u/tools/swiftR41
```

Set the VCS_SWIFT_NOTES environment variable to 1, for example:

```
setenv VCS_SWIFT_NOTES 1
```

Setting this environment variable enables the display of messages from models including messages that tell you if the model is correctly loaded. For example, if PCL code contains printf commands, during the simulation the microprocessor controlled by the PCL code prints out the model messages.

Setting this environment variable is optional but Synopsys recommends that you always set it when you are debugging your design.

Solaris platform

For the Solaris platform, set the environment variable LD_LIBRARY_PATH. If you have already set this environment variable for some other application, you can add to the definition of this environment variable as follows:

```
setenv LD_LIBRARY_PATH ${LMC_HOME}/lib/sun4Solaris.lib:
${LD_LIBRARY_PATH}
```

If you haven't already set this environment variable, enter the following:

```
setenv LD_LIBRARY_PATH ${LMC_HOME}/lib/sun4Solaris.lib
```

HP platform

For the HP platform, set the SHLIB_PATH environment variable.If you have already set this environment variable for some other application, you can add to the definition of this environment variable as follows:

```
setenv SHLIB_PATH ${LMC_HOME}/lib/hp700.lib:${SHLIB_PATH}
```

If you haven't already set this environment variable, enter the following

```
setenv SHLIB_PATH ${LMC_HOME}/lib/hp700.lib
```

Linux

```
% setenv LD_LIBRARY_PATH $LMC_HOME/lib/
x86_linux.lib:$LD_LIBRARY_PATH
```

Generate Verilog Templates

You generate the Verilog template for the model using the `-lmc-swift-template` option. The syntax for the `vcs` command line to generate this models is as follows:

```
vcs -lmc-swift-template modelname
```

This command generates a Verilog template file named *modelname*.swift.v. For example, if you enter the following vcs command:

```
vcs -lmc-swift-template xc4062xl_432
```

VCS writes the xc4062xl_432.swift.v file in the current directory.

This Verilog template file contains a Verilog module definition that contains:

- The special `$vcs_swift` user-defined system task for the SWIFT interface that enables you to use the command channel to the SWIFT interface to pass commands to the model and see messages from the model.
- Declarations for window regs that enable you to see the value of, and in some cases deposit values to, signals in the model. See “Monitoring Signals in the Model Window” on page 10-8.
- Declarations for regs that you use to pass commands to the model.
- Port and reg declarations and assignment statements that are part of a Verilog shell for the model.

When you instantiate the module definition in this Verilog template file, you instantiate the model.

Modifying the Verilog Template File

You can make certain modifications to the contents of the Verilog template file. The modifications you can make are as follows:

Reordering ports in the module header

If, for example, the module header is as follows:

```
module xyz (IO0, IO1, IO2, IO3, IO4);
```

You can reorder the ports:

```
module xyz (IO4, IO3, IO2, IO1, IO0);
```

Concatenating ports in the module header

You can concatenate ports in the module header, for example:

```
module xyz ({IO4, IO3, IO2, IO1}, IO0);
```

Doing so enables you to connect vector signals to the model as follows:

```
wire [3:0] bus4;
...
xyz xyz1( bus4, ...
```

Naming concatenation expressions in the module header

In Verilog you can name concatenation expressions in the port connection list in a module header, for example:

```
module xyz (.IO({IO4, IO3, IO2, IO1}), IO0);
```

This allows you to use name based connections in the module instantiation statement, as follows:

```
wire [3:0] bus4;
..
xyz xyz1( sig1, .IO(bus4), ...
```

Redefining Parameters

The Verilog template file contains a number of parameters that are used for specifying model attributes. In some cases you can modify the parameter definition. For example, the template file contains the following parameter definition:

```
 parameter DelayRange = "MAX";
```

This parameter specifies using the maximum delays of min:typ:max delay triplets in the model. You can change the definition to `"TYP"` or `"MIN"`. There is an alternative to editing the `DelayRange` parameter definition, see “Changing The Timing of A Model” on page 10-16.

For another example, the template file for a memory model might contain the following parameter definition:

```
parameter MemoryFile = "memory";
```

If you know that all instances of this model will need to load memory file mem.dat, you can change this to:

```
parameter MemoryFile = "mem.dat";
```

You can also use defparam statements to change these parameter definitions. For example, if an instance of a memory model has an instance name of test.design.mem1 and this model must load memory file mem1.dat, you can enter the following in the test fixture module:

```
defparam design.mem1.MemoryFile = "mem1.dat";
```

For more information on SmartModel attributes that are parameters in the Verilog template file, see the *SmartModel Library Simulator Interface Manual*.

Monitoring Signals in the Model Window

SWIFT VMC models and SmartModels can have a window that enables you to see the values of certain signals inside the model and for some models also deposit values to these signals. The model specific data sheet tells you the signals that you can monitor in the window and whether you can also deposit values to these signals.

When you generate the Verilog template file for the model, VCS declares window regs in this template file that correspond to these signals inside the model window. By monitoring the value of these regs you monitor the value of the corresponding signals in the model and, when possible, assigning values to these regs deposit values to the corresponding signals in the model.

To enable VCS to declare these regs for SmartCircuit models you must do the following:

1. Create the file listing the SmartCircuit windows, See the *SmartModel Library Users Manual* for a description of how this is done.
2. Create a soft link or copy the Model Control File (MCF) to a file named scf in the current directory. VCS uses this file load the netlist for the SmartCircuit model that contains the signals in the window.

The following are examples of these regs in the Verilog template file:

```
//Generating SmartModel Windows data
  reg [1:0] GEN_CCLK_SPEED;
  reg [4:0] LENGTH_CNT_WIDTH;
  reg [10:0] FRAME_SIZE;
  reg [12:0] DEVICE_FRAMES;
  reg [3:0] CRC_ERROR_CHK;
  reg SYNC_TO_DONE;
```

```
reg [3:0] DONE_ACTIVE;
reg [3:0] IO_ACTIVE;
reg [3:0] DEVICE_STATE;
reg CONFIGURATIONMODE;
```

You enable and disable the monitoring of these window regs with the special `$swift_window_monitor_on` and `$swift_window_monitor_off` system tasks. The syntax for these system tasks are as follows:

```
$swift_window_monitor_on("instance_name"[,"window_reg",
"window_reg",...]);
$swift_window_monitor_off("instance_name"[,"window_reg","w
indow_reg",...]);
```

Where:

instance_name	Specifies the hierarchical name of the instance of the module definition in the Verilog template file.
window_reg	The identifier of the window reg in the Verilog template file. If you do not specify a window reg in these system tasks, you enable or disable the monitoring of all the window regs in the instance.

The following are examples of using these system tasks:

```
$swift_window_monitor_on("test.design.xc4062xl_432_1");
```

This example enables the monitoring of all window regs in the module instance for the model with the hierarchical name test.design.xc4062xl_432_1.

```
$swift_window_monitor_on("test.design.xc4062xl_432_1",
"GEN_CCLK_SPEED","LENGTH_CNT_WIDTH");
```

This example enables the monitoring of the window regs GEN_CCLK_SPEED and LENGTH_CNT_WIDTH in the module instance test.design.xc4062xl_432_1.

After you enable the monitoring or depositing with these system tasks you must specify the monitoring with, for example, the `$monitor` system task and deposit values with procedural assignments to these window regs.

Using LMTV SmartModel Window Commands

VCS has implemented a number of the LMTV SmartModel window commands that were implemented as a command interface between SmartModels and the previous generation of Verilog simulators, Verilog-XL. These commands are user-defined system tasks that we provide for simulating with SmartModels. They also work with VMC models.

These system tasks are as follows:

`$lm_monitor_enable`
: Enables SmartModel Windows for one or more window elements in a specified model instance. This system task is functionally equivalent to the $swift_window_monitor_on system task described in “Monitoring Signals in the Model Window” on page 10-8. Its syntax is as follows:
`$lm_monitor_enable(`*`regname`*`,`*`instance_name`*`, "`*`window_element`*`")`

`$lm_monitor_disable`
: Disables SmartModel Windows for one or more window elements in a specified model instance. This system task is functionally equivalent to the $swift_window_monitor_off system task described in “Monitoring Signals in the Model Window” on page 10-8. Its syntax is as follows:
`$lm_monitor_disable(`*`regname`*`,`*`instance_name`*`, "`*`window_element`*`")`

`$lm_command`

Sends a command to the session or to a model instance. You use this system task to pass Model and Session commands to a SmartModel, SmartBrowser commands to a SmartCircuit model, and PCL commands to a microprocessor model. Its syntax is as follows:

```
$lm_command ("session_cmmd_string" |
"instance_name","model_cmmd_string");
```

`$lm_dump_file`

Dumps the memory contents of the specified instance into the specified file. This works only for memory models. Overwrites the specified file if it already exists. Using this system task eliminates the read cycles required to verify the success of a test. Its syntax is as follows:

```
$lm_dump_file("instance_name","filename"
[,"MEMORY"]);
```

`$lm_load_file`

Loads the memory contents of a file into an instance (the contents of this file are in a specific memory dump format). The instance can be either a programmable device or a memory model. Using this system task eliminates the write cycles required to set up the contents of the model.

You can also use this system task to load a model control file (MCF) during simulation.

Its syntax is as follows:

```
$lm_load_file("instance_name","filename"
[,"MEMORY|JEDEC|PCL|SCF|MCF"]);
```

Model, Session, PCL, and SmartBrowser commands are described in the *SmartModel Library User's Manual*.

Note:

The *instance_name* argument in the `$lm_monitor_enable` and `$lm_monitor_disable` system tasks is not enclosed in quotation marks; whereas it is in the `$lm_command`, `$lm_dump_file`, and `$lm_load_file` system tasks.

For complete and authoritative information on these LMTV SmartModel window commands, see the *SmartModel Library Simulator Interface Manual.* The information on these commands is in the interface information for Verilog-XL but this information is also good for VCS. We implemented these system tasks in VCS for people moving from Verilog-XL to VCS.

You can access these documents at http://www.synopsys.com/products/lm/docs.

Note:

VCS does not support two-dimensional windows for memory models and therefore has no implemented other LMTV window commands.

Entering Commands Using The SWIFT Command Channel

As an alternative to using the LMTV window commands, you can use the SWIFT command channel to pass commands to a SmartModel or a VMC model.

The command channel works by assigning these commands and toggling the values of command channel regs declared in the Verilog template file. The regs to which you assign these values are as follows:

`cmd$str`	The reg to which you assign the command
`do$model$cmd`	When you assign a value of 1 to this reg, the model executes all Model, SmartBrowser, and LMTV commands assigned to reg `cmd$str`.
`do$session$cmd`	When you assign a value of 1 to this reg, the model executes all Session commands assigned to reg `cmd$str`.
`log$file`	The reg to which you assign a logfile name. The model writes to this logfile when `log$on` has a value of 1.
`log$on`	When you assign a value of 1 to this reg you enable the model writing to the logfile you assigned to reg `log$file`.

The following is an example of an initial block that passes commands through the command channel:

```
initial
begin
#1 circuit.model.cmd$str = "show doc";
circuit.model.do$model$cmd=1  ; // 1
#1 circuit.model.do$model$cmd=0  ;
#1 circuit.model.cmd$str = "show timing unit";
circuit.model.do$model$cmd = 1;
#1 circuit.model.do$model$cmd=0  ;
#1 circuit.model.cmd$str = "show version";
```

```
circuit.model.do$model$cmd = 1;
#1;
end
```

The Verilog template files for SmartModel memory models also contain the following reg declarations that allow you to write or dump the contents of a memory to a file:

`mem$dump$file`	The reg to which you assign the name of the file into which the memory model writes its contents. The model writes to this file when `do$mem$dump` has a value of 1.
`do$mem$dump`	Enables writing the memory models contents to the file.

For example, if you had a memory model with the hierarchical name top.asic.mem1 and you wanted to dump its contents to file mem1.1k.dump at time 1000, and mem1.2k.dump at time 2000, you could use the following initial block in your test fixture module:

```
initial
begin
#1000 top.asic.mem1.mem$dump$file = "mem1.1k.dump";
top.asic.mem1.do$mem$dump = 1;
#1000 top.asic.mem1.mem$dump$file = "mem1.2k.dump";
top.asic.mem1.do$mem$dump = 1;
end
```

Using the CLI to Access The Command Channel

You can also use the CLI to access the command channel during simulation, for example:

```
cli_0> set circuit.model.cmd$str = "show doc";
cli_1> once #1;
cli_2> .
cli_3> set circuit.model.do$model$cmd=1;
cli_4> once #1;
cli_5> .
```

Loading Memories at The Start of Runtime

SmartModel memory models have an attribute called `MemoryFile`. You can use this attribute to load the contents of a file into these memories ar runtime. To do so use a `defparam` statement, for example:

```
defparam
test.designinst.modelinst.MemoryFile="mem_vec_file";
```

Here the attribute is treated as if it were a parameter in an instance.

This method enables you to load a memory when simulation starts. To load a memory after simulation starts use the LMTV window command `$lm_load_file` system task. see “Using LMTV SmartModel Window Commands” on page 10-10

Compiling and Simulating a Model

If your design instantiates a SmartModel or a VMC model, you compile your design with the `-lmc-swift` compile-time option. Be sure to also include the Verilog template file on the `vcs` command line, for example:

```
vcs -lmc-swift xc4062xl_432.swift.v test.v design.v
```

This command line results in an executable file named simv. Enter this executable file on a command line to simulate the design that instantiates the model:

```
simv
```

Changing The Timing of A Model

You can enter the `+override_model_delays` runtime option in combination with either the `+mindelays`, `+typdelays`, or `+maxdelays` option to override the `DelayRange` parameter in the template file that specifies the timing used by the model.

If you use this method, all the SmartModel models in you design will use either the minimum, typical, or maximum delays specified by the `+mindelays`, `+typdelays`, or `+maxdelays` option.

If you want to mix the timing used by different models in your design you must instead edit the template files for each model to change the `DelayRange` parameter definition to either `"MIN"`, `"TYP"`, or `"MAX"`.

Synopsys Hardware Models Introduction

Information for the Synopsys Hardware Models Interface is provided in following sections:

- Required Environment Variables
- Using lmvc_template
- Required VCS Command Line Options

Required Environment Variables

The following variables are required to run the LMC Hardware modeler with VCS:

- `LM_DIR`
- `LM_LIB`
- `VCS_HOME`

The variable `LM_DIR` points to LMC SFI (Interface Software) installed area. Specifically it points to LMSI installed `area/sms/lm_dir`. For example if LMSI installed at:

```
/Net/sparrow/mnt/d3/third_party/lm_RMS3.3a/
```

set `LM_DIR` to:

```
/Net/sparrow/mnt/d3/third_party/lm_RMS3.3a/sms/lm_dir
```

The variable `LM_LIB` points to LMC Model Shell Software area. This directory contains different model subdirectories. Some of the model examples are I80287, I80286, I80386 ... etc ..

Each model shell subdirectory contains unique configuration files required to run the LMC Hardware Modeler Interface. These include vector files required to debug hardware the model itself. The *Model_name*.MDL file is the main model file that includes the required configuration file.

LM_LIB can point to multiple subdirectories. Add a ':' to separate different paths. For example:

```
setenv LM_LIB /usr/tools/models:/user/galaxy/fiaz/models
```

The variable VCS_HOME points to VCS installation area.

Note:

In some cases, the model directory does not contain all the files included in *Model_name*.MDL. For example, adapter mapping(.ADP) and package mapping(.PKG) files. These files are located at $LM_DIR/../maps. It is recommended that this directory also be added to LM_LIB.

Using lmvc_template

Utility lmvc_template can be found at *vcs_install_dir*/*platform*/lmc/hm

This utility creates the Verilog-HDL template necessary to incorporate the hardware model into a VCS simulation. The file created will be *model*.lmvc.v

Usage:

```
lmvc_template model-mdl-file
```

This program will look for *model-mdl-file* in the directories indicated by the LM_LIB environment variable.

The port list of the generated template may be altered by the user to match existing instantiations of the model. Port orders may be rearranged and ports may be concatenated together and explicitly named. The name should match vectors on the port instances and allow for connection by name. For example:

```
module mymod (abc, d0, d1, d2, d3, xyz);
```

and you wish to instantiate this module elsewhere as follows:

```
mymod mymod1 (.abc(ABC), xyz(XYZ), .dbus(DBUS));
mymod mymod1 (ABC, DBUS, XYZ);
```

You can adjust the generated port list to be:

```
module mymod (abc, .dbus({d1, d2, d3, d4}), xyz);
```

Note that *only* the port list identified:

```
module mymodel( <port-list> );
```

can be altered. The port and variable definitions and system task must remain as generated.

Required VCS Command Line Options

Compile your description, including the hardware model template and supporting PLI and library files. Usage:

```
vcs a.v b.v ... model.lmvc.v -P lmvc.tab lmvc.o lm_sfi.a
```

Compile the Verilog description with the model template.

Files `lmvc.tab` and `lmvc.o` set up the `$lm*` task in *`model`*`.lmvc.v`. File `lm_sfi.a` contains the calls that provide connection to the hardware modeler.

Files `lmvc.tab` and `lmvc.o` are located at `$VCS_HOME/`*`platform`*`/lmc/hm` and `lm_sfi.a` is located at `$LM_DIR/../lib/`*`platform`*.

Note: The `-lmc-hm` compile-time option automatically includes the command line entries for `lmvc.tab`, `lmvc.o`, and `lm_sfi.a`.

Run Simulation As Normal

You will need an additional passcode to use this interface. Contact Synopsys VCS simulation support at 1-800-VERILOG.

If you wish to turn on test vector logging or timing measurement, you can invoke tasks:

```
'lm_log', 'lm_log_off', 'lm_measure_time'
```

or

```
'lm_measure_time_off'
```

which are contained in the generated template:

```
initial begin
// request hardware modeler to measure time
// (by default delays from a delay file are used)
top.mymod1.lm_measure_time;
// set modeler to log test vectors from 1000 to 2000
#1000 top.mymod1.lm_log("logfilename");
end
```

Changing The Timing of A Model

You can enter the `+override_model_delays` runtime option in combination with either the `+mindelays`, `+typdelays`, or `+maxdelays` option to override the `DelayRange` parameter in the template file that specifies the timing used by the model.

If you use this method, all the hardware models in you design will use either the minimum, typical, or maximum delays specified by the `+mindelays`, `+typdelays`, or `+maxdelays` option.

If you want to mix the timing used by different models in your design you must instead edit the template files for each model to change the `DelayRange` parameter definition to either `"MIN"`, `"TYP"`, or `"MAX"`.

11

Discovery Visual Environment

Discovery Visual Environment (DVE) software is a graphical verification environment that supports debugging for VCS and VCS MX simulations. This release of DVE allows you to work in post-processing and interactive mode.

For information on using DVE, see the *Discovery Visual Environment User Guide*.

Using DVE, you can perform the following tasks:

- Hierarchically expand and view HDL scope information.
- Debug assertions.
- View log, history, and error/warning information related to your analysis.

- Perform TCL tasks by executing scripts from the DVE graphical user interface or executing TCL commands or sourcing scripts through the command-line interface.

Primary DVE Components

This section provides an overview of the following core components of DVE:

- Top Level Window
- Source Window
- Assertion Window
- Wave Window
- List Window
- Schematic Window

Top Level Window

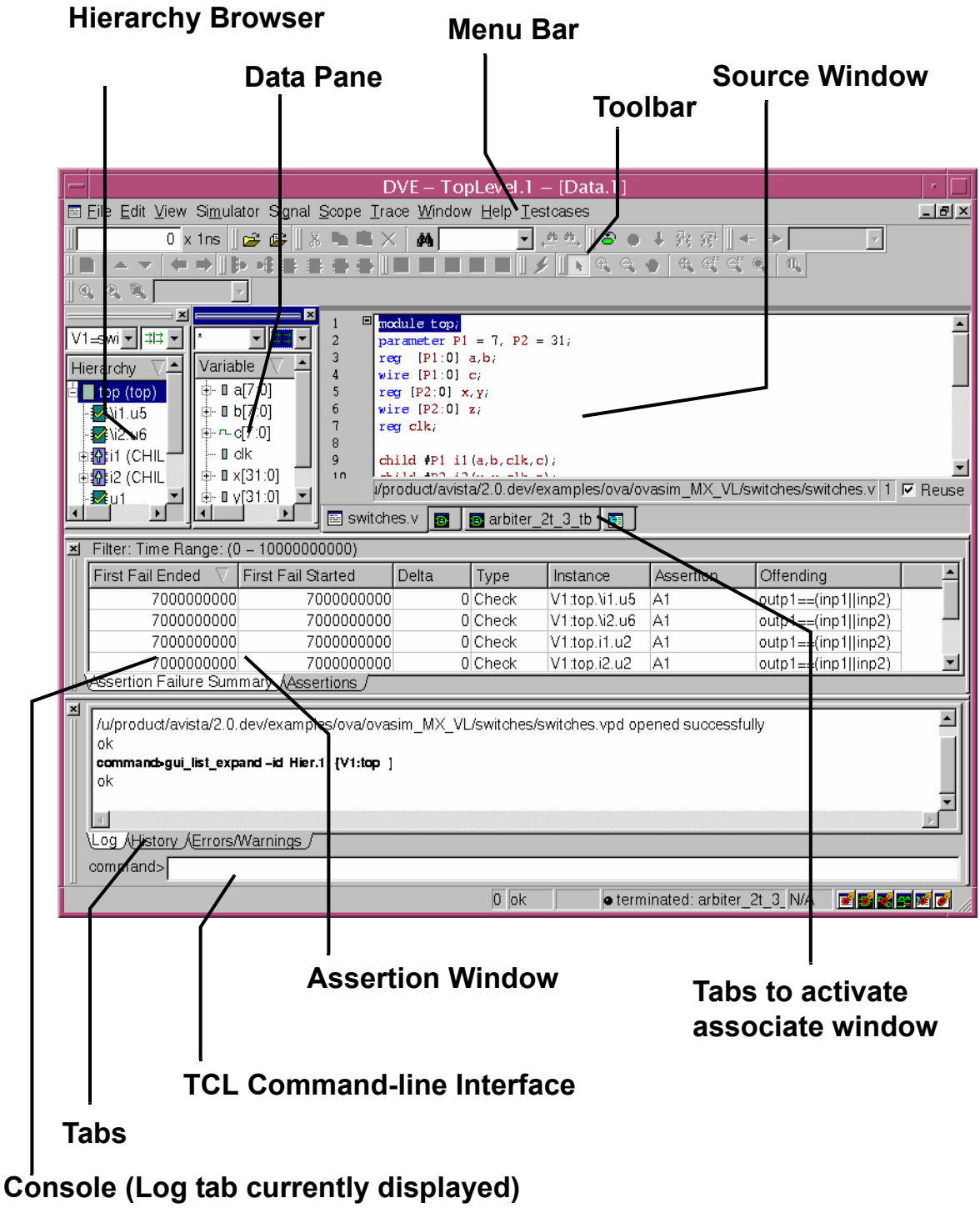
The Top Level Window is a frame that can contain the other windows. A Top Level Window has menus, toolbars and statusbar. From the Top Level Window, you can:

- Use the Menu bar, Toolbar, and Status Bar to run and interact with a simulation or open, view, navigate, edit, and close a design database and set the target for new panes and windows containing source code, waveforms, schematics, and ASCII data. You can specify analysis and search parameters, invoke TCL scripts, run examples, and access Help.

- Use the Hierarchy Browser to view, expand, and select scopes of interest for display in a Waveform Window.
- View and filter signal information. Double click a signal of interest to view source code or drag and drop it into another DVE pane or window. Dragging and dropping is supported in the Data Pane, Source, Wave, List, and Assertion windows.
- View HDL or Assertion source code in the Source Window or in multiple Source Windows.
- View assertion failures and history.
- Perform TCL tasks by executing scripts from the DVE graphical user interface or executing TCL commands or sourcing scripts through the command-line interface.
- Activate any open DVE window or pane regardless of where it is in the desktop window stack.
- Access DVE windows, such as the Waveform Window and List Window, to debug your design.

Figure 11-1 shows an example of the Top Level Window.

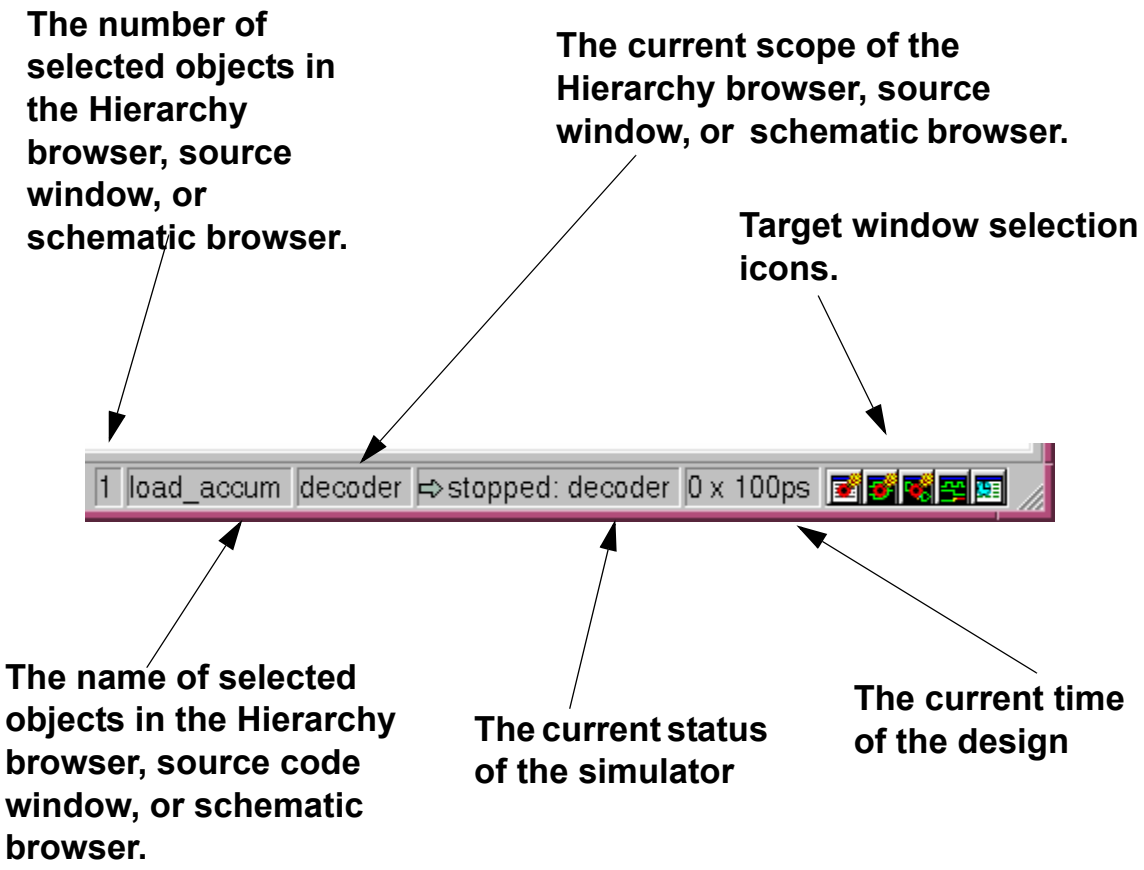
Figure 11-1 DVE Top Level Window Initial View



Status Bar

The status bar in the lower left corner of the Top Level Window displays information about your simulation. Figure 11-2 shows the information displayed in the status bar boxes.Status Bar

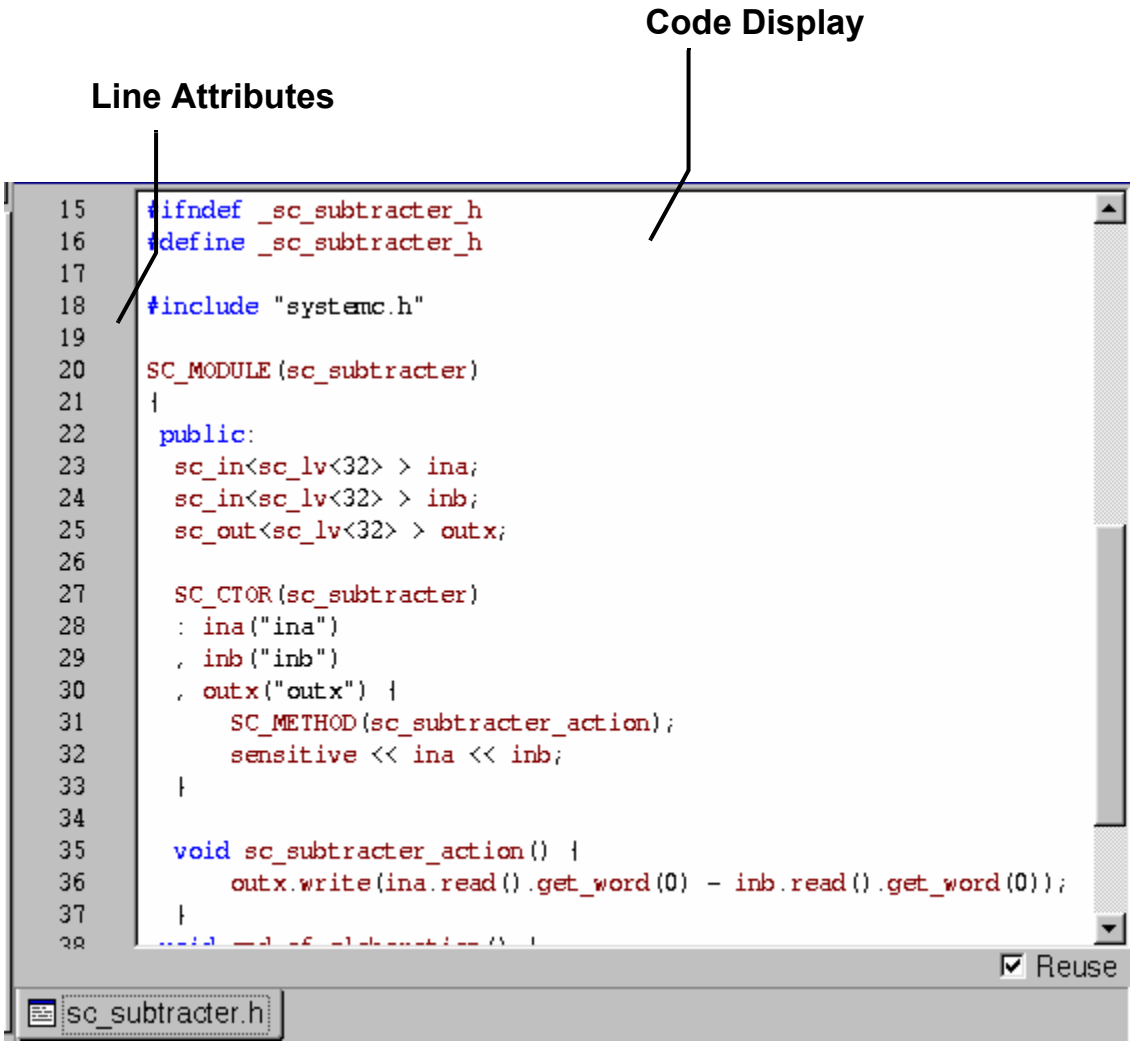
Figure 11-2 Status Bar



Source Window

In addition to viewing source code in the Source Window, you can open multiple tabs or freestanding Source Windows. You display HDL, testbench, and assertion source code by double-clicking an item in a list or by dragging and dropping a signal or assertion from another DVE window.

Figure 11-3 Source Window

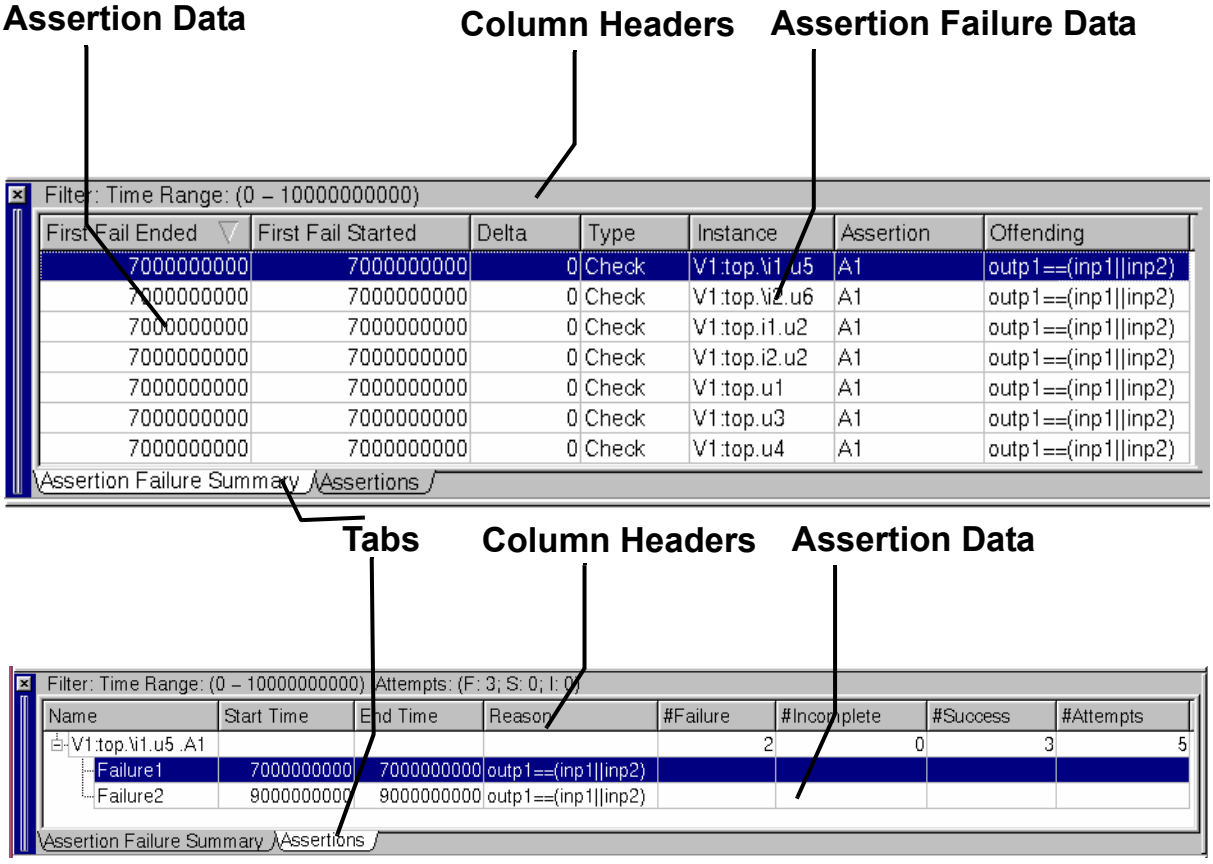


Assertion Window

Use the Assertion Window tabs to view assertions and assertion failures:

- Assertion Failure Summary tab: Displays a summary of one failed attempt for every failing assertion. By default the failed attempt is the first attempt that failed.
- Assertions tab: Totals results for all assertions in a selected block or scope. Also allows analysis of results of specific assertion attempts.

Figure 11-4 Assertion Window tabs



Wave Window

The Wave Window displays

- Waveforms of selected signals in your design.
- Trace information for a particular assertion, along with a waveform of the signals and expressions associated with the assertion.

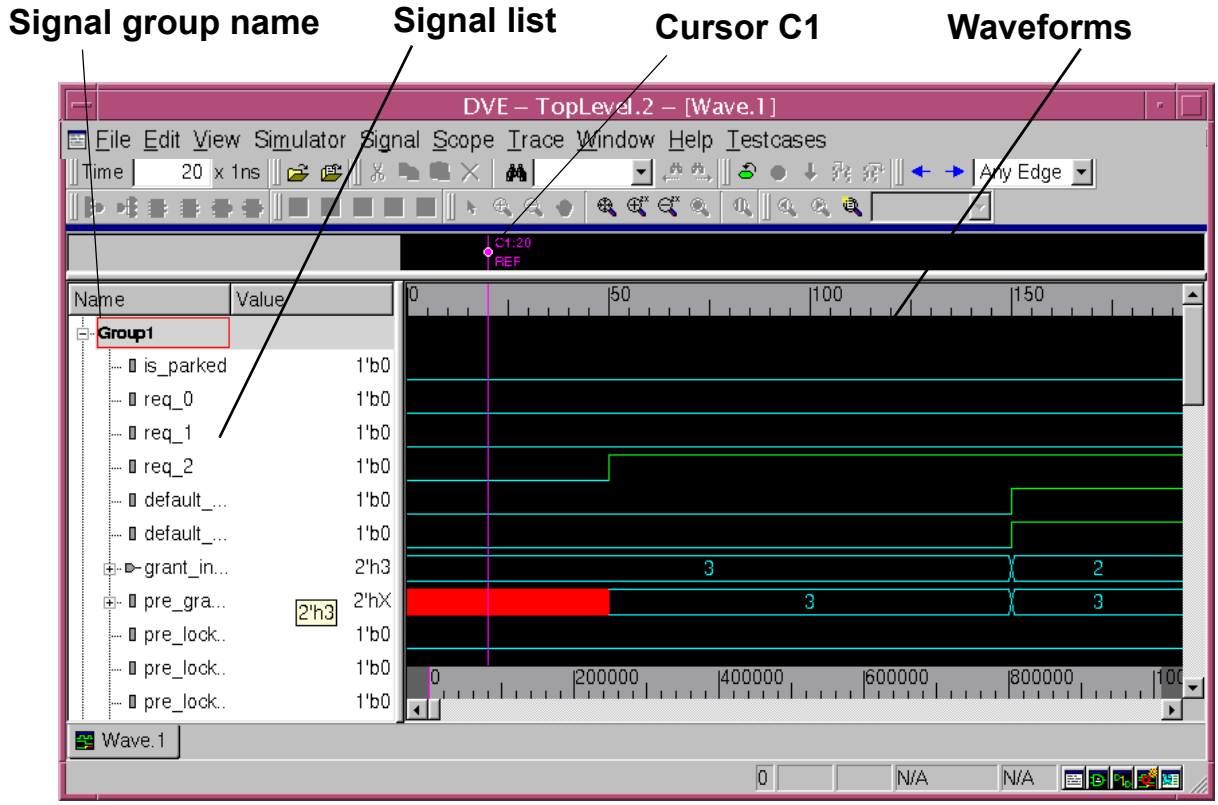
Displaying Signals in the Wave Window

Use the Wave Window to view and highlight signals of interest, advance to various simulation times, and view simulation time deltas.

To view signals in the Waveform Window, you can double click an an assertion summary item or an attempt, drag and drop an item in a list or select an item and select **Add to Waves** from the menu.

Figure 11-5 shows an example of the Waveform Window displaying signals. Cursor 1 (C1) indicates current time.

Figure 11-5 Waveform Window



Viewing Assertions and Assertions Attempts

You debug assertions by first displaying information for a particular assertion failure in the Assertion Window, You can then examine a waveform of the signals. All trace information is color-coded for easy readability and analysis.

Typically, you access and view an assertion by double-clicking a particular assertion in either of the following windows:

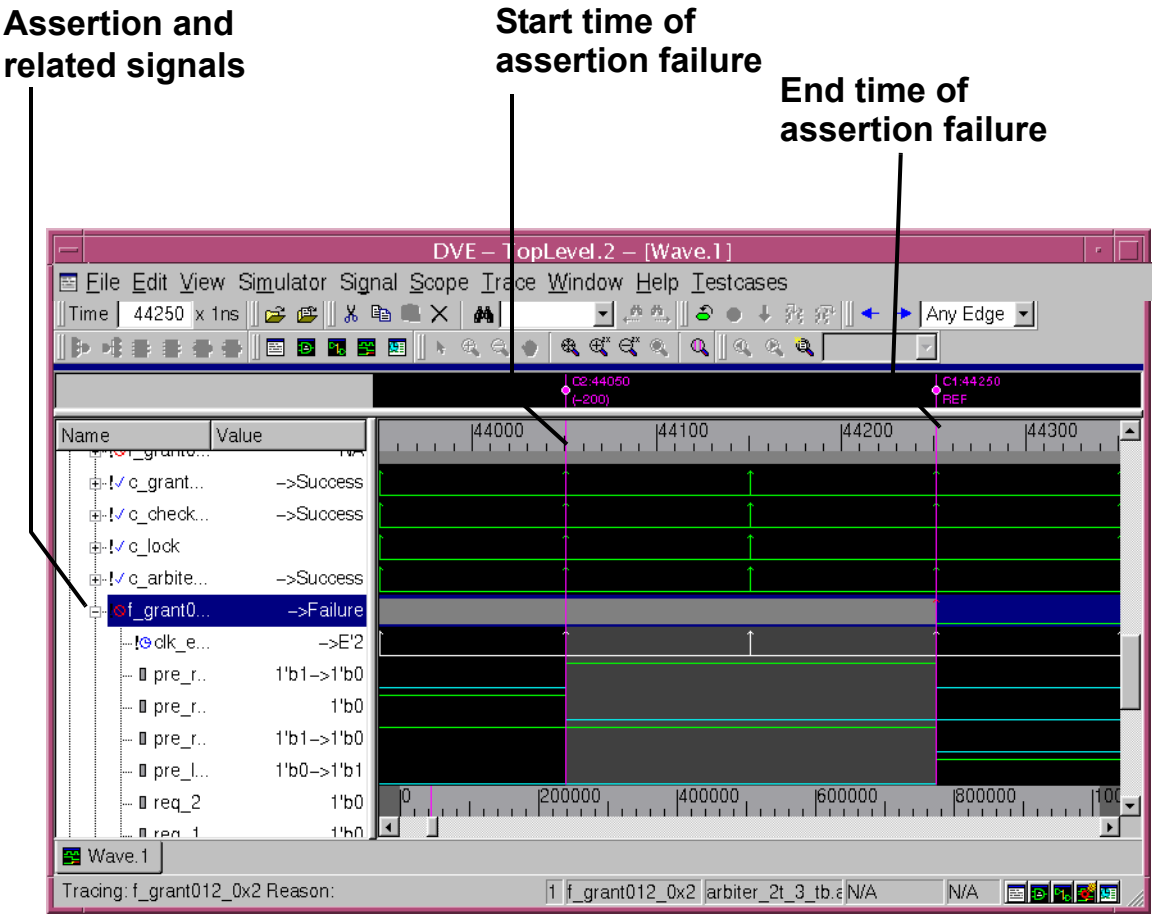
- The Assertion Window (located in the Console of the Top Level Window)
- The Hierarchy Browser by navigating to the assertion.

Or you can drag and drop an OVA assertion unit from the Hierarchy Browser or an OVA or SVA assertion from the Data Window into the the Wave Window (double click to display the source code in the Source Window).

You can then view a trace of a failed assertion by double clicking a failed attempt (a red up arrow) in the Wave Window.

Figure 11-6 shows an example of the Waveform Window. displaying a trace of a failed assertion attempt. The cursors mark the start and end times of the failure and the background color is also different to indicate the assertion time region. This lighter color persists if the cursors are moved to make it easy to identify the assertion region.

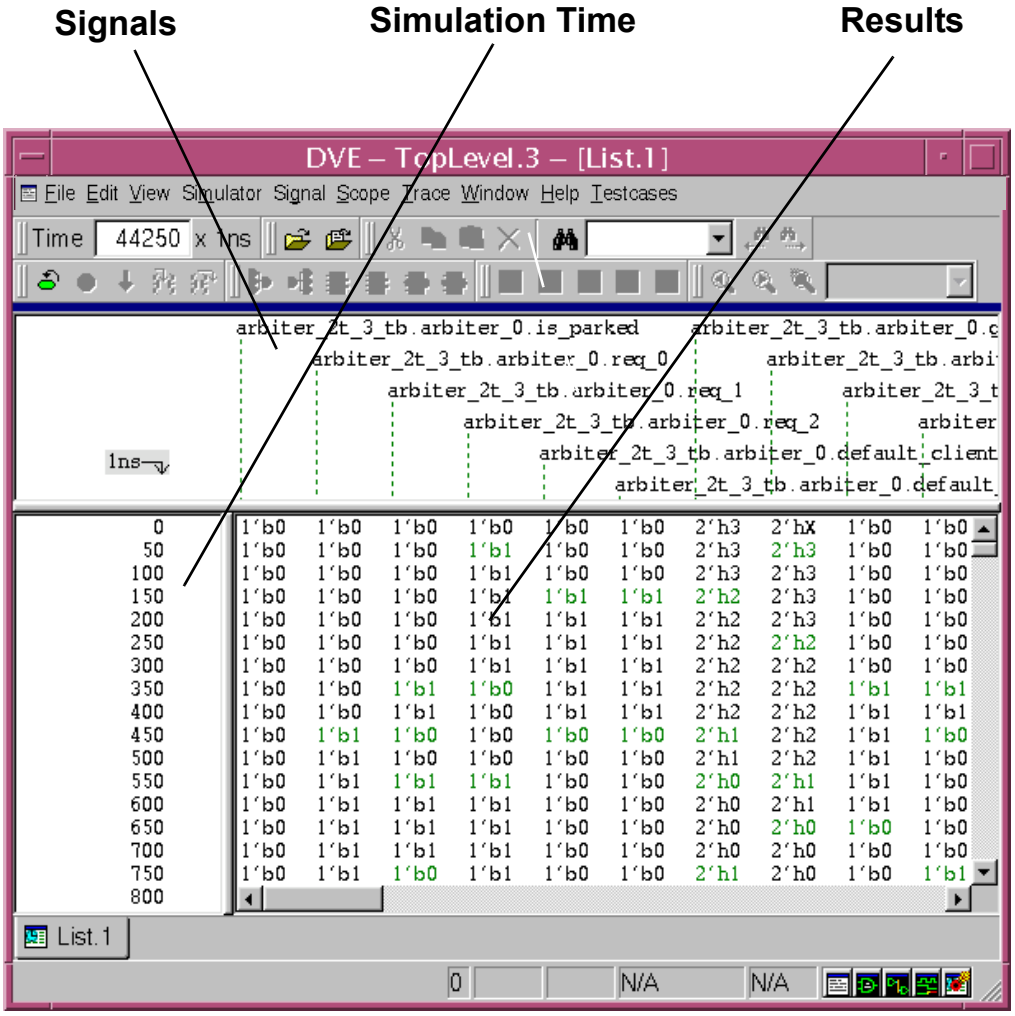
Figure 11-6 Waveform Window displaying a trace of a failed assertion attempt



List Window

Use the List Window to generate ASCII simulation results. The List Window displays simulation data as shown in the Wave Window but in tabular format. It is a scrollable window, showing simulation time on the left.

Figure 11-7 List Window

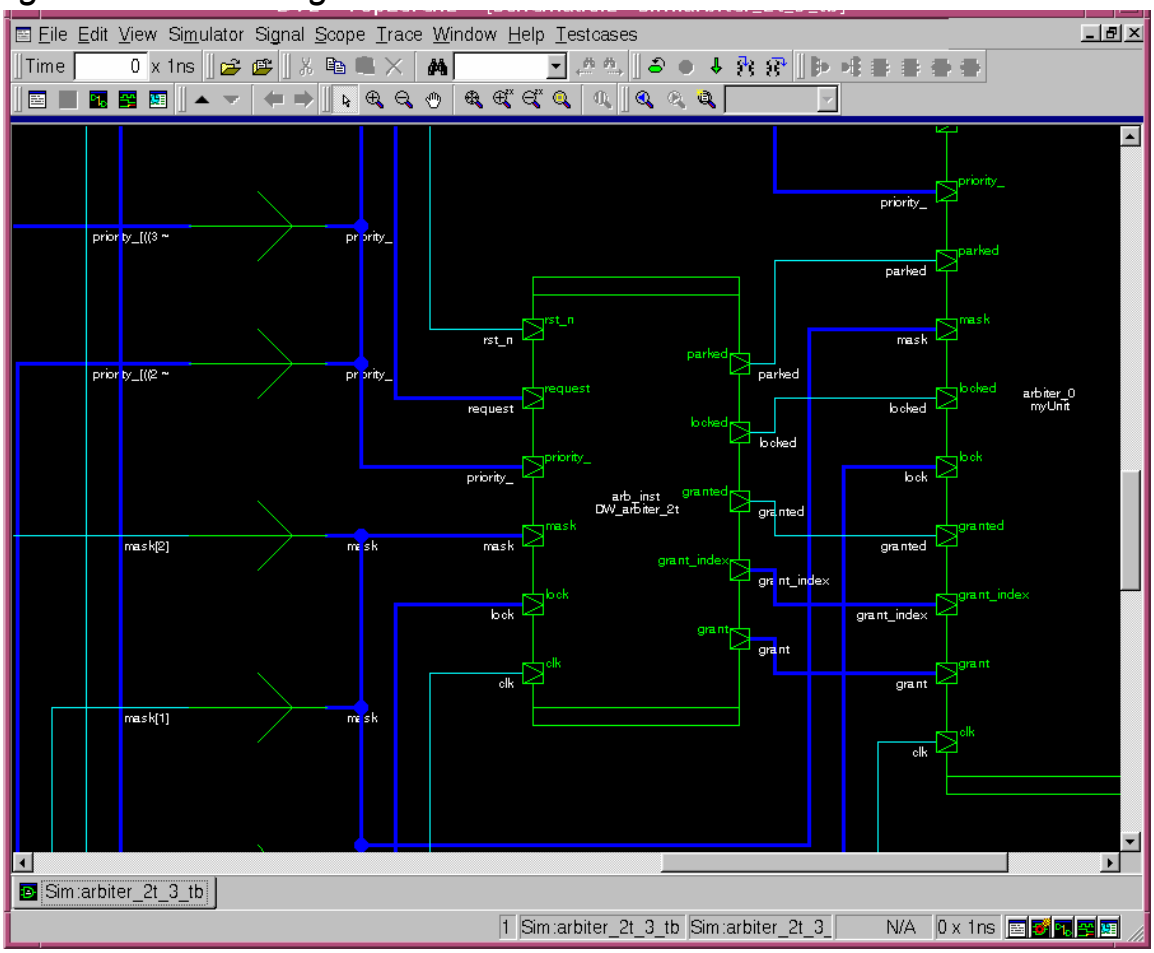


Schematic Window

Schematic views provide a compact, easy-to-read graphical representation of a design. There are two types of schematic views in DVE: design and path.

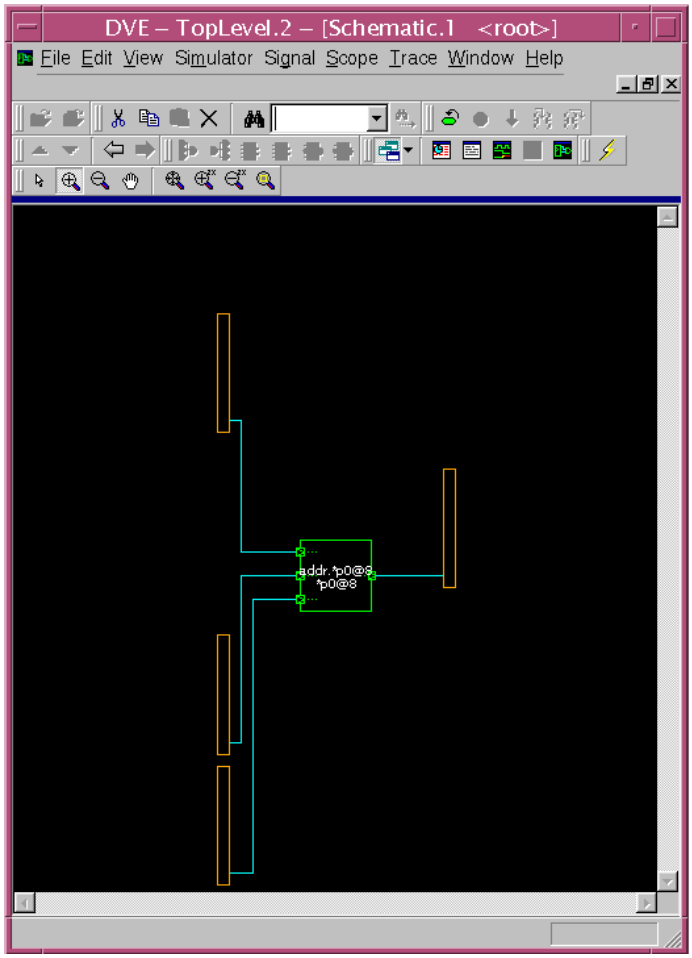
- A design schematic shows the hierarchical contents of a the design or a selected instance and lets you traverse the hierarchy of the design.

Figure 11-8 . Design Schematic



- A path schematic is a subset of the design schematic displaying where signals cross hierarchy levels. Use the path schematic to follow a signal through the hierarchy and display portal logic (signal effects at ports).

Figure 11-9 Path Schematic



12

VCD and VPD File Utilities

VCS comes with a number of utilities for processing VCD and VPD files for certain purposes. These utilities are as follows:

- The vcdpost Utility to create an alternative VCD file that has separate signal entries for each bit of the vector signals or always unique VCD file identifier codes for all signals
- The vcdiff Utility for comparing the simulation data in two VCD, EVCD, or VPD files (described on 12-5)
- The vcat Utility for more easily seeing the data in a VCD file (described on 12-13)
- 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 (described on 12-23)

The vcdpost Utility

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

- 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

In IEEE Std. 1364-1995, section 15, page 212 states that the VCD format does not support a mechanism to dump part of a vector. For this reason, if you enter a bit select or a part-select for a net or register as an argument to the `$dumpvars` system task, VCS records value changes and transition times for the entire net or register in the VCD file. For example, if you enter the following in your source code:

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

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 signal out1 are for the entire signal and not just for the 0 bit.

The vcdpost utility can create an alternative VCD file that defines a separate `$var` section for each bit of the vector signal. The results would be 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, for performance reasons, VCS assigns the same VCD file identifier code to more than one net or register if these nets or registers keep the same value throughout the simulation. for example:

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

Here VCS assigns the `!` identifier code to more than one net.

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

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

The vcdpost Utility Syntax

The syntax for the vcdpost utility is as follows:

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

Where:

`+scalar`
: Specifies creating separate `$var` sections for each bit in a vector signal.This option is the default option and you include it on the command line when you also include the `+unique` option and want to create a VCD file that both scalaraizes 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 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 dumps files and reports any differences found. Dump files consist of 2 sections:

- A header section that reflects the hierarchy (or some subset) of the design that was used to create the dump file, and
- 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.

Two diffs are always performed. First, the header sections are compared by vcdiff. Any signals/scopes that are present in one dump file but are absent in the other are reported to the user.

The second diff compares the value change sections of the dump files, for signals that appear in both dump files. Value change differences are determined based on the final value of the signal in a time step.

Preparing to Run vcdiff

On UNIX platforms the vcdiff utility is installed at $VCS_HOME/bin/vcdiff. You can set the path environment variable to run vcdiff by entering the following:

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

The vcdiff Utility Syntax

The syntax of the vcdiff utility is as follows:

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

Scope/Signal Hierarchy Options.

The following options control how the header sections of the dump files are compared.

-noabsentsig
: Do not report any signals that are present in one dump file but are absent in the other.

`-absentsigscope <scope>`
: Only absent signals in the given scope are reported.

`-absentfile <file>`
: The full path names of all absent scopes/signals are printed to the given file, as opposed to stdout.

`-absentsigiserror`
: If this option is present and there are any absent signals in either dump file, then an error status is returned from vcdiff upon completion even if there weren't any value change differences detected. Without this option being present, absent signals do not cause an error return.

`-allabsentsig`
By default, only the first 10 absent signals are reported. With this option, all absent signals are reported.

`-ignorecase`
This option ignores the case of scope/signal names when looking for absent signals. In effect, all signal/scope names are converted to uppercase before comparison.

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

Specifying Scope(s) to Be Value Change Diffed.

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

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

`-level N`
Limits the depth of scope that value change diffing occurs. For example, if `-level 1` is the only command line option, then only the signals in the top level scopes in the dump file will be value change diffed.

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

`-ignore <file>`
: Removes any signals/scopes contained in the given file from value change diffing. The file contains a set of full path specifications of signals and/or scopes, one per line. Note: `-scope/-level` is applied firs. -include is then applied to the remaining scopes/signals, and finally, `-ignore` is applied.

Specifying when to Perform Value Change Diffing

The following options limit when value change differences are detected.

`-min <time>`
: Specifies the starting time (in simulation units) when value change diffing is to begin (by default, time 0).

`-max <time>`
: Specifies the stopping time (in simulation units) when value change diffing will end. By default. this occurs at the latest time found in either dump file.

`-strobe <first time> <delta time>`
: Only check 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 (above) this option tells vcdiff to look for differences just before the strobe is true.

`-when <expression>`

Differences will only be reported when the given `when` expression is true. Initially this expression can consist only of scalar signals, combined via and, or, xor, xnor, and not operators and employing parenthesis to aid in grouping these infix expressions. Note, operators may be either verilog style (&,|,^,~^,~) or VHDL (and, or, xor, xnor not). All signals used in expressions must be fully specified (complete path, from root).

`-synch <signal>`

Differences are only checked for when the given signal changes value. In effect, the given signal is a "clock" for value change diffing, where diffs are only checked for on transitions (any) of this signal.

`-synch0 <signal>`

As `-sync` (above) except that diffs are only checked for when the given signal transitions to ’0’

`-synch1`

As `-sync` (above) except that diffs are only checked for when the given signal transitions to ’1’

Note: The `-max, -min` and `-when` options must all be true in order for a value change difference to be reported.

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>`
: Ignore value change differences when the amount of time that the signals mismatch is less than the time specified. A value change diff will be reported if the signal that first moved away from matching is the same signal that returns to matching, even if the amount of time that the signals mismatched is less than the given time.

`-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 directionality information than vcd or even vpd files. When this option is used, the strength portion of a signal value will be ignored when checking for differences.

`-compare01xz (evcd only)`

When this option is used (evcd files only) all signal state information is converted to equivalent 4 state values (0,1,x,z) before difference comparison is made. Also the strength information is ignored.

`-xzmatch`

This option equates ’x’ and ’z’ values.

`-xumatch (9-state vpd file only)`

This option equates ’x’ and ’u’ (uninitialized) values.

`-xdmatch (9-state vpd file only)`

This option equates ’x’ and ’d’ (dontcare) values.

`-zdmatch (9-state vpd file only)`

This option equates ’z’ and ’d’ (dontcare) values.

`-zwmatch (9-state vpd file only)`

This option equates ’z’ and ’w’ (weak 1) values. -noxzmatchat0 In conjunction with -xzmatch (above), this option will cause ’x’ and ’z’ value to NOT be treated the same only during time 0.

Specify Output Format Options.

The following optionw change how value change differences are reported.

`-allsigdiffs`

By default, vcdiff only shows the first difference for a given signal. When this option is used, all diffs for a signal are reported (until the maximum number of diffs is reported (see -limitdiffs below).

`-wrapsize <columns>`

Vectors longer than the given size will have their output wrapped to the next line. By default, this value is 64.

`-showmasters` (vcd, evcd files only)

Show Verilog-XL collapsed net masters. VCS can split a collapsed net into several sub-nets when this has performance benefit. When this option is used, and when the master signal (first signal defined on a net) is different in the two dump files, the master signals are reported.

`-limitdiffs <number of diffs>`

By default, vcdiff stops after the first 50 diffs are reported. This option overrides that default. Setting this value to 0 causes all diffs to be reported.

`-geninclude <filename>`

When this option is used, a separate file of the given name is produced in addition to the standard vcdiff output. This file contains a list of signals that had at least one value change difference. The format of the file is one signal per line. Each signal name is a full path name. It is intended that this file be used as input to the vcat tool with vcat's `-include` option.

`-spikes`

A spike is defined as a signal that changes multiple times in a single time step. When this option is used, value change differences detected when the signal has spiked (glitched) are annotated with #'s, and a total count of such diffs is kept and reported.

The vcat Utility

The format of a VCD 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 see the information contained in a VCD file.

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]
```

The functions of the command line options are as follows:

`-deltaTime`
: Specifies writing simulation times as the interval since the last value change rather than the absolute simulation time of the signal transition.

  Without `-deltaTime` a vcat output looks like this:

```
--- TEST_top.TEST.U4._G002 ---
  0      x
  33     0
  20000  1
  30000  x
  30030  z
  50030  x
  50033  1
  60000  0
  70000  x
  70030  z
```

  With `-deltaTime` a vcat output would look like this:

```
--- TEST_top.TEST.U4._G002 ---
```

```
0       x
33      0
19967   1
10000   x
30      z
20000   x
3       1
9967    0
10000   x
30      z
```

`-raw`
Displays “raw” value changed data, organized by simulation time, rather than signal name.

`-min` *`time`*
Specifies a start simulation time from which vcat begins to display data

`-max` *`time`*
Specifies an end simulation time up to which vcat displays data

`-scope` *`instance_name`*
Specifies a module instance. vcat displays data for all signals in the instance and all signals hierarchically under this instance

`-level` *`level_number`*
Specifies how many hierarchical levels down vcat displays data. The starting point is either the top-level module or the module instance specifies with the `-scope` option.

`-include` *`filename`*
Specifies a file that contains a list of module instances and signals. vcat only displays data for these signal or the signals in these instances.

`-ignore` *`filename`*

Also specifies a file that contains a list of module instances and signals. vcat does not display data for these signal or the signals in these instances.

`-spikes`

Specifies indicating all zero-time transitions with the **>>** symbol in the left-most column. In addition, a summary of the total number of spikes seen is printed 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 alphabetizes its display if signals within a module instance. This option disables this alphabetizing.

`-wrapsize` *`size`*

Specifies in value displays for wide vector signals, how many bits to display on a line before wrapping to the next line.

`-showmasters`

Specifies showing Verilog-XL collapsed net masters

`-showdefs`

Specifies displaying signals but not their value changes or the simulation time of these value changes.

`-showcodes`

Specifies also displaying the signal's VCD file identifier code.

`-stdin`
: Enable 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 how the instance applies stimulus to the rest of the design, see “Generating Source Files From VCD Files” on page 12-17.

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 you see, for example that signal top.mid1.midr1 at time 0 had a value of X and at simulation time 2000 (as specified by the `$timescale` section of the VCD file, which VCS derives from the time precision argument of the `` `timescale `` compiler directive) this signal transitioned to 1.

Generating Source Files From VCD Files

vcat can generate Verilog source files that are one the following:

- A module definition that succinctly models how a module instance is driven by a design, that is, a concise test bench module that instantiates the specified instance and applies stimulus to that instance the way the entire design does.
  We call this test bench 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.
  We call this module generation.

Note:

vcat can only generate these source files for instances of module definitions that do not have inout ports.

Test bench generation enables you to focus on a module instance, applying the same stimulus as the design does but at faster simulation because the test bench 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 test bench 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 do this 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 by default named vgen.cfg 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. Specify test bench generation with the keyword `testbench` or specify 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:

```
top.ad1
testbench
//moduleGeneration
module adder (out,in1,in2);
input in1,in2;
output [1:0] out;
```

You can use a different name and location for the configuration file but if you do you must enter it as an argument to the `-vgen` option, for example:

```
vcat filename.vcd -vgen /u/design1/vgen2.cfg
```

Example

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 test bench module named test propagates through an instance of a module name passer before it propagates to an instance of a module named adder. vcat can generate a test bench module to stimulate the instance of adder in the same exact way but in a more concise and therefore faster simulating module.

If we used the example vgen.cfg configuration file on page 12-19 and entered 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
```

Significantly less code to apply the same stimulus with the instance of module passer omitted.

If we revise the vgen.cfg file to have vcat do module generation, the generated source file, moduleGeneration__top_ad1.v, is as follows:

```
module adder (out,in1,in2) ;
input in2 ;
input in1 ;
output [1:0] out ;
reg [1:0] out ;
initial begin
    #0 out = 2'bxx;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
    #10 out = 2'b01;
    #10 out = 2'b10;
    #10 out = 2'b01;
    #10 out = 2'b00;
end
endmodule
```

Notice that the input ports are stubbed and the values from the VCD file are assigned directly to the output port.

The vcsplit Utility

This tool 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). The selection of scopes/signals to be included in the generated file can be made either via a command line argument, or a separate "include" file.

The vcsplit utility has the following syntax:

```
vcsplit [-o output_file] [-scope selected_scope_or_signal]
[-include include_file] [-min min_time] [-max max_time]
[-level n] [-ignore ignore_file] input_file [-v] [-h]
```

Where:

output_file
: Specifies the name of the new VCD/EVCD/VPD file to be generated. If *output_file* is not specified, a default file name (vcsplit.vcd) will be created.

-scope
: 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
: Specifies the name of an include file which specifies a list of signals/scopes whose value changes are to be included in the output file.

The include file will contain one scope or signal per line. Each presented scope/signal must be found in the input VCD, EVCD, or VPD file. If a scope is given, and separately, a signal in that scope is also given, all the signals in that scope will be included, and a warning will be issued.

Note: if both `-include` and `-scope` options are used, all signals and scopes indicated are included.

`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 the input_file is not specified, vcsplit will take its input from stdin. This stdin option for vcd and evcd files is provided so that the user can pipe the output of gunzip to this tool. If an attempt is made to pipe a VPD file through stdin, an error message will be generated and vcsplit will exit with an error status.

`-min`

Specifies time to begin the scan.

`-max`

Specifies the time to stop the scan.

`-ignore`

Specifies the name of the file that specifies a list of signals/scopes whose value changes are to be ignored in the output file.

If neither include_file and selected_scope_or_signal are specified, then all the value changes are included in the output file except those signals/scopes in the ignore_file.

If an include_file and/or a selected_scope_or_signal are specified, all value changes of those signals/scopes that are present in the nclude_file and the selected_scope_or_signal but absent in ignore_file are included in the output file. If a scope is listed in the ignore file, all the signals and the scopes in the scope will be ignored.

`-level`
Reports only n levels hierarchy from top or scope. If neither include_file and selected_scope_or_signal are specified, n is computed from the top level of the design. Otherwise, n is computed from the highest scope included.

`-v`
Causes the current version message to be displayed.

`-h`
Displays a help message explaining usage.

Note: In general, any command line error (such as illegal arguments) that are detected will cause vcsplit to issue an error message and exit with an error status. Specifically:

1. If any errors are detected 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), an error message is reported, and vcsplit exits with an error status.
2. If an error is detected in parsing the input file, an error is reported, and vcsplit exits with an error status.
3. If neither a -scope, an -include or an -ignore option is provided by the user, an error will be reported, and vcsplit will exit with an error status.

Limitations in the Current Version (VPD)

- MDAs are not supported.
- Bit/part selection for a variable is not supported. If this usage is detected, the vector will be regarded as all bits are specified.

13

Two State Simulation

Two state simulation is a high performance option where signals can only have the simulation values, 1 and 0, so they never have the X or Z value, and strength values are not simulated (but if signal strengths are displayed they are always displayed with the strong strength). Two state simulation takes advantage of the fact that after initialization many digital designs predominantly simulate with only 1 and 0 values.

The advantage of two state simulation is that VCS has less simulation data to keep track of so it can run faster. The disadvantage is a potential inaccuracy caused by the absence of the X and Z values and signal strengths. VCS has implemented two state simulation to minimized these inaccuracies.

This chapter describes how to use two state simulation, its contents include:

- Value And Strength Mapping For Two State Simulation — how two state simulation deals with what would be X and Z values
- Signals That Must Retain Four State Simulation — The nets and registers that will continue to have four possible states during two state simulation of your design
- Differences in Initialization — The differences in initial values in two state simulation
- Resolving Multiple Drivers — How two state simulation resolve the values of nets with multiple drivers
- Changes In Simulation Behavior — Cases of different simulation behavior is two state simulation
- PLI Compatibility — PLI considerations for two state simulation
- Specifying Two State Simulation — You specify two state simulation with the `+2state` compile-time option.
- Specifying Four State Simulation for Parts of Your Design — Using metacomments or a configuration file to specify four state simulation for parts of your design

Note:

Two state simulation is disabled if your design back annotates delays from an SDF file.

Value And Strength Mapping For Two State Simulation

As a general rule, VCS replaces the X value by 1 and the Z value by 0 in two state simulation but there are significant exceptions to this general rule such as the initial value of some types of registers, see “Differences in Initialization” on page 13-17.

This rule applies for bus contention when a net has multiple drivers and VCS determines the net’s resolved value. Where in four state simulation VCS would resolve a value of X, in two state simulation VCS resolves a value of 1. See “Resolving Multiple Drivers” on page 13-19.

This rule applies to some undriven nets. In four state simulation when these nets initialize as undriven or become undriven, VCS applies a Z value to them. In two state simulation, VCS applies a 0 value. Nets are undriven nets if they are not connected to the output port of a lower-level module instance, an output terminal of a primitive, or are not assigned a value through a continuous assignment statement. Nets can become unconnected if they are connected to an output terminal of a tri-state primitive (`bufif1`, `bufif0`, `notif1`, and `notif0`).

Also as a general rule in two state simulation VCS displays strengths as the strong strength however this is only how they appear when you use the `%v` format specification. In two state simulation there are no strengths.

Signals That Retain Four State Simulation

To prevent the inaccuracies inherent with two state simulation, particularly mismatches with four state simulation, we have implemented two state simulation in such a way that certain signals will always have four possible states (1, 0, X, and Z) while others have only two possible states (1 and 0).

Some signals retain four state simulation because you specified that they do using metacomments or a configuration file. A `/*4value*/` metacomment in a signal declaration tells VCS that the signal must retain four state simulation, for example:

```
reg /*4value*/ [7:0] r1;
```

For more information about using metacomments and the configuration file see “Specifying Four State Simulation for Parts of Your Design” on page 13-29.

Other signals retain four state simulation for a number of reasons: because their behavior calls for a strength other than strong in certain situations, because they have a different logical behavior than the `wire` type, or because of what they are connected to or the kinds of expressions that include them as operands. In these cases, if they did not retain four state simulation, there would be a greater likelihood of a two state/four state simulation mismatch of results. This section describes the circumstances in which VCS keeps a signal in four state simulation.

Data Types That Need Four State Simulation

You declare certain types of nets, instead of the default `wire` type, because you want them to have a behavior that's different from the `wire` type, such as a strength other than strong in certain situations or a different logical behavior. We have concluded that you declared these types of nets because you wanted their different behavior and their different behavior is not possible in two state simulation. Table 13-1 lists these types of nets and explains why they need to retain four state simulation.

Table 13-1 Data Types Retaining Four State Simulation

Data Type	Reason for Retaining Four State Simulation
`tri1` and `tri0`	As specified in the IEEE Std 1364-1995 page 22, these nets model nets connected to `pullup` and `pulldown` primitives and when not driven by another signal return to the 1 value pull strength and 0 value pull strength respectively.
`trireg`	As specified in the IEEE Std 1364-1995 page 18, a trireg net, when not driven by another signal, returns to its specified capacitive state of a small, medium, or large strength.
`wand` and `triand`	Two state simulation requires replacing an X value with 1. Driving or forcing an X and Z value onto a wand or triand net would therefore require the net to resolve to a 1 value in two state. This would be contrary to the logical AND function of these nets.

Expressions That Require Four State Simulation

The case equality and inequality operators (`===` and `!==`) are for comparing the bits of signals that might contain the X or Z values. You use these operators to look for these values, and typically, in Verilog code that contains these operators, you specify that VCS do something when VCS finds X or Z values in the comparison.

In contrast, the logical equality and inequality operators (== and !=) are not customarily used for bits of signals that might contains X or Z values, and also typically in Verilog code that contains these operators, you do not specify that VCS do something when VCS finds X or Z values in the comparison.

For these reasons, if a signal is an operand in an expression that includes the case equality or inequality operators, then it retains four state simulation, and if a signal is an operand in an expression that include the logical equality or inequality operators, it can have two state simulation. Consider the following Verilog code:

```
if (reg1 == reg5)
 ⋮
if (reg2 != reg5)
 ⋮
if (reg3 === reg5)
 ⋮
if (reg4 !== reg5)
```

Signals reg1 and reg2 can have two state simulation, whereas signals reg3, reg4, and reg5 must retain four state simulation.

Note however that there is a difference in simulation behavior in two state simulation with the logical equality and inequality operators. See “X and Z Values in Expressions” on page 13-23.

Signals in Case Expressions in Case Statements

As stated in IEEE Std 1364-1995, page 33, the comparison in the case equality and inequality operators matches the comparison in a case statement. Therefore if a signal is in the case expression in a `case` statement (or a `casez` or `casex` statement) then the signal must retain four state simulation so VCS can look for X and Z value bits.

In the following code example, signal r1 is in the case expression and therefore must retain four state simulation:

```
case (r1)
1'b1 : $display("r1=1");
1'b0 : $display("r1=0");
1'bx : $display("r1=x");
1'bz : $display("r1=z");
endcase
```

Primitives With A Different Drive Strength

Certain primitives require an output strength other than the strong strength. To enable the different strength on a signal connected to the output, signals connected to these primitives must retain four state simulation. This section describes why this is for signals connected to these primitives.

Signals Connected to pullup and pulldown Primitives

As specified in the IEEE Std 1364-1995 page 67, in the absence of a strength specification, these primitive drive a pull strength of 1 or 0 on nets connected to them. The default condition is therefore for a pull strength, not a strong strength, on nets connected to these primitives. Not to counteract the default condition, and to enable other drive strength specifications on these primitives, nets connected to these primitives retain four state simulation. For example:

```
pulldown pd1 (w1);
pullup (weak1) pu1 (w2);
```

Signals w1 and w2 retain four state simulation.

Signals Connected to Bidirectional Switch Primitives

The bidirectional switch primitives are as follows:

```
tran tranif1 tranif0 rtran rtranif1 rtranif0
```

As specified in the IEEE Std 1364-1995 page 66 and 82, at least in certain circumstances, values propagating through these devices require a strength change.

For this reason signals connected to the inout terminals of these switches (not the control terminals of the `tranif1`, `tranif0`, `rtranif1`, and `rtranif0` switches) must retain four state simulation to make the strength change.

Signals Connected to Pull or Weak Drive Strength Gates

A signal connected to the output terminal of a gate will retain four state simulation if you specify a pull or weak drive strength on the output of a gate and also do either of the following:

- Specify that all the signals that attach to the input terminals retain four state simulation (see “Specifying Four State Simulation for Parts of Your Design” on page 13-29) or in some other way ensure that they retain four state simulation such as also attaching them to a pullup or bidirectional switch primitive, or using them in a case expression or an expression with a case equality or inequality operator, or specifying a four state data type.
- Assigning the Z value to the input signals some time during simulation or in some other way making it possible for VCS to determine, at compile-time, that a Z value will be on the input signals sometime during simulation. If VCS can determine this then the input signals will also retain four state simulation but only if you also specify a pull or weak drive strength on the gate.

Consider the following example:

```
module test;
wire w1,w2,w3,w4,w5;
reg r1,r2;
reg /*4value*/ r3,r4,r5,r6;
reg r7,r8,r9,r10;

and (pull1,weak0) and1 (w1,r1,r2);

and (pull1,weak0) and2 (w2,r3,r4);

and and3 (w3,r5,r6);

and (pull1,weak0) and4 (w4,r7,r8);

initial
```

```
begin
r7=1'bz;
r8=r7;
end

and and5 (w5,r9,r10);

initial
begin
r9=1'bz;
r10=r9;
end

endmodule
```

Signal w1 is in two state simulation, even though the drive strength is pull1 and weak0, because in this example metacomments do not specify that input signals r1 and r2 retain 4 state simulation, nothing in the Verilog code would require r1 to retain four state simulation, and a Z value is not scheduled on these inputs.

Signal w2 retains four state simulation because the drive strength is pull1 and weak0 and metacomments specify that the input signals r3 and r4 retain four state simulation.

Signal w3 is in two state simulation, even though metacomments specify that the input signals r5 and r6 retain four state simulation, because a drive strength of pull or weak is not specified.

Signal w4 retains four state simulation because a drive strength of pull or weak is specified, and Z values are scheduled on the input signals r7 and r8.

Signal w5 is in two state simulation, even though Z values are scheduled on the input signals, because a drive strength of pull or weak is not specified.

If you specify a supply or strong drive strength as the drive strength of the gate, the signal connected to the output terminal does not retain four state simulation. This is because the strong strength is, of course, the default strength for signals and specifying a supply drive strength for a gate is a very unusual practice so retaining four state simulation for supply strength is not implemented.

Signals in The Path For a Z Value

If there is a propagation path between two signals that are four state signals, so that a Z value could propagate from the first to the second of these two signals, then all the signals in this path retain four state simulation so that the Z value can so propagate, for example:

```
module top;
reg r1,r2;
wire /*4value*/ w1,w3;
wire w2;

always @ w1
begin
r1=w1;
r2=r1;
end

assign w2=r2;

bot bot1(w3,w2);

endmodule

module bot(out,in);
input in;
output out;
reg botr1;

always @ in
```

```
botr1=in;

assign out=botr1;
endmodule
```

All the signals in this code example retain four state simulation and because a Z value can propagate through all of them from signal w1 to w3. The path is as follows:

w1 → r1 → r2 → w2 → bot1.in → bot1.botr1 → bot1.out → w3

Similarly If it is possible for a Z value to be driven on a signal, for example if it connects to a gate that can output a Z value, and there is a propagation path for the Z value to a four state signal, then all signals in that path retain four state simulation, for example:

```
module test(in,out);
input in;
output /*4value*/ out;

reg r1,r2;
wire w1;

bufif1 (w1,1'b1,in);

always @ w1
begin
r1=w1;
r2=r1;
end

assign out=r2;

endmodule
```

The `bufif1` gate can output a Z value on to signal w1 and from w1 there is the following propagation path for the Z value:

w1 → r1 → r2 → out

So w1, r1, and r2 all retain four state simulation.

Similarly, if at compile-time VCS sees a simulation event that assigns a Z value to a signal, and there is a propagation path from that signal to a four state signal, the all the signals in the path also retain four state simulation, for example:

```
module test(in,out);
input in;
output /*4value*/ out;

reg r1,r2,r3;

initial
r1=1'bz;

always @ r1
begin
r2=r1;
r3=r2;
end

assign out=r3;

endmodule
```

In the initial block r1 is assigned the Z value, and there is a path from r1 to signal out so all the signals in the path, in this case r3 and r2, as well as r1, retain four state simulation.

Undriven Signals Connected to Four State Signals

Undriven registers (registers to which your Verilog code makes no procedural assignments) that connect to a four state signal retain four state simulation. This connection can be direct, such as through a gate, or continuous or procedural assignment statement, or along a propagation path, for example:

```
module undriven;
wire /*4value*/ w1,w2,w3,w4,w5,w6,w7;
wire w8,w9;
reg r1,r2,r3,r4,r5,r6,r7,r8;
reg /*4value*/ r9;

assign #10 w1 = r1;
assign #10 w2 = r2;
assign #10 w7 = w9;
buf b1 (w3,r3);
not nt1 (w4,r4);
and and1 (w5,r5,r6);

buf b2 (w6,w8),
    b3 (w8,r7);

initial
begin
#100 r1 = 1;
     r9 = r8;
end

endmodule
```

Undriven regs r1 and r2 retain four state simulation through because they connect to four state signals through continuous assignment statements.

Undriven wire w9 does not retain four state simulation because this rule applies only to registers.

Undriven regs r3, r4, r5, and r6 retain four state simulation because they connect to four state signals through `buf`, `not`, and `and` gates.

Undriven reg r7 retains four state simulation because it connects to a four state signal through a path of two `buf` gates.

Undriven reg r8 retains four state simulation because its value is assigned to a four state signal in a procedural assignment statement.

Ports That Connect To Four State Signals

If you connect a four state signal to a port in a module instantiation statement, then that port in that instance retains four state simulation, for example:

```
module test;
reg /*4value*/ r1;
reg r2;
 ⋮
mod mod1 (r1);
mod mod2 (r2);
 ⋮
endmodule

module mod (in);
input in;
 ⋮
endmodule
```

Port mod1.in retains four state simulation. Port mod2.in and signal r2 do not retain four state simulation.

Forced Signals

If you use a force statement to force a value on a signal, even if the forced value is 1 or 0, that signal retains four state simulation.

Continuously Assigned Signals In If Statement Expressions

If you continuously assign a four state signal to another signal and then enter that other signal in an `if` statement expression, then the other signal retains four state simulation, for example:

```
module iffy;
reg /*4value*/ r1;
wire w1;

assign w1 = r1;

initial
begin
 ⋮
if (w1)
 ⋮
end
endmodule
```

Signal w1 retains four state simulation.

Parameters Retain Four State Simulation

You can declare parameters with both the X and Z values. If you do, they continue to have the X or Z values in two state simulation.

Differences in Initialization

Table 13-2 lists the in initial values of various data types in two and four state simulation when they are not assigned a value at time 0. In this table the information for nets is for undriven nets. The initial value of driven nets in two state simulation is explained in “Initial Value of Connected Nets” on page 13-18. Table 13-2 list the values and strengths as they are displayed in four and two state simulation for the various data types. However remember that in fact there are no strength values in two state simulation, this is just how they are displayed when using the `%v` format specification.

Table 13-2 Initial Values in Two and Four State Simulation

Data Type	Initial Value in Four State	Initial Value in Two State
`reg`	`StX`	`St0`
`integer`	`StX`	`St0`
`time`	`StX`	`St0`
`real`	`St0`	`St0` (same as four state)
`realtime`	`St0`	`St0` (same as four state)
`wire`	`HiZ`	`St0`
`wor`	`HiZ`	`St0`
`trior`	`HiZ`	`St0`
`supply1`	`Su1`	`St1`
`supply0`	`Su0`	`St0`

Note that `reg`, `integer` and `time` data types initialize to 0 instead of X in two state simulation which is contrary to the general rule of mapping the X value to 1 in two state simulation.

Also note that the `real` and `realtime` data types have the same values in four and two state simulation. This is because these data types can never have the X or Z values and don't really have strengths in either two or four state simulation but are displayed as strong if you use the `%v` format specification.

Finally note that the `wand`, `triand`, `trireg`, `tri1` and `tri0` data types are not included in Table 13-2. This is because in two state simulation these data types remain in four state simulation, as explained in "Signals That Retain Four State Simulation" on page 13-4.

Initial Value of Connected Nets

For connected nets the initial 1 or 0 value is determined by the drivers on the net and what events on these nets are scheduled by these drivers at simulation time 0.

For nets with only one driver, the initial value is easily determined. In the following example:

```
wire w1,w2;
reg r1;

not not1 (w1,r1);
not not2 (w2,w1);
```

In two state simulation, `reg` r1 initializes to 0 causing `wire` w1 to initialize to 1, which in turn causes `wire` w2 to initialize to 0. Nets with multiple drivers require resolution, see "Resolving Multiple Drivers" on page 13-19.

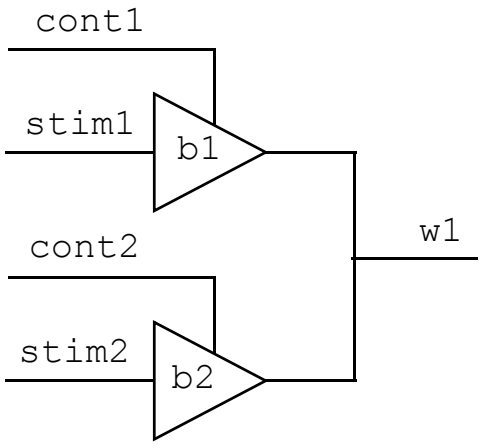
Resolving Multiple Drivers

Nets with multiple drivers often resolve differently in two state simulation and at times these differences can be the basis for performance improvements. Example 13-1 shows a net with multiple drivers.

Example 13-1 Net with Multiple Drivers

```
reg stim1,stim2,cont1,cont2;
wire w1;

bufif1 b1 (w1,stim1,cont1),
       b2 (w1,stim2,cont2);
```



In Example 13-1 wire w1 is driven by two `bufif1` gates, b1 and b2. In four state simulation, because a `bufif1` gate can output the L and H strength values in addition to the 0, 1, X, and Z values, the truth table for resolving the value of wire w1 is in Table 13-3.

Table 13-3 Four State Truth Table for a Net Driven by `bufif1` *Gates*

		b1					
		0	1	X	Z	L	H
b2	0	0	X	X	0	0	X
	1	X	1	X	1	X	1
	X	X	X	X	X	X	X
	Z	0	1	X	Z	L	H
	L	0	X	X	L	L	X
	H	X	1	X	H	X	H

In two state simulation the two `bufif1` gates can only drive 0 and 1 values onto `wire` w1. The truth table for resolving the value of `wire` w1 is considerably simpler and is shown in Table 13-4.

Table 13-4 Two State Truth Table for a Net Driven by `bufif1` *Gate*

		b1			
		0	1	1	0
b2	0	0	1	1	0
	1	1	1	1	1
	1	1	1	1	1
	0	0	1	1	0

The mapping of X and Z to 1 and 0 in two state simulation makes the resolution of the value of the net a basic logical OR function (if X and Z were mapped differently it would be a logical AND function).

This example and its accompanying truth tables illustrate how net resolution in two state simulation can be considerable simplified to a logical function instead of a table lookup and because of this simplification, increases performance.

Changes In Simulation Behavior

The absence of the X and Z values on signals and mapping them to 1 and 0 can cause different simulation behavior. This section describes several ways that a design's behavior can be different during two state simulation so that you can try to avoid these situations and develop a design that has matching behavior in two and four state simulation.

Applying Different Stimulus

Applying the X and Z value in your code applies the 1 and 0 values instead in two state simulation. Consider the following code:

```
not not1 (wire1,reg1);

initial
  begin
      reg1=0;
  #10 reg1=1'bz;
 end
```

In two state simulation there is no simulation event at time 10 because at that time the scheduled event is assigning to reg1 its current value and so there is no transition at that time in reg1 or wire1.

Different Initialization Causing Different Simulation Behavior

The different initial values can cause many kinds of different simulation behavior. for example. When an unconnected net initialized to 1 instead of X in two state simulation, then a subsequent assignment of 1 to that net is not seen as a transition. In the following example:

```
always @ (posedge wire1)
reset=1;
```

In four state simulation, if wire1 initializes to X and a subsequent event deposits a 1 value on wire1, VCS detects a transition on wire1 and executes the always block.

In two state simulation, wire1 would initialize to 1 and VCS would not detect a transition on wire1 when the subsequent event deposits a 1 value on wire1, therefore VCS does not execute the always block.

Missing Rising and Falling Edges

The absence of X and Z values in two state simulation means that the following transitions never occur in two state simulation:

- X to 1
- 1 to X
- Z to 0
- 0 to Z

The absence of these transitions means that VCS does not execute some always blocks that are controlled by edge-sensitive “sensitivity list” event controls. Consider the following code:

```
always @ (negedge wire1)
$display("falling edge on wire1");
```

A transition of 1 to X executes this always block in four state simulation. This transition does not occur in two state simulation.

X and Z Values in Expressions

When you use the logical equality and inequality operators in an expression to compare the X and Z value to a signal in two state simulation, VCS does not replace the X and Z values with 1 and 0 values before it evaluates these expressions. Consider the conditional expression in the following `if` statement:

```
if (wire_1 == 1'bx) reg_1 = reg_3;
```

If the signal wire_1 is in two state simulation, these expressions are never true. If signal wire_1 is in four state simulation, these expressions could be true. For the inequality operator the opposite is true. Consider the conditional expression in the following `if` statement:

```
if(wire_1 != 1'bx)
```

If the signal wire_1 is in two state simulation, these expressions are never false. If signal wire_1 is in four state simulation, these expressions could be false.

Note:

This situation does not arise with the case equality (===) and case inequality (!==) operators. See “Expressions That Require Four State Simulation” on page 13-5.

In assignment statements that assign an expression to a signal, VCS first evaluates the expression and assigns the result to the signal. VCS does not map the X and Z to 1 and 0 before evaluating the expression. If the results of the evaluation of the expression is X or Z, VCS then maps the results to 1 or 0 when it assigns the results to the signal. Usually whether VCS maps X and Z to 1 and 0 before or after it evaluates the expression doesn’t make any difference but in one particular case it does:

```
r1=1’b0+1’bz;
```

The expression resolves to X and then VCS maps the X to 1 and assigns the 1 to r1. If VCS mapped the Z to 0 before evaluating the expression, VCS would assign the resulting 0 to r1.

Tri-State Logic Gates

In two state simulation the `bufif1`, `bufif0`, `notif1`, and `notif0` gates output a 0 value instead of the Z, L, and H values.

Some Strength Specifications Are Ignored

The strength specifications in the following example net declaration and continuous assignment statement are ignored on two state simulation:

```
wire (pull1,weak0) w1;
assign (pull0, weak1) w2 = 1’b0;
```

If you need the pull strength you can use the `pullup` or `pulldown` primitive. Nets connected to these primitives retain four state simulation.

Strength specifications of the pull or weak strength in primitive declarations are not ignored, they result in the net connected to the output terminal retaining four state simulation. See “Signals Connected to Pull or Weak Drive Strength Gates” on page 13-9.

If you want to know if there are strength specifications in your design, include the `+warn2val` compile-time option.

User-Defined Primitives Output Different Values

UDP tables typically contain X values in input and output entries. The following approach is taken to simplify UDP tables for two state simulation:

- Table rows that contain an X value input or current state field are ignored.
- When a table row contains an X value output field, it is not ignored but the X value changes to 1.

In four state simulation if there is not a table row specifying the current state of the input terminals, VCS deposits an X value on the output terminal. In two state simulation, VCS deposits a 1 instead.

Consider the following table entries:

```
table
// in1  in2  : out
    X    0   : 0;
    X    1   : 0;
```

```
    0   X  : 0;
    1   X  : 0;
    0   0  : 1;
    0   1  : 1;
    1   0  : 1;
    1   1  : X;
endtable
```

This UDP will always output a 1 value in two state simulation.

Infinite Value Expressed Differently

In four state simulation. when you divide a value by zero and assign the quotient to a signal, VCS shows the signal's value to be X. In two state simulation, VCS shows the signal's value to be -1. Consider the following code:

```
integer int1, int2, int3;

initial
begin
#10 int1=1/0;
#10 int2=2/0;
#10 int3=3/0;
end
```

In four state simulation there never are transitions on any of these integers. They initialize to X and remain X. In two state simulation, they initialize to 0 and transition to -1.

PLI Compatibility

Maintaining PLI compatibility in mixed 2/4 state simulation is an important consideration. Certain PLI `acc` and `tf` routines (for instance, `acc_fetch_value, acc_set_value, tf_nodeinfo`, etc.) specify logic value of a signal through a pre-defined structure. In VCS's approach, no altering of any user visible data structures (e.g. `s_acc_vecval`) occurs. While reading the value of a 2-state signal, control bits (`bval`) are set to zero by default. Similarly, control bits are ignored when the user tries to write a value to a 2-state signal.

Specifying Two State Simulation

To run two state simulation for your design you must compile your design with the `+2state` compile-time.

Entering this option enables the performance enhancement of only having 1 or 0 values, and no strength values, for all the signals in your design except those that either must retain four state simulation (see "Signals That Retain Four State Simulation" on page 13-4) and those that you specified retaining four state simulation (see "Specifying Four State Simulation for Parts of Your Design" on page 13-29).

When you compile for two state simulation, there are other compile-time options for two state simulation:

`+noinitnegedge`
: Suppresses the execution of an always block at time zero.

In two state simulation a falling-edge-sensitive event control for a net in the “sensitivity list” position on an always block, in some situations, causes VCS to execute the always block at time 0, whereas VCS does not do so in four state simulation. The following is an example of such an event control:

```
always @ (negedge w1)
$display("falling edge event control");
```

To stop the execution of the always block at time 0 in two state simulation, include the `+noinitnegedge` compile-time option.

`+warn2val`

Enables warning messages about possibly ignoring strength specifications in net declarations, primitive declarations, and continuous assignment statements.

In two state simulation, in most cases, strength specifications are ignored. They are always ignored in net declarations and continuous assignments. They are also ignored in primitive declarations unless the strength specification is for either the pull or weak strength (in which case connecting signals retain four state simulation, see “Signals Connected to Pull or Weak Drive Strength Gates” on page 13-9).

When compiling for two state simulation, to see if your design contains strength specifications, including those that VCS ignores and does not ignore, include this compile-time option.

The warning message is as follows:

```
Warning: Strength specification in 2-state mode may be
ignored (filename line number)
```

Specifying Four State Simulation for Parts of Your Design

You can use a metacomment to specify that a signal retains four state simulation and you can use a configuration file to specify that all the signals explicitly or implicitly declared in a module definition retain four state simulation.

Using Metacomments

You use the `/*4value*/` metacomment to specify that a signal retains four state simulation. You use this metacomment in signal declarations after the signal's data type, for example:

```
input /*4value*/ [63:0] data;
reg /*4value*/ [31:0] r1;
wire /*4value*/ [31:0] w1;
```

This metacomment looks like a code comment in the middle of a source line but its syntax is exact; include no white spaces inside it. In the following example white space changes the metacomment to a code comment that VCS ignores:

```
reg /* 4value */ [31:0] r1;
```

There is also the `/*2value*/` metacomment. The officially supported use of this metacomment is for documenting your Verilog code. It many cases this metacomment in a signal declaration will make the signal simulate in two state simulation but using this metacomment for this purpose is not officially supported by VCS.

Using The Configuration File

You specify the configuration file with the `+optconfigfile` compile-time option, for example:

`+optconfigfile+`*`filename`*

The VCS configuration file enables you to enter statements that specify:

- Using the optimizations of Radiant technology on part of a design.
- Enabling PLI ACC write capabilities for memories or for part of the design, or disable them for the entire design
- Four state simulation for part of a design.

The syntax of each type of statement for four state simulation is as follows:

`module {`*`list_of_module_identifiers`*`} {4value};`

or

`tree [(`*`depth`*`)] {`*`list_of_module_identifiers`*`} {4value};`

Where:

`module`
: Is a keyword that specifies that either two or four state simulation for the signals declared in all instances of the modules in the list of module identifiers.

`list_of_module_identifiers`
: Is a comma separated list of module identifiers enclosed in curly braces: `{ }`

`4value`
: Is a keyword that specifies four state simulation for the signals in the modules in the list of module identifiers. Enclosed in curly braces: { }

`tree`
: Is a keyword that specifies four state simulation applies to the signals in all instances of the modules in the list and also apply to the signals in all module instances hierarchically under these module instances.

`depth`
: Is an integer that specifies how far down the module hierarchy, from the specified modules, you want to apply four state simulation. You can specify a negative value. A negative value specifies descending to the leaf level and counting up levels of the hierarchy to apply four state simulation. This specification is optional. Enclose this specification in parentheses: `()`

Unlike using the configuration file to specify the optimizations of Radiant technology, you cannot use the configuration file to specify four state simulation for the following:

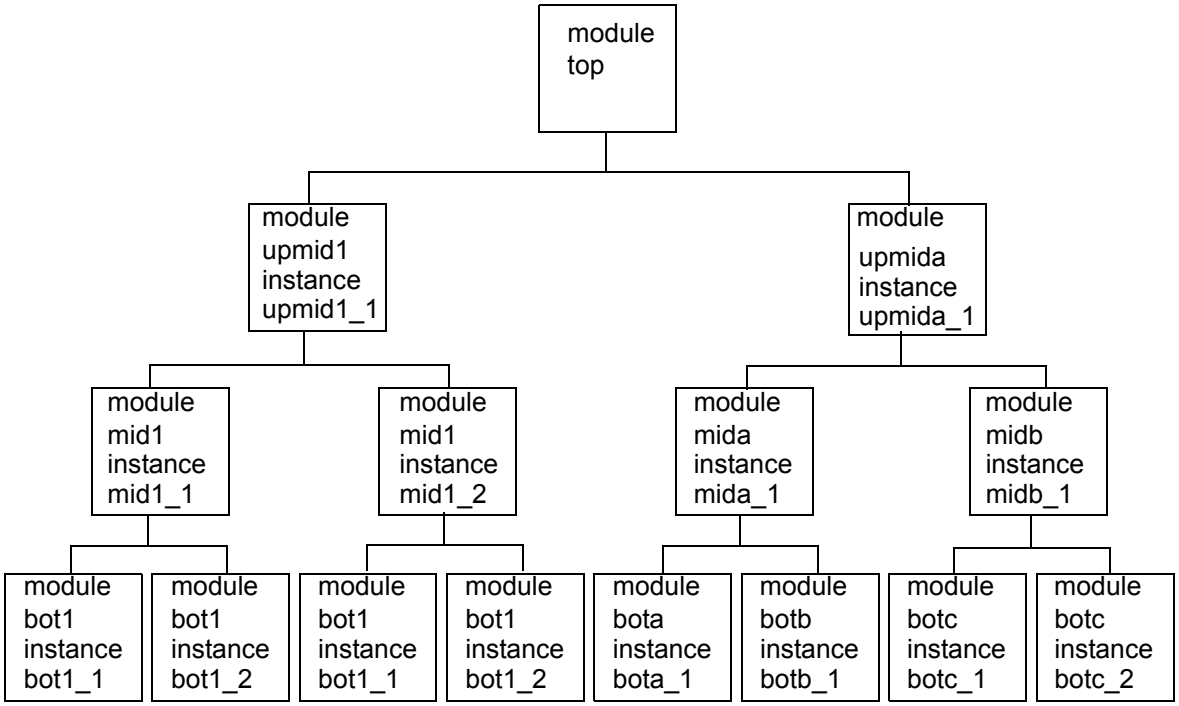
- specific module instances specified by their hierarchical name
- specific signals also specified by their hierarchical name

You can use the configuration file to specify both Radiant technology optimizations and two or four state simulation to modules.

Configuration File Examples

The following are configuration file examples that apply to the example design hierarchy shown in Figure 13-2. In Figure 13-2 the module identifiers (another word for module names) and instance identifiers (names) are shown but the configuration file only uses module identifiers. For these examples the entire design is in design.v.

Figure 13-2 Design Hierarchy



module statement example

If the configuration file, named design.config, contained the following:

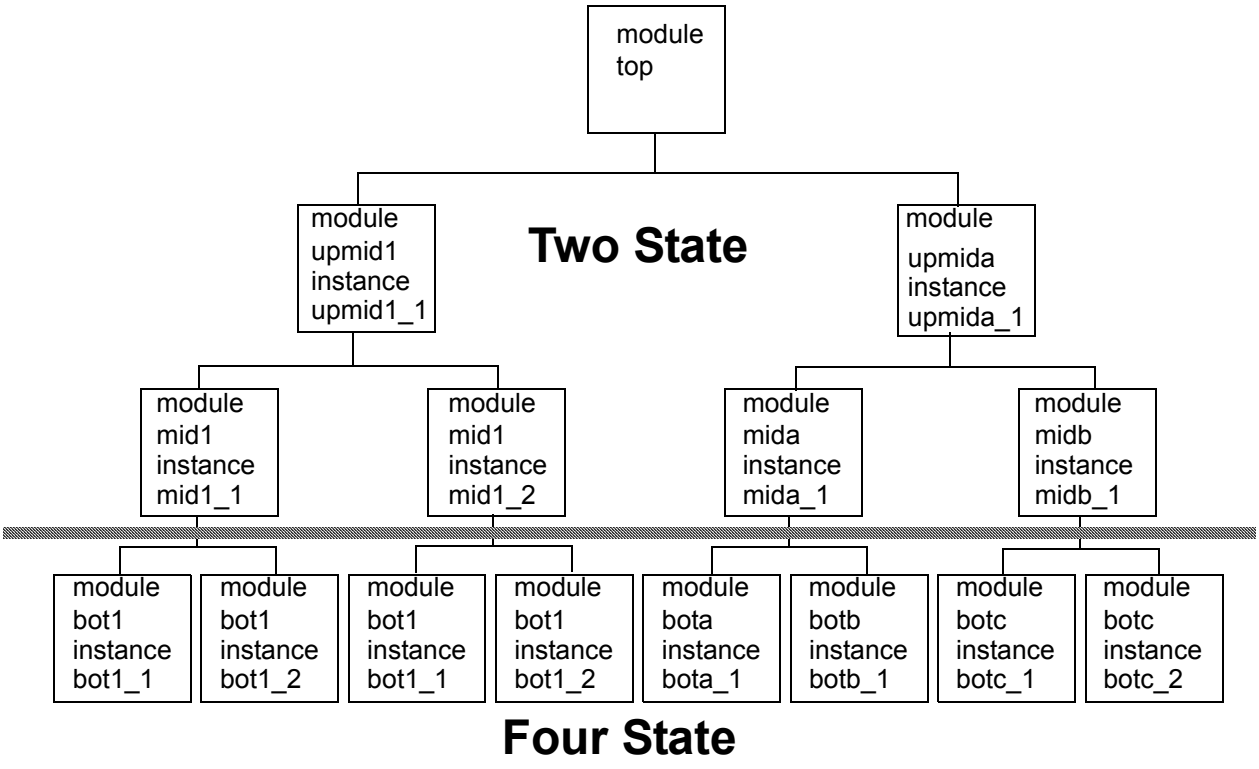
```
module {bot1,bota,botb,botc} {4value};
```

When you enter the following command line:

```
vcs design.v +2state +optconfigfile+design.config
```

The design is partitioned into two and four state simulation as shown in Figure 13-3.

Figure 13-3 module statement partition



tree statement example

If the configuration file contains the following tree statement:

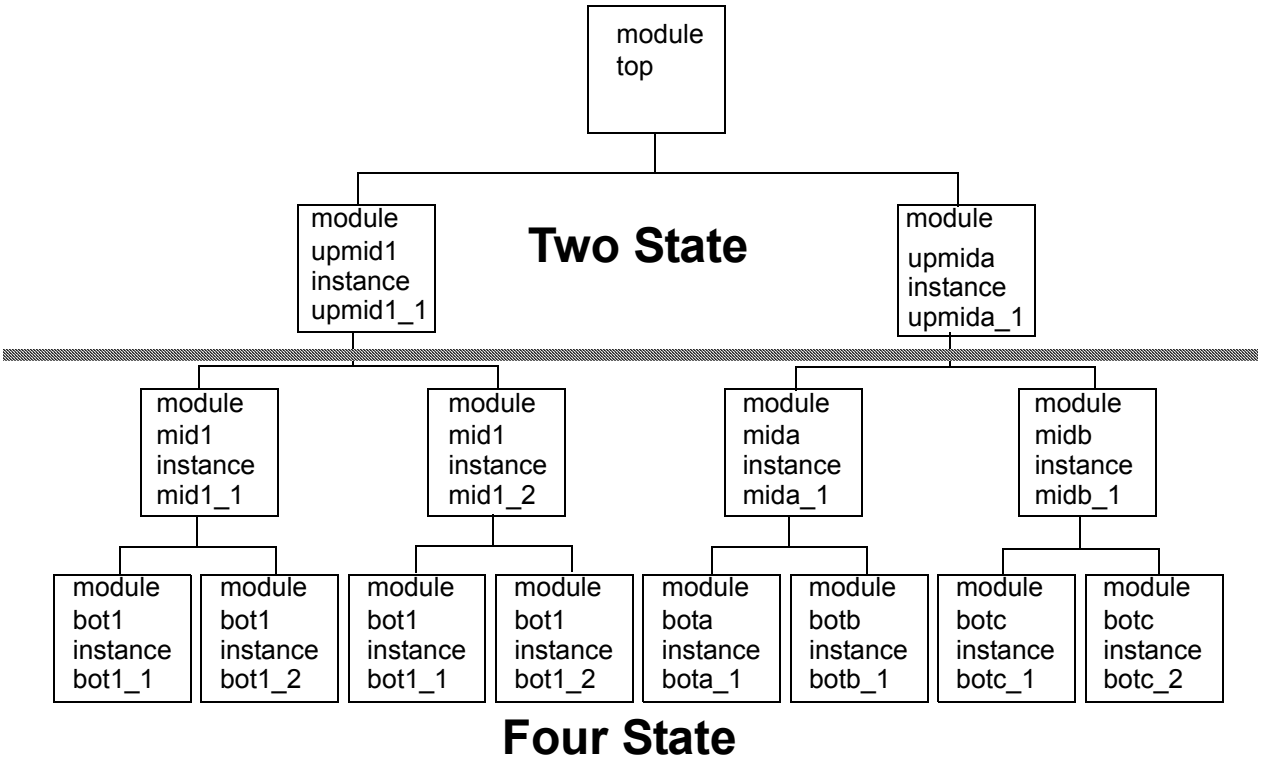
```
tree {mid1,mida,midb} {4value};
```

When you enter the following command line:

```
vcs design.v +2state +optconfigfile+design.config
```

The design is partitioned into two and four state simulation as shown in Figure 13-4.

Figure 13-4 tree statement partition



tree statement with depth specification example

If the configuration file contains the following tree statement:

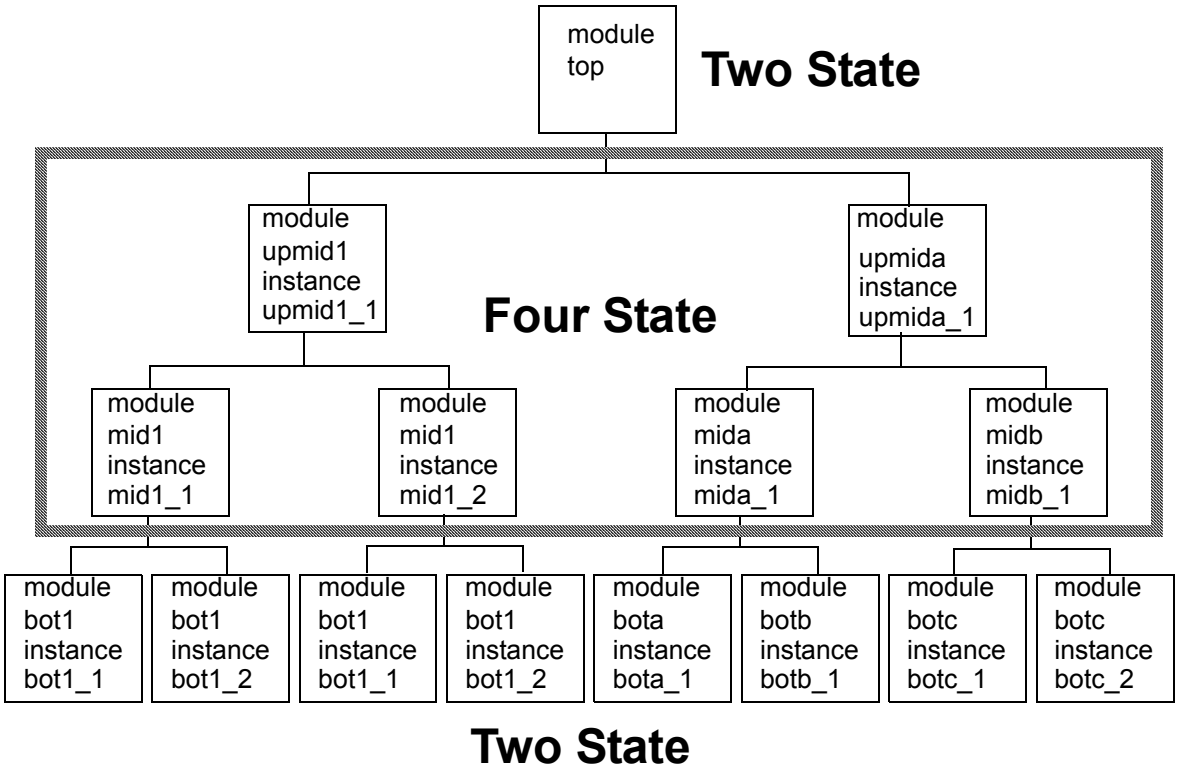
```
tree (1){upmid1,upmida} {4value};
```

When you enter the following command line:

```
vcs design.v +2state +optconfigfile+design.config
```

The design is partitioned into two and four state simulation as shown in Figure 13-5.

Figure 13-5 tree statement with depth specification partition



A depth level of 1 means that four state simulation applies to the instances of the specified modules and all module instances one level down in the hierarchy.

tree statement with negative depth specification example

If the configuration file contains the following tree statement:

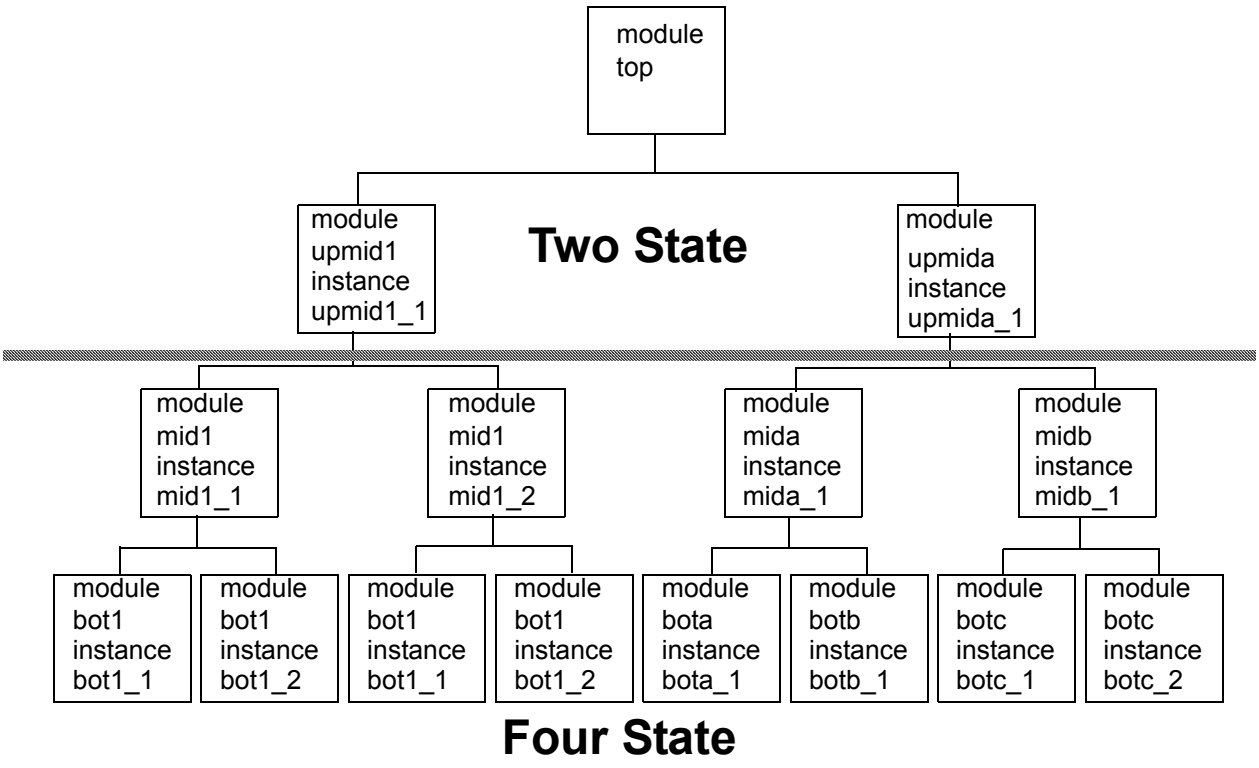
```
tree (-2){top} {4value};
```

When you enter the following command line:

```
vcs design.v +2state +optconfigfile+design.config
```

The design is partitioned into two and four state simulation as shown in Figure 13-6:

Figure 13-6 tree statement with negative depth specification partition



VCS descended the hierarchy under module top to the leaf level to apply two state simulation and then from that level ascended to a second level to also apply two state simulation.

Four State Simulation Can Extend Across Design Partition

When you use the configuration file to specify four state simulation for module definitions whose instances are on one layer of the design hierarchy, VCS also assigns what you specify to nets and registers in the adjacent layers that attach to ports in the modules you specify.

Consider the following code:

```
module top;
reg r1,r2;
wire w1,w2;
four_state fs1 (w1,r1);
 ⋮

endmodule

module four_state(fsout,fsin);
output fsout;
input fsin;
reg fsr1;
bottom bot1 (fsout, ,fsr1,fsin);
 ⋮

endmodule

module bottom (botout1,botout2,botin1,botin2);
output botout1,botout2;
input botin1,botin2;
 ⋮

endmodule
```

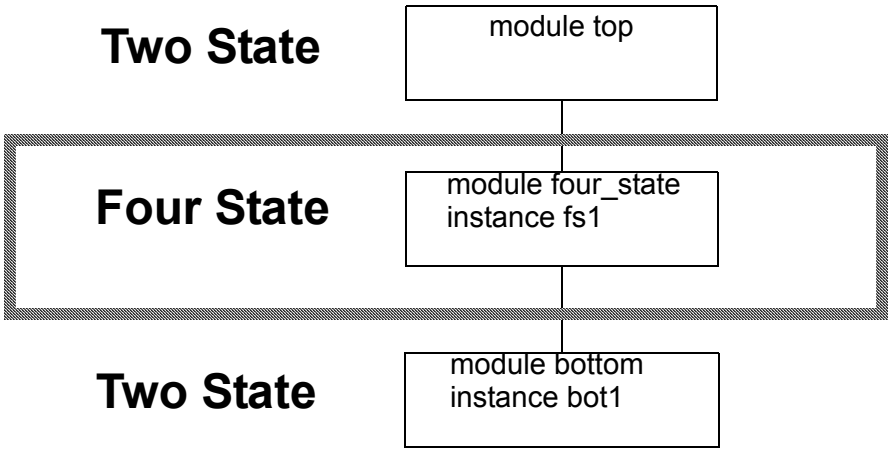
If the configuration file contains the following:

```
module {four_state} {4value};
```

The configuration file specifies four state simulation for module four_state.

The design is partitioned as shown in Figure 13-7.

Figure 13-7 Multi-Layer Partition for Two and Four State Simulation



The following signals, not declared in module four_state, retain four state simulation:

top.r1

top.w1

top.fs1.bot1.botout1

top.fs1.bot1.botin1

top.fs1.bot1.botin2

14

Using OpenVera Assertions

This chapter introduces the OpenVera Assertions (OVA) language, and explains how to compile and run OVA within VCS.

The following topics are covered:

- Introducing OVA
- OVA Flow
- Checking OVA Code with the Linter Option
- OVA Runtime Options
- OpenVera Assertions Post-Processing
- Viewing Output Results
- Inlining OVA in Verilog
- Using Verilog Parameters in OVA Bind Statements
- OVA System Tasks and Functions

Introducing OVA

OVA is a clear, easy way to describe sequences of events and facilities to test for their occurrence. With clear definitions and less code, testbench design is faster and easier, and you can be confident that you are testing the right sequences in the right way.

As a declarative method, OVA is much more concise and easier to read than the procedural descriptions provided by hardware description languages such as Verilog. With OVA:

- Descriptions can range from the most simple to the most complex logical and conditional combinations.
- Sequences can specify precise timing or a range of times.
- Descriptions can be associated with specified modules and module instances.
- Descriptions can be grouped as a library for repeated use. OVA includes a Checker Library of commonly used descriptions.

Built-in Test Facilities and Functions

OVA has built-in test facilities to minimize the amount of code that you need to write. In addition, OVA works seamlessly with other Synopsys tools to form a complete verification environment. OVA performs the following tasks:

- Tests Verilog, VHDL, and mixed-HDL designs using VCS and VCS-MX.
- Automatically tests and reports results on all defined sequences. You just write the definitions.

- Produces results that can be viewed with DVE or VirSim.
- Can be monitored and controlled as part of a Vera testbench.

VCS also has functional coverage that provides you with code coverage information about your OVA code.

Using OVA Directives

OVA uses two directives:

- The `assert` directive, consists of mostly temporal expressions and is used to define a property of a system that is monitored to provide the user with a functional validation capability. Properties are specified as temporal expressions, where complex timing and functional relationships between values and events of the system are expressed.
- The `cover` directive consists of event coverage expressions used to record all successful matches of the coverage expression. When the event expression results in a match, the cover always increments a counter. Multiple matches per attempt may be generated and reported. With compile time option `-ova_enable_diag`, if the match is the first success of the attempt, then the cover directive also increment a second counter first_matches.

How Sequences Are Tested Using `assert`

Testing starts with a *temporal assertion file*, which contains the descriptions of the sequences and instructions for how they should be tested. OVA is designed to resemble Verilog with similar data types, operators, and lexical conventions.

A typical temporal assertion file consists mostly of *temporal expressions*, which are the descriptions of the event sequences. *Events* are values or changes in value of any Verilog regs, integers, or nets. (Events declared in Verilog cannot be part of an expression.) Temporal expressions can be combined to form longer or more complex expressions. The language supports not only linear sequences but logical and conditional combinations.

The basic instruction for testing is a *temporal assertion*. Assertions specify an expression or combination of expressions to be tested. Assertions come in two forms: *check*, which succeeds when the simulation matches the expression, and *forbid*, which succeeds when the simulation does not match.

The temporal expressions and assertions must also be associated with a clock that specifies when the assertions are to be tested. Different assertions can be associated with different clocks. A clock can be defined as posedge, negedge, or any edge of a signal; or based on a temporal expression. Also, asynchronous events can use the simulation time as a clock.

An assertion can be associated with all instances of a specified module or limited to a specific instance.

Example 14-1 shows an example temporal assertion file. It tests for a simple sequence of values (4, 6, 9, 3) on the device's output bus.

Example 14-1 Temporal Assertion File, cnt.ova

```
/* Define a unit with expressions and assertions (or select
one from the Checker Library).
*/
unit step4
  #(parameter integer s0 = 0)      // Define parameters
  (logic clk, logic [7:0] result); // Define ports

  // Define a clock to synchronize attempts:
```

```
  clock posedge (clk)
  {
    // Define expressions:
    event t_0 : (result == s0);
    event t_1 : (result == 6);
    event t_2 : (result == 9);
    event t_3 : (result == 3);
    event t_normal_s: t_0 #1 t_1 #1 t_2 #1 t_3;
  }

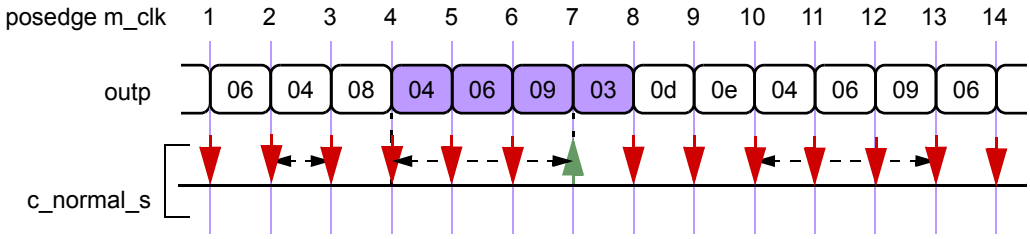
  // Define an assertion:
  assert c_normal_s : check(t_normal_s, "Missed a step.");

endunit

/* Bind the unit to one or more instances in the design.
*/
// bind module cnt : // All instances of cnt or
bind instances cnt_top.dut : // one instance.
  step4 start_4   // Name the unit instance.
  #(4)            // Specify parameters.
  (m_clk, outp); // Specify ports.
```

When the temporal assertion file is compiled and run with a simulator, the assertions are continuously tested for the duration of the simulation. New attempts to match each assertion to the simulation's values are started with every cycle of the assertion's associated clock. Each attempt continues until it either fails to match or succeeds in matching the complete expression. See Figure 14-1. The up arrow at clock tick 7 indicates a match that started at tick 4. The down arrows are failures. The failure or success of each attempt is logged to a file that can be reviewed later.

Figure 14-1 Assertion Attempts for cnt.ova



A Vera testbench can monitor and control the testing. Using built-in object classes, you can stop and start attempts to match the selected assertion; monitor attempts, failures, and successes; and synchronize the testbench with the testing process.

How Event Coverage Is Tested Using `cover`

The `cover` directive records only successful matches. You can specify cover directive specific to your design or use it with assertions statements.

With the default compile time options, only one counter is generated for each cover directive. This counter is incremented each time the event expression matches. At the end of a default simulation, the number of total matches is reported in the example:

```
unit_instance_name cover_name, int_val total match
```

You can also increment a second counter using the compile time option `ova_enable_dialog`. In this case, the first counter is the same as the default counter, and the second counter reports the number of total matches of the event expression.

OVA Flow

OVA uses the following flow:

1. Create a temporal assertion or cover file. See the OpenVera Assertions Language Reference Manual ($VCS_HOME/doc/UserGuide/ova_lrm.pdf).

   Start with simple temporal expressions and then combine them to form complex sequences. Simple expressions compile and run faster, and might use less memory.

   Files named with an .ova extension (*filename*.ova) are recognized as assertion and cover files.

2. Compile and simulate the design including Temporal Assertions files and options on the vcs and simv command lines.

3. After running the simulation, verify the results:

   - See “Viewing Results in a Report File” on page 14-40.

   - See “Viewing Results with Functional Coverage” on page 14-41.

   - Results can also be monitored through a Vera testbench.

Checking OVA Code with the Linter Option

The linter option adds predefined rules Rule violations other than syntax / semantic errors are not errors in the strict sense of the word, but are warnings of potential performance issues or differences in intended and real semantics of the OVA expressions.

The linter option has two sets of rules:

- General Rules (GR) that are applicable to both simulation and formal verification tools, synthesis tools such as VCS and verification tools
- Magellan Rules (MR) that are specific to Magellan (or other similar formal verification tools).

Upon detecting a violation of any one of the rules and normal parsing errors, the linter will output a message in the same format as the OVA parser does now. It will prefix the message by ERROR, WARNING or RECOMMENDATION, depending on the case as detailed below.

The rules as listed next are classified as "e" for ERROR, "w" for WARNING, and "r" for RECOMMENDATION.

Each rule contains two information blocks. The first one is the error message text output by the linter. The second provides further information on the kind of problem identified.

Applying General Rules with VCS

The linter option in VCS is `-ova_lint`.

If used without any Verilog design file, only the OVA files are analyzed (including bind statements), and potential problems are reported. The command is:

```
vcs ova_files_list -ova_lint
```

If used with Verilog and OVA files (or inlined OVA), the compilation proceeds normally while detailed linting analysis is also done. If no fatal error is found, the simulation can go ahead and any recommendations from the linter can be incorporated for later runs.

Linter General Rule Messages:

This section lists the messages generated by the general rules and describes the condition that caused the message along with a suggested remedy.

GR1: WARNING "assert forbid" used on an event that contains an "if" without an "else".

Example:

```
event e: if a then #1 b;
assert c: forbid(e);
```

Whenever "a" is false, the assertion will fail because the "if - then" implication is satisfied. This may not be, however, what is intended. A modification to consider is to change the event definition as follows:

```
event e: a #1 b;
```

GR2: WARNING "ended" or "matched" is used on an event that contains an "if" without an "else"

Example:

```
event e1: if a then #1 b;
event e2: if ended e1 then #1 c;
```

Whenever "a" is false, event "e1" will match because the "if - then" implication is satisfied. Which means that "e2" will also trigger and try to match on "c" at the next clock tick. This may not be, however, what is intended. A modification to consider is to change event "e1" as follows:

```
event e1: a #1 b;
```

GR3: WARNING "if" appears in the middle of a longer sequence, or a composition of sequences where both contain an "if" without an "else"

Example:

```
event ev: if a then #1 (if b then #1 c);
```

If "a" is true, then in the next cycle event e will match even if "b" is false. This may not be the intended behavior.

Consider changing the event as follows:

```
event ev: if a then (#1 b #1 c);
```

Note 1: The portion of the original event "ev" in parentheses could have been an event declared separately and instantiated in "e", thus possibly hiding the fact that it contains an "if".

Note 2: The following use of if-then-else can be useful, however:

```
event ev: if a then #1 if b then d else e;
```

where a, b, c, d and e are some boolean expressions.

GR6: WARNING OVA * repetition factor contains a 0.

Example:

```
a*[m..n] #k b;
```

where m is a 0 or a parameter with default value of 0

Please consider rephrasing the expression using for instance a disjunction, e.g., if m is 0 then (a*[1..n] #k b) || (#(k-1) b);

GR8: WARNING "matched" used in the same clock domain (one clock tick delay)

Example:

```
clock posedge clk {
        event e1: ... ;
        event e2: if matched e1 then ... ;
}
```

The successful match on "e1" would only be detected in "e2" at the subsequent posedge of "clk". Consider changing "e2" as follows:

```
event e2: if ended e1 then ... ;
```

The "ended" operator transfers the match of "e2" to "e2" at the same "posedge clk".

GR11: RECOMMENDATION event contains a large delay or repetition interval.

Example:

```
event e1: if posedge a then a*[1..1000] #1 b;
```

or

```
event e2: a #[1..10000] b;
```

or

```
event e3: a #10000 b;
```

Consider using a variable to count clock ticks if there are no overlapped transactions.For example, the first case:

```
logic [9:0] cnt = 11'b0;
clock ... {
        cnt <= reset ? 10'd0 :
                        posedge a ? 10'd1 :
                                cnt > 10'd1000 ? cnt :
                                        cnt = cnt + 10'd1;
event e1: if posedge a then ((cnt <= 11'd1000) && a) * [1..] #1 b;
}
```

Alternately, if overlaps are possible consider using time stamps stored in a queue (see for instance the OVA standard checker unit "ova_req_ack_unique").

GR14: RECOMMENDATION top-level conjunctions over events in "check" assertions.

Example:

```
clock posedge clk {
   event e1: ...;
   ...
   event eN: ...;
   event e: if a then e1 && e2 && e3 && ... ;
}
assert c: check(e);
```

Consider placing an assertion on each individual event as follows:

```
assert c1: check (if a then e1);
assert c2: check (if a then e2);
assert c3: check (if a then e3);
...
```

This creates more assertions but they may execute faster because they can be considerably simpler than event "e".

GR15: RECOMMENDATION Simulation time is used as the sampling clock

The event used in an assertion does not have a sampling clock. The simulation time will be used in that case which may lead to inefficient simulation. Consider whether such fine sampling is required in your application.

GR17: RECOMMENDATION for loop over events / assertions with large (>20) ranges with loop index appearing in delay or repetition operators.

Consider using variables and / or compacting into a single event. For an example see the OVA standard checker "ova_req_ack_unique". It contains two versions of a checker, version 0 that is suitable for small ranges of parameters, and version 1 that uses a time stamp and a queue when the loop ranges become large (thus creating too many events / assertions).

GR19: WARNING the "past" operator is over a long time interval (> 1000).

Consider if the same property can be expressed differently without the use of a deep look into the past (or in the future using # delay which would have a similar problem.)

GR25: WARNING a conditional check assertion that involves open-ended delay interval in the consequent (unbounded eventuality).

Example:

```
event e: if c then some_sequence1 #[1..] some_sequence2;
```

This assertion cannot fail in a simulation because of the failure of "some_sequence2" because "some_sequence2" could occur after the simulation ends.

Consider placing an upper bound on the delay interval as follows:

```
event e: if c then some_sequence1 #[1..20] some_sequence2;
```

GR26: WARNING OVA variable not initialized to a known value.

Consider initializing the variable to 0. For Example:

```
logic v = 1'b0;
```

This may be needed if the OVA checker is to be used with formal tools.

GR27: WARNING assert check over a non-bool sequence that does not start with an "if".

Example:

```
event e: a #1 b;
assert c: check(e);
```

The assertion will fail any time "a" is false. Unless used as a coverage assertion that is supposed to track the occurrences of the sequence "a" followed by "b" in the simulation trace, or when the sampling clock is some irregular event from the design, the usefulness of the assertion as a checker should be reconsidered, as it would require "a" to hold at every clock tick in order to have even a chance to succeed.

GR31: RECOMMENDATION multiple attempts may be triggered for the same check.

Example:

Suppose that "req" must return to 0 between activations, and if asserted it must remain so until "ack" is received, then the following sequence and an assertion on it would create unnecessary additional attempts to be triggered for the same req-ack transaction:

```
event e: if req then req*[1..] #0 ack;
```

Consider replacing it by:

```
event e: if posedge req then req*[1..] #0 ack;
```

The modified event will generate only one non-vacuous attempt for each assertion of "req".

GR32: WARNING the unit instance name in a bind statement is missing.

The problem is that an automatically generated name is inserted by the compiler which makes it more difficult for the user to locate the checker instance.

GR34: WARNING multiplication *N (by a constant) is the last operator on an expression - consider changing to repetition *[N].

Example:

```
event e: a #[1..] b*3;
```

Often [] is omitted by accident from the repetition operation. Consider changing the expression to event e:

```
a #[1..] b*[3];
```

GR35: WARNING a bitwise operator (&, |, ~, etc.) is used on a boolean expression.

Example:

```
logic a; logic b;
event e: (a & b) == 0 #1 (~b == 0);
```

The problem is that with bitwise operations the word extension to 32 bits as implied by the "0" operand and the bitwise operations may produce unwanted results. Consider rewriting as follows:

```
event e: (a && b) == 0 #1 (!b == 0);
```

GR36: RECOMMENDATION open-ended interval delay used in an event to which ended or matched is applied.

Example:

```
event e1: a #[1..] b;
event e2: ended e1 #1 ... ;
```

The problem is that any evaluation attempt that matches on "a" will remain active till the end of simulation, waiting for (yet another) occurrence of "b". Most likely this is not intended. Consider rewriting "e1" to terminate on the first match of "b" as follows:

```
event e1: a #1 (!b)*[0..] #1 b;
```

Applying Magellan Rules for Formal Verification

This section describes use of Magellan Rules (MR) for checking OVA code to be used with formal verification tools such as Magellan.

The compile-time option for enabling MR rules is `-ova_lint_magellan`.

Linter General Rule Messages:

This section lists the messages generated by the general rules and describes the condition that caused the message along with a recommended alternate model.

MR1: ERROR the unit instance name in a bind statement is missing. The instance name must be provided.

MR2: WARNING an assertion is stated solely over an OVA variable value.

Example:

```
logic [bw-1:0] tmp = 1'b0;
clock posedge clk {
         tmp <= c1 ? reg_A :
                        c2 ? reg_B;
         event e: |tmp == 1'b1;
}
```

```
 assert c: check(tmp);
```

This type of an assertion cannot be used effectively as an assumption in Magellan because it requires constraining the signals driving the OVA variable "tmp" in the past clock tick. This is not possible when doing random simulation constrained by OVA assertions. The result is that no constraint is imposed. Consider correcting the assertion as follows:

```
logic [bw-1:0] tmp;
assign tmp = c1 ? reg_A :
                      c2 ? reg_B;
 clock posedge clk {
          event e: |tmp == 1'b1;
 }
 assert c: check(tmp);
```

The constraint is now applied in the current clock cycle rather than in the past one.

MR3: WARNING the "matched" or "ended" operator appears in the consequent sequence of a "check" assertion or as part of the sequence in a "forbid" assertion.

The problem is that this form of an assertion cannot be used effectively in random simulation under OVA constraints / assumption because it requires constraining inputs in past clock cycles.

Example:

```
clock posedge clk {
        event e1: c #1 d;
        event e2: if a then matched e1;
}
assert c: check(e2);
```

Try to rewrite event "e2" without the use of "matched". In the case of our simple example, the solution is simple due to the one cycle delay introduced by "matched":

```
clock posedge clk {
//      event e1: c #1 d; -- do not use
        event e2: if a then c #1 d;
}
assert c: check(e2);
```

In general the transformation may not be as simple as in the above example. It may be preferable to approach the problem differently right from the start rather than trying to rewrite the case later.

MR5: ERROR simulation time is used as the OVA clock.

The notion of simulation time is not available in Magellan. You may create an explicit periodic clock that provides some notion of time advancement, however.

MR6: ERROR case equality is used in expressions. This is non-synthesizable.

MR7: ERROR comparisons with 'z' and 'x' values is used. This is non-synthesizable.

MR8: WARNING an uninitialized OVA variable is used.

An uninitialized variable may cause a simulation - formal mismatch due to differences in interpreting the initial unknown value "x".

Compiling Temporal Assertions Files

Temporal assertions files are compiled concurrently with Verilog source files. You can use a set of OVA-specific compile-time options to control how VCS compiles the temporal assertions files.

Note:

When you use the OVA compile-time options, VCS creates a Verification Database directory in your current directory (by default named simv.vdb). VCS writes intermediate files and reports about OpenVera Assertions in subdirectories within in this directory.

The following compile-time options are for OVA:

`-ovac`

Starts the OVA compiler for checking the syntax of OVA files that you specify on the vcs command line. This option is for when you first start writing OVA files and need to make sure that they can compile correctly.

`-ova_cov`

Enables viewing results with functional coverage.

`-ova_cov_events`

Enables coverage reporting of expressions.

`-ova_cov_hier` *`filename`*

Limits functional coverage to the module instances specified in *filename*. Specify the instances using the same format as VCS coverage metrics. If this option is not used, coverage is implemented on the whole design.

`-ova_debug | -ova_debug_vpd`

Required to view results with DVE or VirSim.

`-ova_dir` *`pathname`*

Specifies an alternative name and location for the Verification Database directory. There is no need to specify the name and location of the new Verification Database directory at runtime, the simv executable contains this information.

If you move or rename this directory after you create the simv executable, you include this option at runtime to tell VCS its new name or location.

`-ova_file` *`filename`*

Identifies *filename* as an assertion file. Not required if the file name ends with .ova. For multiple assertion files, repeat this option with each file.

`-ova_filter_past`

For assertions that are defined with the past operator, ignore these assertions where the past history buffer is empty. For instance, at the very beginning of the simulation the past history buffer is empty. So, a check/forbid at the first sampling point and subsequent sampling points should be ignored until the past buffer has been filled with respect to the sampling point.

`-ova_enable_diag`

Enables further control of result reporting with runtime options.

Used with the cover directive, `ova_enable_diag` generates a second counter to report the number of times an attempt to match the event expression succeeds for the first time.

The counters are reported in the form

```
unit_instance_name cover_name,  int_val  total match,
int_val  first match
```

For example it can be

top.\gen_2.aova2 .cover_temp_no_vacuous_f_eq_t, 4 total match, 4 first match

`-ova_inline`
Enables compiling of OVA code that is written inline with a Verilog design.

Note:
You can also use the VCS `-f` option to specify a file containing a list of all absolute pathnames for Verilog source files and compile-time option. You can pass all OVA compile-time options through this file, except `-ova_debug`.

OVA Runtime Options

The following runtime options are available for use with OVA:

`-ova_quiet [1]`
Disables printing results on screen. The report file is not affected. With the 1 argument, only a summary is printed on screen.

`-ova_report [`*`filename`*`]`
Generates a report file in addition to printing results on your screen. Specifying the full path name of the report file overrides the default report name and location, which is ./simv.vdb/report/ova.report.

`-ova_verbose`
Adds more information to the end of the report including assertions that never triggered and attempts that did not finish, and a summary with the number of assertions present, attempted, and failed.

A set of runtime options are also available for controlling how VCS writes its report on OpenVera Assertions. You can use these options only if you compiled with the `-ova_enable_diag` compile-time option.

`-ova_filter`

Blocks reporting of trivial if-then successes. These happen when an if-then construct registers a success only because the if portion is false (and so the then portion is not checked). With this option, reporting only shows successes in which the whole expression matched.

`-ova_max_fail` *N*

Limits the number of failures for each assertion to *N*. When the limit is reached, the assertion is disabled. *N* must be supplied, otherwise no limit is set.

`-ova_max_success` *N*

Limits the total number of reported successes to *N*. *N* must be supplied, otherwise no limit is set. The monitoring of assertions continues, even after the limit is reached.

`-ova_simend_max_fail` *N*

Terminates the simulation if the number of failures for any assertion reaches *N*. *N* must be supplied, otherwise no limit is set.

`-ova_success`

Enables reporting of successful assertion matches in addition to failures. The default is to report only failures.

For a `cover` statement, triggers the match (success) message in the following format:

```
Ova [i][j]: ".fileName", <line>: <hierCoverName> started
at 900s covered at 900s [optional custom msg]
```

where `i` and `j` are the severity and category values associated with the cover statement, respectively. `ova_success` is under the control of `-ova_quiet`.

Functional Code Coverage Options

Functional coverage is code coverage for your OVA code. With functional coverage enabled, the `cover` statement is treated in the same manner as an `assert` statement. The runtime options are enabled by the `-ova_cov` compile-time option. These runtime options are as follows:

`-ova_cov`
: Enables functional coverage reporting.

`-ova_cov_name` *`filename`*
: Specifies the file name or the full path name of the functional coverage report file. This option overrides the default report name and location. If only a file name is given, the default location is used resulting in: ./*simv*.vdb/fcov/*filename*.db.

`-ova_cov_db` *`path/filename`*
: Specifies the path and filename of an initial coverage file. An initial coverage file is needed to set up the database. By default, an empty coverage file is loaded from the following directory: simv.vdb/snps/fcov.

OpenVera Assertions Post-Processing

This section describes use of OpenVera Assertions post-processing (OVAPP) functionality. It contains the following topics:

- Overview
- OVAPP Flow
- Building and Running a Post-Processor

- Multiple OVA Post-Processing Sessions in One Directory

Overview

You can use VCS to build a post processor from a compiled design and temporal assertion files. You then run the post-processor, using either DVE or VirSim or the post-processor CLI supplied with OVAPP, as you would a simulation compiled with the design and OVA files together. This approach allows you to:

- Post-process a compiled design several times with different temporal assertions files each time. You can observe the effects of different assertion scenarios and collecting several distinct sets of functional coverage data.
- Develop assertions incrementally over a series of post-processing runs, improving and augmenting the assertions in the process.

OVAPP Flow

The following steps show a typical flow for post-processing a compiled VCS design with temporal assertions.

1. To use the post-processor CLI as the debugging tool, include the `$vcdpluson` or the `$dumpvars` system task in your Verilog code.
2. Compile your design in VCS with the `-ova_PP` compile-time option.

   Note: Use the `$vcdpluson` or the `$dumpvars` system task in your Verilog code to create dump files and enable CLI functionality. Do not use the `-ova_debug` compile-time option.
3. Simulate the design to create a VPD or VCD file.

4. Build the post-processor.
5. Run the post-processor using DVE or VirSim or the post-processor CLI.

Building and Running a Post-Processor

The procedure to build and run a post-processor is as follows:

1. Include either of the following in your Verilog code:
   - The `$vcdpluson` system task to tell VCS to write a VPD file during simulation.

     Note: Use the -PP compile-time option to write the VPD files as described in the following step.
   - The `$dumpvars` system task to tell VCS to write a VCD file during simulation.

   If VCS writes a VCD file, the post-processor will call the vcd2vpd utility to translate it into a VPD file.

Note: Do not use the `-ova_PP` compile-time option to generate design dumps for OVAPP. Such dumps will not contain all the correct hierarchies needed.

2. Compile your design with the `-ova_PP` compile-time option, for example:

```
vcs -f filename -ova_PP [-PP] [-o simv_name]
[-ova_dir directory_path]
```

In this example compilation command line:

`-f filename`
: Specifies a file containing the source files, and perhaps compile-time options. This compile-time option is not specifically related to OVA post-processing.

`-ova_PP`
: Tells VCS to record in the verification database directory design information that is used by the post-processor.

  By default VCS also creates the simv.vdb directory in the current directory when you include this option. We call this directory the verification database directory. VCS writes post-processing data in a subdirectory in the verification database directory

`-PP`
: Optional compile-time option that enables VCS to write a VPD file. This option also enables VCS to recognize the `$vcdpluson` system task. This compile-time option is not specifically related to OVA post-processing.

`-o simv_name`
: This compile-time option is not specifically related to OVA post-processing. It specifies an alternative name, and possibly a different location, for the simv executable. Because the executable and this directory have the same name (but not the same extension), you will alter the name and location of the verification database directory.

`-ova_dir directory_path`
Specifies an alternative name and location for the verification database directory. This option supersedes the `-o` option in naming and locating the verification database directory.

During compilation VCS displays the following message:

```
Generating OVA post-processing data ...
```

3. Simulate the design. There are no runtime options needed for the post-processor. As usual, if you used the `-o` compile-time option to specify the name of the simv executable, you enter this name, instead of simv, to start the command line.

   During simulation VCS writes the VPD file.

4. The next step is to build the post-processor from the post-processing data and one or more temporal assertions files. temporal assertions file. You specify building the post-processor and the temporal assertions file with the `-ova_RPP` compile-time option, for example:

   ```
   vcs -ova_RPP filename.ova...[-o simv_name] [-ova_dir
   directory_path] [-ova_cov]
   ```

   In this example compilation command line:

   `-ova_RPP`
   Tells VCS to compile the post-processing engine. VCS writes it in a subdirectory in the verification database directory (VDB).

   `filename.ova`
   Is a temporal assertions file whose assertions you want compiled into the post-processing engine. You can specify more than one temporal assertions file.

`-o simv_name`

If you included the `-o` option when you compiled your design, also include it on this command line to tell VCS where to look for the VDB where the new data generated by the current command/step will be written.

Important:

Including this option also creates the *simv_name*.daidir direct access interface directory. This directory enables you to use CLI commands during post-processing.

`-ova_dir directory_path`

Specifies the verification database directory that VCS searches for the information about your design that is used by the post-processor.

`-vdb_lib directory_path`

Specifies an alternative location for the VDB that VCS searches for data generated during a prior step.
VCS first searches the directory specified by the `-ova_dir` option, then the directory specified by this option. If it does not find this information in either directory, the compilation fails.
If you include this option without the `-ova_dir` option, VCS searches the directory specified by this option, then the simv.vdb directory in the current directory.

`-ova_cov`

Enables the post-processor to gather OVA functional coverage information.

5. The last step is to start the post-processor. You do this with an `ovapp` command line. Its syntax is as follows:

```
ovapp [-vdb_lib directory_path][-vpd filename.vpd]
[-vcd filename.vcd] [-cli [-daidir=pathname.daidir] |
-ova_report [filename] -ova_cov]
[-ova_name session_name] [-o simv_name]
```

[*other_OVA_options*]

Where:

`-vdb_lib directory_path`
Specifies the Verification database directory that contains the dynamic library, the post-processing engine.

`-vpd filename.vpd`
Specifies the VPD file. If the filename is vcdplus.vpd and it is in the current directory, you can omit this option.

`-vcd filename.vcd`
Specifies the VCD file. The post-processor will call the vcd2vpd utility to translate it to a VPD file name vcdplus.vpd. If the VCD file is named verilog.dump and it is in the current directory (if the current directory doesn't contain a file named vcdplus.vpd). You can omit this option and the post-process will use the verilog.dump file.

`-cli`
Specifies that post-processing starts with a command line prompt for entering CLI commands. These CLI commands are the same as the VCS CLI commands plus additional ones for OVA post-processing. See "OVA Post-Processing CLI Commands" on page -30.

`-daidir=pathname.daidir`
Specifies the direct access interface directory used by the post-processor for CLI commands.

`-ova_report [filename]`
The post-processor writes a report file in the simv.vdb/report directory. The report contains messages displayed during post-processing.The default name of this report is ova.report.

`-ova_cov`
Tells the post-process to also gather functional OVA coverage data.

`-ova_name session_name`
Changes the name, but not the extension, of the generated files in the verification database directory. Generated files are those specified by other options such as `-ova_report`. See “Using Multiple Post-Processing Sessions” on page -31.

`-o simv_name`
Specifies the executable name so the post-processor know the name and location of the post-processing engine and the verification database directory. If you used the `-o` compile-time option when you built the post-processor, you need to enter it again on this command line so that the post-processor will know the name and location of the dynamic library.

OVA Post-Processing CLI Commands

When you include the `-cli` option, the post-processor displays a CLI command prompt just like the VCS CLI command prompt:

```
cli_0>
```

You can enter any VCS CLI command at this prompt, such as those for moving up and down the hierarchy and displaying values. There are also special CLI commands for OVA post-processing. The special CLI commands are as follows:

`pp_fastforward time`
Advances post-processing to the specified simulation time.

`pp_rewind time`
Returns post-processing to the specified previous simulation time.

`ova_trace_off instance_hierarchical_name assertion_name time`

Disables the tracing of the specifies assertion in the specified instance, at the specified simulation time.

`ova_trace_off assertion_hierarchical_name`
Disables tracing of the specified assertion the next time.

`ova_trace_on instance_hierarchical_name assertion_name time`

Enables the tracing of the specified assertion in the specified instance, at the specified simulation time.

`ova_trace_on assertion_hierarchical_name`
Enables tracing of the specified assertion name the next time.

Using Multiple Post-Processing Sessions

You can repeatedly run the post-processor using different input (either different temporal assertion files or different waveforms). This section describes how to use the `-ova_name` option to generate a unique report for each session. For example:

```
vcs -ova_RPP first.ova

ovapp -ova_name first -ova_report

vcs -ova_RPP second.ova

ovapp -ova_name second -ova_report
```

After these two post-processing sessions, the simv.vdb/report directory contains:

```
first.report   second.report
```

Multiple OVA Post-Processing Sessions in One Directory

You can run multiple OVA post-process sessions in the same directory. While only one design should be simulated in any one directory, any number of OVA assertion sets and any number of stimulus waveform patterns can be executed against that one design from within the same working directory.

In this section, reference will be made to the following four-step post-process flow:

1. Capture waveforms from a simulation of the base design.
2. Compile the base design for post-processing (`vcs -ova_PP`).
3. Compile the OVA checkers against the base design (`vcs -ova_RPP`).
4. Replay the captured waveforms against the compiled checkers (`ovapp`).

Note that step 1 could also be run after step 2 using the executable generated as part of the compilation of the base design. Note also that step 3 could be run more than once with different OVA checkers and that step 4 could be run more than once with different captured waveform files as input.

In each of these steps, information generated by one step is read and used by the immediately succeeding step. In step 3, the skeleton design data generated in step 2 is used to compile the OVA checkers into an engine that will be used during the post-processing replay. In step 4, the compiled OVA runtime engine and checker database are used during the replay. All this intermediate data is stored in predetermined locations in the vdb directory.

You can select two separate vdb directories via the command line at each step in the flow. One of these directories takes the same basename as the simulation executable (derived from the `-o` option). This is the vdb directory into which the results of the current step are written.

Note:

In this discussion, the vdb directory derived from the basename given with the `-o` option will be referred to as simv.vdb for simplicity.

The other directory is a read-only directory that you can specify with the `-vdb_lib` option. This directory is used as the source for files generated by the previous step, but the files are not located in the simv.vdb directory (as specified by the `-o` option). If no `-vdb_lib` option is given, all intermediate files are expected to be found in the simv.vdb directory (the simple/default case).

To run multiple independent versions of either step 3 or step 4, each command must specify a directory into which it will write its results. Specify the directories with the `-o` *`basename`* option and argument (the OVA intermediate files, for example, will be written to a directory with the given *basename* plus the .vdb extension). The *basename* given for each unique run of each step must be unique.

Each command will also include a pointer to the vdb directory of the previous step against which this step is being performed. This path is specified with the `-vdb_lib` option. The arguments for both `-o` and `-vdb_lib` can include full path names if desired.

Here is an example of the overall flow:

1. `vcs -ova_PP -o simv1 verilog_files`
2. `vcs -ova_RPP -vdb_lib simv1.vdb -o simv2 ova_files`
3. `ovapp -vdb_lib simv2.vdb -o simv3 -vcd dumpfile`

As you can see, in step1, we pass simv1 as the basename for the intermediate output files. In step 2, we pass the vdb directory created as a result of step1 as the argument of the `-vdb_lib` option and give VCS a second basename, simv2, for the new intermediate file directory. When VCS tries to locate a file in the simv2.vdb directory (during step 2) and cannot find the file, it next checks the directory specified by the `-vdb_lib` option. This way, you can run many unique step 2 OVAPP compilations using the same base design without running the risk of overwriting necessary files generated by the base design compilation.

In addition to the `-o` and `-vdb_lib` options, other compile-time options must be taken into consideration when running multiple compilations, simulations, or post-process runs in the same directory. The following sections describe in detail some of the issues to consider.

Interactive Simulation

The initial compile can be used as an interactive/batch simulation, with or without OVA checking. Simply add `-PP` or `-I` to the step 1 compile command line.

Waveform dump files

The post-processing run requires a waveform dump file. This file can be either a VCD file or a VPD file. The file can be generated by any simulation which is based on the same design as was compiled in step1 (the `-ova_PP` step). The dump file can be generated by the executable resulting from the Step1 compile, an earlier interactive simulation of the same design, or some other simulation.

This dump file must contain, as a minimum, any signal referenced by the OVA checkers to be post-processed. The signals used by the OVA checkers must be included in a `$dumpvars` or `$vcdpluson` dumped scope (an instance specified in one of these system tasks) in order to be visible by the post-process checkers. Automated dumping of OVA signals is not supported.

The runtime option `-vcd` *`filename`* can be used to name the VCD file if there is a `$dumpvars` system task in the design. The `+vpdfile` *`filename`* option and argument can be used to name the VPD file if there is a `$vcdpluson` system task in the design.

If you include the signals referenced by the checkers in `$vcdpluson` and `-PP` or `$dumpvars, the referenced signals will be dumped at compile time`.

Note: Do not use -ova_debug to generate dump files, since the generated files will not contain all the right hierachies.

It is recommended that the dump files be explicitly named in all cases. Both the `simv` executable and the `ovapp` executable use the same default name for the dumpfile output. If you dump a VPD file by it's default name in step 1 and use this file as the input to `ovapp`, it is possible that the input dumpfile can be overwritten.

Note that the step 3 (`ovapp`) waveform input file is specified with the `-vcd` or `-vpd` options. `-vcd` *`vcdfile`* tells `ovapp` to read a VCD file, while `-vpd` *`vpdfile`* tells `ovapp` to read a VPD file. This `-vcd` option is NOT the same option used by VCS to name the output waveform file. Caution in using this option is recommended.

Note that OVAPP internally requires a VPD file. Therefore, if you pass a VCD file to OVAPP, it calls the vcd2vpd converter prior to the post-processing run. This means that using VPD files to capture the waveforms to be re-played will result in better overall performance.

Dumping Signals Automatically

If you include the OVA checkers to the compilation step before the waveform replay dump is generated, these signals will be included in the dump automatically.

Debugging

When the OVA files are compiled in step 2 (`-ova_RPP`), debugging is enabled by default, the `-ova_enable_diag` and `+cli` compile-time options are entered automatically. This is to support the OVAPP debugger.

PLI or other 3rd party tools

If the initial compilation includes PLI or other 3rd party interfaces that use PLI/DKI to interface to VCS, a daidir directory will be generated during the step1 compile. To keep from corrupting this directory, the `-o` option for the step 2 compile ***must*** name a different basename from that used for the step1 compile. For safety, it's best to always include the `-o` option and use a unique basename for each compilation.

Incremental compile

The step 2 OVA engine compile (`-ova_RPP`) uses incremental compile in order to speed up the compilation process for multiple runs. Incremental compile generates intermediate files in a directory named csrc by default. If you enable the incremental compile feature in the step1 compile, it is possible for the csrc intermediate files to be corrupted. If you use incremental compile, you should also add the `-Mdir=`*`dirname`* option to the command line to direct the VCS compiler to store it's intermediate files in a unique directory.

Inlined OVA

The step1 compile command can include the `-ova_inline` option to enable the capture and processing of OVA statements inlined as pragmas in the Verilog source. These pragmas are not copied to the skeleton design and thus will not be processed during the post-processing playback. This is in keeping with the strategy of post-processing, which allows playback one or more sets of signal waveforms against one or more sets of OVA checkers. Post-processing was designed to playback against checkers defined in independent stand-alone OVA files (those given on the step 2 (`-ova_RPP`) command line).

Inlined OVA statements will not, however, interfere with the post-processing execution of the OVA statements compiled in step 2. Inlined OVA may be freely included in the design for interactive/ debugging purposes, and inlined statements will simply be excluded from the post-processing runs.

Reporting

By default, the assertion report generated by the OVA engine is written into./simv.vdb/report/ova.report. In the case of multiple post-processing runs, there is a chance the file will be overwritten. For each run, it is suggested that the `-ova_report` *`name`*`.report` and `-ova_name` *`name`* options be used to ensure that any report files generated will be stored under unique names.

Coverage

To enable functional coverage, use the `-ova_cov` option during the Step 2 (`-ova_RPP`) compile (also enter `-ova_cov_events` to see coverage of events). During the post-processing run, the `-ova_cov` option must again be given (as a runtime option) to actually turn on coverage capture.

By default, the coverage from all post-processing runs with a given compiled OVA image is captured in a single database. If you need to generate reports for each post-process run separately, use the `-ova_cov_name` *`name`* option to assign a unique name to each post-processing run. The various databases are stored under the simv.vdb directory in either case. Coverage reporting can include a single post-processing run or a merged set of runs, as described in the OVA chapter of the *VCS/VCSi User Guide*.

Coverage reports are generated with the fcovReport utility. The fcovReport command line should include the `-ova_cov_db` *`vdbdir`* option to point to the directory where the global coverage database resides, and the `-ova_cov_report` *`name`* to point to the path and name of the report file.

Things to watch out for

- If you pass *path/name* (a full path) to `-ova_cov_report`, fcovReport does not automatically create the directories in *path*. If the *path* does not exist, the report is not created and no error message is generated.

- The `-ova_inline` compile-time option should not be included with the `-ova_PP` compile-time option. If both options are present on the same command line, and inlined OVA references to checker library elements are included in the design, an error message will result. This error can be ignored and both the interactive simulation and the post-process compile should run fine.

Note:
Inlined OVA is not recognized for post-processing.

- If you use the `+vpdfile+`*filename* option to name the debug VPD file in the `ovapp` step, an informational message referring to a dummy file will be emitted. This message can be safely ignored.

- If the design from step 1 (`-ova_PP`) is re-compiled, the step 2 OVA compilations might have to be re-run. If there are no structural changes to the design (no hierarchy changes and no added/deleted signals), it may not be necessary to re-compile the OVA files. However, the step 1 design re-compile should not be done while step 2 OVA compilations are running, because the design compile deletes and re-generates the skeleton design file, even if there are no changes to the actual design.

Viewing Output Results

There are several ways to view results of a simulation involving OVA. This section covers the following topics:

- Viewing Results in a Report File
- Viewing Results with Functional Coverage
- Viewing Results with VirSim

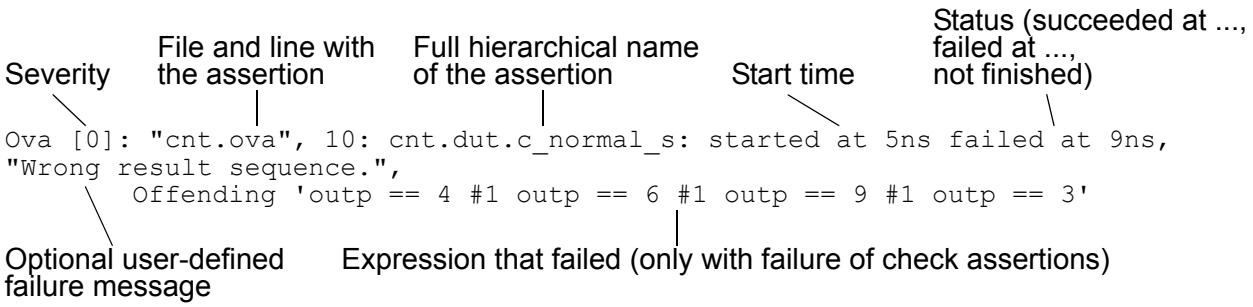
Viewing Results in a Report File

A report file is created when the `-ova_report` runtime option is used. The report file name is ova.report unless you specified a different name in the run command. This default ova.report file is stored in directory *simv*.vdb/report, where the .vdb directory is the root directory at the same level where the design is compiled and simv is stored.

To override this default name and location for the report file, use the `-ova_report` runtime option and provide the full path name of the report file.

The report file is replaced with each simulation run. If you want to save the results, be sure to rename the file before running the simulation again.

Assertion attempts generate messages with the following format:



Viewing Results with Functional Coverage

After running a series of simulations, you can generate a report summarizing the coverage of the assertions and expressions in two ways.

- With the default report, you can quickly see if all assertions were attempted, how often they were successful, and how often they failed. Potential problem areas can be easily identified. The report can cover one test or merge the results of a test suite. The report is presented in HTML and you can customize it with a Tcl script.
- An assertion and event summary report describes the total number of assertions and events details of their performance. This list can be filtered by category and severity to report matching assertions.

Using the Default Report

The default report shows the number of assertions and expressions that:

- Were attempted

- Had successes
- Had failures

Coverage is broken down by module and instance, showing for each assertion and expression, the number of attempts, failures, and successes. Because if-then constructs register a success anytime the if portion is false (and so the then portion is not checked), the report also shows the number of *real successes* in which the whole expression matched. This works with nested if statements too.

Functional coverage can also grade the effectiveness of tests, producing a list of the minimum set of tests that meet the coverage target. Tests can be graded on any of these metrics:

- Number of successful assertion attempts versus number of assertions (*metric* = SN)
- Number of failed assertion attempts versus number of assertions (*metric* = FN)
- Number of assertion attempts versus number of assertions (*metric* = AN)
- Number of successful assertion attempts versus number of assertion attempts (*metric* = SA)
- Number of failed assertion attempts versus number of assertion attempts (*metric* = FA)

To generate a report, run the following command:

```
fcovReport [options]
```

Assertion and Event Summary Report

Since no changes are introduced to the data collection process when generating functional coverage reports, you can produce different reports from a simulation. One report could show all assertions and events; another report could show assertions filtered by category and severity.

The assertion and event summary report generates four html files:

- report.index.html displays total assertions and events and details including:
  - Assertions with at least 1 real success
  - Assertions with at least 1 failure
  - Assertions with at least 1 incomplete
  - Assertions without attempts
  - Events with at least 1 attempt
  - Events with at least 1 real match
  - Events without any match or with only vacuous matches
  - Events without any attempts

  report.index.html also contains links to the other three files.
- tests.html describes the tests merged to generate the report.
- hier.html displays a hierarchical report table showing a list of instances, the number of assertions in each instance, and the number of events in each instance.

- category.html is generated when `-ova_cov_category` and/or `-ova_cov_severity` are used to filter results. Tables display functional coverage results for assertions showing the assertions having the category and severity specified along with number of attempts, successes, failures' and incompletes.

To generate the assertion and event summary report, run the fcovReport command after compilation and simulation:

```
fcovReport [-ova_cov_severity value,...]
[-ova_cov_category value,...]
```

Command Line Options

The command line options are as follows:

`-e TCL_script | -`
Use this option to produce a custom report using Tcl scripts or entering Tcl commands at standard input (keyboard). Most of the other fcovReport options are processed before the Tcl scripts or keyboard entries. The exception is -ova_cov_report, which is ignored. Its function should be in the Tcl.

`-e TCL_script`
Specifies the path name of a Tcl script to execute. To use multiple scripts, repeat this option with each script's path name. They are processed in the order listed.

`-e -`
Specifies your intent to enter Tcl commands at the keywoard.

The Tcl commands provided by VCS, that you can input to fcovReport for OVA coverage reports (see “Tcl Commands for SVA and OVA Functional Coverage Reports” on page 15-102), you can also input to assertCovReport for SystemVerilog assertion (SVA) coverage.

`-ova_cov_cover`
Specifies reporting of cover directive information only.

`-ova_cov_db` *`path`*
Specifies the path of the template database. If this option is not included, fcovReport uses simv.vdb.

`-ova_cov_events`
Specifies reporting only about OVA events.

`-ova_cov_grade_instances` *`target`*`,`*`metric`*
`[,` *`time_limit`*`]`
Generates an additional report, grade.html, that lists the minimum set of tests that add up to the target value for the metric (see previous page for *metric* codes). The grading is by instance.

`-ova_cov_grade_modules` *`target`*`,` *`metric`*
`[,` *`time_limit`*`]`
Generates an additional report, grade.html, that lists the minimum set of tests that add up to the target value for the metric (see previous page for *metric* codes). The grading is by module.

`-ova_cov_map` *`filename`*

Maps the module instances of one design onto another while merging the results. For example, use this to merge the functional coverage results of unit tests with the results of system tests. Give the path name of a file that lists the hierarchical names of from/to pairs of instances with one pair per line:

`from_name to_name`

The results from the first instance are merged with the results of the second instance in the report.

`-ova_cov_merge` *`filename`*

Specifies the path name of a functional coverage result file or directory to be included in the report. If *filename* is a directory, all coverage result files under that directory are merged. Repeat this option for any result file or directory to be merged into this report. If this option is not used, fcovReport merges all the result files in the directory of the template database (specified with `-ova_cov_db` or *simv*.vdb/snps/fcov by default).

`-ova_cov_report` *`name | path/name`*

Specifies the base name for the report.The fcovReport command creates an HTML index file at *simv*.vdb/reports/*name*.fcov-index.html and stores the other report files under *simv*.vdb/reports/*name*.fcov.

If you give a path name, the last component of the path is used as the base name. So the report files are stored under *path*/*name* and the index file is at *path*/*name*.fcov-index.html.
If this option is not included, the report files are stored under *simv*.vdb/reports/report.fcov and the index file is named report.fcov-index.html.

Customizing The Report with Tcl Commands

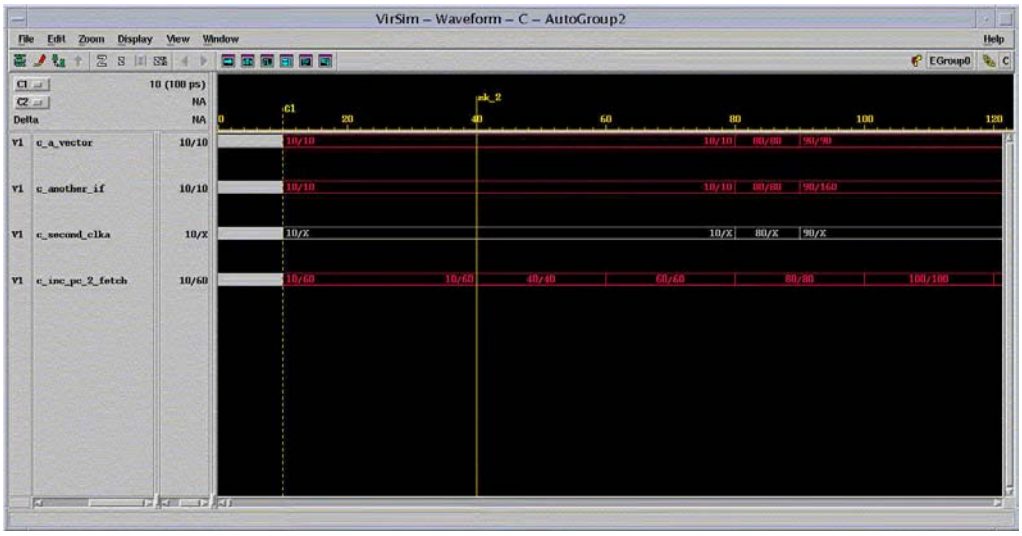
After you enter fcovReport you can enter Tcl commands to modify the report. These commands also work in assertCovReport that you use for SystemVerilog assertions coverage reports. See “Tcl Commands for SVA and OVA Functional Coverage Reports” on page 15-102.

Viewing Results with VirSim

If a design is compiled with the `-ova_debug` or `-ova_debug_vpd` option, the OVA results from a simulation run can be viewed in VirSim similar to the other simulation results. This section describes viewing OVA results and assumes that you are familiar with using VirSim. For more information, see the *VirSim User Guide*.

To view the assertion results, start VirSim and load the VPD file from the simulation run. Then go to the scope where the assertion is declared. The scope shows the assertion plus design signals and OVA variables related to the assertion. Drag the assertion and any other signals of interest into the Waveform Viewer. Use the features of VirSim to debug the results in the same way as debugging other simulation results. Figure 14-2 shows the Waveform Viewer with four assertions.

Figure 14-2 VirSim Display of Assertion Results



An assertion shows each evaluation attempt as a colored box: green for success, red for failure, gray for incomplete. A solid, gray bar from the beginning of the simulation time indicates that no attempts have started yet. The left edge of the box marks the start time of the attempt. However, because attempts can overlap, the right edge is not meaningful and the length of the box does not represent the length of the sequence. (The right edge is just the beginning of the next attempt.)

The start and end times are noted in the box as start/end. An end time of “X” means incomplete. An asterisk (*) means the left-most characters were truncated for lack of space. An empty box is also because of lack of space. To see the times, zoom in.

Expanding an assertion displays three component “signals” (see Figure 14-2):

- “clk” shows the ticks of the clock used by the assertion.

- “result” shows the result of each attempt: a green up arrow for success, a red down arrow for failure, a gray line for incomplete. The result markers are placed at the start times of the attempts.
- “end_time” notes the end time for each attempt. An “X” means incomplete.

Figure 14-2 VirSim Display with Expanded Assertion Results

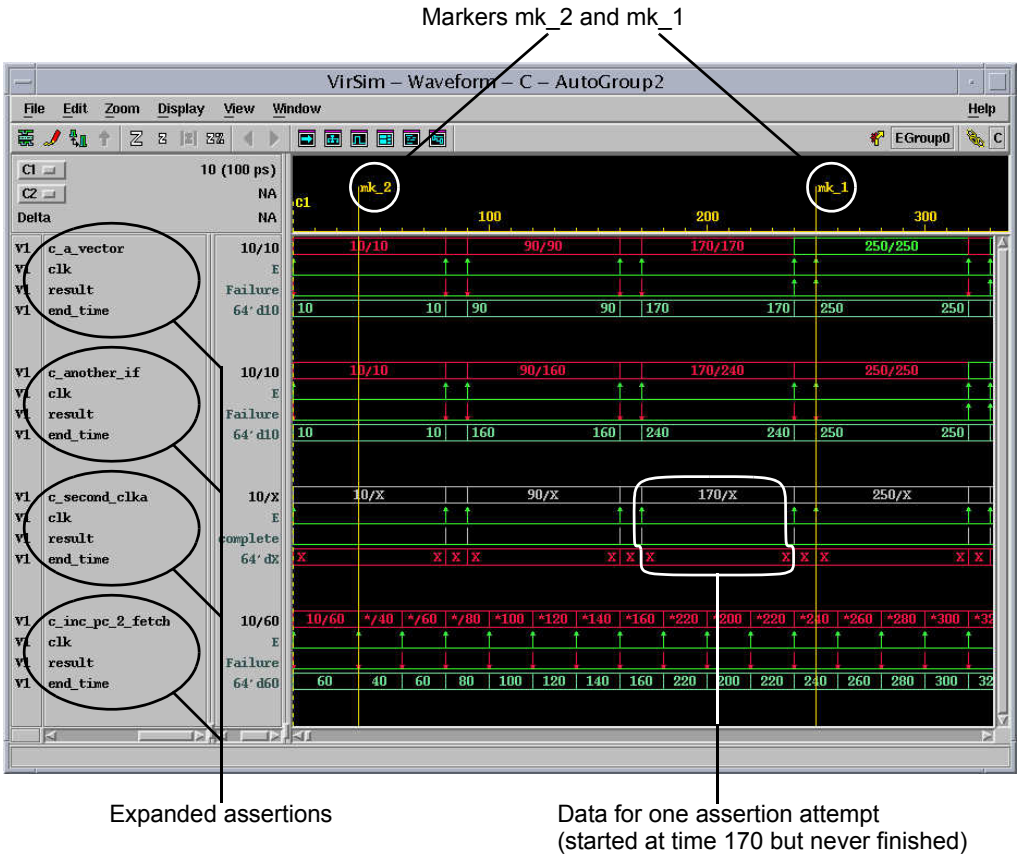


Figure 14-2 shows four assertions that have been expanded. The vertical markers, mk_1 and mk_2, highlight typical display elements.

Marker mk_1 (at time tick 250) is at the beginning of an attempt on each of the first three assertions. Assertion c_a_vector shows a success that started and ended at 250. Assertion c_another_if shows a failure that started and ended at 250. Assertion c_second_clka shows an incomplete attempt that started at 250 and never ended.

Assertion c_inc_pc_2_fetch, at marker mk_2, shows a display of overlapping attempts. The first box for the assertion shows an attempt that started at 10 and ended at 60. The second box shows an attempt that started and ended at 40—after the first attempt started but before it ended.

Assertion c_inc_pc_2_fetch also shows truncated start/end times marked with asterisks. You can also see that this assertion was evaluated with a different clock than the other three assertions.

Using OVA with Third Party Simulators

Synopsys has developed OVAsim, a PLI application that enables you to run non-Synopsys simulators using OVA (OVA). With OVAsim, you can create a powerful system of assertion-based checkers. Because the checkers are compiled independently of a specific simulator, they can be used with any major simulator, packaged with IP designs, and shipped to any customer.

OVAsim works by compiling the OVA code into a shared object and creating a wrapper file that forms the link between the checkers and the design. This wrapper file provides ports to the signals of interest in the design and contains all necessary PLI calls. Also, because the OVA code specifically refers to the ports of the wrapper file, it is largely insulated from design changes. The wrapper file and shared object generated by OVAsim are compiled and run as a part of the design by the simulator.

For information on OVAsim, contact vcs_support@synopsys.com.

Inlining OVA in Verilog

Inlined OVA enables you to write any valid OVA code within a Verilog file using pragmas. In most usage cases, the context is inferred automatically and the OVA code will be bound to the current module.

You can use this process with or without regular OVA files. The results for both inlined and regular (OVA) assertions are reported together.

Inlined OVA is enabled in VCS by the `-ova_inline` command line switch.

This section covers the following topics:

- Specifying Pragmas in Verilog
- Methods for Inlining OVA
- Case Checking
- General Inlined OVA Coding Guidelines

Specifying Pragmas in Verilog

Inlined OVA is specified in Verilog code using pragmas. Several different forms are accepted, including C and C++ style comments, and modified C++ multi-line comments.

The general C++ style form is as follows:

```
/* ova first_part_of_pragma
...
last_part_of_pragma
*/
```

You can also use the following modified C++ approach:

```
//ova_begin
// pragma_statement
//...
//ova_end
```

For a single-line pragma, you can use the following C form:

```
// ova pragma_statement;
```

Assertions can be placed anywhere in a Verilog module and they use predefined units, including those in the Checker Library.

Methods for Inlining OVA

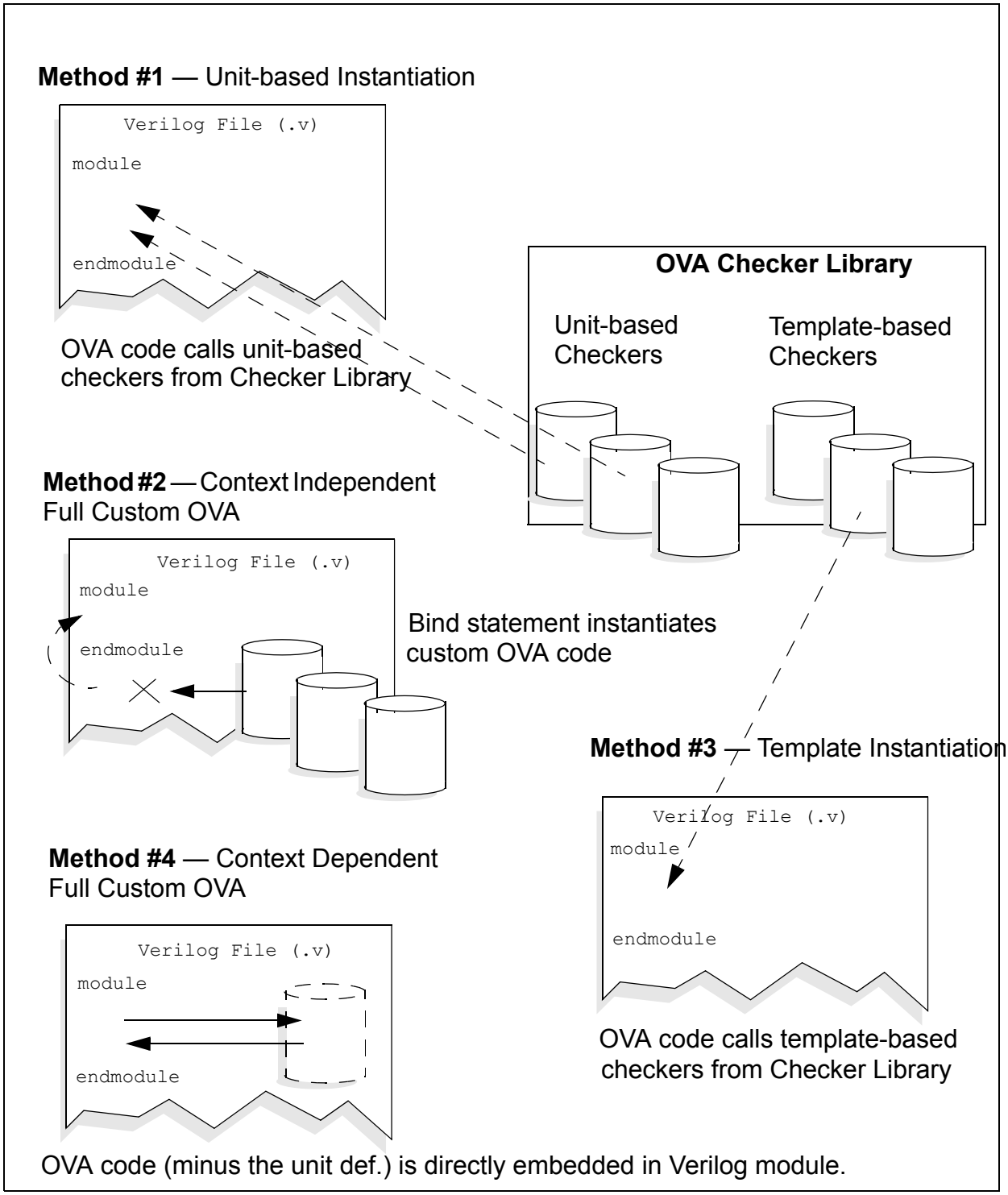
There are four basic methods you can use to inline OVA within Verilog:

- Unit Instantiation Using the Unit-based Checker Library (recommended for using Synopsys-developed OVA checkers)

- Context-Independent Full Custom OVA (uses custom-developed OVA code that resides in a Verilog file)
- Template Instantiation Using the Template-Based Checker Library
- Context-Dependent Full Custom OVA

These methods are described in detail throughout this section. The graphic in Figure 14-2 provides an overview of these methods.

Figure 14-3 Methods for Inlining OVA within Verilog



Unit Instantiation Using the Unit-Based Checker Library

The easiest and most efficient method to inline OVA is to instantiate a unit-based checker from the OVA Checker Library (For more information on the Checker Library, see the *OpenVera Assertions Checker Library Reference Manual*). The context of the checker is automatically inferred based upon the location of the OVA pragma statement within a module of Verilog code. To use this method, you must include the bind keyword within a valid OVA pragma statement.

The syntax options for specifying unit-based assertions are as follows:

C++ Style:

```
/* ova
bind unit_name [inst_name] [#(param1, ...,paramN)] [(port1, ..., portN)];
*/
```

Modified C++ Style:

```
//ova_begin
//bind unit_name [inst_name] [#(param1, ...,paramN)] [(port1, ..., portN)];
//OVA_END

*/
```

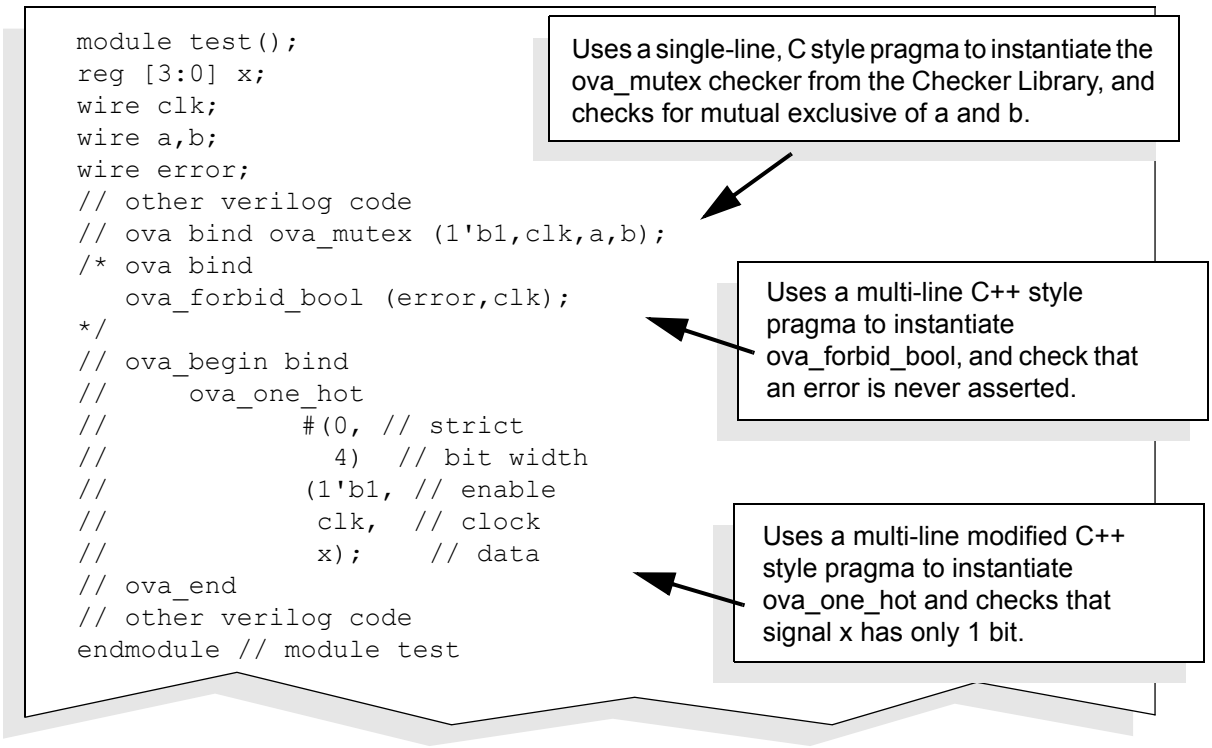
C Style:

```
// ova bind unit_name [inst_name] [#(param1, ...,paramN)] [(port1, ..., portN)];
```

Note: In all syntax styles, you can also split a pragma statement into separate lines as shown below:

```
//ova bind
//ova unit_name [instance_name]
//ova [#(parameter1, ..., parameterN)]
//ova [(port1, ..., portN)];
```

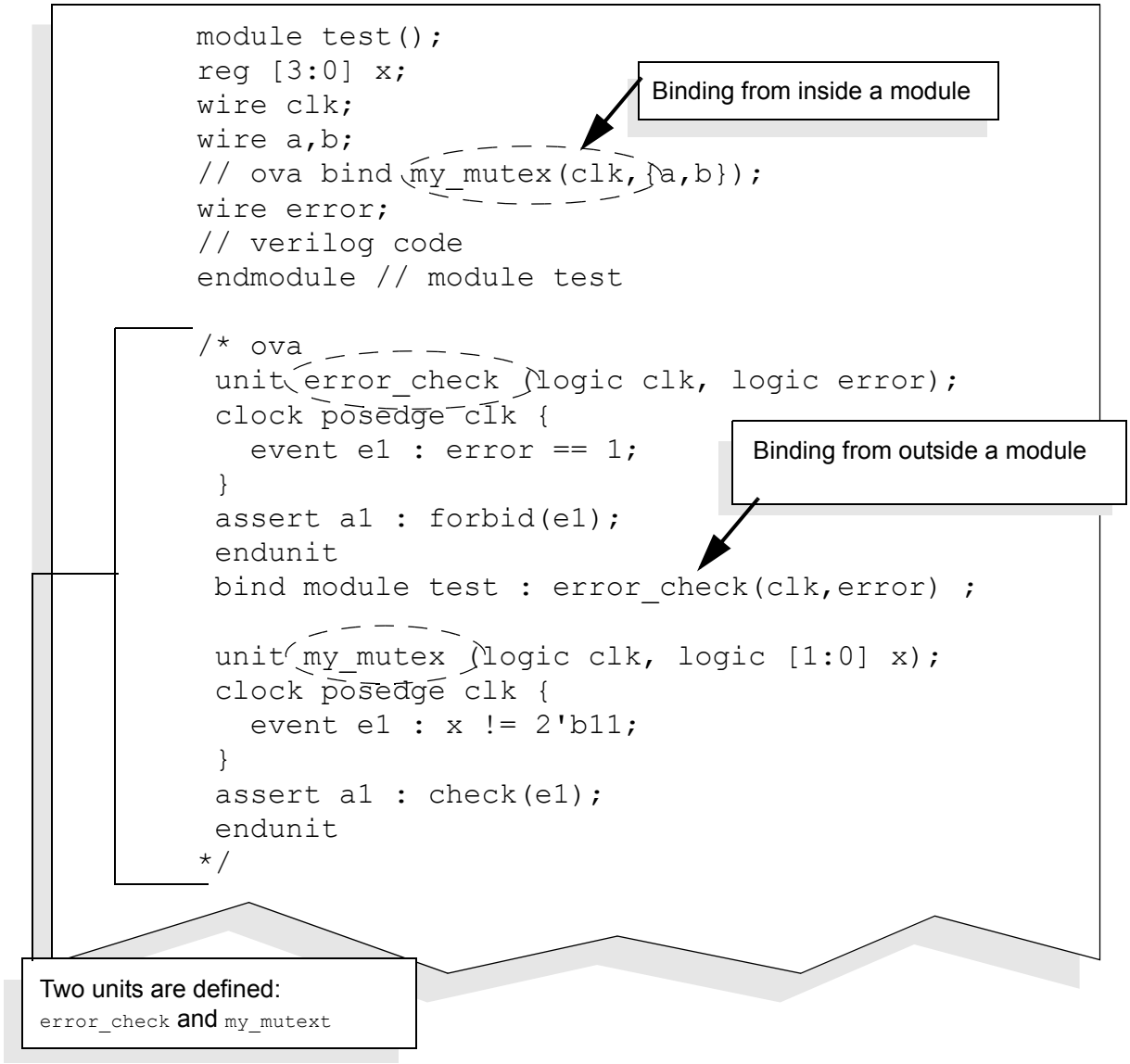
The following example shows how to instantiate a checker, called `ova_one_hot`, from the OVA Checker Library:



Instantiating Context-Independent Full Custom OVA

You can inline OVA within Verilog by instantiating independent custom OVA code located in the Verilog file but outside a Verilog module definition. The unit definition associated with the code must be specified outside a Verilog module.

This method is demonstrated in the following example:



In the previous example, the bind statement (`// ova bind my_mutex(clk,{a,b});`) calls independent OVA code located outside Verilog module. You can instantiate the independent OVA code as many times as needed anywhere in the Verilog code. The context of the OVA code within the Verilog code is automatically inferred based upon the location of the bind statement.

Template Instantiation Using the Template-Based Checker Library

You can instantiate any template-based checker from the Checker Library in your Verilog code. The context of the checker is automatically inferred based on the location of the call from within Verilog.

Note the following construct usages and limitations in Template Instantiation:

- Clocks must use edge expressions (unit-based checkers use Verilog style ports)
- You can specify the default clock in conjunction with the `check_bool` and `forbid_bool` checkers, however, it does not work with other templates or units. The following example shows a supported default clock:

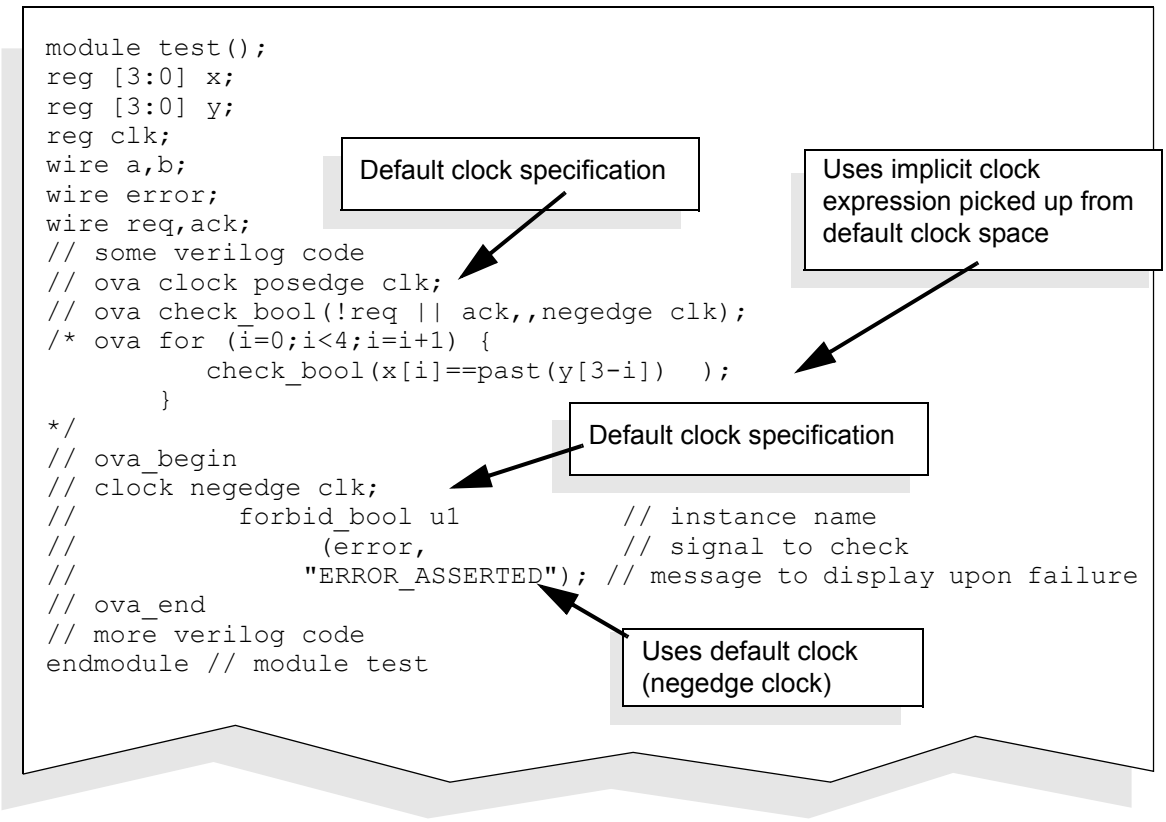
  ```
  //ova clock posedge clk;
  ```

  Note that each sequence or boolean expression is associated with a clock. The clock determines the sampling times for variable values.
- Both for loops and nested for loops are supported, as shown below:

  ```
  for (name=expr;name op1 expr;
                   name=name op2 expr)
  {
  for loop body
  }
  ```

- You cannot mix a template instantiation and unit instantiation within the same OVA pragma using the multi-line C++ and modified C++ pragma specification formats. You can, however, specify a template instantiation and a unit instantiation using a separate single-line C pragmas.

The following example demonstrates how to implement template instantiation using the template-based checker library:



The example calls the `check_bool` template from the OVA checker library. Note that the default clock, (`// ova clock posedge clk;`), must be a local signal, and can be boolean expression. It works only works for the "`check_bool`" and "`forbid_bool`" templates, and does not work with other templates.

Inlining Context-Dependent Full Custom OVA

You can directly inline any custom OVA code, except for the unit definition, within a Verilog module. In this case, the unit and binding definitions are implicit.

The following example demonstrates this method:

```
module test();
reg clk;
wire a,b;
// other verilog code
// ova_begin
// clock posedge clk {
//   event e1 : a #1 b;
// }
// assert a1 : forbid(e1);
// ova_end
/* ova
   clock posedge clk {
      event e2 : ended(e1) #2 b;
   }
   assert a2 : forbid(e2);
*/
// more verilog code
endmodule // module test
```

Uses modified C++ style pragma to specify custom OVA code that defines event e1 and assertion a1.

Uses C++ style pragma to specify custom OVA code that defines event e2 and assertion a2.

Case Checking

You can perform two types of case checking when using inlined OVA:

- *Parallel* — The actual case selection value must select exactly one of the specified cases in the statement.
- *Full* — The actual case selection value must fall into the range specified by the case items. A case containing the default item is by definition full.

To control parallel case checking, use the `parallel_case` statement:

```
//ova parallel_case on | off ;
```

When `on` is specified, all case statements until the end of the module and the entire hierarchy underneath will be checked to ensure the rules of “parallel case” execution, unless overridden by another command or a local override pragma.

If the *off* argument is specified, parallel case checking is disabled unless overridden by another command or a pragma associated with a case statement.

When the pragma is not specified the default is *off*.

To control full case checking:

```
//ova full_case on | off ;
```

When `on` is specified, all case statements until the end of the module and the entire hierarchy underneath will be checked to ensure the rules of “full case” execution, unless overridden by another command or a local override pragma.

If the `off` argument is specified, full case checking is disabled unless overridden by another command or a pragma associated with a case statement.

When the pragma is not specified, the default is `off`.

The following rules govern parallel and full case commands:

- The commands must precede any module instance and any case statement in the module.
- If such a command is not provided, the default from the parent module is taken. (It can be by default *off*.) Also, you must make sure that every instance of the child module receives the *same* specifications. To avoid this limitation, widely used modules should include case-checking specifications.
- If a case statement appears in a function or a task, the module that contains the function or task declaration determines the default case checks (unless overridden by a local case pragma on the case statement).

Context-Dependent Assertion Pragmas

OVA includes three local assertions that depend on the context in which they are placed in the Verilog code:

```
// ova parallel_case;

// ova full_case;

// ova no_case;
```

The pragma must be placed immediately following a *case (epxr)* statement. If placed anywhere else, an error will be reported. They apply only to the associated case statement. The defaults continue to apply elsewhere.

The `parallel_case` and `full_case` statements enable their own type of case check on the associated case statement and disable the other type. *Unless* both are present.

The `no_case` statement disables any case checking on that case statement.

The following rules govern assertion pragmas:

- These assertions verify the correct operation of case statements. They are triggered any time the case statement is executed. That is, no sampling of signals by a clock takes effect.
- The same pragma may be applied multiple times within a module. Each appearance will be considered an invocation of the assertion for its associated statement.
- If no such pragma assertion is associated with a case statement then the default setting established by an // ova command takes effect.
- Each // ova pragma may contain only one assertion terminated by “;”.
- Multiple case pragmas can be associated with a case statement, each on a separate line.
- The `no_case` pragma takes precedence over any other specification.

General Inlined OVA Coding Guidelines

Note the following guidelines when coding Inlined OVA:

- Since OVA pragmas are declarative in nature, they do not need to placed within procedural code — for example, within tasks, functions, and always/initial/forever blocks.
- Cross-module References (XMRs) are not allowed to concurrently reside within a custom OVA description.
- Template instantiation will be treated as an OVA description (the XMR restriction also applies).
- Unit binding is allowed inside ‘module endmodule’, but the keyword ‘bind’ needs to be specifed in order to distinguish it from template instantiation. Unit binding (with the keyword ‘bind’) can have XMRs in its connections; however, using XMRs in connections without the bind keyword will cause a warning statement. Inlined bindings have the same requirements and limits as bindings created in an OVA source file.
- Inlined OVA cannot be read directly by third-party simulators. You can, however, use VCS to produce output files that can be read by third-party simulators.
- Each instance must be terminated by a “;” (semi-colon).
- Both positional and named (explicit) association of actual arguments are supported. However, the two styles cannot be used simultaneously in an instance.

Using Verilog Parameters in OVA Bind Statements

This section contains the following topics:

- Introduction
- Use Model
- Post-processing Flow

Introduction

This section describes the enhancement to VCS that allows Verilog parameters to be used in temporal context in OVA bind statements. This enhancement is under an optional switch. At this time, use of Verilog parameters is limited to inline bind statements, but simple workarounds exist to allow binding of non-inline OVA while still allowing full use of Verilog parameters.

Definitions

For purposes of this document, a value is used in *temporal context* if it is used as a delay (or delay range), a sequence length (or length range), or as a repeat count in an OVA event or assert statement. A value is used in *static context* if it is used to compute the bit width or bounds of a vector in an OVA unit statement.

Use Model

The current OVA use model allows the use of Verilog parameters only in static context. That is, the vector widths of an OVA unit may be defined using Verilog parameters. This continues to be the case even after this enhancement.

However, the OVA compiler does not have access to the Verilog parameters at the time the OVA state machine is compiled. The values of the parameters must be extracted and passed to the OVA compiler in order to use these values in temporal context.

The enhancement described in this document will cause the Verilog parameters to be expanded at the time the inlined OVA is processed by the OVA extractor, since the OVA binds and the design hierarchy are both available at that time. This implies that Verilog parameters can only be used in temporal context if the parameter was first expanded during the inlining phase. For that reason, only inline OVA binds will be able to pass Verilog parameter values to a unit if the parameter is to be used in a temporal context.

Enabling Verilog Parameter Expansion

To enable the substitution of Verilog parameters during inlining, the `-ova_exp_param` option must be used on the command line at compile time. In addition, the `-ova_inline` option must be enabled in order to enable the OVA inline processing.

Limitations on the Input

Only those binds found in single-line pragmas will be scanned for Verilog parameters (this is an internal limitation of the OVA inline parser). In order to use a Verilog parameter in temporal context via an OVA bind statement, the bind statements must be of the form:

```
// ova bind module foo : unitA u1 #( count ) ( ... );
// ova bind module bar : unitB u2 #( delay ) ( ... );
// ova bind ...etc...
```

Multiple-line inline pragmas (mostly used for full-language inlining) follow a different flow when they are processed and will not work if Verilog parameters are used in temporal constructs (but they will continue to work correctly for binds containing constants or Verilog parameters used only in static context). For example, the following binds would not be legal if count and delay were used by the bound unit in temporal context:

```
/*
ova bind module foo : unitA u1 #( count ) ( ... );
ova bind module bar : unitB u2 #( delay ) ( ... );
*/
// ova_begin
ova bind module foo : unitA u1 #( count ) ( ... );
ova bind module bar : unitB u2 #( delay ) ( ... );
// ova_end
```

The parameter names are resolved in the context of the module to which the unit is bound. If the unit parameter is used in temporal context, the Verilog parameter bound to that unit must not use a cross-module reference (cross-module references are allowable if the parameter is only used in static context). Both module and instance binding is supported by this enhancement. But only binds directly found in pragmas in the Verilog file will be recognized. Bind statements inside files which are included in one or more Verilog files via `'include` will not be processed.

Table 14-4 Table 1. Verilog parameter use model summary

Parameter context:	**Static (bus width)**	**Temporal (delay, repeat, ...)**
Type of bind	Module or instance	Module or instance
Location of bind	Inline or separate file	Inline flow only
XMR parameters allowed	Yes	No
Pragma format	Single or multiple line	Single line only (// ova bind)
Can put bind in include file	Yes	No

Recommended Methodology

To make use of Verilog parameters in temporal context, the binds that use such parameters must follow the inlined OVA flow. For OVA binds which are already inlined into the RTL, no additional work is required. It is not necessary, however, that the bind statements be inserted directly into the RTL source. Rather, a separate file (such as dummy.v) could be created to hold all the binds in the design that come from separate (non-inlined) OVA files. Giving this file a ".v" extension and passing it to the compiler with `-ova_inline` and `-ova_exp_param` enabled is enough to get the inline preprocessor to substitute the parameter value for the parameter name before passing the converted OVA code to the OVA compiler.

To avoid future problems, we also recommend moving the binds that are not already inlined into one or more of these dummy Verilog files. That way, if the contents of the OVA unit change at some point in the future (for example, a new parameter used in temporal context is added when there was previously no such parameter), the bind will continue to work as expected. Also, while other OVA code (such as units or templates) can be added to these dummy Verilog files, we recommend against that, as there could be some chance of confusing the inline proc essor (which, at this point, does not use a very sophisticated parser).

Caveats

The inline pre-processor writes the extracted inlined OVA into a file called generated.ova under the ova.vdb directory hierarchy. In the past, this file could be copied into a user-level file and used in subsequent simulation runs as a block of extracted OVA. This can still be done, to a certain extent. However, one of the following must be true:

- The -ova_exp_param option is not enabled, or
- Any modules to which units using Verilog parameters are bound occur only once in the design, or
- Multiple instances of any modules to which units using Verilog parameters are bound use the same value for each parameter across all instances.

In other words, if any one module is instanced multiple times with different parameter values for two or more of the instances, then the parameter expansion that occurs at the time the inlined OVA is extracted will render the generated.ova file unusable for other purposes.

Post-processing Flow

A small change to the post-processing flow is necessary in order for this enhancement to have an effect on the OVA code compiled for post-processing. Recall that, normally, inlined OVA is not supported in the post-processing flow. However, it is still possible to invoke the inline pre-processing step as part of this flow.

Use Model

The existing post-processing flow consists of three basic steps:

1. Compilation of the design (Verilog) files using `-ova_PP`,
2. Compilation of the OVA source files using `-ova_RPP`
3. Replay of the saved simulation vectors with `ovapp`

Using Verilog parameters in OVA bind statements

The first step generates a skeleton Verilog file containing the hierarchy of the design and the nets converted to registers (for playback). It also preserves the Verilog parameters. This skeleton Verilog file must be compiled in the second step before the playback can occur. It is this compilation step that we will exploit to allow Verilog parameters to be used in the post-processing flow.

The change to the flow involves the same dummy Verilog file that was discussed in the sections above. This file, which is a disguised as a Verilog file but, in reality, contains nothing but inlined bind statements, is passed to the compiler during the `-ova_RPP` (second) step. In addition, the `-ova_inline` and `-ova_exp_param` options are added to the `-ova_RPP` compile step. This will cause the inline pre-processor to pick up the binds along with the remainder of the OVA code (which should still be in separate ".ova" files) and expand the parameters according to the values found in the skeleton design file.

OVA System Tasks and Functions

OVA system tasks and functions enable you to set conditions and control the monitoring of assertions, and specify the response to failed assertions.

The following topics are covered:

- Setting and Retrieving Category and Severity Attributes
- Starting and Stopping the Monitoring of Assertions
- Controlling the Response to an Assertion Failure
- Task Invocation from the CLI
- Debug Control Tasks
- Calls From Within Code

Setting and Retrieving Category and Severity Attributes

You can use the following system tasks to set the category and severity attributes of assertions:

`$ova_set_severity("`*`assertion_full_hier_name`*`", `*`severity`*`)`
Sets the severity level attributes of an assertion. The severity level is an unsigned integer from 0 to 255.

`$ova_set_category("`*`assertion_full_hier_name`*`", `*`category`*`)`
Sets the category level attributes of an assertion. The category level is an unsigned integer from 0 to 2^{24} - 1.

You can use the following system functions to retrieve the category and severity attributes of assertions:

`$ova_get_severity("`*`assertion_full_hier_name`*`")`
Returns unsigned integer.

`$ova_get_category("`*`assertion_full_hier_name`*`")`
Returns unsigned integer.

After specifying these system tasks and functions, you can start or stop the monitoring of assertions based upon their specified category or severity. For details on starting and stopping assertions based on their category specification, see ìCategory and Severity-Based Monitoringî on page 15-40. For details on starting and stopping assertions based on their severity specification, see the next section, "Starting and Stopping the Monitoring of Assertions."

Starting and Stopping the Monitoring of Assertions

There are several methods you can use to start and stop the monitoring of assertions:

- Global Monitoring — Starts or stops assertions based upon their hierarchical location level at or below specified modules, entities, or scopes (note that these tasks overwrite any other tasks that control monitoring).
- Category-Based Monitoring — Starts or stops assertions based upon their category specifications.
- Severity-Based Monitoring — Starts or stops assertions based upon their severity specifications.
- Name-Based Monitoring — Starts or stops assertions based on the specified name of the assertion.

Global Monitoring

The `$ova_start` and `$ova_stop` system tasks enable you to control the monitoring of assertions based on the specified hierarchical location of the module, scope, or entity. Note that these tasks are specified on a global level, and overwrite any other tasks that start or stop assertion monitoring.

To start level-based monitoring, use the `$ova_start` system task:

```
$ova_start[(levels [, module_or_scope_arguments])];
```

The integer argument *levels* indicates how many levels in the hierarchy at and below the specified modules, entities, and scopes the OVA monitoring is to start. Zero indicates all levels below the specified ones. Negative values are not allowed.

Within OVA and Verilog code, arguments can be specified for one or more module names as quoted text strings (e.g.,"*module_name*") and/or instance scopes as Verilog scope names (unquoted), separated by commas.

Scope resolution follows the Verilog Standard (IEEE Std 1364-2001) Section 12.5. That is, if a scope argument is specified, then the path name is first considered as relative to the module where the task was called. If it does not match any valid relative path, it is matched against paths one level up. If no match then one more level up, etc. For example, if scope *a.b.c* is an argument of a task called from module *m*, the scope name is first interpreted as a path starting at an instance *a* in *m.* If such a path does not exist, it is applied to a father module of *a.* If the path does not exist in the root module of m, the path is an error.

If there are no arguments, i.e., `$ova_start`, the task applies to the entire design. It is equivalent to `$ova_start(0)`.

Note that the system task is ignored if monitoring was already started in the specified scope, module, or entity.

To stop monitoring, use the `$ova_stop` system task:

```
$ova_stop[(levels [, module_or_scope_arguments])];
```

Similar to the `$ova_start` system task, the integer argument *levels* indicates how many levels in the hierarchy at and below the specified modules, entities, and scopes the OVA monitoring is to stop. Zero indicates all levels below the specified ones.

Arguments can be specified for one or more modules, entities, and instance scopes as in the case of `$ova_start`. The effect of a module or entity argument is to stop monitoring assertions for all instances of the module and their respective hierarchies under each instance as indicated by levels. The effect of a scope argument is to stop monitoring assertions for that instance scope and the hierarchy under it as indicated by levels. If module, entity and scope arguments are omitted then levels applies to all the root modules of the design.

If there are no arguments, i.e., `$ova_stop`, the task applies to the entire design. It is equivalent to `$ova_stop(0)`.

The system task is ignored if the monitoring has already been stopped in the specified scope or module.

OVA monitoring is started automatically at the beginning of simulation. This is for compatibility with the non-inlined version where monitoring starts immediately at time 0. To control monitoring with the `$ova_start` and `$ova_stop` tasks, `$ova_stop` must be issued at time 0 — e.g., in an initial block.

This is similar to the use of `$dumpvars` where dumping starts immediately after calling this task and can be inhibited by calling $dumpoff right after. Here, the equivalent of calling `$dumpvars` is implicit in the start of simulation.

Examples:

```
$ova_start;   // Start assertions in the whole design.
```

```
$ova_start(0) // Start assertions in the whole design.

$ova_stop(3)  // Stop assertions in all top modules
              // to three levels below.

$ova_start(1, "mod1") // Start assertions in all
           // instances of module mod1 at that level only.

$ova_stop(0, i1.mod1) // Stop assertions in all of
        // the hierarchy at and below scope i1.mod1.

$ova_stop(3, i1.mod1) // Stop assertions in the hierarchy
               // started at scope i1.mod1 to a depth
```

Category and Severity-Based Monitoring

To control category and severity-based assertion monitoring, you can use the following system tasks:

`$ova_category_start(`*`category`*`)`
: Starts all assertions associated with the specified *category*. The category level is an unsigned integer from 0 to 2^{24} - 1

`ova_category_stop(`*`category`*`)`
: Stops all assertions associated with the specified *category*.

`$ova_severity_start(`*`severity`*`)`
: Starts all assertions associated with the specified *severity* level. The severity level is an unsigned integer from 0 to 255.

`$ova_severity_stop(`*`severity`*`)`
: Stops all assertions associated with the specified *severity* level.

Name-Based Monitoring

To control monitoring of assertions based upon the specified names of assertions, use the following system tasks:

`$ova_assertion_stop("`*`fullHierName`*`")`
Stops the monitoring of the specified assertion (string).

`$ova_assertion_start("`*`fullHierName`*`")`
Starts the monitoring of the specified assetion (string).

Controlling the Response to an Assertion Failure

You can specify a response to an assertion failure, based upon its severity or category, using the $ova_severity_action or $ova_category_action system tasks. The syntax for these two tasks is as follows:

```
$ova_severity_action(severity, action);
```

or

```
$ova_category_action(category, action);
```

Note the following syntax elements:

`severity`
Severity to be associated with the action.

`category`
Category associated with the action.

action

Can be specified as "continue", "stop" or "finish". The action, which must be quoted as shown, is associated globally with the specified severity level, for all modules and instances. The default is "continue". The actions are specified as follows:

- "stop" — Stops the simulation with $stop semantics. All OVA attempts are also suspended and can be resumed.
- "finish" — Terminates the simulation with $finish semantics.
- "continue" — Outputs the message associated with the assertion failure but otherwise has no effect on the simulation.

Display Custom Message for an Assertion Failure

You can display a custom message upon failure of an assert statement check or forbid. The assert statement accepts an action block as follows:

```
assert [name] [[index]] : check | forbid
        (sequence_expr | event_name [, message[, severity [,
        category]]]) [action_block];
```

Where:

```
action_block ::=  [else display_statement]
```

display_statement is similar to the standard $display system task:

```
$display(formatting_string, list_of args);
```

If both *message* and a $display are specified, then both are output upon assertion failure.

The following expression and system function calls are allowed in the `list_of args`.

`expression`

Output the value of the (bitvector) expression at the time of assertion failure (or success). It can be any expression over formal port/parameter names and ova variables. It CANNOT contain any OVA - specific operators such as the temporal operators `past`, all `edge` operatprs, and `count`.

The evaluation of the uses the sampled values of the variables as in the assertion.

`$ova_current_time`

Returns current simulation time as a 64 bit entity. (Same as type "time" in Verilog.)

`$ova_start_time`

Returns assertion start time as a 64 bit entity. (Same as type "time" in Verilog.)

`$ova_global_time_unit`

Returns global time unit as a string. (The definition of a string is the same as in Verilog.)

Task Invocation from the CLI

Any CLI mode (command option `+cli+`*`n`* where *`n`* = 1-4 or `+cli` with no numerical value):

`$ova_stop` *`levels`*

Applied down to depth levels with respect to the present scope.

`$ova_start` *`levels modname`*

Applied to all instances of module "*`modname`*" down to a depth of levels. Note that no XMR is allowed, however module names are allowed without a quoted string.

`$ova_severity_action` *`level`* "*`action`*"

Sets the action for severity level where action is continue, stop, or finish.

The commands can also be issued using the format used in task calls ó that is, arguments in parentheses separated by commas. Module names must be quoted as in the task calls.

Note that you can specify all OVA system tasks and functions described in this chapter (`$ova_set_severity`, `$ova_set_category`, etc.) at the CLI using the documented syntax.

Debug Control Tasks

When dumping of OVA assertion results for viewing using VirSim is enabled using the `-ova_debug` or `-ova_debug_vpd` options, the debug information is dumped for all enabled OVA assertions (enabled using `$ova_start` or by default from time 0 if not stopped by $ova_stop).

Dumping can be turned off by calling the following task:

```
$ovadumpoff;
```

in the Verilog code. It can be turned on again by calling the following task:

```
$ovadumpon;
```

The effect of calling these tasks is from the time of the call on.

If dumping of Verilog signals is not enabled by calling the system task $dumpvars, OVA dumping will also include all signals that are referred to by the assertions. This information is placed in a VPD file, vcdplus.vpd, in the simulation run directory.

Calls from within Code

The `$ova_start`, `$ova_stop`, and `$ova_severity_action` system tasks are provided to control the monitoring of assertions, such as delaying the start of assertion monitoring and terminating the monitoring based on some conditions. For example, you can start monitoring after some initial condition, such as reset, has been satisfied.

To start monitoring, use the `$ova_start` system task:

```
$ova_start[(levels [, module, entity, or scope arguments])];
```

Whenever this statement is executed, assertion monitoring starts. The command is ignored if monitoring was already started in the specified scope, module, or entity.

The integer argument *levels* indicates how many levels in the hierarchy at and below the specified modules, entities, and scopes the OVA monitoring is to start. Zero indicates all levels below the specified ones. Negative values are not allowed.

Within OVA and Verilog code, arguments can be specified for one or more module names as quoted text strings (e.g.,“module_name”) and/or instance scopes as Verilog scope names (unquoted), separated by commas.

Scope resolution follows the Verilog Standard (IEEE Std 1364-2001) Section 12.5. That is, if a scope argument is specified, then the path name is first considered as relative to the module where the task was called. If it does not match any valid relative path, it is matched against paths one level up. If no match then one more level up, etc. For example, if scope *a.b.c* is an argument of a task called from module *m*, the scope name is first interpreted as a path starting at an instance *a* in *m.* If such a path does not exist, it is applied to a father module of *a.* If the path does not exist in the root module of m, the path is an error.

If there are no arguments, i.e., `$ova_start`, the task applies to the entire design. It is equivalent to `$ova_start(0)`.

To stop monitoring, use the `$ova_stop` system task:

```
$ova_stop[(levels [, module, entity, or scope arguments])];
```

Whenever this statement is executed, the assertion monitoring stops. All attempts that are in progress are reset. The command is ignored if the monitoring has already been stopped in the specified scope or module.

The integer argument *levels* indicates how many levels in the hierarchy at and below the specified modules, entities, and scopes the OVA monitoring is to stop. Zero indicates all levels below the specified ones.

Arguments can be specified for one or more modules, entities, and instance scopes as in the case of `$ova_start`. The effect of a module or entity argument is to stop monitoring assertions for all instances of the module and their respective hierarchies under each instance as indicated by levels. The effect of a scope argument is to stop monitoring assertions for that instance scope and the hierarchy under it as indicated by levels. If module, entity and scope arguments are omitted then levels applies to all the root modules of the design.

If there are no arguments, i.e., `$ova_stop`, the task applies to the entire design. It is equivalent to `$ova_stop(0)`.

OVA monitoring is started automatically at the beginning of simulation. This is for compatibility with the non-inlined version where monitoring starts immediately at time 0. To control monitoring with the `$ova_start` and `$ova_stop` tasks, `$ova_stop` must be issued at time 0, e.g., in an initial block.

This is similar to the use of `$dumpvars` where dumping starts immediately after calling this task and can be inhibited by calling $dumpoff right after. Here, the equivalent of calling `$dumpvars` is implicit in the start of simulation.

Examples:

```
$ova_start;        // Start assertions in the whole design.

$ova_start(0)      // Start assertions in the whole design.

$ova_stop(3)       // Stop assertions in all top modules
                   // to three levels below.

$ova_start(1, "mod1")      // Start assertions in all
              // instances of module mod1 at that level only.

$ova_stop(0, i1.mod1) // Stop assertions in all of
              // the hierarchy at and below scope i1.mod1.
```

```
$ova_stop(3, i1.mod1) // Stop assertions in the hierarchy
             // started at scope i1.mod1 to a depth 3.
```

To specify the response to an assertion failure, use the `$ova_severity_action` system task:

```
$ova_severity_action(level, action);
```

Note the following syntax elements for `$ova_severity_action`:

`level`
: Severity level to be associated with the action.

`action`
: Can be specified as `"continue"`, `"stop"` or `"finish"`. The action, which must be quoted as shown, is associated globally with the specified severity level, for all modules and instances. The default is `"continue"`. The actions are specified as follows:

  - `"stop"` — Stops the simulation with `$stop` semantics. All OVA attempts are also suspended and can be resumed.
  - `"finish"` — Terminates the simulation with `$finish` semantics.
  - `"continue"` — Outputs the message associated with the assertion failure but otherwise has no effect on the simulation.

Developing A User Action Function

Instead of specifying "`continue`", "`stop`" or "`finish`" as the action argument to the `$ova_severity_action` system task, you can specify a function that you develop to perform the action. To enable this feature the new struct types:

Ova_AssertionSourceInfo

This struct has the following fields:

lineNo
: represents the line number in the file where the assertion was written.

fileName
: represents the filename where the assertion was written in HDL or ova source code.

OvaAssertionData;

This struct has the following fields:

Ova_AssertName name
: This represents the full hierarchical name of the assertion.

Ova_ExprType exprType
: This represents the type of assertion(event/check/forbid).

Ova_AssertionSourceInfo srcBlock
: This represents the source file information for the assertion.

unsigned int severity:8
: This represents the severity assigned to the assertion. It is an eight bit integer constant.

category:24
: This represents the category assigned to the assertion. It is a 24 bit integer constant.

Ova_String userMessage
: This represents the custom user message assigned to the assertion.

Ova_TimeL startTime
: This represents the start time for the assertion as an unsigned long long.

Ova_TimeL endTime
: This represents the end time for the assertion as an unsigned long long.

For non-temporal assertions, startTime and endTime will always be the same.

The following is the prototype of the user action functions:

```
typedef void (*OvaAssertionFailCB)(OvaAssertionData
asData);
```

For example, a sample callback function would look like:

```
void my_user_action(OvaAssertionData assertionData)
{
⋮
}
```

15

Using SystemVerilog

This chapter begins with examples of the constructs in SystemVerilog that VCS has implemented. These constructs include:

- New data types including
  - C-like data types for encapsulation and more compact code and new four-state and two-state data types for design data.
  - user defined data types and enumerations
  - structures and unions
- Multi-dimensional arrays
- New ways of writing to Variables
- New operators from C
- New do while procedural statement

- New types of always blocks
- New rules for user-defined tasks and functions
- New assertions
- New ways to represent the design hierarchy
- Interfaces

The chapter then describes:

- How to enable the use of SystemVerilog code
- Controlling how VCS uses SystemVerilog assertions, including:
  - compile-time and runtime options
  - options for coverage
  - How to view results coverage
  - New system tasks for assertions
  - OVA system tasks that also work with SystemVerilog assertions

SystemVerilog Data Types

SystemVerilog has several new data types.

Variable Data Types for Storing Integers

VCS has implemented the following SystemVerilog variable data types for storing integers:

data type	States	Default	Description
`char`	two state	signed	A C-like data type, 8 bit integer
`shortint`	two state	signed	16 bit integer
`int`	two state	signed	32 bit integer
`longint`	two state	signed	64 bit integer
`byte`	two state	signed	8 bit integer or ASCII character
`bit`	two state	unsigned	user-defined vector size
`logic`	four state	unsigned	user defined vector size

Notice that some of these data types default to signed values. The Verilog-2001 standard has the `unsigned` reserved keyword. In SystemVerilog you can use it in the variable declaration to change one of these default signed data types to unsigned, for example:

```
longint unsigned liu;
```

You can also use the `signed` keyword to make the `bit` and `logic` data types store signed values.

Note:

In SystemVerilog the Verilog-2001 `integer` data type defaults to a signed value.

The two state data types begin simulation with the 0 value. Assignments of the Z or X values to these data types result in an assignment of the 0 value.

There is also the `void` data type that represents non-existent data. This data type can be used to specify that a function has no return value, for example:

```
function void nfunc (bit [31:0] ia);
⋮
endfunction
```

User Defined Data Types

You can define your own data types from other data types using the `typedef` construct like in C, for example:

```
typedef logic [63:0] mylog;
mylog m1, m2, m3, m4;
```

Following the `typedef` keyword are the SystemVerilog data type for the user defined type, the optional bit-width, and the name of the user defined type.

You can use a user defined type, that is, declare a signal that has a user defined type, before the definition of the user defined type, for example:

```
typedef mine;
mine p;
⋮
typedef int mine;
```

You enable this use of a user defined type by entering the `typedef` keyword and the name of the user defined type without the SystemVerilog data type.

Enumerations

You can declare a set of variables that have a set of values. These are called enumerated data types, for example:

```
enum shortint{green,yellow,red} northsouth,eastwest;
```

In this example we have declared `shortint` variables northsouth and eastwest, that can hold a set of unassigned constants named green, yellow, and red.

The default data type for an enumerated data type is `int`. If you omit the data type in the enumeration, the variables declared have the `int` data type.

You can make assignments to the constants in the enumeration, for example:

```
enum{now=0,next=1,old=2}cs,ns,tmp;
```

You can declare a user defined data type and then use it as the base type for an enumeration, for example:

```
typedef bit [1:0] mybit;
typedef enum mybit {red=2'b00, green=2'b01, blue=2'b10,
                    yellow=2'b11} colors;
```

You can use the following data types as the base type of an enumeration:

```
reg logic int integer shortint longint byte
```

Unpacked dimensions are not allowed in the base type.

Methods for Enumerations

SystemVerilog contains a number of methods for enumerations. These methods to the following:

`.first`
: Displays the value of the first constant of the enumeration.

`.last`
: Displays the value of the last constant of the enumeration.

`.next`
: Displays the value of the next constant of the enumeration.

`.prev`
: Displays the value of the previous constant in the enumeration.

`.num`
: Displays the total number of constants in the enumeration.

`.name`
: Displays the name of the constant in the enumeration.

The following is an example of the use of these methods:

```
module top;
typedef enum {red, green, blue, yellow} Colors;
```

Here is an enumeration named Colors. It has four constants named red, green, blue and yellow and ranging in value from 0 to 3.

```
Colors color = color.first;
```

We declare a Colors variable named color and initialize it to the value of the first constant in the enumeration.

```
initial
begin
$display("Type Colors:\n");
$display("name\tvalue\ttotal\tprevious\tnext\tlast\tfirst\n");
forever begin
  $display("%0s\t%0d\t%0d\t%0d\t\t%0d\t%0d\t%0d\n",
  color.name,color,color.num,color.prev,color.next,color.last,
  color.first);
  if (color == color.last) break;
  color = color.next;
  end
end
endmodule
```

Results from the VCS simulation follow:

```
Type Colors:

name    value   total   previous  next    last   first

red     0       4       3         1       3      0

green   1       4       0         2       3      0

blue    2       4       1         3       3      0

yellow  3       4       2         0       3      0
```

Structures and Unions

You can declare C-like structures and unions. The following are some examples of structures:

```
struct { logic [31:0] lg1; bit [7:0] bt1; } st1;
struct {
         bit [2:0] bt2;
         struct{
             shortint st1;
             longint  lg1;
             } st2;
       } st3;
```

Here are three structures, in them:

- The first structure is named st1 and it contains a 32 `bit` logic variable named lg1 and an 8 bit `bit` variable named bt1.
- The second structure named st3 contains a structure named st2. Structure st3 also contains a `bit` variable named bt2.
- Structure st2 contains a `shortint` variable named st1 (same name as the first structure, this is okay, but not recommended, because structure st1 is not in the same hierarchy as `shortint` st1). Structure st2 contains a `longint` variable named lg1. Notice there is also an lg1 in structure st1.

You can assign values to and from the members of these structures using the structure names as the building blocks of hierarchical names, for example:

```
initial
begin
logic1 = st1.lg1;
longint1 = st3.st2.lg1;
st1.bt1 = bit1;
```

```
end
```

You can make assignments to an entire structure if you also make the structure a user-defined type, for example:

```
typedef struct { logic [7:0] lg2; bit [2:0] bit2;} st4;
st4 st4_1;

initial
st4_1={128,3};
```

The keyword `typedef` makes the structure a user defined type. Then we can declare a variable of that type. Then we can assign values to the members of the structure. Here lg2 is assigned 128 and bit2 is assigned 3.

A structure can be packed or unpacked. A packed structure is packed in memory without gaps so that its members represent bit or part selects of a single vector. This isn't true with unpacked structures. You specify a packed structure with the `packed` keyword. By default structures are unpacked. The following is an example of a packed structure:

```
struct packed { bit [15:0] left; bit [7:0] right;} lr;
```

With a packed structure you can access the values of the members by accessing bit or part selects of the packed structure's vector, for example:

```
initial
begin
lr.left='1;
lr.right='0;
mbit1=lr[23:8];
mbit2=lr[7:0];
$display("mbit1=%0b mbit2=%0b",mbit1,mbit2);
```

```
end
```

In SystemVerilog you can enter an unsized literal single bit preceded by an apostrophe ′ as shown in the assignments to members left and right. All bits the variable are assigned the unsized literals single bit. In this case left is filled with ones and right is filled with zeroes.

Here the `$display` system task displays the following:

```
mbit1=1111111111111111 mbit2=0
```

VCS has not implemented an unpacked union. In a packed union all members must be packed structures, packed arrays (see “SystemVerilog Arrays” on page 15-12), or integer data types, all with the same size, for example:

```
typedef struct packed { bit [15:0] b1; bit [7:0] b2;} st24;
typedef union packed{
        st24 st24_1;
        reg [23:0] r1;
        reg [7:0][2:0] r2;
        } union1;
```

In this union all the members, structure st24, reg r1, and reg r2 have 24 bits.

You can access the members of the union as follows:

```
union1 u1;
bit [7:0] mybit1;
reg [7:0]myreg1;
initial
begin
mybit1=u1.st24_1.b2;
myreg1=u1.r2[2];
end
```

Structure Expressions

You can use braces and commas to specify an expression to assign values to the members of a structure, for example:

```
typedef struct{
    bit bt0;
    bit bt11;
    } struct0;
struct0 s0 = {0,1};
```

In this example, in the declaration of struct0 structure s0, bt0 is initialized to 0 and bt11 is initialized to 1, because they are listed that way in the declaration of structure struct0.

You can use the names of the members in a structure expression so that you don’t have to assign values in the order of the members in the declaration, for example:

```
typedef struct{
    bit bt1;
    bit bt2;
    } struct1;
struct1 s1 = {bt2:0, bt1:1};
```

You can use the default keyword to assign values to unspecified members. You can also use a structure expression in a procedural assignment statement, for example:

```
typedef struct{
    logic l1;
    bit bt3;
    }struct2;
struct2 s2;

initial
s2 = {default:0};
```

SystemVerilog Arrays

SystemVerilog has packed and unpacked arrays. In a packed array all the dimensions are specified to the left of the variable name. In an unpacked array the dimensions are to the right of the variable name, for example:

```
bit [1:0] b1;           // packed array
bit signed [10:0] b2;   // packed array
logic l1 [31:0];        // unpacked array
```

Packed arrays can only have the following variable data types: `bit`, `logic`, and `reg`. You can make a packed array of any net data type.

Unpacked arrays can be made of any data type.

When assigning to and from an unpacked array the following rules must be followed:

- You cannot assign to them an integer, for example:

```
logic l1 [31:0];        // unpacked array
⋮
l1 = '0;
```

- You cannot treat them as an integer in an expression, for example:

```
logic l1 [31:0];        // unpacked array
⋮
reg2 = (l1 + 2);
```

- You can only assign another unpacked array with the same number of dimensions, all with the same size.

When assigning to a packed array you can assign any vector expression, for example:

```
bit [1:0] b1;           // packed array
bit signed [10:0] b2;   // packed array
logic [7:0] l2;         // packed array
⋮
b1={r1,r2};
b2='1;
l2=b2[7:0];
```

Multiple Dimensions

You can have multi-dimensional arrays where all the dimensions are packed or some dimensions are packed and others unpacked. Here is an example of all dimensions packed:

```
logic [7:0][3:0][9:0] log1;
```

Here, is a single entry of forty bytes. All dimensions are packed, so in an assignment to this array, you reference the dimensions from left to right. To assign to the left-most bit in the left most dimensions, do the following:

```
log1[7][3][9]=1;
```

Here is an example of none of the dimensions packed:

```
logic log2 [15:0][1:0][4:0];
```

Here are ten entries of two bytes. Like when all dimensions are packed, when all dimensions are unpacked, in an assignment to this array, you reference the dimensions from left to right.To assign to the left-most bit in the left most dimensions, do the following:

```
log2[15][1][4]=1;
```

Here is an example in which some dimensions are backed, but another is not:

```
logic [11:0][1:0] log3 [6:0];
```

Here are seven entries of three bytes. In an assignment to this array, you reference the unpacked dimensions, followed by the packed ones. To assign to the left-most bit in the left most dimensions, do the following:

```
log3[6][11][1]=1;
```

In these assignments the last packed dimension can be a part select, or a slice, for example:

```
log3[6][11][1:0]=2'b11;
log3[6][11:10]=2'b00;
```

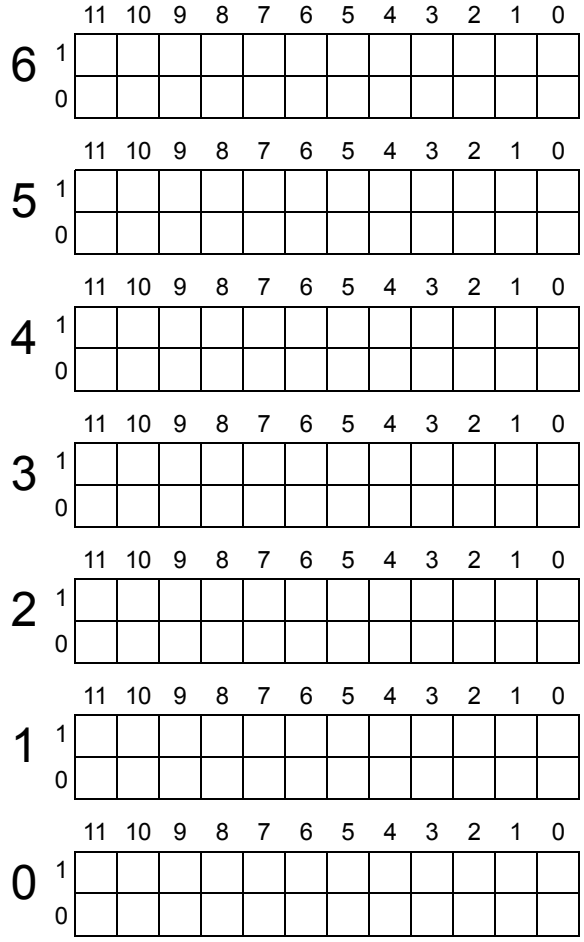
Indexing and Slicing Arrays

SystemVerilog has both part-selects and sslices. To illustrate this concept, consider the last example multidimensional array:

```
logic [11:0][1:0] log3 [6:0];
```

A graphic representation of the packed and unpacked dimensions of this array is in Figure 15-1.

Figure 15-1 Packed and Unpacked Multi-dimensional Array

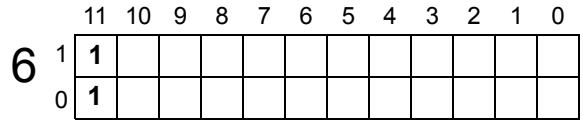


The first of the previous two assignment statements:

```
log3[6][11][1:0]=2'b11;
```

This is an assignment to a part select. It is an assignment to contiguous bits on a single packed dimension

Figure 15-2 A SystemVerilog Part Select

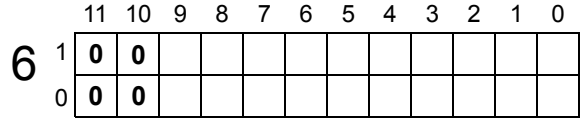


The second of the two previous assignment statements:

```
log3[6][11:10]=2'b00;
```

This is an assignment to a slice. A slice is a joint part select in contiguous elements of a multidimensional array. This slice assignment is shown in Figure 15-3.

Figure 15-3 A SystemVerilog Slice



Writing to Variables

SystemVerilog changes one of the basic concepts of Verilog: that variables are only written to by procedural statements. In SystemVerilog there are several other ways to write to a variable, as the following code example illustrates:

```
module dat;
logic log1,log2,log3;
longint loi1;
byte byt1;
wire w1;

assign log1=w1;      //continuous assignment to logic
```

```
assign loi1=w1;      //continuous assignment to longint
assign byt1=w1;      //continuous assignment to byte
buf b1 (log2,w1);    //connect to output terminal
dev dev1(log3,w1);   //connect to output port
endmodule

module dev(output out,input in);
assign out=in;
endmodule
```

As you can see, in SystemVerilog, you can write to a variable using a continuous assignment, connecting it to an output terminal of a primitive, or and output port of a module instance. Doing so is called using a structural driver.

There are some limitations on structural drivers on variables:

- A variable cannot have a behavioral driver (assigned a value by a procedural assignment statement) and a structural driver.
- A variable, unlike a net, cannot have multiple structural drivers.
- A variable still cannot connect to an `inout` port of a module definition.

Note:

You can also declare a variable to be an `input` port, this is also using a structural driver, See “New Port Connection Rules for Variables” on page 15-80.

Force and Release on SystemVerilog Variables

You can enter `force` and `release` statements for SystemVerilog data types. The following SystemVerilog code declares such data types and `force` and `release` statements for them:

```
module test;
int int1,int2;

initial
begin
    force int1=100;
    force int2=100;
    #100 release int1;
    release int2;
end
endmodule
```

Automatic Variables

You cannot force a value on to an automatic variable, for example:

```
task automatic my_aut_task;
⋮
begin
⋮
#1 force mat=1; // causes this warning:
⋮
end
endtask
```

Doing so makes the task static and VCS displays the following warning message:

```
Warning-[MNA] Making Task or Function Non-Automatic
Disable/Force/Release/Assign/Deassign inside Automatic Task
is not supported.
     "filename.v", line_number:
     task automatic task_name;
```

Also any cross-reference to or from an automatic variable makes the task static. This happens with a `force` statement, but also with any kind of assignment statement. All of the assignment statements and `force` statements in the following code result in VCS compiling the task as a static task:

```
initial
begin
#5 r5=my_aut_task.mat;
#5 my_aut_task.mat =1;
#5 force my_aut_task.mat=1;
#5 force r5=my_aut_task.mat;
end
```

VCS displays the following warning message:

```
Warning-[MNA] Making Task or Function Non-Automatic
     Making Task non-automatic due to Xmr access into it.
     "filename.v", line_number:
     task automatic task_name;
```

Multiple Drivers

If different elements of an unpacked array have structural and procedural drivers, Then you cannot enter a `force` statement for any of the elements in the unpacked array.

A procedural driver is a procedural assignment statement.

A structural driver is, for example, a continuous assignment or a connection to an output or inout port in a module instance.

```
module mult_driver;
int unpacked_int [3:0];
assign unpacked_int[3]=1; // structural driver
dev dev1(unpacked_int[3],1); // structural driver

initial
```

```
begin
unpacked_int[2]=1;  // procedural driver
//force unpacked_int[1]=1; //force statement is not valid
end
endmodule

module dev(out,in);
output [31:0] out;
input [31:0] in;
assign out=in;
endmodule
```

The `force` statement causes the following error message:

```
Error-[IFRLHS] Illegal force/release left hand side
    Force/Release of an unpacked array which is driven by
a mixture of structural and procedural assignments is not
valid
    The offending expression is signal_name
    "mult_driver.v", 10: force signal_name[1] = 1;
```

This restriction does not hold for unpacked structures, see “Structures” on page 15-25:

Release Behavior

When VCS executes a `release` statement on a SystemVerilog variable, the release behavior depends on how that variable obtained the value it had before VCS executed the previous `force` statement on it.

- If it was a structural driver, which is to say a connection to a module or primitive instance or by a continuous assignment or a procedural continuous assignment, the variable returns to its previous value immediately.

- If it was a behavioral driver, otherwise known as a procedural assignment statement, the variable returns to that previous value when or if VCS executes the procedural assignment statement again. Until VCS executes the assignment again, the variable keeps its forced value.

The following SystemVerilog illustrates this behavior:

```
module test;
logic l1, l2, l3;
⋮
assign #10 l1=1;
⋮
always @(posedge clk)
begin
assign l2=1;
l3=1;
#10 force l1=0;
    force l2=0;
    force l3=0;
#10 release l1;
    release l2;
    release l3;
end
endmodule
```

Signal l1 is continuously assigned a value of 1 at simulation time 10. Assuming a rising edge on the clock signal at simulation time 100, a procedural continuous assignment assigns the value of 1 to signal l2 and a procedural assignment assigns the value of 1 to signal l3.

At simulation time110, all three signals are forced to a 0 value.

At simulation time 120, when VCS executes all three `release` statements, signals l1 and l2 return to their 1 values, but signal l3 remains at 0 until the next rising edge on the clock.

Integer Data Types

The SystemVerilog LRM lists the following data types as integer data types:

```
shortint int longint byte bit logic reg integer time
```

All of these data types can hold an integer value. With the exception of the `time` data type, you can force a value on to an entire signal with these data types or a bit-select or part-select of these data types. The following are examples of these data type declarations and valid `force` statements to the most significant bits in them:

```
shortint si1;
int int1;
longint li1;
byte byte1;
bit [31:0] bit1;
logic [31:0] log1;
reg [31:0] reg1;
integer intg1;

initial
begin
force si1[15]=1;
force int1[31]=1;
force li1[63]=1;
force byte1[7]=1;
force bit1[31]=1;
force log1[31]=1;
force reg1[31]=1;
force intg1[31]=1;
end
```

Notice that a bit-width was not specified for the `shortint`, `int`, `longint`, `byte`, or `integer` data types in their declarations, but these `force` statements to these bit-selects are valid. This is because these data types have a known number of bits, with little-endian numbering.

Force statements to bit or part-selects of the `real`, `time`, or `realtime` data types are not possible.

Unpacked Arrays

You can make `force` statements to an element of an unpacked array or a slice of elements, for example:

```
int int1;
logic l1 [2:0];
bit bit1 [2:0];
reg reg1 [2:0];
byte byte1 [2:0];
int int2 [2:0];
shortint si1[2:0];
longint li1[2:0];
integer intg1[2:0];

initial
begin
int1=2;
force l1[int1]=3;
force bit1[int1]=3;
force reg1[int1]=3;
force byte1[int1]=3;
force int2[int1]=3;
force si1[int1]=3;
force li1[int1]=3;
force intg1[int1]=3;
end
```

You can force an entire unpacked array, a slice of elements, or a single element, for example:

```
int int1 [2:0];
int int2 [2:0];

initial
begin
force int1={1,1,1};
force int2[2:1]=int1[2:1];
force int2[0]=0;
end
```

Like with `force` statements, you can enter a `release` statement for an entire array, a slice of elements in the array, or a single element.

You can use a variable to specify the element of an unpacked array, for example:

```
int int1;
logic l1 [2:0];

initial
begin
int1=2;
force l1[int1]=3;
end
```

If however, you have and unpacked array of packed bits, the packed bits must be specified with a value or a parameter, not a variable, for example:

```
int int1=1;
parameter p1=4;
logic log1 [2:0];
logic [5:3]log2[2:0];

initial
begin
force log1[int1]=1;
force log2[int1][p1]=1;
end
```

The `const` constant construct will not work for specifying packed bits.

Structures

You can force and release both entire packed and unpacked structures, or individual members of the structure, for example:

```
typedef struct{
    int int1;
    bit [31:0] packedbit;
    integer intg1;
} unpackedstruct;

typedef struct packed{
    logic [2:0] log1;
    bit [2:0] bit1;
}packedstruct;

module test;

unpackedstruct ups1;
unpackedstruct ups2;
packedstruct ps1;
packedstruct ps2;

assign ups1.int1=30;

initial
begin
  #0 ups1.packedbit[0]=0;
  #20 force ups1.int1=1;
  #20 force ups2=ups1;
  #20 release ups2.int1;
  #20 release ups2;
  #20 force ps1.log1=1;
  #20 force ps1.bit1=0;
  #20 force ps2=ps1;
  #20 release ps1;
end
endmodule
```

Using the VPI

you can force and release on SystemVerilog variables using the VPI. The following conditions will apply:

- VPI force and release should behave exactly in the same manner as procedural statement `force` and `release`.
- If the vpi_put_value call contains a time delay information, VCS ignores it.
- If the vpi_put_value call contains a value with vpiReleaseFlag, VCS ignores the value argument of vpi_put_value.
- You cannot apply vpi_put_value to an entire struct. You can only apply it to individual members.

In the following example, SystemVerilog code makes a PLI call $forceX. In the PLI code, the SystemVerilog variables in that module are iterated and once the handle of the required variable is obtained, vpi_put_value is used with vpiForceFlag to force a value on to the variable. In the example, value 7 is forced on the variable through an argument (value_s) of vpi_put_value function call.

```
module dut(input int I, output int O);
int y;

always @(I) begin
     #10 y = I + 50;           // 2
     ⋮
     #20 $forceX();            // 3
     ⋮
     #20 $releaseX();          // 4
end
endmodule

// PLI code
```

```
     ⋮
void forceX()
{
   vpiHandle  var;
   int flag = vpiForceFlag;
   s_vpi_value value_s = {vpiIntVal};
   value_s.value.integer = 7;
     ⋮
        vpi_put_value(var, &value_s, NULL, flag);
     ⋮
}

void releaseX()
{
   vpiHandle  var;
   int flag = vpiReleaseFlag;
   s_vpi_value value_s = {vpiIntVal};
   value_s.value.integer = 70;
     ⋮
        vpi_put_value(var, &value_s, NULL, flag);
     ⋮
}
```

At time 40 ns, another PLI call $releaseX is called, which will release the previously forced object.

With respect to the behavior, an object can be forced from within SystemVerilog and released from PLI or vice versa. Otherwise both force and release can happen from PLI. In all these cases the behavior of force and release would be the same as SystemVerilog force and release. All the rules that govern the force and release on various data types from within SystemVerilog code will also apply to VPI force and release.

SystemVerilog Operators

SystemVerilog include the other assignment operators in C:

```
=+ -= *= /= %= &= |= ^= <<= >>= <<<= >>>=
```

The following table shows a few uses of these assignment operators and their equivalent in Verilog-2001.

operator example	Verilog-2001 equivalent
`b += 2;`	`b = b + 2;`
`b -= 2;`	`b = b - 2;`
`b *= 2;`	`b = b * 2;`

SystemVerilog includes the `++` increment and `--` decrement operators. The following table show how they work.

operator example	Verilog-2001 equivalent
`++b;` or `b++;`	`b = b + 1;`
`--b;` or `b--;`	`b = b - 1;`

You can place the `++` and `--` operators to the left or the right of the variable operand. Doing so makes a difference when these operators and their operand are in a larger expression, but enabling you to do so is not yet implemented in VCS, so you can only use them as simple assignment statements.

New Procedural Statements

In SystemVerilog `if` and `case` statements, including `casex` and `casez`, can be qualified by the `unique` or `priority` keyword and there is a `do while` statement.

unique and priority if and case Statements

The keyword `unique` preceding an `if` or nested `else if` statement specifies that one, and only one, conditional expression must be true. So in the following code:

```
unique if (l2!=0) $display("l2!=0");
else if (l2==3) $display("l2=3");
```

There are two conditional expressions: `(l2!=0)` and `(l2==3)`.

VCS evaluates these conditional expressions in parallel and, if both are true, it is a warning condition and VCS displays the following warning message:

```
RT Warning: More than one conditions match in 'unique if'
statement.
     "filename.v", line line_number, at time        sim_time
```

If neither conditional expression is true, it is also a warning condition and VCS displays the following warning message:

```
RT Warning: No condition matches in 'unique if' statement.
     "filename.v", line line_number, at time        sim_time
```

The keyword `priority` preceding an `if` or nested `else if` statement specifies that one conditional expression must be true. So in the following code:

```
priority if (l4!=0) $display("l4!=0");
else if (l4==3) $display("l4=3");
```

There are two conditional expressions: `(l4!=0)` and `(l4==3)`.

VCS evaluates these conditional expressions in sequence to see if they are true. VCS executes the first statement controlled by the first conditional expression that is true. In this example, therefore, VCS would display:

```
l4!=0
```

The `priority` keyword allows more than one conditional expression to be true. In this example the conditional expression `(l4==3)` could also be true but this would not be a warning condition.

If neither conditional expression were true, it would be a warning condition and VCS displays the following warning message:

```
RT Warning: No condition matches in 'priority if' statement.
     "filename.v", line line_number, at time        sim_time
```

The keyword `unique` preceding a `case`, `casex`, or `casez` statement specifies that one, and only one, case item expression can have the value of the case expression. So in the following code:

```
unique case (l2)
    0: $display("l2=%0d",l2);
    1: $display("l2=%0d",l2);
    !0: $display("l2 also !0");
endcase
```

There is the case expression `l2` and three case item expressions: `0`, `1`, and `!0`.

VCS evaluates these case item expressions in parallel and, if more than one has the value of the case expression, it is a warning condition and VCS displays the following warning message:

```
RT Warning: More than one conditions match in 'unique case'
statement.
     "filename.v", line line_number, at time        sim_time
```

If none of the case item expressions have the value of the case expression, it is also a warning condition and VCS displays the following warning message:

```
RT Warning: No condition matches in 'unique case' statement.
     "filename.v", line line_number, at time        sim_time
```

The keyword `priority` preceding a `case`, `casex`, or `casez` statement specifies that one case item expression must have the value of the case expression. So in the following code:

```
priority case (l4)
     0: $display("l4 !1");
     1: $display("l4=%0d",l4);
endcase
```

If `l4`'s value is neither `0` or `1`, it is a warning condition and VCS displays the following warning message:

```
RT Warning: No condition matches in 'priority case'
statement.
     "filename.v", line line_number, at time        sim_time
```

do while Statement

SystemVerilog has the `do while` statement. It's an alternative to the Verilog `while` statement, where an action is performed and then a condition is evaluated. Here is an example:

```
do #1 i1++; while (r1 == 0);
```

Here VCS repeatedly does these two things: increments signal i1 and then check to see if signal r1 is at 0. If when it checks r1 is no longer at 0, it stops incrementing i1. With a `while` statement, VCS would check on r1 before incrementing i1.

SystemVerilog Processes

SystemVerilog identifies the Verilog `always` block and its three variations, as static processes (There are dynamic processes that are not yet implemented). These three variations are as follows:

- `always_comb`
- `always_latch`
- `always_ff`

SystemVerilog also sees continuous assignment statements as static processes and you can use them to continuously assign to not just nets but also variables.

The always_comb Block

An `alway_comb` block models combinational logic. This block is to circumvent the problem of an `always` block with a missing `else` statement resulting in an unintended latch. The following is an example of an `always_comb` block:

```
always_comb
begin
  if (!mode)
    var1 = sig1 + sig2;
  else
    var1 = sig1 - sig2;
  var2 = sig1 + sig2;
end
```

SystemVerilog uses the term sensitivity list (from VHDL) and in an `always_comb` block the signals in the right-hand side of the assignment statements are inferred to be in the sensitivity list, meaning that any transition in these signals causes the execution of the `always_comb` block. In this example any transition in signals sig1 and sig2 cause the always block to execute.

To make sure that there is consistency between the expressions on the right-hand side of the assignments and the variables on the left-hand side, an `always_comb` block also executes at time zero, after the `initial` and `always` blocks that start at time zero begin executing.

There is a rule that there cannot be any other procedural assignment statements in the rest of the design that assign values to variables on the left-hand side of the assignment statements in an `always_comb` block. In this example there can be no other assignment statements in the design assigning values to var1 and var2.

An `alway_comb` block is similar to the `always` block with and implicit event control expression list, for example:

```
bit sig1 = 1;
bit sig2 = 1;

always @*
begin
  if (!mode)
    var1 = sig1 + sig2;
  else
    var1 = sig1 - sig2;
  var2 = sig1 + sig2;
end
```

This `always` block executes at time zero only because there were transitions on the signals in its implicit sensitivity list at time zero, in this example on signals sig1 and sig2. If there were to such time zero transitions, this `always` block would not execute at time zero. The `always_comb` block always executes at time zero so the `always_comb` block more accurately models proper wired logic behavior.

Another difference between such an `always` block and a similar `always_comb` block is that an `always_comb` block executes after all the `always` and `initial` blocks execute at time zero. The `always` block with and implicit event control expression list, if it executes at time zero, executes in no predictable order with the other `always` or `initial` blocks that execute at time zero.

If you have more than one `always_comb` block, there is no way to predict the order in which they execute at time zero other than that they execute after the `always` and `initial` blocks that execute at time zero.

You can consider an `always_comb` block, for example:

```
always_comb
bit1 = bit2 || bit3;

assign wire1 =  bit1;
```

To be analogous to a continuous assignment statement:

```
assign wire1 = bit2 || bit3;
```

The always_latch Block

The `always_latch` block models latched behavior, combinational logic with storage. The following is an example of an `always_latch` block:

```
always_latch
if (clk) sigq <= sigd;
```

The `always_latch` block is similar to the `always_comb` block in the following ways:

- It also has an inferred sensitivity list, it executes when there is a transition in the signals in the right-hand side of assignment statements.
- There can be no other assignment statements in the design that assign values to the variables in the left-hand side of its assignment statements.
- It always executes at time zero.

The difference between the `always_latch` and `always_comb` block is for synthesis. It's a way to make clear that you intend a latch for the code in the always_latch block.

The always_ff Block

The `always_ff` block models synthesizable sequential behavior. The following is an example of an `always_ff` block:

```
always_ff @(posedge clk or negedge reset)
   if (!reset)
       q <= 0;
   else
       q <= d;
```

An `always_ff` block can only have one event control.

Tasks and Functions

SystemVerilog changes tasks and functions in the following ways:

- Easier ways to declare task and function ports
- Function inout and output ports
- Void functions
- Tasks no longer requiring `begin end` or `fork join` blocks
- Returning values from a task or function before executing all the statements in a task or function

Tasks

The following is an example of a valid SystemVerilog task. Note the differences from what would be a Verilog-2001 task.

```
task task1 (input [3:2][1:0]in1, in2, output bit
```

```
[3:2][1:0]out);
logic tlog1,tlog2;
tlog1=in1[3][1];
⋮
#10 out[3][1]=tlog1;
⋮
tlog2=in2;
endtask
```

Lets take a look at a number of lines in the task:

```
task task1 (input [3:2][1:0]in1, in2, output bit
[3:2][1:0]out);
```

The task header declares three ports:

- in1 is declared as an `input` port. The keyword `input` is necessary only because it is a multi-dimensional packed array. In SystemVerilog a port can be a multi-dimensional array, packed or unpacked. In SystemVerilog task ports also have data types. The default data type is `logic`, so in1 has the `logic` data type.

  If in1 were an unpacked multi-dimensional array, you would not need the keyword `input` to make it an `input` port:

  ```
  in1 [3:2][1:0]
  ```

  instead of

  ```
  input [3:2][1:0] in1
  ```

- in2 takes both defaults. The default direction is an `input` port and the default data type is `logic`.
- out deviates from both defaults and so it must be specified as an `output` port with the `bit` data type. It is also a multi-dimensional packed array.

```
logic tlog1,tlog2;
```

Local scalar variables tlog1 and tlog2 are declared, with the `logic` data type. Ports have a default data type, local variables do not.

```
tlog1=in1[3][1];
⋮
#10 out[3][1]=tlog1;
⋮
tlog2=in2;
```

Notice that these assignment statements are not inside a `begin end` or `fork join` block. By default statements in a task are executed sequentially, just like they were in a `begin end` block. If you want, you can enclose these statements in a `begin end` or `fork join` block.

Functions

The following is an example of a valid SystemVerilog function and the code that calls the function. Note the differences from what would be a Verilog-2001 function.

```
function reg [1:0] outgo(reg [3:2][1:0] in1,in2, output int
out);
int funcint;
funcint = in1[3] >> 1;
⋮
if (in2==0)
   return 0;
out = funcint;
⋮
outgo = funcint;
endfunction
```

```
initial
begin
⋮
#1 reg2=outgo(reg1,log1,int2);
⋮
end
```

Lets take a look at a number of lines in the function:

```
function reg [1:0] outgo(reg [3:2][1:0] in1,in2, output int
out);
```

The function header specifies that the function name is outgo. It declares that it returns a two-bit value of the `reg` data type (A SystemVerilog function can also return a multi-dimensional array, or a structure or union). The default data type of the return value is `logic`. The header declares Three ports:

- in1 is an input port of the `reg` data type. It is a multi-dimensional packed array. In SystemVerilog function ports, like task ports, can be a multi-dimensional array. Also like task ports, function ports default to the `input` direction, so port in1 is an `input` port.
- in2 is a scalar port that takes both defaults, `input` direction and the `logic` data type.
- out is an `output` port so the direction must be specified, it is of the `int` data type so that too is specified. In SystemVerilog a function can have an `output` or an `inout` port.

```
int funcint;
```

Local scalar variable funcint is declared, with the `int` data type. Ports have a default data type, local variables do not.

```
funcint = in1[3] >> 1;
```

```
⋮
out = funcint;
⋮
if (in2==0)
   return 0;
outgo = funcint;
```

Notice that, just like SystemVerilog tasks, these assignment statements are not inside a `begin end` or `fork join` block. By default statements in a function are executed sequentially, just like they were in a `begin end` block. If you want, you can enclose these statements in a `begin end` or `fork join` block.

In these procedural statements:

1. The local variable funcint is assigned a shifted value of an element in the multi-dimensional array of input port in1.
2. The new value of funcint is assigned to the output port named out
3. SystemVerilog functions can have a `return` statement that overrides an assignment to the function name. Here if input port in2 equals zero, the function returns zero.
4. If the value of in2 is not zero, the value of the local variable funcint is returned by the function.

```
#1 reg2=outgo(reg1,log1,int2);
```

In this statement that calls the function (SystemVerilog function calls are expressions unless they are `void` functions, see below), signal reg2 is assigned the return value of the function. Signals reg1 and log1 input values to the function and the value of the output port is a structural driver of signal int2.

SystemVerilog also allow `void` functions that do not have a return value. A `void` function is called as a statement not as an expression, as it is in non-void functions. The following is an example of a `void` function and the code that calls it:

```
function void display(bit in1);
bit funcbit;
funcbit=in1;
$display("bit1=%0b",funcbit);
//return 1'bz;
endfunction

initial
begin
bit1=1;
display(bit1);
end
```

A void function cannot contain a `return` statement.

SystemVerilog Assertions

SystemVerilog assertions (SVA), just like OpenVera assertions (OVA), are a shorthand way to specify how you expect a design to behave and have VCS display messages when the design does not behave as specified. You can use both to accomplish the same thing. SystemVerilog assertions are in the SystemVerilog 3.1a standard promulgated by Accellera, the electronics industry wide organization for the advancement of hardware description languages. OpenVera assertions are part of the Synopsys proprietary OpenVera standard.

VCS has implemented both types of SystemVerilog assertions:

- immediate assertions
- concurrent assertions

Immediate assertions are a test of an expression when VCS executes the immediate assertion. An immediate assertions is a statement in procedural code.

Concurrent assertions specify how the design behaves during a span of simulation time.

Immediate Assertions

An immediate assertions resembles a conditional statement in that it has a boolean expression that is a condition. When VCS executes the immediate assertion it tests this condition and if it is true, VCS executes some statements in what is called a pass action block. If the condition is not true VCS executes statements in what is called a fail action block.

The following is an example of an immediate assertion:

```
module test;
⋮
initial
begin:named
⋮
a1:assert (lg1 && lg2 && lg3)
  begin: pass
    $display("%m passed");
    ⋮
  end
else
```

```
      begin: fail
        $display("%m failed");
        ⋮
      end
end
```

In this example the immediate assertion is labeled a1, and its expression `(lg1 && lg2 && lg3)` is the condition. If the condition is true, VCS executes the begin-end block labeled pass, this is the pass action block (it is not required to name this action block). If the condition is not true, VCS executes the begin-end block labeled fail (it is also not required to name this action block), it follows the keyword `else`.

If, for example, this immediate assertion passes, the condition is true, VCS displays the following:

```
test.named.a1 passed
```

Concurrent Assertions Overview

Concurrent assertions consists of one or more properties. A property consists of a clock signal and one or more sequences. In a property you can either specify a sequence of values on signals and the simulation time that occurs between these values, specified as clock ticks, or instantiate a sequence that you declare. You can declare a sequence and then use it as a building block in a property.

Sequences

A sequence enables you to build and manipulate sequential behavior. The following is an example of a sequence:

```
sequence s1;
```

```
sig1 ##1 sig2;
endsequence
```

Sequence s1 specifies that signal sig1 is true and then one clock tick later signal s2 is true. In this case the clock signal and the edge that specifies the clock tick, are in a property definition that instantiates this sequence. The ## operator specifies a delay of a number of clock ticks (the value you specify must be a non-negative integer).

You can specify any number of signal in the sequential expression and the logical negation operator, for example:

```
sequence s2;
sig1 ##1 sig2 ##2 !sig3;
endsequence
```

Sequence s2 specifies that signal sig1 must be true, and one clock tick later signal s2 must be true, and then two clock ticks after that signal s3 must be false.

You can use a sequence in another sequence as a building block in another sequence, for example:

```
sequence s1;
sig1 ##1 sig2;
endsequence

sequence s3;
s1 ##2 !sig3;
endsequence
```

Here sequence s1 is used in sequence s3.

You can declare a sequence in the following places in your code:

- In a module definition

- In an Interface definition
- At `$root` (in SystemVerilog `$root` means outside of any other definition, a sequence defined in `$root` is globally accessible).

Note:
The SystemVerilog LRM says that you can declare a sequence in a module definition but never in an `always` or `initial` block.

Using Formal Arguments in a Sequence

You can specify formal arguments in a sequence and then make substitutions when you use it in another sequence, for example:

```
sequence s1(sig3,sig4);
sig3 ##1 sig4;
endsequence

sequence s2;
s1(sig1,sig2) ##1 sig5;
endsequence
```

Specifying a Range of Clock Ticks

You can specify a range of clock ticks in the delay, for example:

```
sequence s1;
sig1 ##[1:3] sig2;
endsequence
```

This sequence specifies that sig1 must be true and then sig2 must be true at either the first, second, or third subsequent clock tick.

You can specify that the range end at the end of the simulation with the `$` token, for example:

```
sequence s1;
```

```
sig1 ##[1:$] sig2;
endsequence
```

The operands in the sequences need to be boolean expressions, they don't have to be just signal names, for example:

```
sequence s1;
sig1 == 1 ##1 sig3 || sig4;
endsequence
```

Unconditionally Extending a Sequence

You can unconditionally extend a sequence by using the literal true value 1 or a text macro defined as 1, for example:

```
sequence s2;
sig1 ##1 sig2 ##2 !sig3 ##3 1;
endsequence
```

Here sig1 must be true, one clock tick later sig2 must be true, then two clock ticks later sig3 must be false, but the sequence doesn't end until three clock ticks later. Extending a sequence can be handy when you are using a sequence in another sequence.

Using Repetition

There are three operators for specifying the repetition of a sequence:

- The consecutive repetition operator `[*`
- The goto repetition operator `[->`
- The non-consecutive repetition operator `[=`

The consecutive repetition operator `[*` is for specifying consecutive repetitions of a sequence, for example, the following sequence:

```
sequence s1;
sig1 ##1 sig2 ##1 sig2 ##1 sig2;
endsequence
```

Can be shortened to the following:

```
sequence s1;
sig1 ##1 sig2 [*3];
endsequence
```

Note:

The value you specify with the `[*` operator must be a positive integer.

You can use repetition in a range of clock ticks, for example:

```
sequence s1;
(sig1 ##2 sig2) [*1:3];
endsequence
```

This sequence specifies that the sequence is run at the length of itself, or the length of itself doubled, or tripled. This sequence is the equivalent of all of the following:

```
sequence s1;
(sig1 ##2 sig2);
endsequence

sequence s1;
(sig1 ##2 sig2 ##1 sig1 ##2 sig2);
endsequence

sequence s1;
(sig1 ##2 sig2 ##1 sig1 ##2 sig2 ##1 sig1 ##2 sig2);
endsequence
```

You can specify an infinite number of repetitions with the $ token, for example:

```
sequence s1;
(sig1 ##2 sig2) [*1:$];
endsequence
```

Note:

`##1` is automatically added between repetitions.

The goto repetition operator `[->` (non-consecutive exact repetition) specifies the repetition of a boolean expression, such as:

```
a ##1 b [->min:max] ##1 c
```

This is equivalent to:

```
a ##1 ((!b [*0:$] ##1 b)) [*min:max]) ##1 c
```

Adding the range specification to this allows the construction of useful sequences containing a boolean expression that is true for at most N occurrences:

```
a ##1 b[->1:N] ##1 c // a followed by at most N occurrences
                     // of b, followed by c
```

The non-consecutive repetition operator `[=` extends the goto repetition by extra clock ticks where the boolean expression is not true.

```
a ##1 b [=min:max] ##1 c
```

This is equivalent to:

```
a ##1 ((!b [*0:$] ##1 b)) [*min:max]) ##1 !b[*0:$] ##1 c
```

The above expression would pass the following sequence, assuming that 3 is within the min:max range.

```
a c c c c b c c b c b d d d c
```

Specifying a Clock

You can specify a clock in a sequence, for example:

```
sequence s1;
@(posedge clk) sig1 ##1 sig2;
endsequecle
```

This sequence specifies that the clock tick is on the rising edge of signal clk.

Value Change Functions

You can also include the following system functions in a sequential expression. They tell you about value changes between clock ticks:

`$sampled(`*`expression`*`)`
: Returns the sampled value of the expression with respect to the last occurence of the clocking event.

`$rose(`*`expression`*`)`
: If the expression is just a signal, returns 1 if the least significant bit of the signal changed to 1 between clock ticks. If the expression is more than one signal and an operator, for example `sig1 + sig2`, returns 1 if the value of the least significant bit in the evaluation of the expression changes from 0, X, or Z to 1.

`$fell(`*`expression`*`)`
: If the expression is just a signal, returns 1 if the least significant bit of the signal changed to 0 between clock ticks. If the expression is more than one signal and an operator, for example `sig1 + sig2`, returns 1 if the value of the least significant bit in the evaluation of the expression changes from 1, X, or Z to 0.

`$stable(`*`expression`*`)`
: Returns 1 if the value of the expression does not change between clock ticks. A change in a four state signal from X to Z returns false.

The following is an example of using these system functions:

```
sequence s1;
$rose(sig1) ##1 $stable(sig2 && sig3);
endsequence
```

Anding Sequences

You can use the `and` operator to specify that two sequences must occur (succeed), but not necessarily at the same time. The following is an example:

```
sequence s1;
sig1 ##1 sig2;
endsequence

sequence s2;
sig3 ##2 sig4;
endsequence
sequence s3;
s1 and s2;
endsequence
```

Sequence s3 succeeds when both sequences s1 and s2 succeed. The time of the success of s3 is whenever the last of s1 or s2 succeed.

Intersecting Sequences (And with Length Restriction)

You use `intersect` operator to specify the match of the operand sequential expressions at the same clock tick, for example:

```
sequence s1;
l1 ##1 l3;
endsequence

sequence s2;
l2 ##1 l4;
endsequence

sequence s3;
s1 intersect s2;
endsequence
```

In this example sequence s3 can match because sequences s1 and s2 both have the same number of clock ticks, sequence s3 does match when sequences s1 and s2 match.

Oring Sequences

You can use the `or` operator to specify that one of two sequences must succeed. The following is an example:

```
sequence s1;
sig1 ##1 sig2;
endsequence

sequence s2;
sig3 ##2 sig4;
endsequence

sequence s3;
s1 or s2;
endsequence
```

Sequence s3 succeeds when either sequences s1 and s2 succeed but only when sequences s1 and s2 start at the same time.

Only Looking For The First Match of a Sequence

The `first_match` operator specifies that a sequential expression matches only once. After its first success, VCS no longer looks for subsequent matches.

```
sequence s1;
first_match(l1 ##[1:2] l2);
endsequence
```

In s1, if l1 is true at the first clock tick, the expression could match at the next clock tick, or the one after that, but the use of the `first_match` operator means that VCS does not monitor for the second possible match if the first possible match occurs.

Conditions for Sequences

You can use the `throughout` operator to specify a condition that must be met throughout the sequence in order for the sequence to succeed, for example:

```
sequence s1;
(sig3 || sig4) throughout sig1 ##1 sig2;
endsequence
```

For this sequence to succeed, not only must sig1 be true and then in the next clock tick, sig2 is true, but also for both clock ticks the expression `(sig3 || sig4)` must evaluate to true.

Specifying that Sequence Match Within Another Sequence

You use the `within` operator to require that one sequence to begin and match when or after another starts but before or when the other one matches, for example:

```
sequence s1;
l1 ##3 l4;
endsequence

sequence s2;
l2 ##1 l3;
endsequence

sequence s3;
s2 within s1;
endsequence
```

Sequence s1 requires three clock ticks, sequence s2 only requires one clock tick, So it is possible for s2 to begin and end during s1, signal l2 to toggle to true after l1 does, a clock tick later, l3 toggles to true, and l4 toggling to true a clock tick later.

Using The End Point of a Sequence

Sequences have an `ended` method that you can use in another sequence to specify that the other sequence includes the end of the first sequence, for example:

```
sequence s1;
@(posedge clk) sig1 ##1 sig2;
endsequence

sequence s2;
s1.ended ##1 sig3;
endsequence
```

Note:

If you are referencing a sequence in another sequence, and you are using the ended method in the second sequence, the referenced sequence must specify its clock tick. The first sequence above does so by beginning with `@(posedge clk)`.

You cannot use the ended method on a sequence with formal arguments. Use (or instantiate) such a sequence in another sequence, ant then you can use the method on the other sequence, for example:

```
sequence s1(sig3,sig4);
@(posedge clk) sig3 ##1 sig4;
endsequence

sequence s2;
s1(sig3,sig4);
endsequence

sequence s3;
s2.ended ##1 sig1;
endsequence
```

Level Sensitive Sequence Controls

You can use a SystemVerilog assertion sequence as an event control expression. You can also use a sequence as the conditional expression in a `wait`, `if`, `case`, `do while`, or `while` statements, if you use the `triggered` sequence method.

The `triggered` sequence method evaluates to true if the sequence successfully completes during the same time step. You use this method in an expression that includes the sequence name, immediately followed by a period (.) and then the keyword `triggered`, for example:

```
if (sequence1.triggered)
```

Level sensitive sequence controls are documented in section 8.11, starting on page 93, of the SystemVerilog 3.1a Language Reference Manual.

The following annotated code example shows using a sequence for these purposes:

```
module test;
logic l1,l2,clk;

sequence s1;
@ (posedge clk) l1 ##1 l2;
endsequence
```

Sequence s1 specifies that when there is a rising edge on variable clk, variable l1 is true, and with the next rising edge on clk, variable l2 is true.

```
initial
begin
clk=0;
#4 l1=1;
#10 l2=1;
#3 $finish;
end

always
#5 clk=~clk;
```

There will be a rising edge on variable clk at time 5 and 15. Simulation ends at time 17. At the first rising edge, l1 will be true, at the second rising edge, l2 will be true. The sequence will occur.

```
always @(s1)
$display("sequence s1 event control at %0t\n",$time);
```

Sequence s1 is an event control expression.

```
initial
begin
wait (s1.triggered)
$display("wait condition s1.triggered\n");
```

The `triggered` method with sequence s1 is the conditional expression for the `wait` statement.

```
if (s1.triggered)
$display("if condition s1.triggered\n");
case (s1.triggered)
1'b1 : $display("case condition s1.triggered happened\n");
1'b0 : $display("s1.triggered did not happen\n");
endcase
do
   begin
     $display("do while condition s1.triggered\n");
     $finish;
   end
while (s1.triggered);
end

endmodule
```

The `triggered` method with sequence s1 is also the conditional expression for the `if`, `case`, and `do while` statements.

Sequence s1 does occur, so VCS displays the following:

```
sequence s1 event control at 15

wait condition s1.triggered

if condition s1.triggered

case condition s1.triggered happened
```

```
do while condition s1.triggered
```

Properties

A property says something about the design, so a property evaluates to true or false.

Concurrent assertions use properties and properties contain sequences, either instantiated or containing sequential expressions like a sequence. Both of the following sequences and all but the last of the following properties are valid:

```
sequence s1;
sig1 ##1 sig2;
endsequence

sequence s2;
@(posedge clk) sig3 ##1 sig4;
endsequence

property p1;
@(posedge clk) s1;
endproperty

property p2;
@(posedge clk) s2;
endproperty

property p3;
@(posedge clk2) sig1 ##1 sig2;
endproperty

property p4;
@(posedge clk2) s2;  //illegal
endproperty
```

The last property is invalid because it instantiates a sequence with a different clock tick than the clock tick for the property. In the valid properties you see the following:

- How to specify the clock tick in the property for the sequence it instantiates.
- How you can specify a clock tick both in the property and in the sequence instantiated in the property, if they specify the same clock tick.
- That instead of instantiating a sequence you can include a sequential expression like `sig1 ##1 sig2`.

You can declare a property in the following places in your code:

- In a module definition
- In an interface definition
- At `$root` (in SystemVerilog `$root` means outside of any other definition, a property defined in `$root` is globally accessible).

Note:
The SystemVerilog LRM says that you can declare a property in a module definition but never in an `always` or `initial` block.

Using Formal Arguments in a Property

Like sequences, you can include formal arguments in properties. Unlike sequences you cannot use or instantiate a property in another property. You use or instantiate a property in a concurrent assertion, for example:

```
property p3 (sig1,sig2);
@(posedge clk2) sig1 ##1 sig2;
endproperty
```

```
a1: assert property (p3(sig3,sig4));
```

Here the property uses signals sig1 and sig2 in its sequential expression, but signals sig3 and sig4 replace them in the assertion declaration.

Implications

Property implications, contain a boolean or sequential expression, called an antecedent, which must be true or match before VCS starts to monitor the events in another sequence or sequential expression, called the consequent, to see if that sequence matches.

There are two types of implications:

- Overlapping implications where the antecedent and the first event in the declared sequence or sequential expression happen during the same clock tick. For overlapping implications you enter the `|->` operator between the antecedent and the consequent.
- Non-overlapping implications where there is a clock tick delay between the antecedent and the consequent. For non-overlapping implications you enter the `|=>` operator between the antecedent and the consequent.

The following are examples of the VCS implemented implication constructs:

```
sequence s1;
sig1 ##1 sig2;
endsequence

property p1;
@(posedge clk) (sig3 || sig4) |-> s1;
endproperty
```

Property p1 contains an overlapping implication. It specifies checking that `(sig3 && sig4)` is true and if so, during the same clock tick, checking to see if sig1 is true, and then, a clock tick later, checking to see if sig2 is true.

```
property p2;
@(posedge clk) (sig1 ##1 sig2) |-> (sig3 ##1 sig4);
endproperty
```

Property p2 also contains an overlapping implication. In p2 the antecedent is a sequential expression.

```
property p3;
@(posedge clk) (sig3 ##1 sig4) |=> ##1 (sig1 ##1 sig2);
endproperty
```

Property p3 contains a non-overlapping implication. The first event is the sequential expression, sig1 being true, must happen one clock tick after the antecedent expression is true.

Remember that a property is either true or false, so for a property to be true, by default the antecedent must be true and the consequent must succeed. If you include the keyword `not` between the implication operator and the consequent, the property is true if the consequent does not succeed, for example:

```
property p4;
@(posedge clk) (sig3 && sig4) |-> not (sig1 ##1 sig2);
endproperty
```

Property p4 is true if, when `(sig3 && sig4)` is true, sig1 is not true, or if it is, one clock tick later, sig2 is not true.

Inverting a Property

The keyword `not` can also be used before the declared sequence or sequential expression, or if it contains an implication, before the antecedent, to make the property true, if it otherwise would be false, or make the property false if it otherwise would be true, for example:

```
sequence s1;
sig1 ##1 sig2;
endsequence

property p1;
@(posedge clk) not s1;
endproperty
```

Property p1 is true if sig1 is never true, or if it is, one clock tick later sig2 is never true.

```
sequence s2;
@(posedge clk2) sig4 ##1 sig5;
endsequence

property p2;
not s2;
endproperty
```

Property p2 is true if sig4 is never true, or if it is, one clock tick later sig5 is never true.

```
property p3;
@(posedge clk) not (sig3 && sig4) |-> not sig1 ##1 sig2;
endproperty
```

Property p3 is true if the implication antecedent is never true, or if it is, the consequent sequential expression succeeds. Notice here that the keyword `not` occurs twice in the property.

Past Value Function

SystemVerilog has a `$past` system function that returns the value of a signal from a previous clock tick. The following is an example of its use:

```
property p1;
@(posedge clk) (cnt == 0) ##3 ($past(cnt,3)==0);
endproperty
```

This rather elementary use of the `$past` system function returns the value of signal cnt from three clock ticks ago. The first argument, an expression, is required. The second argument, a number of clock ticks previous to the current clock tick, is optional and defaults to 1.

The disable iff Construct

The `disable iff` construct enables the use of asynchronous resets. The specify a reset condition in which all attempts that have started for properties immediately succeed and all subsequent attempts succeed as soon as they start.

The following shows a use of the `disable iff` construct:

```
initial
begin
clk=0;
rst=0;
sig1=1;
#7 sig2=1;
⋮
end
always
#5 clk=~clk;

sequence s1;
```

```
sig1 ##1 sig2;
endsequence

property p1;
@(posedge clk) disable iff (rst) s1;
endproperty

a1: assert property (p1);
```

If during simulation sig2 turns false the property no longer succeeds. If, some time later, rst turns true, the property starts to succeed again. If rst turns false again, the property once again no longer succeeds.

assert Statements

VCS never checks a property or a sequence unless it is instantiated in a concurrent assertion. The concurrent assertion enforces the property or sequence as a checker of that property or sequence.

A concurrent assertion takes the form of an `assert` statement. The following is an example of an `assert` statement:

```
a1: assert property (p1);
```

In this `assert` statement:

`a1:`
: Is the instance name of the concurrent assertion. Concurrent assertions have hierarchical name beginning with the hierarchical name of the module instance in which they are declared, and ending with this instance name. Instance names are optional.

`assert`
: Is a keyword for declaring a concurrent assertion.

`property`
: Is a keyword for instantiating both a property or a sequence.

`p1`
: Is the property instantiated in the concurrent assertion. You could also have specified a sequence instead of a property.

You can declare a concurrent assertion, and enter an `assert` statement, in the following places in your code:

- In a module definition
- In an Interface definition
- In `$root`

Note:

- In the VCS implementation, you can declare a concurrent assertion in a module definition including inside an `always` block but not in an `initial` block. The Accellera SystemVerilog LRM allows concurrent assertions in `initial` blocks.

If a property has formal arguments you can replace them with other signals as shown in “Using Formal Arguments in a Property” on page 15-58.

cover Statements

The `cover` statement calls for the monitoring of a property or a sequence. VCS looks for matches, how often the property was true or how often the sequence occurred. When simulation is over, VCS displays the results of this monitoring.

A `cover` statement is syntactically similar to an `assert` statement. The following is an example of a `cover` statement:

```
c1: cover property (p1);
```

In this `cover` statement:

`c1:`
: Is the instance name of the `cover` statement. `cover` statements have hierarchical name beginning with the hierarchical name of the module instance in which they are declared, and ending with this instance name. Instance names are optional.

`cover`
: Is a keyword for declaring a `cover` statement.

`property`
: Is a keyword for instantiating both a property or a sequence.

`p1`
: Is the property instantiated in the cover statement. You could have specified a sequence instead of a property.

The Following SVA code:

```
sequence s1;
@(posedge clk) sig1 ##[1:3] sig2;
endsequence

sequence s2;
sig3 ##[1:3] sig4;
endsequence

property p1;
@(posedge clk) sig1 && sig2 |=> s2;
endproperty

a1: assert property (p1);
a2: assert property (@(posedge clk)s1);
```

```
c1: cover property (p1);
c2: cover property (@(posedge clk)s1);
endmodule
```

Contains two `cover` statement and VCS, for example, displays the following after simulation as a result of these `cover` statements:

```
"exp3.v", 31: test.c1, 9 attempts, 16 total match, 7 first match, 1 vacuous match
"exp3.v", 32: test.c2, 9 attempts, 21 total match, 8 first match, 0 vacuous match
```

This display is explained as follows:

- In the first line:
  - The `cover` statement is in source file exp3.v.
  - The instance of the `cover` statement is test.c1. It is declared in module test.
  - There were nine attempts to cover the property p1.
  - In those nine attempts there were 16 times that the properly was true. There can be more than one match in a attempt. In this case in property p1, in sequence s2, a match can occur over a range of clock ticks and in this case more than once in the range.
  - There were seven first matches. The property was true seven times at the start of the range.
  - There was a vacuous match. Property p1 contains and implication. The antecedent `sig1 && sig2` was false making the implication vacuously true because it doesn't mean that the consequent sequence `s2` occurred.
- In the second line:
  - The `cover` statement is in source file exp3.v.

- The instance of the `cover` statement is test.c2. It is declared in module test.
- There were nine attempts to cover the sequence s1.
- In those nine attempts there were 21 times that the sequence occurred.
- There were no vacuous matches because the `cover` statement does not instantiate a property with an implication.

You can declare a `cover` statement, in the following places in your code:

- In a module definition
- In an Interface definition
- In `$root`

Note:

In the VCS implementation, you can declare a `cover` statement in a module definition including inside an always block but not in an `initial` block. The Accellera SystemVerilog LRM allows `cover` statements in `initial` blocks.

`cover` statements, unlike `assert` statements, only have a pass action block, not a fail action block, in the VCS implementation.

Action Blocks

`assert` statements can have a pass and a fail action block, and `cover` statements can have a pass action block. The pass block executes when the `assert` or `cover` statement succeeds. The fail block, that follows the keyword `else`, executes when the `assert` statement fails. The following are examples of these blocks:

```
a1: assert property (p1)
begin
  $display("p1 succeeds");
  passCount ++;
end
else
begin
  $display("p1 does not succeed");
  failCount ++;
end

c1: cover property (p1)
begin
 $display("p1 covered");
 coverCount ++;
end
```

Binding an SVA Module to a Design Module

You can define a module that contains just SVA `sequence` and `property` declarations, and `assert` and `cover` statements. The module ports are signals in these declarations and statements. These ports are also signals in a design module (a module that contains behavioral or RTL code or other types of design constructs).

you can then bind the SVA module to the design module and it is the same as instantiating the SVA module in the design module. The following is an example of a design module, and SVA module and a `bind` directive that binds the SVA module to the design module:

```
module dev;
logic clk,a,b;
⋮
endmodule

module dev_checker (input logic CLK, input logic A,
```

```
input logic B);
property p1;
  @(posedge CLK) A ##1 B;
endproperty

a1: assert property(p1) else $display("p1 Failed");
endmodule

bind dev dev_checker dc1 (clk,a,b);
```

In this `bind` directive:

`bind`
: Is a keyword that starts the bind directive

`dev`
: is the module identifier (name) of the module to which you want to bind the SVA module

`dev_checker`
: Is the module identifier of the SVA module

`dc1`
: Is an instance name of the SVA module.

`(clk,a,b)`
: Is a port connection list to the SVA module

You can also bind an SVA module to a design module instances, for example:

```
module top;
logic clk,a,b;
⋮
dev d1 (clk,a,b);

endmodule

module dev (input logic clk, input logic a, input logic b);
```

```
⋮
endmodule

module dev_checker (input logic clk, input logic a, input
logic b);
property p1;
  @(posedge clk) a ##1 b;
endproperty

a1: assert property(p1) else $display("p1 Failed");
endmodule

bind top.d1 dev_checker dc1 (clk,a,b);
```

In this `bind` directive top.d1 is an instance of module dev.

IMPORTANT:

Binding to an instance that is generated using a `generate` statement is not supported.

Parameter Passing in a bind Directive

The module containing the SVA code that is bound to the design module is instantiated in the bind directive and as such you can use parameter passing in the instantiation. The following is an example:

```
`timescale 1ns/1ns
module dev;
logic clk,sig1,sig2;
⋮
endmodule

module dev_check(input logic CLOCK,input logic SIG1, input
logic SIG2);
parameter ticks =5;

property p1;
@(posedge CLOCK) SIG1 ##ticks SIG2;
```

```
endproperty

a1: assert property(p1) else $display("\n\np1 failed\n\n");
endmodule

bind dev dev_check #(10) dc1 (clk,sig1,sig2);
```

Notice that module dev_check, that contains the SVA code, also has a parameter for specifying the number of clock ticks in the property. In the parameter declaration it has a value of 5, but its value is changed to 10 in the `bind` directive, like in any other module instantiation.

The VPI for SVA

This VPI is to enable you to write an applications that reacts to SVA events and to enable you to write SVA waveform, coverage, and debugging tools.

Note:

To use this API you need to include the sv_vpi_user.h file along with the vpi_user.h file in the $VCS_HOME/include directory. See section 28 of the SystemVerilog 3.1a LRM

This section describes the differences between the VCS implementation and section 28 of SystemVerilog 3.1a LRM.

- In subsection 28.3.2.2 “Extending vpi_get() and vpi_get_str,” use `vpiDefFileName` instead of `vpiFileName`.
- In subsection 28.4.1 “Placing assertion system callbacks,” `cbAssertionSysStop` is not supported.
- In subsection 28.4.2 “Placing assertion callbacks,” the `failEpr` step information is not supported.

- In subsection 28.5.1 “Assertion system control,” `vpiAssertionSysStart` and `vpiAssertionSysStop` are not supported.
- In subsection 28.5.2 “Assertion control,” `vpiAssertionDisableStep` and `vpiAssertionEnableStep` are not supported.

Also the following, in the static part, are not supported:

- Assignments inside sequence expressions
- Bit-selects or part-selects of formal arguments inside a sequence expression

System Verilog Assertion Local Variable Debugging

VCS includes four callback types that you can use to debug SVA local variables. You register these callback types on a VPI assertion handle using the vpi_register_assertion_cb method.

These callback types are as follows:

`CreationcbAssertionLocalVarCreated`
: VCS calls this callback type when VCS creates the SVA local variable. This happens when a new SVA attempt starts.

`cbAssertionLocalVarUpdated`
: VCS calls this callback type when it updates an SVA local variable. The new value might be same as the old value.

`cbAssertionLocalVarDuplicated`
: VCS calls this callback type when it duplicates an SVA local variable. This happens when an attempt forks off into multiple paths.

`cbAssertionLocalVarDestroyed`

VCS calls this callback type when it destroys an SVA local variable. This happens when an assertion succeeds or fails.

All these callback types return a handle to the local variable that caused the event. Use the handle provided in the callback to get the name and value of the local variable. Your application is responsible for keeping track of the destroyed local variable handles. Using a destroyed variable handle results in unpredictable behavior.

You use the vpi_register_assertion_cb method to register a callback on an SVA. Whenever a local variable event happens on that SVA, VCS invokes the corresponding callback with that local variable handle embedded in the attempt information.

Note:

You do not have the flexibility to register a callback on a single local variable or part (bit or part select) of the variable. For embedding the local variable handle the attempt info structure is extended as follows.

```
typedef struct t_vpi_attempt_info {
  union {
    vpiHandle failExpr;
    p_vpi_assertion_step_info step;
    p_vpi_attempt_local_var_info local_var_info;
  } detail;
  s_vpi_time attemptStartTime; /* Time attempt triggered */
} s_vpi_attempt_info, *p_vpi_attempt_info;

typedef struct t_vpi_attempt_local_var_info {
  vpiHandle localVar;
} s_vpi_attempt_local_var_info,
*p_vpi_attempt_local_var_info;
```

Note that if VCS duplicates the SVA local variable, the returned vpi_attempt_info structure contains the handle to the new local variable. Your application needs to keep track of all copies of local variable for a particular attempt.

These callback types have the following imitations:

- The name or fullname of the SVA local variable does not contain the name of sequence or property it is declared in.
- The local variable handle supports only vpiType, vpiName, vpiFullName and getValue. There is no support for other properties defined on normal VPI variables.
- VCS treats XMR (cross module reference) SVA local variables as normal SVA local variables, so you can not get the XMR part in the local variable name. vpiFullName considers the sequence or property instantiated as if it is declared in the scope containing the assertion.
- Change in part of variable, for example a one bit change in a vector results in a callback of the full variable. Your application is responsible for identifying the changed part.
- No debug support for structure/union/classes in the initial implementation. Your application can get callbacks on other local variables in the assertion. VCS ignores only unsupported callback types.

Hierarchy

SystemVerilog contains enhancements for representing the design hierarchy:

- The `$root` top-level global declaration space
- New data types for ports
- Instantiation using implicit .name connections
- Instantiation using implicit .* connections
- New port connection rules for variables

The `$root` Top-Level Global Declaration Space

In SystemVerilog there is the `$root` top-level declaration space which not only contains all uninstantiated modules, but also interface definitions must be declared here (in the current implementation interface definitions cannot be inside module definitions or other interfaces), user-defined tasks and functions, parameter, nets and variables, and type definitions.

Some examples of `$root` declarations are as follows:

```
parameter msb=7;

typedef int myint;

wire w1,w2,w3,w4;

logic clk;

and and1 (w2,w3,w4);
```

```
tran tr1 (w1,w4);

primitive prim1 (out,in);
input in;
output out;
table
// in : out
    0 : x;
    1 : 0;
    x : 1;
endtable
endprimitive

event recieved_data;

task task1 (input [3:2][1:0]in1, in2, output bit
[3:2][1:0]out);
logic tlog1,tlog2;
tlog1=in1[3][1];
⋮
#10 out[3][1]=tlog1;
⋮
tlog2=in2;
endtask

function void left (output myint k);
   k = 34;
   $display ("entering left");
endfunction

interface try_i;
wire [7:0] send, receive;
endinterface

module top1(w1);
output w1;
endmodule

module top2(w1);
input w1;
endmodule
```

All constructs that are declared or defined in `$root` are, for the most part, accessible to the entire design. Any module, no matter its place in the hierarchy, can use the parameter, use the type, read or write to these variables, use the named event, call the task and function, or instantiate the interface. The gate and switch primitive cannot be instantiated in the rest of the design (they are already instantiated in `$root`) but the rest of the design can write to their inputs and read their outputs. The UDP can be instantiated in the rest of the design. The module definitions, not instantiated elsewhere, are top-level modules. Note that they connect to `$root` level wire w1.

The Accellera SystemVerilog 3.1a specification says that `$root` can also contains lone procedural statements (`$root` cannot have `initial` or `always` blocks) and SystemVerilog assertions sequence, property and assertion declarations, but this is not yet implemented in VCS.

New Data Types for Ports

In SystemVerilog a module `input` or `output` port can be any net data type or any variable data type including an array, a structure, or a union, for example:

```
typedef struct {
   bit bit1;
   union packed{
            int int1;
            logic log1;
            } myunion;
} mystruct;

module mod1(input int in1, byte in2, inout wire io1, io2,
             output mystruct out);
:
```

```
.
endmodule
```

In the module header for module mod1:

1. The first port is named in1. It is specified as an `input` port with the `int` data type.
   If we omitted both the direction and data type, SystemVerilog expects the port to be declared following the header.
   If only the direction is omitted, it defaults to `inout`.
   If only the data type is omitted, it defaults to the `wire` net data type (which you can change with the `` `default_nettype `` compiler directive to another net data type).
2. The second port is named in2. No direction is specified so it inherits the direction from the previous port, so in2 is an `input` port with the `byte` data type.
3. The third port is named io1. It's specified as an `inout` port with the `wire` data type.
4. The fourth port is named io2. Not being the first port in the list, it inherits the direction and data type from port io1.
5. The last port is named out. It is an output port that is the structure mystruct.

You still can only use net data types for `inout` ports.

The Accellera SystemVerilog 3.1a specification says that named events can also be ports, but this is not implemented in VCS.

Instantiation Using Implicit .name Connections

In SystemVerilog if the name and size of a port matches the name and size of the signal that connects to that port, you can make connections in any order without matching the order of the ports or using a name based connection list where you have to enter each port and the signal that connects to it, for example:

```
module top;
logic [7:0] log1;
byte byt1 [1:0];

dev dev1(.log1,.byt1);
endmodule

module dev(input byte byt1 [1:0], input logic [7:0] log1);
⋮
endmodule
```

Module top instantiates module dev. In the module instantiation statement in module top, the connection list has signal log1 first, followed by signal byt1. In the module header for module dev, the port connection list has port byt1 first followed by port log1.

In Verilog-2001 or Verilog-1995 you would need a name based connection list in the module instantiation statement:

```
dev dev1(.log1(log1),.byt1(byt1));
```

Instantiation Using Implicit .* Connections

Also in SystemVerilog, if the name and size of a port matches the name and size of the signal that connects to that port, you use a period and as asterisk to connect the signals to the ports, similar to

an implicit event control expression list for an always block, for example:

```
module top;
logic [7:0] log1;
byte byt1 [1:0];

dev dev1(.*);
endmodule

module dev(input byte byt1 [1:0], input logic [7:0] log1);
⋮
endmodule
```

New Port Connection Rules for Variables

SystemVerilog allows you to declare an `input` port to be a variable. This is another way to use a structural driver, see “Writing to Variables” on page 15-16. If you declare an `input` port to be a variable, you cannot have multiple drivers, so you cannot do any of the following:

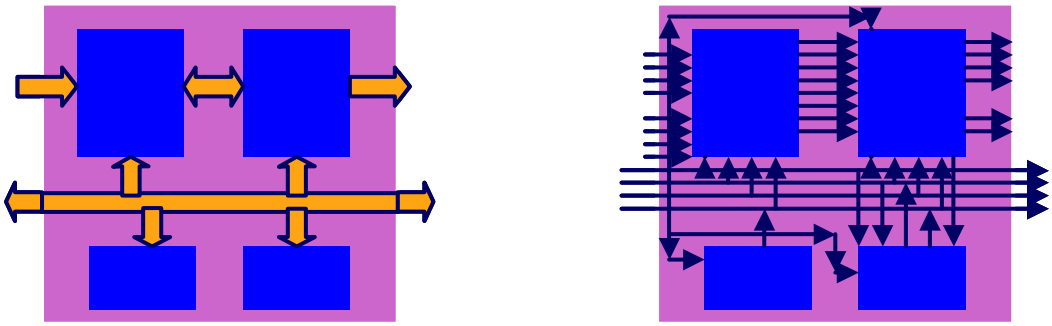
- Assign values to the variable with a procedural assignment statement or a user-defined task enabling statement.
- Assign values to it with a continuous assignment statement or have values propagate to it from a module instance, gate, switch-level primitive, or UDP.

Like Verilog-2001 and Verilog-1995, SystemVerilog allows you to declare an `output` port to be a variable and prohibits the same for `inout` ports.

Interfaces

Interfaces were developed because most bugs occur between blocks in a design and interfaces help eliminate these wiring errors. Interfaces are a way of encapsulating the communication and interconnect between these blocks, but they are also more than just that. They help you to develop a divide and conquer methodology and are re-usable in other places in a design and in other designs.

Figure 15-4 Block Diagrams



Consider the wide arrows in the block diagram on the left to be interfaces. They are more than just wire bundles, They are an implementation of how to communicate between blocks. As interfaces, they help you to focus on how information is communicated between block.

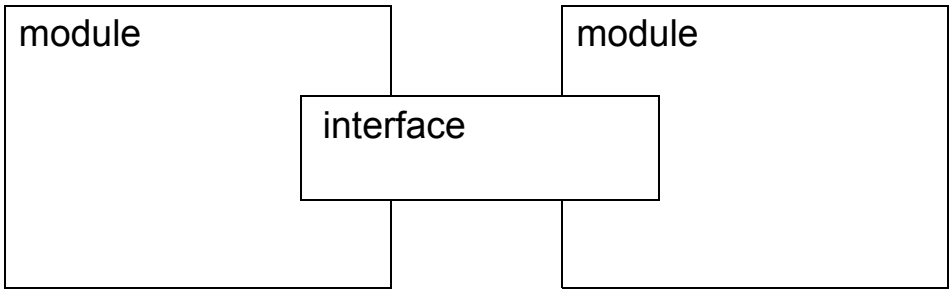
At its simplest level, an interface encapsulated communication like a struct encapsulates data:

```
typedef struct{
int int1;
logic [7:0] log1;
} s_type;
```

```
interface intf;
int int1;
wire [7:0] w1;
endinterface
```

Think of a wire as a built-in interface.

An interface is like a module that straddles other modules.

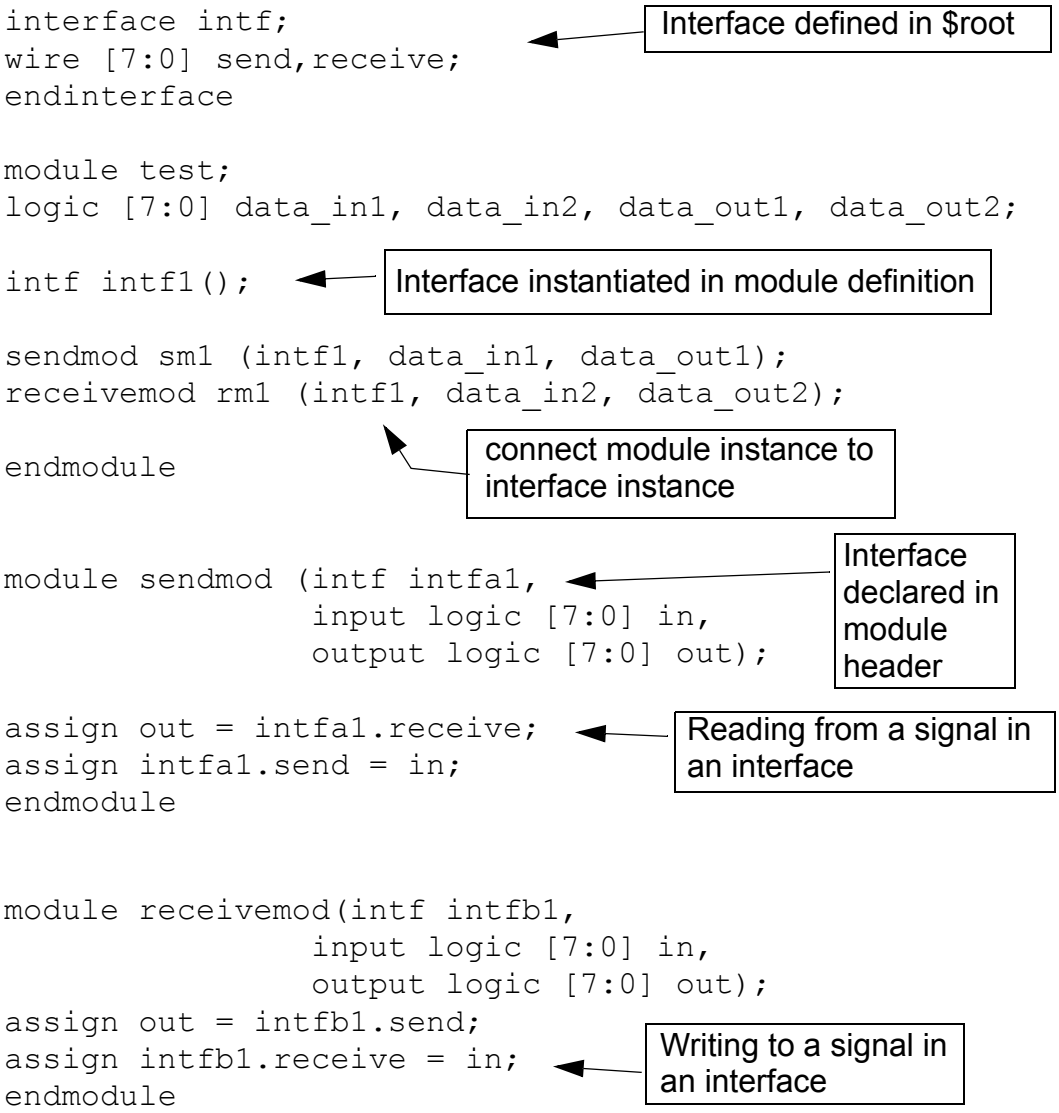


Interfaces help you to maintain your code, for example, to add a signal between blocks in your design, you only need to edit the interface instead of editing the module headers of both modules and the module instantiation statements for these modules.Interfaces are not just wires. Interfaces can contain the following:

- Variables and parameters that can be declared in one location and used in various modules.
- Tasks or functions that can be used by all instances that connect to these interfaces.
- Procedures for checking and other verification operations.

Example 15-5 introduces a basic interface and the code that uses it.

Example 15-5 Basic Interface



As illustrated in this example:

- In the VCS implementation, interface definitions must be in the `$root` declaration space, outside of any module definition. The the Accellera SystemVerilog 3.1a specification, this is not the

case, and interface definitions can be in module definitions or nested in other interface definitions.

- To use an interface to connect a module instance to another, you must instantiate the interface in the module that instantiates the two module instances.
- You also must declare the interface in the port connection list of the module headers of the two modules you are connecting with the interface. Note that the interface instance names in the port connection lists don't have to match the interface instance name in the module that instantiates these module, not do the instance names have to match in the port connection lists of the modules connected by the instance. In this example we have the following instance names for the same interface: intf1, intfa1, and intfb1.
- To read from or write to a signal in an interface, you reference it by *`interface_instance_name.signal_name`*.

This basic example, meant only to introduce interfaces, doesn't do much to show you the utility of an interface. In fact, in this example the signals in the interface, send and receive, are functionally the same as `inout` ports in the module definitions and two nets with the `wire` data type connecting these `inout` ports.

What if your intent is for the instance of module sendmod to always send data to module receivemod through one signal, in this case signal send, and receive data from module receivemod from the other signal, signal receive? This basic interface isn't doing the job for you. You can use an interface construct called a `modport` to add directionality to signals in an interface.

Using Modports

A modport specifies direction for a signal from a “point of view.” With these two signal in the interface we can specify two modports or “points of view” and the two modules connected by the interface will use different modports, and have different points of view.

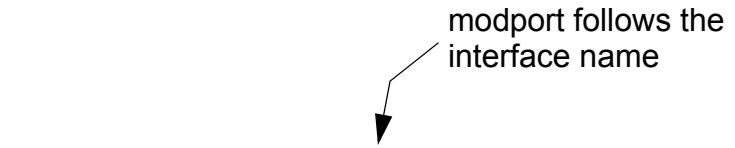
Let’s modify the interface definition by adding two modports.

```
interface intf;
logic [7:0] send,receive;
modport sendmode (output send, input receive);
modport receivemode (input send, output receive);
endinterface
```

Modules that use `modport` sendmode, have an `output` port named send and an `input` port named receive. Modules that use `modport` receivemode, have an `input` port named send and an `output` port named receive.

The data type is also changed. This isn’t done to enable the modport. It’s to enable the modules connected by this interface, that we will also modify, to make procedural assignments to the signals.

Now let's modify the module definition for module sendmod:




```
module sendmod (intf.sendmode intfa1,
                input logic [7:0] in,
                output logic [7:0] out);
always @(intfa1.receive) out = intfa1.receive;
always @(intfa1.receive) intfa1.send = in;

endmodule
```

In the module header, in the connection list in the header where using the interface is declared, the `modport` is also declared, using the following syntax:

interface_name.modport_name

Module receivemod is also modified:

```
module receivemod(intf.receivemode intfb1,
                input logic [7:0] in,
                output logic [7:0] out);
always @(intfb1.send) out = intfb1.send;
always @(intfb1.send) intfb1.receive = in;
endmodule
```

Modports also control the visibility of signals declared in and interface. If a signal in an interface is not also specified in a `modport`, then modules that use the `modport` cannot access the signal.

Functions in Interfaces

If we defined a user-defined task or function in the interface, we can use the `import` keyword in the `modport` to make it accessible to the module instances connected by the interface and that use the `modport`, for example:

```
interface intf;
logic [7:0] send,receive;

function automatic logic parity(logic [7:0] data);
return (^data);
endfunction

modport sendmode (output send, input receive,
                  import function parity());
modport receivemode (input send, output receive,
                     import function parity());
endinterface
```

Using the keyword `import` and specifying whether it is a task or function in the `modport`, enables module connected by the interface to use the function named parity. Using a function in an interface is called using a method.

```
module sendmod (intf.sendmode intfa1,
                input logic [7:0] in,
                output logic [7:0] out,
                output logic out_parity);

always @(intfa1.receive)
begin
out = intfa1.receive;
out_parity = intfa1.parity(intfa1.receive);
intfa1.send = in;
end

endmodule
```

This module uses the method called parity, using the syntax:

interface_instance_name.method_name

Enabling SystemVerilog

You tell VCS to compile and simulate SystemVerilog code with the `-sverilog` compile-time option. No runtime option is necessary.

IMPORTANT:

Radiant Technology (`+rad`) does not work with SystemVerilog design construct code, for example structures and unions, new types of always blocks, interfaces, or things defined in `$root`.
The only SystemVerilog constructs that work with Radiant Technology are SystemVerilog assertions that refer to signals with Verilog-2001 data types, not the new data types in SystemVerilog.

Disabling Unique and Priority Warning Messages

By default VCS displays warning messages in certain situations when you enter `unique if`, `unique case`, `priority if` and `priority case` statements, for example:

```
RT Warning: More than one conditions match in 'unique if'
statement.
     "filename.v", line line_number, at time        sim_time

RT Warning: No condition matches in 'unique if' statement.
     "filename.v", line line_number, at time        sim_time
```

```
RT Warning: No condition matches in 'priority if' statement.
     "filename.v", line line_number, at time        sim_time

RT Warning: More than one conditions match in 'unique case'
statement.
     "filename.v", line line_number, at time        sim_time

RT Warning: No condition matches in 'unique case' statement.
     "filename.v", line line_number, at time        sim_time

RT Warning: No condition matches in 'priority case'
statement.
     "filename.v", line line_number, at time        sim_time
```

You can suppress these warning messages with the following compile-time option and keyword argument.

`-ignore` *keyword_argument*

The keyword arguments are as follows:

`unique_checks`
: Suppresses warning messages about `unique if` and `unique case` statements.

`priority_checks`
: Suppresses warning messages about `priority if` and `priority case` statements.

`all`
: Suppresses warning messages about `unique if`, `unique case`, `priority if` and `priority case` statements.

Controlling How VCS Uses SystemVerilog Assertions

For SystemVerilog assertions there are the following features:

- Compile-Time and Runtime Options for SystemVerilog Assertions
- Ending Simulation at a Specified Number of Assertion Failures
- Entering System Verilog Assertions as Pragmas
- Using SystemVerilog Assertions in an MX Design
- Options for SystemVerilog Assertion Coverage
- Reporting on SystemVerilog and OpenVera Assertions Coverage
- Assertion Monitoring System Tasks
- Using Assertion Categories

Compile-Time and Runtime Options for SystemVerilog Assertions

There is the following compile-time option:

`-assert` *`keyword_argument`*

The keyword arguments are as follows:

`enable_diag`
Enables further control of results reporting with runtime options.

`filter_past`
For assertions that are defined with the `$past` system task, ignore these assertions when the past history buffer is empty. For instance, at the very beginning of the simulation the past history buffer is empty. So the first sampling point and subsequent sampling points should be ignored until the past buffer has been filled with respect to the sampling point. Using this keyword filters out vacuous successes too.

`dumpoff`
Disables the dumping of SVA information in the VPD file during simulation.

There is the following runtime option:

`-assert` *`keyword_argument`*

The keyword arguments are as follows:

`dumpoff`
Disables the dumping of SVA information in the VPD file during simulation.

`filter`
Blocks reporting of trivial implication successes. These happen when an implication construct registers a success only because the precondition (antecedent) portion is false (and so the consequent portion is not checked). With this option, reporting only shows successes in which the whole expression matched.

`finish_maxfail=`*`N`*
Terminates the simulation if the number of failures for any assertion reaches *`N`*. *`N`* must be supplied, otherwise no limit is set.

`global_finish_maxfail=`*`N`*

Stops the simulation when the total number of failures, from all SystemVerilog assertions, reaches *`N`*.

`maxfail=`*`N`*

Limits the number of failures for each assertion to *`N`*. When the limit is reached, the assertion is disabled. *`N`* must be supplied, otherwise no limit is set.

`maxsuccess=`*`N`*

Limits the total number of reported successes to *`N`*. *`N`* must be supplied, otherwise no limit is set. The monitoring of assertions continues, even after the limit is reached.

`nopostproc`

Disables the display of the SVA coverage summary at the end of simulation. This summary looks like this for each `cover` statement:

```
"source_filename.v", line_number:
cover_statement_hierarchical_name number attempts,
number total match, number first match, number vacuous
match
```

`quiet 0 | 1`

0

Disables messages, in standard output, about assertion failures.

1

Disables messages, in standard output, about assertion failures, but displays a summary of them at the end of simulation. The never triggered assertions are also reported.

`report[=path/filename]`

Generates a report file in addition to printing results on your screen. By default this file's name and location is ./assert.report, but you can change it to where you want by entering the filename path name argument.

The filename can start with a number or letter. The following special characters are acceptable in the filename: `%`, `^`, and `@`. Using the following unacceptable special characters: `#`, `&`, `*`, `[]`, `$`, `()`, or `!` has the following consequences:

- A filename containing `#` or `&` results in a filename truncation to the character before the `#` or `&`.
- A filename containing `*` or `[]` results in a `No match` message.
- A filename containing `$` results in an `Undefined variable` message.
- A filename containing `()` results in a `Badly placed ()'s` message.
- A filename containing `!` results in an `Event not found` message.

`success`

Enables reporting of successful matches, and successes on `cover` statements, in addition to failures. The default is to report only failures.

`verbose`

Adds more information to the end of the report specified by the `report` keyword argument and a summary with the number of assertions present, attempted, and failed.

You can enter more than one keyword, using the plus + separator, for example:

```
-assert maxfail=10+maxsucess=20+success+filter
```

Ending Simulation at a Specified Number of Assertion Failures

There are two ways to end simulation when the total number of failures from all assertions reaches a specified number:

- Using the `-assert global_finish_maxfail=`*`N`* runtime option and argument, see "Compile-Time and Runtime Options for SystemVerilog Assertions" on page 15-90.
- Using the `$ova_set_global_finish_maxfail` system task.

The `$ova_set_global_finish_maxfail` takes an argument which is an expression, for example:

```
$ova_set_global_finish_maxfail(100);
```

This expression does not need to be a constant expression, for example:

```
$ova_set_global_finish_maxfail(reg1 + reg2);
```

When VCS executes this system task, the current value of the expression argument determines the total number of assertion failures that ends simulation.

Entering System Verilog Assertions as Pragmas

If your Verilog code also has to be read by a tool that has not implemented SystemVerilog Assertions, you can enter your SVA code as pragmas (or metacomments) so that the other tool ignores the SVA code, and tell VCS to compile the SVA code by including the `-sv_pragma` compile-time option. The following is an example of SVA code as pragmas:

```
// sv_pragma sequence s1;
// sv_pragma @(posedge clk) sig1 ##[1:3] sig2;
// sv_pragma endsequence

/* sv_pragma
sequence s2;
sig3 ##[1:3] sig4;
endsequence

property p1;
@(posedge clk) sig1 && sig2 => s2;
endproperty

a1: assert property (p1);
a2: assert property (@(posedge clk)s1);
c1: cover property (p1);
c2: cover property (@(posedge clk)s1);
*/
```

The `sv_pragma` keyword must immediately follow the characters that begin the comment: `//` for single line comments and `/*` for multi-line comments.

Note:

This feature is intended allow SVA code as pragmas. When you include the `-sv_pragma` compile time option VCS compiles all the contents in the comment, not just the SVA code in the comment. If the multi-line comment is the following:

```
/* sv_pragma
a1: assert property (p1);
a2: assert property (@(posedge clk)s1);
c1: cover property (p1);
c2: cover property (@(posedge clk)s1);

initial
$display("$display with SVAs");
*/
```

VCS displays the `$display with SVAs` character string at runtime.

Using SystemVerilog Assertions in an MX Design

If you instantiate a Verilog module in a VHDL design entity, you must analyze the source file containing the module using the vlogan utility. If the module includes SystemVerilog assertions, you must include the `-sverilog` option on the vlogan command line.

Options for SystemVerilog Assertion Coverage

SystemVerilog assertion coverage is monitoring the design for when assertions are met and not met. Coverage results are on assertions, not the properties or sequences that might be instantiated in these assertions. See “Reporting on SystemVerilog and OpenVera Assertions Coverage” on page 15-98

To enable and control assertion coverage, VCS has the following compile-time options:

`-cm assert`
: Compiles for SystemVerilog assertions coverage. `-cm` is not a new compile-time option but the `assert` argument is new. This option and argument must also be entered at runtime.

`-cm_assert_hier` *`filename`*
: Limits assertion coverage to the module instances specified in *`filename`*. Specify the instances using the same format as VCS coverage metrics. If this option is not used, coverage is implemented on the whole design.

There are also the following runtime options for assertion coverage:

`-cm assert`
: Specifies monitoring for System Verilog assertions coverage. Like at compile-time, `-cm` is not a new runtime option but the `assert` argument is new.

`-cm_assert_dir` *`path/filename`*
: Specifies the path and filename of an initial coverage file. An initial coverage file is needed to set up the database. By default, an empty coverage file is loaded from the following directory: simv.vdb/snps/fcov.

`-cm_assert_name` *`path/filename`*
: Specifies the file name or the full path name of the assertion coverage report file. This option overrides the default report name and location, which is ./simv.vdb/fcov/results.db If only a file name is given, the default location is used resulting in: ./*simv*.vdb/fcov/*filename*.db.

Reporting on SystemVerilog and OpenVera Assertions Coverage

After running a series of simulations, you can generate a report summarizing the coverage of both kinds of assertions. With this report, you can quickly see if all assertions were attempted, how often they were successful, and how often they failed. Potential problem areas can be easily identified. The report can cover one test or merge the results of a test suite. The report is written in HTML and you can customize it with a TCL script.

The default report shows the number of assertions that:

- Were attempted
- Had successes
- Had failures

Coverage is broken down by module and instance, showing for each assertion and expression, the number of attempts, failures, and successes.

Assertion coverage can also grade the effectiveness of tests, producing a list of the minimum set of tests that meet the coverage target. Tests can be graded on any of these metrics:

- Number of successful assertion attempts versus number of assertions (*metric* = SN)
- Number of failed assertion attempts versus number of assertions (*metric* = FN)
- Number of assertion attempts versus number of assertions (*metric* = AN)

- Number of successful assertion attempts versus number of assertion attempts (*metric* = SA)
- Number of failed assertion attempts versus number of assertion attempts (*metric* = FA)

To generate a report, run the following command:

```
assertCovReport [options]
```

The command line options are as follows:

`-e` *`TCL_script`* `|` `-`
Use this option to produce a custom report using Tcl scripts or entering Tcl commands at standard input (keyboard). Most of the other assertCovReport options are processed before the Tcl scripts or keyboard entries.

`-e` *`TCL_script`*
Specifies the path name of a Tcl script to execute. To use multiple scripts, repeat this option with each script's path name. They are processed in the order listed.

`-e -`
Specifies your intent to enter Tcl commands at the keyboard.

The Tcl commands provided by VCS, that you can input to fcovReport for OVA coverage reports (see "Tcl Commands for SVA and OVA Functional Coverage Reports" on page 15-102), you can also input to assertCovReport for SystemVerilog assertion (SVA) coverage.

`-cm_assert_cov_cover`
Specifies reporting only about `cover` statements.

`-cm_assert_cov`
Specifies reporting only about `cover` and `assert` statements (no OVA coverage).

`-cm_assert_category` *`category_val`*
`[,`*`category_val...`*`]`
Reports only on assertions specified by category value. You can specify any number of category values, separating them with commas.

`-cm_assert_cov_events`
Specifies only reporting on OpenVera assertions events.

`-cm_assert_dir` *`path`*
Specifies the path of the template database. If this option is not included, assertCovReport uses simv.vdb.

`-cm_assert_grade_instances` *`target`*`,`*`metric`*
`[,` *`time_limit`*`]`
Generates an additional report, grade.html, that lists the minimum set of tests that add up to the target value for the metric (see previous page for *metric* codes). The grading is by instance.

`-cm_assert_grade_modules` *`target`*`,` *`metric`*
`[,` *`time_limit`*`]`
Generates an additional report, grade.html, that lists the minimum set of tests that add up to the target value for the metric (see previous page for *metric* codes). The grading is by module.

`-cm_assert_map` *`filename`*
Maps the module instances of one design onto another while merging the results. For example, use this to merge the assertion coverage results of unit tests with the results of system tests. Give the path name of a file that lists the hierarchical names of from/to pairs of instances with one pair per line:

`from_name to_name`

The results from the first instance are merged with the results of the second instance in the report.

`-cm_assert_merge` *`filename`*

Specifies the path name of a assertion coverage result file or directory to be included in the report. If *filename* is a directory, all coverage result files under that directory are merged. Repeat this option for any result file or directory to be merged into this report. If this option is not used, assertCovReport merges all the result files in the directory of the template database (specified with `-cm_assert_dir` or *simv*.vdb/snps/fcov by default).

`-cm_assert_report` *`name | path/name`*

Specifies the base name for the report.The assertCovReport command creates an HTML index file at *simv*.vdb/reports/*name*.fcov-index.html and stores the other report files under *simv*.vdb/reports/*name*.fcov.

If you give a path name, the last component of the path is used as the base name. So the report files are stored under *path*/*name* and the index file is at *path*/*name*.fcov-index.html.
If this option is not included, the report files are stored under *simv*.vdb/reports/report.fcov and the index file is named report.fcov-index.html.

`-cm_assert_severity` *`int_val`* `[,`*`int_val`*`...]`

Reports only on OpenVera assertions specified by severity value. You can specify any number of severity values, separating them with commas. Only OpenVera assertions can have a severity, SystemVerilog assertions cannot.

Tcl Commands for SVA and OVA Functional Coverage Reports

You can produce a custom report with Tcl commands that you enter in assertCovReport. These Tcl commands also work in fcovReport when you want a costom report about OVA (OpenVera assertions) coverage. These commands are in Table 15-6.

The Tcl command descriptions frequently refer to a bin. A bin is a coverage value container in the coverage database. There are two kinds of bins: boolean (which hold 0 or 1 values) and count. Count bins can hold any unsigned value that can be stored in 32 bits.

Table 15-6 Tcl Commands for SVA and OVA Functional Coverage Reports

Command	Return Value	Description
`fcov_get_assertions -instance` *handle*	array of handles	Returns an array of handles to the assertions of the instance with the specified handle.
`fcov_get_assertions -module` *handle*	array of handles	Returns an array of handles to the assertions of the module with the specified handle.
`fcov_get_bins -assertion` *handle*	array of handles	Returns an array of handles to the bins of the assertion with the specified handle.
`fcov_get_category -assertion` *handle*	int	Returns the category of the assertion with the specified handle expressed as an integer.
`fcov_get_children -instance` *handle*	array of handles	Returns an array of handles for the child instances that contain at least one assertion under their hierarchy. The parent instance is with the specified handle.
`fcov_get_coverage -bin` *handle*	int	Returns the coverage count associated with the bin with the specified handle. It can be 0 or the count of the times the bin was covered.

Command	Return Value	Description
`fcov_get_flag -bin handle`	string	Returns the flag associated with the bin with the specified handle. It can be: “illegal”, “ignored”, or “impossible”.
`fcov_get_handle -instance name`	handle	Returns a handle for the instance specified with its full hierarchical name.
`fcov_get_handle -module name`	handle	Returns a handle for the specified module.
`fcov_get_handle -instance | -module name1 -assertion name2`	handle	Returns the handle to the specified assertion in the specified instance or module. The assertion name must follow the convention for the full name of an assertion.
`fcov_get_handle -instance | -module name1 -assertion name2 -bin name3`	handle	Returns the handle to the specified bin for the specified assertion in the specified instance or module. Current names are: “ 1attempts”, “ 2failures”, “ 3allsuccesses”, “ 4realsuccesses”, and “ 5events” (note that all bin names start with a space).
`fcov_get_instances`	array of handles	Returns an array of handles for the instances that contain at least one assertion under their hierarchy.
`fcov_get_instances -module handle`	array of handles	Returns an array of handles for the instances that contain at least one assertion under their hierarchy for the module with the specified handle.
`fcov_get_modules`	array of handles	Returns an array of handles for the modules that contain at least one assertion.
`fcov_get_name -object handle`	string	Returns the name of the object the specified handle.
`fcov_get_no_bins -assertion handle`	int	Returns total bins for the assertion with the specified handle.
`fcov_get_no_of_assertions`	int	Returns total number of assertions in the design..

Command	Return Value	Description
`fcov_get_no_of_assertions -instance` *handle*	int	Returns total number of assertions for the instance with the specified handle.
`fcov_get_no_of_assertions -module` *handle*	int	Returns total number of assertions for the module with the specified handle.
`fcov_get_no_of_assertions_attempted`	int	Returns total number of assertions attempted in the design.
`fcov_get_no_of_assertions_attempted -instance` *handle*	int	Returns total number of assertions attempted for the instance with the specified handle.
`fcov_get_no_of_assertions_attempted -module` *handle*	int	Returns total number of assertions attempted for the module with the specified handle.
`fcov_get_no_of_assertions_failed`	int	Returns total number of assertions that failed in the design.
`fcov_get_no_of_assertions_failed -instance` *handle*	int	Returns total number of assertions that failed for the instance with the specified handle.
`fcov_get_no_of_assertions_failed -module` *handle*	int	Returns total number of assertions that failed for the module with the specified handle.
`fcov_get_no_of_assertions_succeeded`	int	Returns total number of assertions that succeeded/matched in the design.
`fcov_get_no_of_assertions_succeeded -instance` *handle*	int	Returns total number of assertions that succeeded/matched for the instance with the specified handle.
`fcov_get_no_of_assertions_succeeded -module` *handle*	int	Returns total number of assertions that succeeded/matched for the module with the specified handle.
`fcov_get_no_of_children -bin` *handle*	int	Returns total number of child bins whose parent is the bin handle handle.
`fcov_get_no_of_children -instance` *handle*	int	Returns total number of child instances containing at least one assertion under their hierarchy. The parent instance is with the specified handle.

Command	Return Value	Description
`fcov_get_no_of_instances`	int	Returns total number of instances that contain at least one assertion under their hierarchy.
`fcov_get_no_of_instances -module` *handle*	int	Returns total number of instances containing at least one assertion under their hierarchy for a module with the specified handle.
`fcov_get_no_of_modules`	int	Returns total number of modules that contain at least one assertion.
`fcov_get_no_of_topmodules`	int	Returns total number of top level modules that contain at least one assertion under their hierarchy. (Top level modules are instances.)
`fcov_get_severity -assertion` *handle*	int	Returns the severity of an assertion with the specified handle expressed as an integer.
`fcov_get_topmodules`	array of handles	Returns an array of handles for the top level modules that contain at least one assertion under their hierarchy. (Top level modules are instances.)
`fcov_get_type -bin` *handle*	string	Returns the type of the bin with the specified handle. It can be: “bool” or “count”.
`fcov_get_type -object` *handle*	string	Returns the type of the object with the specified handle. The type can be “module”, “instance”, “assertion”, or “bin”.
`fcov_grade_instances -target` *value1* `-metric` *code* `[-timeLimit` *value2*`]`	string	Returns a list of the minimum set of tests that add up to the target value for the metric code. Each test is accompanied by the accumulated coverage value including that test. The grading is by instance.
`fcov_grade_modules -target` *value1* `-metric` *code* `[-timeLimit` *value2*`]`	string	Returns a list of the minimum set of tests that add up to the target value for the metric code. Each test is accompanied by the accumulated coverage value including that test. The grading is by module.

Command	Return Value	Description
`fcov_is_cover_prop -assertion` *handle*	int	Returns 1 if the assertion is the cover directive for the property.
`fcov_is_cover_seq -assertion` *handle*	int	Returns 1 if the assertion is the cover directive for the sequence.
`fcov_load_design -file` *name*	empty string “”	Unloads any existing design and data, including all name maps. Then loads a design with the specified path name. (The search rules are described in the table for options.)
`fcov_load_test -file` *name*	empty string “”	Loads a test with the specified name. The name can be a file name or full path name. (The search rules are described in the table for options.)
`fcov_load_test_grade -file` *name*	empty string “”	Loads a file with the specified name for coverage grading. The name can be a file name or full path name. (The search rules are described in the table for options.)
`fcov_map_hier -from` *hier_name* `-to` *hier_name*	empty string “”	Maps coverage of instance (and the hierarchy under it) with the specified hierarchical name *hier_name* to another instance for all subsequent merges.
`fcov_set -bin` *handle* `-coverage` *int*	empty string “”	Set the coverage count *int* for the bin with the specified handle. It can be 0 or the count of the times the bin was covered.
`fcov_set -bin` *handle* `-flag` *name*	empty string “”	Set the flag with *name* for the bin with the specified handle. The flag can be: “illegal”, “ignored”, or “none”.
`fcov_write_coverage -file` *name*	empty string “”	Writes current coverage to a file with the specified name. The name can be a file name or full path name. (The search rules are described in the table for options.)

The names of bins in the coverage database are as follows:

For SVA assertions:

1attempts
: Holds the count of the attempts.

2failures
: Holds the count of failed attempts.

3allsuccess
: Holds the count of succesfull attempts.

4realsuccess
: Holds the count of attempts with nonvacuous success.

8incompletes
: Holds the count of unterminated attempts.

For SVA Property coverage:

1attempts
: Holds the count of attempts.

2failures
: Holds the count of failed attempts.

3allsuccess
: Holds the count of succesfull attempts.

6vacuoussuccess
: Holds the count of vacuous successes.

8incompletes
: Holds the count of unterminated attempts

For SVA sequence coverage:

1attempts
: Holds the count of attempts.

3allsuccess
: Holds the count of succesfull attempts.

7firstmatches
Holds the count of generated .matched events.

8incompletes
Holds the count of unterminated attempts.

For OVA assertions:

1attempts
Holds the count of attempts.

2failures
Holds the count of failed attempts.

3allsuccess
Holds the count of succesfull attempts.

4realsuccess
Holds the count of attempts with nonvacuous success.

8incompletes
Holds the count of unterminated attempts.

For OVA events:

1attempts
Holds the count of attempts.

2failures
Holds the count of failed attempts.

3allsuccess
Holds the count of succesfull attempts.

4realsuccess
Holds the count of attempts with nonvacuous success.

5events
Holds the count of generated event matches.

8incompletes
Holds the count of unterminated attempts.

The assertCovReport Report Files

When you compile for SystemVerilog or OpenVera assertion coverage, by including the `-cm assert` compile-time option and keyword argument, VCS creates the simv.vdb directory in the current directory.

When you monitor for SystemVerilog or OpenVera assertion coverage, by including the `-cm assert` runtime option and keyword argument, VCS creates the fcov directory in the simv.vdb directory. This directory contains binary data about SystemVerilog assertions coverage (and if also included, about OpenVera assertions coverage).

When you run assertCovReport, This utility does the following:

1. Creates the reports directory in the simv.vdb directory and writes the report.index.html file in the reports directory.
2. Creates the report.fcov directory in the reports directory and writes in the report.fcov directory the following files: category.html, hier.html, and tests.html.

Note:

If you include the `-cm_assert_report` *`name`* option on the assertCovReport command line, you see the following differences:

- The report.fcov directory is named *name*.fcov
- The report.index.html file is named *name*.index.html

The report.index.html File

This file begins with tables of numerical data about the assertions in your design. This information has a title for a type of information about your assertions with a total number and percentage under it. In most cases this title is blue and is a hypertext link to a list of these assertions further down in this file. These tiles and what the values under them mean are as follows:

Total number of Assertions
: The total number of SystemVerilog `assert` statements and OpenVera `assert` directives in your design.

Assertions not Covered
: The total number and percentage of the SystemVerilog `assert` statements for properties that never matched and the OpenVera `assert` directives for sequential expressions that never occur during simulation.

Assertions with at least 1 Real Success
: SystemVerilog and OpenVera assertions can fail at certain times during simulation and succeed at other times. This it the total number an percentage of SystemVerilog and OpenVera assertions that had a least one success during simulation.

Assertions with at least 1 Failure
: The total number an percentage of SystemVerilog and OpenVera assertions that had a least one failure during simulation.

Assertions with at least 1 Incomplete
: Assertions specify a property about the design that can occur over a span of simulation time. This is the total number and percentage of assertions that VCS began to monitor, but simulation ended before the design matched the behavior specified in the assertion.

Assertions without Attempts

VCS looks to see if the design matches the behavior in the assertion when there is a transition on a clock signal for the property (you can specify the type of transition). If none of these clock signal transitions occur, then there is no attempt at the assertion. This is the total number and percentage of assertions with no attempts.

Total number of Cover Directives for Properties

The argument to a SystemVerilog assertion `cover` statement can be the name of a defined and declared property (between the `property` and `endproperty` keywords) or the argument can be just the building blocks of a property between parentheses (a clock signal and a sequential expression).

This value is the total number of SystemVerilog assertions `cover` statements with the name of a property as their argument.

Cover Directive for Property Not Covered

The total number and percentage of SystemVerilog assertion `cover` statements with the name of a property is their argument, where the design's behavior never matches the property specified in the `cover` statement.

Cover Directive for Property with Matches

The total number and percentage of SystemVerilog assertion `cover` statements with the name of a property as their argument, where the design's behavior, at least some of the simulation time, matches the property specified in the `cover` statement.

Cover Directive for Property with Vacuous Matches

A property for a SystemVerilog `cover` statement automatically matches if there is an implication for the property and the antecedent condition is never true, for example:

```
sequence s5;
sig11 ##1 sig12;
```

```
endsequence

property p5;
@(posedge clk1) (sig9 ##1 sig10) |-> s5;
endproperty

c4: cover property (p5);
```

If the antecedent `(sig9 ##1 sig10)` never occurs, property p5 matches, even if the consequent sequential expression `sig11 ##1 sig12` never occurs. This is why such a match is vacuous or empty.

This information is the total number and percentage of SystemVerilog assertion `cover` statements with the name of a property as their argument, and the total number of OpenVera `cover` directives, where there is a vacuous match.

Total number of Cover Directives for Sequences

A SystemVerilog `cover` statement can have a sequence name for an argument instead of a property, for example:

```
sequence s8;
@(posedge clk1) sig17 ##[1:5] sig18;
endsequence

c7: cover property (s8);
```

or

```
sequence s8;
sig17 ##[1:5] sig18;
endsequence

c7: cover property (@(posedge clk1) s8);
```

This information is the total number of such `cover` statements.

Cover Directive for Sequence not Covered

Total number and percentage of SystemVerilog `cover` statements, where the argument is a sequence that did not occur during simulation.

Cover Directive for Sequence with All Matches

If there is a cycle delay range expression in a sequence, for example:

```
sequence s1;
@(posedge clk1) sig1 ##[1:5] sig2;
endsequence
```

After sig1 is true, there could be a match after one, two, three, four, or five rising edges on clk1, if sig2 is also true. If all these matches happen for all attempts, then this is a sequence of all matches.

This information is the total number and percentage of SystemVerilog `cover` statements with sequences with all matches.

Cover Directive for Sequence with First Matches

Total number and percentage SystemVerilog `cover` statements where the argument is a sequence, and there was a cycle delay range expression, and the first possible match for the sequence occurred.

Total number of Events

In OpenVera assertions, you can define a sequential expression as an event. This information is the total number of such events.

Events Not Covered

Total number and percentage of OpenVera events that did not occur (not covered).

Events with at least 1 real Match
: Total number and percentage of OpenVera events that occurred (were covered).

Events without any match or with only vacuous match
: Total number and percentage of OpenVera events for which there was no match or only a vacuous match.

Events without any Attempts
: Total number and percentage of OpenVera events for which there were no attempts.

At the bottom of the file are hypertext links to the other files written by assertCovReport:

- Lists of tests merged to generate this report (tests.html)
- Hierarchical coverage report (hier.html)
- Category based coverage report (category.html)

These links are followed by general information about generating this report.

The tests.html File

You can use the `-cm_assert_map` and `-cm_assert_merge` command line options to merge the results from one design to another. This file indicates the path names of the results.db files from which assertCovReport merges the results.

The category.html File

You can use the `$ova_set_category` and `$ova_set_severity` system tasks to set the category and severity of a SystemVerilog assertion and an OpenVera assertion.

You can also use the `(* category=`*`number`*` *)` SystemVerilog attribute to set the category of a SystemVerilog assertion.

This file shows you the assertions in your design according to category and severity.

The report begins with categories you used for the SystemVerilog `assert` statements or OpenVera `assert` directives. It shows you the total number and percentage of each category number (integer). The word Category preceding a category number is a hypertext link to a list of `assert` statements or directives with that category number, showing you the hierarchical name, number of attempts, successes, failures, and incompletes.

Next is the same information for the severity numbers you assigned to your SystemVerilog `assert` statements or OpenVera `assert` directives. The word Severity is a hypertext link to a similar list of `assert` statements or directives with that severity number.

Next is the same information for the category numbers you used for the SystemVerilog `cover` statements where the argument is a property, and the OpenVera `cover` directives.

Next is the same information for the severity numbers you used for the SystemVerilog `cover` statements where the argument is a property, and the OpenVera `cover` directives.

Next is the category numbers you used for the SystemVerilog `cover` statements where a sequence is the argument.

Next is the severity numbers you used for the SystemVerilog `cover` statements where a sequence is the argument.

The hier.html File

The report begins with a list of the module instances in the design and the number of the following in the instances:

- The number (integer) of SystemVerilog `assert` statements or OpenVera `assert` directives.
- The number (integer) of SystemVerilog `cover` statements where the argument is a sequence.
- The number (integer) of SystemVerilog `cover` statements where the argument is a property, and OpenVera `cover` directives.
- The number (integer) of OpenVera events.

Each number is a hypertext link that takes you to a list of each type of statement, directive, or event. For `assert` statements or directives, the list shows you the number of attempts, successes, failures, and incompletes. For `cover` statements or directives and events, the list shows you the number of attempts, all matches, first matches, and incompletes.

Assertion Monitoring System Tasks

For monitoring SystemVerilog assertions we have developed the following new system tasks:

```
$assert_monitor
$assert_monitor_off
$assert_monitor_on
```

IMPORTANT:

Enter these system tasks in an initial block. Do not enter these system tasks in an always block.

The `$assert_monitor` system task is analogous to the standard `$monitor` system task in that it continually monitors specified assertions and displays what is happening with them (you can have it only display on the next clock of the assertion). Its syntax is as follows:

```
$assert_monitor([0|1,]assertion_identifier...);
```

Where:

0
: Specifies reporting on the assertion if it is active (VCS is checking for its properties) and for the rest of the simulation reporting on the assertion or assertions, whenever they start.

1
: Specifies reporting on the assertion or assertions only once, the next time they start.

If you specify neither 0 or 1, the default is 0.

assertion_identifier...
: A comma separated list of assertions. If one of these assertions is not declared in the module definition containing this system task, specify it by its hierarchical name.

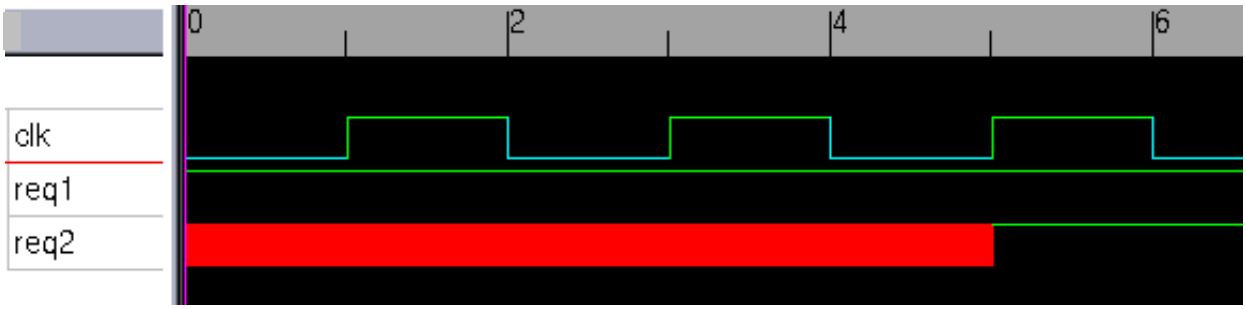
To explain what it displays take, for example, the following assertion:

```
property p1;
 @ (posedge clk) (req1 ##[1:5] req2);
endproperty

a1: assert property(p1);
```

For property p1 in assertion a1, a clock tick is a rising edge on signal clk. When there is a clock tick VCS checks to see if signal req1 is true, and then to see if signal req2 is true at any of the next five clock ticks.

In this example simulation, signal clk initializes to 0 and toggles every 1 ns, so the clock ticks at 1 ns, 3 ns, 5 ns and so on.



A typical display of this system task is as follows:

```
Assertion test.a1 ['design.v'27]:
5ns: tracing "test.a1" started at 5ns:
                attempt startingfound: req1looking for: req2 or
          any
5ns: tracing "test.a1" started at 3ns:
                trace: req1 ##1 anylooking for: req2 or any
                failed: req1 ##1 req2
5ns: tracing "test.a1" started at 1ns:
                trace: req1 ##1 any[* 2 ]looking for: req2 or any
                failed: req1 ##1 any ##1 req2
```

Breaking this display into smaller chunks:

```
Assertion test.a1 ['design.v'27]:
```

The display is about the assertion with the hierarchical name test.a1. It is in the source file named design.v and declared on line 27.

```
5ns: tracing "test.a1" started at 5ns:
                attempt startingfound: req1looking for: req2 or
          any
```

At simulation time 5 ns VCS is tracing test.a1. An attempt at the assertion started at 5 ns. At this time VCS found req1 to be true and is looking to see if req2 is true one to five clock ticks after 5 ns. Signal req2 doesn't have to be true on the next clock tick, so req2 not being true is okay on the next clock tick; that's what looking for "or any" means, anything else than req2 being true.

```
5ns: tracing "test.a1" started at 3ns:
                trace: req1 ##1 anylooking for: req2 or any
                failed: req1 ##1 req2
```

The attempt at the assertion also started at 3 ns. At that time VCS found req1 to be true at 3 ns and it is looking for req2 to be true some time later. The assertion "failed" in that req2 was not true one clock tick later. This is not a true failure of the assertion at 3 ns, it can still succeed in two more clock ticks, but it didn't succeed at 5 ns.

```
5ns: tracing "test.a1" started at 1ns:
                trace: req1 ##1 any[* 2 ]looking for: req2 or any
                failed: req1 ##1 any ##1 req2
```

The attempt at the assertion also started at 1 ns. `[*` is the repeat operator. `##1 any[* 2 ]` means that after one clock tick, anything can happen, repeated twice. So the second line here says that req1 was true at 1 ns, anything happened after a clock tick after 1 ns (3 ns) and again after another clock tick (5 ns) and VCS is now looking for req2 to be true or anything else could happen. The third line here says the assertion "failed" two clock ticks (5 ns) after req1 was found to be true at 1 ns.

The $assert_monitor_off and $assert_monitor_on system tasks turn off and on the display from the $assert_monitor system task, just like the $monitoroff and $monitoron system turn off and on the display from the $monitor system task.

Assertion System Functions

The assertion system functions are $onehot, $onehot0, and $isunknown. Their purposes are as follows:

$onehot
: Returns true if only one bit in the expression is true.

$onehot0
: Returns true if at the most one bit of the expression is true (also returns true if none of the bits are true).

$isunknown
: Returns true if one of the bits in the expression has an X value. In the VCS implementation, this function also returns true if one of the bits in the expression has a Z value.

The following is an example of their use:

```
a1: assert property (@ (posedge clk) $onehot({lg1,lg2}));
a2: assert property (@ (posedge clk) $onehot0({lg1,lg3}));
a3: assert property (@ (posedge clk)$isunknown({r1,r2,r3}));
```

Another useful function is $countones. This function returns the number of 1s in a bit vector expression.

Using Assertion Categories

You can categorize assertions and then enable and disable them by category. There are two ways to categorize SystemVerilog assertions:

- Using OpenVera assertions system tasks for categorizing assertions
- Using attributes

After you categorize assertions you can use these categories to stop and restart assertions.

Using OpenVera Assertion System Tasks

VCS has a number of system tasks and functions for OpenVera assertions that also work on SystemVerilog assertions. These system tasks do the following:

- Set a category for an assertion
- Return the category of an assertion

These system tasks are as follows:

`$ova_set_category("`*`assertion_full_hier_name`*`", `*`category`*`)`
or

`$ova_set_category(`*`assertion_full_hier_name`*`, `*`category`*`)`
System task that sets the category level attributes of an assertion. The category level is an unsigned integer from 0 to 2^{24} - 1.

Note:

These string arguments, such as the full hierarchical name of an assertion, can be enclosed in quotation marks or not. This is true when using these system tasks with SVA. They must be in quotation marks when using them with OVA.

`$ova_get_category("`*`assertion_full_hier_name`*`")`
or

`$ova_get_category(`*`assertion_full_hier_name`*`)`
System function that returns an unsigned integer for the category.

Using Attributes

You can prefix an attribute in front of an `assert` statement to specify the category of the assertion. The attribute must begin with the `category` name and specify an integer value, fort 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
```

IMPORTANT:

In a `generate` statement the category value cannot be an expression, the attribute in the following example is invalid:

```
genvar g;
generate
for (g=0; g<1; g=g+1)
begin:loop
(* category=g+1 *) a3: assert property (s2);
end
endgenerate
```

If you use a `parameter` for a category value, the parameter value can be overwritten in a module instantiation statement.

You can use these attributes to assign categories to both named and unnamed assertions, for example:

```
(* category=p+1 *) a1: assert property (p1);
(* category=1 *) assert property (s1);
```

The attribute is retained in a tokens.v file when you use the `-Xman=0x4` compile-time option and keyword argument.

Stopping and Restarting Assertions by Category

The are also OpenVera assertions system tasks for starting and stopping assertions that also work on SystemVerilog assertions. These system tasks are as follows:

`$ova_category_start(`*`category`*`)`
System task that starts all assertions associated with the specified *category*.

`$ova_category_stop(`*`category`*`)`
System task that stops all assertions associated with the specified *category*.

Using Mask Values to Stop and Restart Assertions

There are system tasks for both OpenVera and SystemVerilog assertions that allow you to use a mask to determine if a category of assertions should be stopped or restarted. These system tasks are `$ova_category_stop` and `$ova_category_start`. They have matching syntax.

```
$ova_category_stop(categoryValue, maskValue[, globalDirective]);
```

Where:

`categoryValue`
Because there is a *`maskValue`* argument, this argument now is the result of an anding operation between the assertion categories and the *`maskValue`* argument. If the result matches this value, these categories stop. As seen in “Stopping and Restarting Assertions by Category” on page 15-123, without the *`maskValue`* argument, this argument is the value you specified in `$ova_set_category` system tasks or `category` attribute.

`maskValue`
Is a value that is logically anded with the category of the assertion. If the result of this and operation matches the *`categoryValue`*, VCS stops monitoring the assertion.

globalDirective

Can be either of the following values:

0

Enables an `$ova_category_start` system task, that does not have a *globalDirective* argument, to restart the assertions stopped with this system task.

1

Prevents an `$ova_category_start` system task that does not have a *globalDirective* argument from restarting the assertions stopped with this system task.

```
$ova_category_start(categoryValue, maskValue[, globalDirective]);
```

Where:

categoryValue

Because there is a *maskValue* argument, this argument now is the result of an anding operation between the assertion categories and the *maskValue* argument. If the result matches this value, these categories start. As seen in “Stopping and Restarting Assertions by Category” on page 15-123, without the *maskValue* argument, this argument is the value you specified in `$ova_set_category` system tasks or `category` attribute.

maskValue

Is a value that is logically anded with the category of the assertion. If the result of this and operation matches the *categoryValue*, VCS starts monitoring the assertion.

globalDirective

Can be either of the following values:

0
: Enables an `$ova_category_stop` system task, that does not have a *`globalDirective`* argument, to stop the assertions started with this system task.

1
: Prevents an `$ova_category_stop` system task that does not have a *`globalDirective`* argument from stopping the assertions started with this system task.

Examples

This first example stops the odd numbered categories:

```
$ova_set_category(top.d1.a1,1);
$ova_set_category(top.d1.a2,2);
$ova_set_category(top.d1.a3,3);
$ova_set_category(top.d1.a4,4);
⋮
$ova_category_stop(1,'h1);
```

The categories are masked with the *`maskValue`* argument and compared with the *`categoryValue`* argument:

	bits	*`categoryValue`*	
category 1	001		
`maskValue`	1		
result	1	1	match
category 2	010		
`maskValue`	1		
result	0	1	no match
category 3	011		
`maskValue`	1		

result	1	1	match
category 4	100		
maskValue	1		
result	0	1	no match

1. VCS looks at the least significant bit of each category and logically ands that LSB to the *maskValue* argument, which is 1.
2. The results of these anding operations, 1 or true for categories 1 and 3, and 0 or false for categories 2 and 4, is compared to the *categoryValue*, which is 1, there is a match for categories 1 and 3.
3. VCS stops the odd numbered categories.

Here is another example. This one uses the *globalDirective* argument:

```
$ova_set_category(top.d1.a1,1);
$ova_set_category(top.d1.a2,2);
$ova_set_category(top.d1.a3,3);
$ova_set_category(top.d1.a4,4);
⋮
$ova_category_stop(1,'h1,0);
$ova_category_stop(0,'h1,1);
⋮
$ova_category_start(1,'h1);
$ova_category_start(0,'h1);
```

In this example:

1. The two `$ova_category_stop` system tasks stop first the odd numbered assertions and then the even numbered ones. The first `$ova_category_stop` system task has a *`globalDirective`* argument that's 0, the second has a *`globalDirective`* argument that's 1.
2. The first `$ova_category_start` system task can restart the odd numbered assertions but the second `$ova_category_start` system task can't start the even numbered assertions.

16

Using the VCS/SystemC Cosimulation Interface

The VCS/SystemC cosimulation interface provides a means by which the SystemC modeling environment and VCS can cooperate to simulate a system described in SystemC and Verilog.

VCS contains a built-in SystemC simulator that is compatible with OSCI SystemC 2.0.1. By default, when you use the interface VCS runs its own SystemC simulator. No setup is necessary.

You also have the option of installing the OSCI SystemC simulator and have VCS run it to cosimulate using the interface. See “Using a Customized SystemC Installation” on page 16-28.

With the interface you can use the most appropriate modeling language for each part of the system and verify the correctness of the design. For example, VCS/SystemC cosimulation interface allows you to:

- Use a SystemC module as a reference model for the Verilog RTL design under test in your testbench.
- Import legacy Verilog IP into a SystemC description.
- Import third-party Verilog IP into a SystemC description.
- Export SystemC IP into a Verilog environment when only a few of the design blocks are implemented in SystemC.
- Use systemC to provide stimulus to your design.

The VCS/SystemC cosimulation interface creates the necessary infrastructure to cosimulate SystemC models with Verilog models. The infrastructure consists of the required build files and any generated wrapper or stimulus code. VCS writes these files in subdirectories in the ./csrc directory. To use the interface, you don't need to do anything to these files.

During cosimulation, the VCS/SystemC cosimulation interface is responsible for:

- Synchronizing the SystemC kernel and VCS
- Exchanging data between the two environments

The usage models for the VCS/SystemC cosimulation interface, depending on the type of cosimulation you want to perform, are the following:

- Verilog Designs Containing SystemC Modules

- SystemC Designs Containing Verilog Modules

Notes:

- There are examples of Verilog instantiated in SystemC and SystemC instantiated in Verilog in the $VCS_HOME/doc/examples/osci_dki directory.
- The interface supports the following compilers:

  Linux: you can use the gnu 3.2.1 or 2.96 compilers.

  Solaris: SC 6.2, gcc 2.95.2, gcc 3.2.2

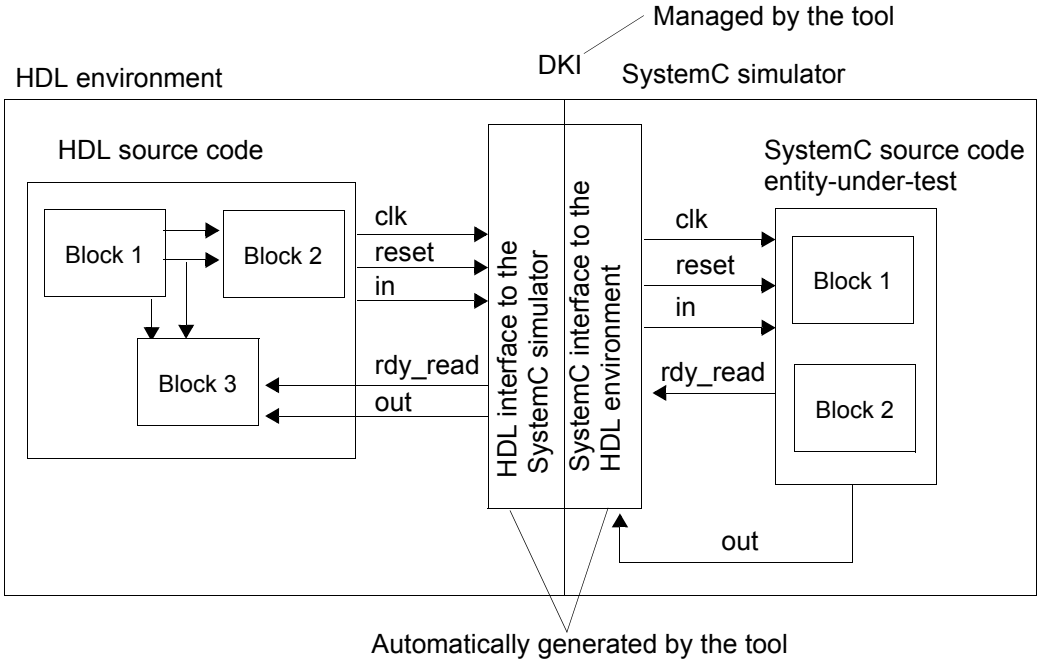
  HP: aCC 3.35
- Do not enter the -e *new_name_for_main* compile-time option when using the interface.

Verilog Designs Containing SystemC Modules

To cosimulate a Verilog design that contains SystemC modules, you import one or more SystemC instances into the Verilog design. Using the VCS/SystemC cosimulation interface, you generate a wrapper and include it in the Verilog design for each SystemC instance. The ports of the created Verilog wrapper are connected to signals attached to the ports of the corresponding SystemC modules.

Figure 16-1 illustrates VCS DKI communication.

Figure 16-1 VCS DKI Communication of an Verilog Design Containing SystemC Modules



Input Files Required

To run cosimulation with a Verilog design containing SystemC instances, you need to provide the following files:

- SystemC source code

  You can directly write the entity-under-test source code or generate it with other tools.

  - Any other C or C++ code for the design

- Verilog source code (.v extension) including

  - A Verilog module definition that instantiates the interface wrapper and other Verilog modules. These wrapper files are generated by a utility and you don't need to do anything to these files, see "Generating The Wrapper for SystemC Modules" on page 16-6 and "Instantiating The Wrapper and Coding Style" on page 16-9.
  - Any other Verilog source files for the design

- An optional port mapping file. If you do not provide this file, the interface uses the default port mapping definition. For details of the port mapping file, see "Using a Port Mapping File" on page 16-21.
- An optional data type mapping file. If you don't write a data type mapping file, the interface uses the default one in the VCS installation. For details of the data type mapping files, see "Using a Data Type Mapping File" on page 16-22.

Supported Port Data Types

SystemC types are restricted to the `sc_clock`, `sc_bit`, `sc_bv`, `sc_logic`, `sc_lv`, `sc_int`, `sc_uint`, `sc_bigint`, and `sc_biguint` data types. Native C/C++ types are restricted to the `uint`, `uchar`, `ushort`, `int`, `bool`, `short`, `char`, `long` and `ulong` types

Inout ports that cross the cosimulation boundary between SystemC and Verilog must observe the following restrictions:

- SystemC port types must be `sc_inout_rv<>` or `sc_inout_resolved` and must be connected to signals of type `sc_signal_rv<>` or `sc_signal_resolved`.

- You need to create a port mapping file, as described in “SystemC Designs Containing Verilog Modules” on page 16-12, to specify the SystemC port data types as `sc_lv` (for a vector port) or `sc_logic` (for a scalar port).

Generating The Wrapper for SystemC Modules

You use the syscan utility to generate the wrapper and interface files for cosimulation. This utility creates the csrc directory in the current directory, just like VCS does when you include compile-time options for incremental compilation. The syscan utility writes the wrapper and interface files in subdirectories in the ./csrc directory.

There is nothing you need to do to the files that syscan writes. VCS will know to look for them when you include the compile-time option for using the interface, see “Compiling a Verilog Design Containing SystemC Modules” on page 16-11.

The syntax for the syscan command line is as follows:

```
syscan [options] filename[:modulename]
[filename[:modulename]]*
```

Where:

filename[:*modulename*] [*filename*[:*modulename*]]*
: Is how you specify all the SystemC files in the design. There is no limit to the number of of files. The entries for the SystemC files that contain modules that you want to instantiate also include a colon : followed by the name of the module. If :*modulename* is omitted, the .cpp files will be compiled and added to the design's database so the final vcs command will be able to bring together

all the modules in the design. You do not need to add `-I$VCS_HOME/include` or `-I$SYSTEMC/include`

`[options]`

Are any of the following:

```
[-cflags "flags"] [-cpp g++]
[-port port_mapping_file][-Mdir=directory_path]
[-help|-h] [-v] [-o name] [-V] [-vcsi]
```

These options are defined as follows:

`-cflags "flags"`

Passes flags to the C++ compiler.

`-cpp path_to_compiler`

You can specify a path to a compiler, or if your environment is set up to pick up a default compiler, the basename of the compiler.

If this option is omitted syscan uses the default compiler, the version of Solaris CC, HP aCC, or Linux g++ that your environment picks up.

- For Solaris, to use a gnu compiler instead of the default Sun CC compiler, enter `-cpp g++` or `-cpp path_to_g++`.
- For Linux, the default g++ will be tested for 2.96 or 3.2.1 compatibility. If you wish to use a compiler other than your default, enter `-cpp path_to_g++`.

If you enter the `-cpp` option on the `syscan` command line, you must also enter it on the `vcs` command line.

If you override your default gnu compiler, you must enter the `-cc path_to_gcc` compile-time option on your `vcs` command line.

`-port` *`port_mapping_file`*
Specifies a port mapping file, see “SystemC Designs Containing Verilog Modules” on page 16-12.

`-Mdir=`*`directory_path`*
This option works the same way that the `-Mdir` VCS compile-time option works If you are using the `-Mdir` option with VCS, you should use the `-Mdir` option with syscan to redirect the syscan output to the same location that VCS uses.

`-help|-h`
Displays the command line syntax, options, and example command lines.

`-v`
Displays the version number.

`-o` *`name`*
The syscan utility uses the specified name instead of the module name as the name of the model. Do not enter this option when you have multiple modules on the command line, doing so results in an error condition.

`-V`
Displays code generation and build details. Use this option if you are encountering errors or are is interested in the flow that builds the design.

`-vcsi`
Prepares all SystemC interface models for simulation with VCSi.

You don’t specify the data type mapping file on the command line, See “Using a Data Type Mapping File” on page 16-22.

The following is an example generates a Verilog wrapper:

```
syscan -cflags "-g" sc_add.cpp:sc_add
```

The following example compiles the rest of your .cpp files:

```
syscan -cflags "-g" sc_sub.cpp multiply.cpp
display.cpp
```

Instantiating The Wrapper and Coding Style

You instantiate the wrapper just like a Verilog module. For example, take the following SystemC module in a file named sc_add.h:

```
SC_MODULE(sc_add)
{
 public:
  sc_in<sc_lv<32> > ina;
  sc_in<sc_lv<32> > inb;
  sc_out<sc_lv<32> > outx;

  SC_CTOR(sc_add) {
      SC_METHOD(sc_add_action);
      sensitive << ina << inb;
  }

  void sc_add_action() {
      outx.write(ina.read().get_word(0) +
        inb.read().get_word(0));
  }
};
```

The module name is sc_add, so you instantiate it as follows In the Verilog part of the design:

```
sc_add add1(value1, value2, add_wire);
```

Use a Verilog wire to connect a SystemC module output port to any other (i.e SystemC or Verilog) module input port or Verilog register. Use a Verilog wire to connect a SystemC module input port to any other module output port or Verilog register.

Controlling Time Scale and Resolution in a SystemC Module

To control the time resolution of your SystemC module,create a static global object that initializes the timing requirements for the module. This can be a separate file that is included as one of the .cpp files for the design. The following is an example of such a file:

```
#include <systemc.h>

class set_time_resolution
{
public:
  set_time_resolution()
  {
    sc_set_time_resolution(10, SC_PS);
    sc_set_default_time_unit(100, SC_PS);
  }
};

static int SetTimeResolution()
{
  new set_time_resolution();
  return 42;
}

static int time_resolution_is_set = SetTimeResolution();
```

Compiling a Verilog Design Containing SystemC Modules

To compile your Verilog design for using the interface include the `-sysc` compile-time option, for example:

```
vcs -sysc top.v dev.v
```

When you compile with this option, VCS looks in the csrc directory for the subdirectories containing the interface and wrapper files needed to instantiate the SystemC design in the Verilog design.

There are no runtime options needed to run the cosimulation. Start simulation with the simv command line and any runtime options that you want to use.

Using GNU Compilers on SUN Solaris

On Solaris the default compiler is Sun Forte CC. You can specify a different compiler with the `-cpp` and `-cc` compile-time options. The interface supports the gcc 3.2.2 and 2.95.2 compilers.

If you use the `-cpp g++` option on the `syscan` command line, you must also use it on the `vcs` command line, for example:

```
syscan -cpp g++ -cflags -g sc_add.cpp:sc_add

syscan -cpp g++ -cflags "-g" sc_sub.cpp multiply.cpp
display.cpp

vcs -cpp g++ -sysc top.v dev.v
```

The default compiler on solaris is the Sun Native compiler CC. If you wish to use a gnu compiler, and your environment is set up to pick up the correct gcc and g++ executables, then you can use `-cpp g++` to indicate this.

If your environment does not choose the gnu compiler you wish to use, you need to specify the path to the compiler executables as follows:

```
syscan -cpp /usr/bin/g++ sc_add.cpp:sc_add
syscan -cpp /usr/bin/g++ sc_sub.cpp multiply.cpp
vcs -cc /usr/bin/gcc -cpp /usr/bin/g++ -sysc top.v dev.v
```

Using GNU Compilers on Linux

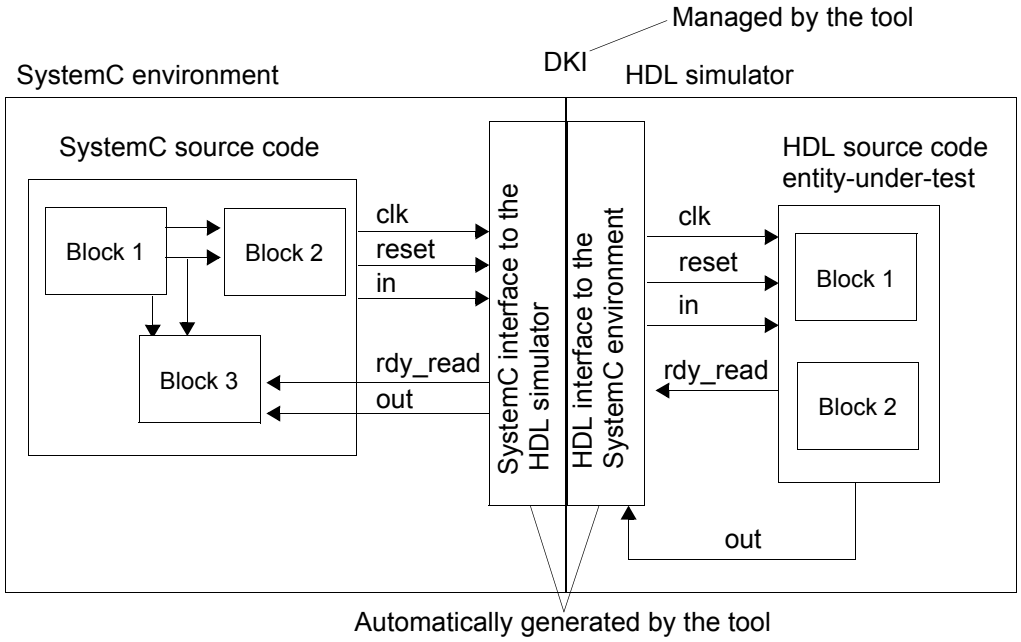
On Linux the default compiler is gcc. You can specify a different compiler with the `-cpp` and `-cc` compile-time options. The interface supports the gcc 3.2.1 and 2.96 compilers.

SystemC Designs Containing Verilog Modules

To cosimulate a SystemC design that contains Verilog modules, you import one or more Verilog instances into the SystemC design. Using the VCS/SystemC cosimulation interface, you generate a wrapper and include it in the SystemC design for each Verilog instance. The ports of the created SystemC wrapper are connected to signals attached to the ports of the corresponding Verilog modules.

Figure 16-2 illustrates the VCS direct kernel interface (DKI) communication.

Figure 16-2 VCS DKI Communication of SystemC Design Containing Verilog Modules



Input Files Required

To run cosimulation with a SystemC design containing Verilog modules, you need to provide the following files:

- Verilog source code (.v extension)
  - You can directly write the entity-under-test Verilog code or generate it with other tools. The Verilog description represented by the entity-under-test can be Verilog code of any complexity (including hierarchy) and can use any language feature VCS supports.
  - Any other Verilog source files that are necessary for the design

- SystemC source code including:
  - A SystemC top-level simulation (sc_main) that instantiates the interface wrappers and other SystemC modules
  - Any other SystemC source files for the design
- An optional port mapping file. If you do not provide this file, the interface uses the default port mapping definition. For details of the port mapping file, see “SystemC Designs Containing Verilog Modules” on page 16-12.
- An optional data type mapping file. If you don’t write a data type mapping file, the interface uses the default one in the VCS installation. For details of the data type mapping files, see “Using a Data Type Mapping File” on page 16-22.

Generating The Wrapper for a Verilog Module

You use the vlogan utility with the `-sysc` option to generate and build the wrapper and interface files for cosimulation. This utility creates the ./csrc directory in the current directory, just like VCS does when you include compile-time options for incremental compilation. The vlogan utility writes the wrapper and interface files in subdirectories in the ./csrc directory.

Note:
The vlogan utility is also used in mixed-HDL simulation for instantiating a Verilog module into a VHDL architecture.

There is nothing you need to do to the files that vlogan writes. VCS will know to look for them when you include the compile-time option for using the interface, see “Compiling a Verilog Design Containing SystemC Modules” on page 16-11.

The syntax for the vlogan command line is as follows:

```
vlogan -sysc -sc_model modulename file.v [-cpp g++]
[-sc_portmap port_mapping_file]
[-Mdir=directory_path] [-V]
```

Where:

`-sysc`
Specifies generating a SystemC wrapper.

`-sc_model` *`modulename file.v`*
Specifies the module name and its Verilog source file.

`-cpp` *`path_to_compiler`*
You can specify a path to a compiler, or if your environment is set up to pick up a default compiler, the basename of the compiler. If this option is omitted syscan uses the default compiler, the version of Solaris CC, HP aCC, or Linux g++ that your environment picks up.

- For Solaris, to use a gnu compiler instead of the default Sun CC compiler, enter `-cpp g++` or `-cpp` *`path_to_g++`*.
- For Linux, the default g++ will be tested for 2.96 or 3.2.1 compatibility. If you wish to use a compiler other than your default, enter `-cpp` *`path_to_g++`*.

If you enter the `-cpp` option on the `syscan` command line, you must also enter it on the `vcs` command line.

If you override your default gnu compiler, you must enter the `-cc` *`path_to_gcc`* compile-time option on your `vcs` command line.

`-sc_portmap` *`port_mapping_file`*
Specifies a port mapping file, see “SystemC Designs Containing Verilog Modules” on page 16-12.

`-Mdir=`*`directory_path`*

This option works the same way that the `-Mdir` VCS compile-time option works. If you are using the `-Mdir` option with VCS, you should use the `-Mdir` option with vlogan to redirect the vlogan output to the same location that VCS uses.

`-V`

Displays code generation and build details. Use this option if you are encountering errors or are interested in the flow that builds the design.

To generate the wrapper and interface files for a Verilog module named adder, in a Verilog source file named adder.v, instantiated in SystemC code in top.cpp, you would enter the following:

```
vlogan -sysc -sc_model adder -sc_portmap the.map
adder.v
```

Instantiating The Wrapper

You instantiate a Verilog module adder in your SystemC code like a SystemC module. For example, take the following Verilog module in a file called adder.v:

```
module adder (value1, value2, result);
 input [31:0] value1;
 input [31:0] value2;
 output [31:0] result;
 reg [31:0] result_reg;

 always @(value1 or value2) begin
   result_reg <= value1 + value2;
 end

 assign result = result_reg;
```

```
  endmodule
```

The module name is adder, so you instantiate in your SystemC code in top.cpp as follows:

```
#include adder.h
int sc_main(int argc, char *argv[]){
   sc_clock clock ("CLK", 20, .5, 0.0);
   sc_signal<sc_lv<32> > value1;
   sc_signal<sc_lv<32> > value2;
   sc_signal<sc_lv<32> > result;

   // Verilog adder module
   adder adder1("adder1");
   adder1.value1(value1);
   adder1.value2(value2);
   adder1.result(result);

   sc_start(clock, -1);
 }
```

One of the generated files will be modulename.h, which should be included in your .cpp file.

Compiling a SystemC Design Containing Verilog Modules

Compile your design with the following command:

```
syscsim dev.v other_source_files
compile-time_options
```

In this example, dev.v might contain Verilog code utilized by the adder.v module above.

When you compile with this option, VCS looks in the ./csrc directory for the subdirectories containing the interface and wrapper files needed to connect the Verilog and SystemC parts of the design.

Note:

On the HP platform, when simulating a SystemC design containing Verilog modules, the `-Mloadlist` option to VCS must not be used. This will interfere with the use of this option by the cosimulation interface.

The output of compilation is an executable file named simv, in the current directory.

Using GNU compilers on SUN Solaris

On Solaris the default compiler is Sun Forte CC. You can specify a different compiler with the `-cpp` and `-cc` compile-time options. The interface supports the gcc 3.2.2 and 2.95.2 compilers.

If you use the `-cpp g++` option on the `vlogan` command line, you must also use it on the `syscsim` command line, for example:

```
vlogan -sysc -cpp g++ -sc_model adder adder.v

syscsim dev.v -cpp g++ other_options
```

Another example:

```
vlogan -sysc -cpp /usr/bin/g++ -sc_model adder
adder.v

syscsim -cc /usr/bin/gcc -cpp /usr/bin/g++ dev.v
other_options
```

Using GNU Compilers on Linux

On Linux the default compiler is gcc. You can specify a different compiler with the `-cpp` and `-cc` compile-time options. The interface supports the gcc 3.2.1 and 2.96 compilers.

Elaborating The Design

When SystemC is at the top pf the design hierarchy, you instantiate Verilog code in the SystemC code. The elaboration of the simulation is then automatically done in two steps when you enter the `syscsim` command and VCS creates the simulation:

1. The first step creates a temporary simulation executable that contains all SystemC parts but does not yet contain any HDL (Verilog, VHDL, ...) parts. VCS then starts this temporary executable to find out which Verilog instances are really needed. All SystemC constructors and end_of_elaboration() methods will be executed. However, simulation will not start.
2. VCS creates the final version of the simv file containing SystemC as well as all HDL parts. The design is now fully elaborated ready to simulate.

As a side effect of executing the temporary executable during step 1, you will see that the following message is printed:

```
INFO: Exiting prematurely since $SYSTEMC_ELAB_ONLY is set
```

If your simulation contains statements that should be executed during step 1, guard these statements with a check for environment variable SYSTEMC_ELAB_ONLY or function

```
extern "C" bool hdl_elaboration_only()
```

Both will be set/yield true only during this extra execution of simv during step 1.

For example, guard statements like this:

```
module constructor:
  if (! hdl_elaboration_only()) {
      ... open log file for covergae data ...
  }
module destructor:
  if (! hdl_elaboration_only()) {
      ... close log file for covergae data ...
  }
```

Specifying Runtime Options

You start simulation with a simv command line. You enter SystemC parameters or arguments (analogous to VCS runtime options), to control the simulation of the SystemC part of the design, on the simv command line, for example:

```
simv scarg1 scarg2
```

If you also want to enter a VCS runtime option, to control the simulation of the Verilog part of the design, you can do so by preceding it with the `-verilogrun` option on the simv command line, for example:

```
simv scarg2 -verilogrun -q
```

If you want to enter more than one VCS runtime option, enclose the runtime options in quotation marks, for example:

```
simv scarg1 -verilogrun "-q -l log1"
```

Using a Port Mapping File

You can provide an optional port mapping file for the `syscan` command with the `-port` option. If you specify a port mapping file, any module port that is not listed in the port mapping file will be assigned the default type mapping.

A SystemC port has a corresponding Verilog port in the wrapper for instantiation. The syscan utility either uses the default method for determining what type of Verilog port it writes in the wrapper or uses the entry for the port in the port mapping file.

A port mapping file is an ASCII text file. Each line defines a port in the SystemC module, specifying the following:

- The port name
- bit width
- The corresponding Verilog “type” using keywords that specify if it is scalar, unsigned vector, or signed port
- The SystemC or Native C++ type

A line beginning with a pound sign (#) is a comment.

Example 16-3 shows a port mapping file.

Example 16-3 Verilog Port Mapping File

```
#    Port name     Bits     Verilog type  SystemC type
     in1           8        signed        sc_int
     in2           8        bit_vector    sc_lv
     clock         1        bit           sc_clock
     out1          8        bit_vector    sc_uint
     out2          8        bit_vector    sc_uint
```

The valid Verilog port “types,” which are case-insensitive, are as follows:

- `bit` — specifies a scalar (single bit) Verilog port.
- `bit_vector` — specifies a vector (multi-bit) unsigned Verilog port. (`bitvector` is a valid alternative.)
- `signed` — specifies a Verilog port that is also a reg or a net declared with the `signed` keyword and propagates a signed value.

SystemC types are restricted to the `sc_clock`, `sc_bit`, `sc_bv`, `sc_logic`, `sc_lv`, `sc_int`, `sc_uint`, `sc_bigint`, and `sc_biguint` data types.

Native C/C++ types are restricted to the `bool`, `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong` data types.

Using a Data Type Mapping File

When running a VCS/SystemC simulation, the interface propagates data through the module ports from one language domain to another. This can require the interface to translate data from one data type representation to another. This translation is called mapping and is controlled by data type mapping files.

The data type mapping mechanism is similar to that used for port mapping (see “Using a Port Mapping File” on page 16-21), but is more economical and requires less effort to create and maintain. Because the data type mapping is independent of the ports, you can create one or more default mappings for a particular type that will be used for all ports, rather than having to create a port map for every port of each new HDL wrapper model.

Data type mapping files map types, so that ALL ports of that type on ALL instances will now be assigned the specified mapping.

The data type mapping file is named cosim_defaults.map. The interface looks for and reads the data mapping file in the following places and in the following order:

1. In $VCS_HOME/include/cosim
2. In your $HOME/.synopsys_ccss directory
3. In the current directory.

An entry in a later file overrules an entry in an earlier file.

Each entry for a SystemC type has the following:

1. It begins with the keyword `Verilog`.
2. It is followed by the bit width. For vectors, an asterisk (*) is a wildcard to designate vectors of any bit width not specified elsewhere in the file.
3. The corresponding Verilog “type” using keywords that specify if it is scalar, unsigned vector, or signed port, the same keywords used in the port mapping file.
4. The SystemC or Native C++ type

Example 16-4 shows an example of a data type mapping file.

Example 16-4 Data Type Mapping File

```
################################################
# Mappings between SystemC and Verilog datatypes
################################################
Verilog  * bit_vector       sc_bv
Verilog  1 bit              bool
Verilog  * bit_vector       int
Verilog  * signed           int
Verilog  1 bit              sc_logic
Verilog  1 bit              sc_bit
Verilog  * bit_vector       char
Verilog  * bit_vector       uchar
Verilog  * bit_vector       short
Verilog  * bit_vector       ushort
Verilog  * bit_vector       uint
Verilog  * bit_vector       long
Verilog  * bit_vector       ulong
```

Debugging the SystemC Code

To debug just the SystemC code in the combined simulation, do the following:

1. Run syscan with the `-cflags "-g"` option to build the SystemC source code for debugging.
2. Start the C++ debugger on the simv executable file.

If you are using the Sun Forte compiler:

```
dbx ./simv
```

If you are using the Gnu compiler on Solaris or Linux:

Run both syscan and VCS with the `-cpp g++` option

```
gdb ./simv
```

If you are using the aCC compiler on the HP platform, you can use either the gdb or wdb debugger:

```
gdb ./simv
wdb ./simv
```

You can now set and stop at breakpoints in your SystemC code.

Debugging The Verilog Code

To debug the Verilog code, create the simv executable with the `-RI` option to start VirSim for interactive debugging, for example if you instantiate SystemC code in Verilog code:

```
vcs -sysc -RI -o simv top.v
```

If you instantiate Verilog code in SystemC:

```
syscsim -RI -o simv dev.v
```

Debugging Both the Verilog and SystemC Portions of a Design

To debug both the SystemC and Verilog portions of your design:

1. Run syscan with the `-cflags "-g"` option to build the SystemC source code for debugging.
2. Include the `-I`and `-line` compile-time options on the vcs or syscsim command line to compile the Verilog part of the design for post-processing debug tools and for Verilog source code debugging.

   Enter the `-RIG` option for an existing simv executable, for interactive debugging using VirSim.

   To compile and interactively debug a Verilog design containg SystemC modules, enter command lines like the following:

   ```
   vcs -sysc -I -line top.v
   vcs -RIG
   ```

   To compile and interactively debug a SystemC design containing Verilog modules, enter command lines like the following:

   ```
   syscsim -I -line
   syscsim -RIG
   ```

3. Start the C++ debugger on the simv executable file. As VirSim is already running the simv executable, you must attach your debugger to the simv process.

To find the simv executable process id, execute the following command:

```
% ps -e | grep simv
12216 pts/1      0:00 simv
```

You then can launch your debugger as outlined above, but provide the process id from the ps command as the third argument to the debugger:

```
% gdb ./simv 12216
```

Using the Built-in SystemC Simulator

VCS contains a built-in SystemC which it uses by default. No setup is necessary. Using this simulator does not require setting an environment variable (in earlier releases, the interface required the SYSTEMC environment variable).

Compatibility with OSCI SystemC 2.0.1

The built-in SystemC simulator is binary compatible to the OSCI SystemC 2.0.1 simulator. That means that you can link into a simv executable objects files (*.{o,a,so}) compiled with the OSCI SystemC 2.0.1 simulator.

Supported Compilers

The following compilers are supported:

- Linux: gcc 2.96 and gcc 3.2.1
- Solaris: SC 6.2, gcc 2.95.2, gcc 3.2.2
- HP: aCC 3.35

Compiling Source Files

If you need to compile source files that include systemc.h and you do this in your own environment and not with the syscan script, then add compiler flag `-I$VCS_HOME/include/systemc`.

Using a Customized SystemC Installation

You can install the OSCI SystemC simulator and tell VCS to use it for Verilog/SystemC cosimulation. To do so you set the SYSTEMC_OVVERRIDE environment variable to the directory where you installed it, for example:

```
setenv SYSTEMC_OVERRIDE /net/user/systemc-2.0.1
```

The OSCI SystemC simulator installation must have the usual OSCI structure of $SYSTEMC_OVERRIDE/include and $SYSTEMC_OVERRIDE/lib-<arch>/libsystemc.a.

The installation must contain the patches required for the cosimulation interface.

Compatibility with OSCI SystemC 2.0.1

The built-in SystemC simulator is binary compatible to the OSCI SystemC 2.0.1 simulator. That means that you can link into a simv executable objects files (*.{o,a,so}) compiled with the OSCI SystemC 2.0.1 simulator.

Supported Compilers

The following compilers are supported:

Linux: gcc 2.96 and gcc 3.2.1

Solaris: SC 6.2, gcc 2.95.2, gcc 3.2.1 (Note that gcc 3.2.1 is not supported in 7.2 beta.)

HP: aCC 3.35

Compiling Source Files

If you need to compile source files that include systemc.h and you do this in your own environment and not with the syscan script, then add compiler flag `-I$VCS_HOME/include/systemc`.

A

VCS for the Verilog XL User

It is Synopsys' intent that VCS conform to the Verilog language standard as determined by the IEEE. In practice, the complexity of the Verilog specification makes it impossible for all Verilog simulators to behave exactly the same in all cases.

If you find a difference between Verilog-XL® and VCS simulation behavior, it is important to distinguish between ambiguity in the source model (race conditions) and differences between the simulators that fall outside the range of the IEEE Std 1364-1995 Verilog specification.

When VCS and other simulators produce different results for the same model, most likely a race condition is the cause. All race conditions are some variation of using or setting a data value at the same time step that the data value is changing. The most common cause of race conditions is the use of zero-delay elements.

If a race condition is detected it should be fixed before proceeding.

Race Debugging Differences

A common problem is that VCS and Verilog-XL results are different during race debugging. In most cases this is due to a race condition in the test bench. A race condition occurs when Verilog source code is written in an ambiguous coding style such that a simulator may correctly produce two or more different results.

In some cases race conditions have existed in user's code for years, hidden by the event-ordering of Verilog-XL. VCS is more aggressive in optimizing the ordering of events, and takes advantage of the natural parallelism of the language. As a result, you may be surprised that a correct VCS simulation of a design produces different results than in Verilog-XL.

Support Tools

Two very useful tools *vcat* and *vcdiff* (in the $VCS_HOME/bin directory) are provided. These tools display and compare VCD dumpfiles. Type *vcat* and *vcdiff* on the command line for help using these utilities.

It is advisable to know as much as possible about the circuit, in particular the time at which the circuit should reset. Then run:

```
vcdiff -min time_after_reset -max max_time_of_interest
verilogxl.dump vcs.dump
```

and analyze the data. View the code manually or with the debugger Source window. In both simulators, run for similar periods of time,

then turn on line tracing using the command `trace` in VCS. Compile with `-line` and type `trace` in the CLI in VCS vs. Verilog-XL `$db_??` (do `$db_help` to see commands in Verilog-XL).

The line numbers can now be compared for the two log files.

Evaluation of Expressions in Verilog-XL

Verilog-XL will not evaluate the whole expression if it determines that the condition can never be true. VCS will, however, evaluate the whole expression. If this is the case the code should be rewritten using a work-around example as follows:

```
if(0 && SIG())  // only VCS will call SIG()
if(SIG() && 0)  // VCS & V-XL will call SIG()
if(1 && SIG(2) && 0)  // VCS & V-XL will call SIG()
```

original code:

```
if(!A || (func1(179) && func2(179)))
```

modified code:

```
if(! A || func1(179))
if(!A || ! func2(179))
```

Initial vs. always Constructs

In both VCS and Verilog-XL there is no priority between initial and always constructs. If source code has been ordered differently the results will vary because there is no guaranteed order of execution. For example:

```
module test;
```

```
    `ifdef INITIAL_FIRST
        initial
        begin
        $display("INITIAL");
        #50 $finish;
        end

        always
        begin
        $display("ALWAYS");
        #100; end
    `endif

    `ifdef ALWAYS_FIRST
        always
        begin
        $display("ALWAYS");
        #100;
        end

        initial
        begin
        $display("INITIAL");
        #50 $finish;
        end
    `endif
endmodule
```

Incompatibilities with Verilog-XL

This section describes the known differences in behavior between Verilog-XL and VCS, our intent is to be as compatible as possible with Verilog-XL and to ensure Verilog-XL compatibility beyond what is specified by IEEE Std 1364-1995.

System Tasks Not Implemented

Annex F of IEEE Std 1364-1995 describes useful system tasks that not part of the standard Verilog language. Some of these system tasks are not yet implemented in VCS, see “Nonstandard System Tasks Not Supported in VCS” on page 2-119.

There are some system tasks that are implemented in Verilog-XL but are not described in IEEE Std 1364-1995 including its Annex F. These system tasks are not implemented in VCS, see “Verilog-XL System Tasks Not Supported in VCS” on page 2-119.

PLI

VCS supports IEEE Std 1364-1995 PLI standards with the exception of VPI routines which VCS does not support. See Chapter 7, “Using the PLI,” for more information.

VCS does not support Verilog-XL specific tasks that are not part of IEEE Std 1364-1995. If the routine has no meaning in VCS it can be easily stubbed out using:

```
void no_good (int x) {}
```

Non-Supported Verilog Tasks

If your existing user PLI application relies on calls not supported by VCS, you see an `undefined symbol` error message when compiling the design.

If a user task has no meaning in VCS and the user doesn’t want to change the code, it can be stubbed out by adding a line to the VCS PLI table file .tab file as follows:

```
$no_good task //file.tab
```

CLI

VCS CLI is different from Verilog-XL. VCS is a compiled simulator and has it's own command line language. It is recommended that you compile Verilog-XL scripts. If this is not possible then the scripts will have to be converted to VCS format. Example:

`A=100` has to be converted to `set A=100`

Tasks

VCS does not support the running of Verilog user-defined tasks on the CLI command line (PLI tasks are supported). Use the following instead:

```
reg [7:0] task_to_run;
always @ task_to_run
begin
     if(task_to_run=="A")     task_A;
     if(task_to_run=="B")     task_B;
end
```

Then in the CLI: `set task_to_run=”A”` and continue the simulation.

Port Coercion and Port Collapsing

Verilog-XL implements an optimization called port collapsing that combines the stored state for connected nets in the design. A result is that port direction is often ignored in Verilog-XL; that is, a signal value can propagate in through an output and out through an input. The result is that a port's direction might be incorrectly declared but will still simulate.

In order to be Verilog-XL compatible, VCS implements *port coercion*, which automatically converts any input or output ports that may be used in a manner not consistent with their declaration into inout ports. To see which nets are being coerced during compile, run `vcs` with the `-notice` option as follows:

```
% vcs -notice a.v
```

Port coercion can be controlled by using the compiler directives `` `noportcoerce `` and `` `portcoerce ``. These turn port coercion off and on, respectively. They are useful for turning off port coercion around modules (such as library modules) where you can easily verify that port directions are correct. They should be used with caution.

TF Routine Compatibility

VCS implements the TF routines as specified by IEEE Std 1364-1995. However, it is noted that routine `tf_spname()` has the same functionality as `tf_mipname()`.

Module Path Delays on Bidirectional Nets

If a bidirectional net is connected to an inout port that has an associated module path delay, the delay may be ignored in VCS.

Library Searching

Rules for searching libraries for unresolved modules are very complicated, especially with the `+librescan` and `+liborder` options, and not specified by IEEE Std 1364-1995. We have tried to make VCS compatible with Verilog-XL for all cases, but because Verilog-XL sometimes violates its own documentation you may find a case in which VCS behavior is unexpected.

Event Control with Memories

VCS recognizes individual memory elements in event controls, such that in this expression:

```
reg [31:0] mem[1:1000];
always @(mem[1]) ...
```

The event will only trigger when memory element 1 changes, not when any of the other 999 memory elements change.

Disable Statements

A nonblocking assignment statement that VCS has already evaluated and scheduled cannot be disabled by either a disable statement or by a procedural continuous assignment to the destination register of the nonblocking assignment.

A disable of an outer named block or task from within a function is ignored. That is, a block that calls a function cannot be disabled by that function.

A disable of a named block containing a force of a register does not reinstate a continuous assign.

Case Expressions

Case expressions are evaluated slightly differently in VCS than in Verilog-XL. We believe that the Verilog-XL behavior is erroneous, and that VCS handles this case correctly. Specifically, a case selector expression is evaluated in the context of the case expression. For example:

```
case (exp)
     sel: statement
endcase
```

The width of *sel* is determined by *exp*. In Verilog-XL, if *exp* is a constant, then *sel* is self-determined, whereas in VCS, *sel* is always evaluated in the context of *exp*. In Verilog-XL, `case(1)` and `case(1+0)` are different, while in VCS they are the same. This is seen particularly in a construct like:

```
case (1)
     1'b0:                  <s1>
     ~1'b0:                 <s2>
endcase
```

In Verilog-XL, this matches the second case, but if `1` is replaced by `1+0`, it will not match either case. The correct way to write this to avoid this incompatibility is to instead write:

```
case (1'b1) ...
```

Assignment to Parameters

Some versions of Verilog-XL allow parameters to be assigned values in procedural assignments. Recent versions of Verilog-XL treat this as an error, as does VCS.

Vectored Nets

VCS ignores the `vectored` and `scalared` keywords. All nets are treated as scalar nets which can be accessed with bit and part selects. This should not impact the simulation of any design.

$dist_uniform

In contrast to `$random`, `$dist_uniform` does not give exactly the same random number sequence as Verilog-XL. The result is different by 1 when used with some large ranges. Because the resulting random number sequence is in fact uniformly distributed, this is still correct behavior.

Use of Conditioners in Timing Checks

VCS, like Verilog-XL, supports timing check conditioners as in the following example where the setup check on `data` every positive edge of `clk` is conditioned by the signal `rst`:

```
$setup(data, posedge clk &&& rst, 1.0);
```

However, VCS enables the timing check only when `rst` is `1`. Alternatively if the condition is `~rst`, then the timing check is enabled only when `rst` is `0`.
Verilog-XL also enables the timing check if the conditioner is value `z` (undriven). In both Verilog-XL and VCS, if the conditioner is an unconnected port, the timing check is disabled.

Module Input Port Delay (MIPD)

With module input port delay annotation, VCS applies the input port delay to all the structural and behavioral constructs in the module, while Verilog-XL can only apply them to structural constructs. Consider the following example:

```
module top;
reg a;
wire in1=a;
sub u1(in1,...);
initial begin
     $monitor($time,,”%b”,u1.in1);
     #100 a = 1‘b0;
end
endmodule;
module sub(in1,..);
endmodule
```

Assume that the module input `port u1.in1` has a module input port delay of 5. The change will be observed via the `$monitor` at time 105 in VCS, and at time 100 in Verilog XL.

SDF INTERCONNECT Delays

The syntax for INTERCONNECT entries in SDF files is as follows:

```
(INTERCONNECT port_instance port_instance delval_list)
```

Where the first *port_instance* is an instance of an output or inout port and the second *port_instance* is an instance of an input or inout port. The *delval_list* argument is a list of delay values.

In Verilog-XL, when you back annotate these INTERCONNECT delays, Verilog-XL looks down the fanout in the module of the second *port_instance* and applies the delay to the first gate that it finds in each of the paths of the fanout in that module. Verilog-XL does not apply the INTERCONNECT delay to the net that connects the two port instances.

VCS can and does apply the delay to the net that connects the two port instances. Unlike the implementation in Verilog-XL, VCS accurately behaves according to the SDF 3.0 specification that states the following: “The INTERCONNECT entry is for the specification of interconnect delays (actual or estimated) that are modeled independently for each driver-to-driven path. Both start and end points for the delay path are specified.”

This kludge in Verilog-XL can lead to problems that Verilog-XL avoids by preventing you from using some rather conventional modeling techniques, for example:

```
module DFLOP(clock,data,q);
input clock;
input [31:0] data;
output [31:0] q;
reg [31:0] qreg;
assign q=qreg;

always @ clock
qreg=data;
endmodule
```

If there is an INTERCONNECT delay between some other module port instance and an instance of input port clock in this example, Verilog-XL issues an error message. VCS has no trouble with the INTERCONNECT delay.

Verilog-XL needs to find a gate so to enable INTERCONNECT delays you must do the following:

```
module DFLOP(clock,data,q);
input clock;
input [31:0] data;
output [31:0] q;
reg [31:0] qreg;
assign q=qreg;
buf (clock_,clock);

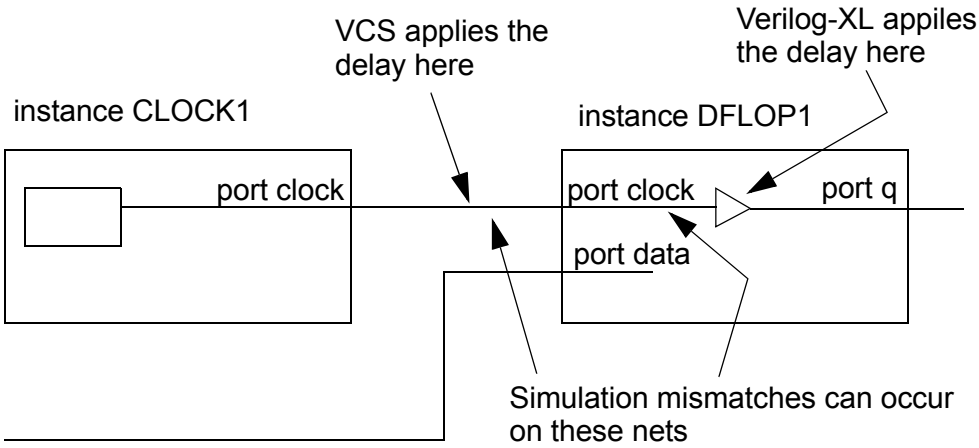
always @ clock_
qreg=data;
endmodule
```

Here Verilog-XL applies the INTERCONNECT delay to the buf gate.

The fact that VCS can apply INTERCONNECT delays to nets that connect port instances can lead to simulation mismatches between VCS and Verilog-XL.

When you have an INTERCONNECT entry in the SDF file such as the following:

```
(INTERCONNECT CLOCK1/clock DFLOP1/clock(5:5:5))
```



Getting the Most Out of Compiled Simulation

Some differences between Verilog-XL and VCS are due to the compiled nature of VCS versus the interpreted nature of Verilog-XL. VCS's intent is to overcome the natural differences in the two simulation environments as seen by the user, but some differences remain as described in this section.

Working with Multiple Test Fixtures or Test Stimuli

In some design methodologies, the different tests that are to be run on a particular design are defined by different files of Verilog source code, rather than by a memory image or other data-driven method. Since VCS is a compiled simulator, you need to adapt the data-driven methodology to avoid a full recompile for each test.

Even though VCS incremental compile is fast, we strongly discourage you from adopting a methodology that requires a VCS compile for every test. The advantage of VCS is that you compile once for each design, and reap the benefit again and again for each test run on the design.

The solution to this issue is to compile a superset of all your tests into one binary, and enable the appropriate test via a `plusarg` to the run-time command-line.

Suppose, for instance, that you have a design comprised of two files `a.v` and `b.v`, and you need to run 100 tests `test1.v` to `test100.v` each containing stimulus for the design as follows:

```
module test1;
initial begin
     driver.request_to_own;              // series of
     driver.acknowledge;                 // task calls
end
endmodule
```

With Verilog-XL, to run all tests you are used to entering:

```
verilog a.v b.v test1.v
verilog a.v b.v test2.v
.verilog a.v b.v test100.v
```

With VCS, you need to modify the test files to read:

```
module test1;
initial begin
     if ($test$plusargs("test+001+")) begin
          $display("Running test 001");
          driver.request_to_own;              // series of
          driver.acknowledge;                 // task calls
     end
end
endmodule
```

Compile all of the design files with VCS and run with the appropriate `plusarg` to enable a particular test:

```
vcs a.v b.v test1.v test2.v ... test100.v
<details of VCS compile omitted>
simv +test+001+

Running test001
V C S         S i m u l a t i o n              R e p o r t
Time: 0
CPU Time: 0.033 seconds; Data structure size: 0.0Mb
```

One potential pitfall to this approach is that the system function `$test$plusargs(`*`string`*`)` matches any `plusarg` having *`string`* as a prefix. Therefore, use leading-zero padded numbering or a trailing `+` to make each string unique.

A second approach to solving the problem of test-specific Verilog source code is to move the stimulus into a memory image format read and parsed at run-time by `$readmem` and associated Verilog code. This works well for stimulus that is easily encoded into a data format, such as an instruction stream to stimulate a CPU design.

A third solution is to apply stimulus for each test through the PLI at run-time. The PLI offers great flexibility for arbitrarily stimulating the design, but requires a substantial investment in coding the PLI application and porting the stimulus. Note that all the solutions presented in this section are compatible with Verilog-XL.

Alternate Interactive Commands

VCS uses its own command line interface, which is different from Verilog-XL. If your methodology relies on `$input` files containing Verilog source code, you can usually convert them to Verilog source files and compile them along with the rest of your model, using `plusargs` as shown above to distinguish between different tests of the design.

Gate-level Timing

VCS supports the compile-time options `+typdelays`, `+mindelays`, and `+maxdelays`, which are equivalent to Verilog-XL functionality. But these are compile-time options not run-time options. So, you need to recompile your design to change default delays. SDF annotation does not require a recompile of the design.

B

Source Protection

Source protection changes the source description of a Verilog model so that others cannot read or understand it but they can still simulate it. Source protection enables you to protect proprietary information about your designs but enables users in companies that purchase your designs to simulate them.

VCS provides three ways to protect your source description:

- Mangling of source description:
  Mangling replaces identifiers in the source description with identifiers that obscure the design. Mangling does not change the structure of the source description in that the keywords and syntax remains intact.
- Encrypting of source description:
  Encrypting makes the entire source description unreadable. SDF files can also be encrypted, however, you cannot mangle or make a binary executable from SDF files.

- Creating a VMC model from source description:
  A VMC model is an executable binary that you compile from your Verilog model. To simulate the VMC model you link it to VCS. VMC is a separate Synopsys product.

These source protection methods vary in the levels of security they provide, their impact on performance, platform and vendor independence, and PLI and CLI access as described in Table B-1, Source Protection Methods.

Table B-1 Source Protection Methods

Method	Level of Security	Performance	Platform Independence	Vendor Independence	PLI and CLI Access
Mangling	Alters only the identifiers. You can read the structure of the source description.	Same as VCS or third party Verilog simulator	Yes, the output is an ASCII file	Yes, you can simulate the resulting model in any Verilog simulator	Yes, however the identifiers are difficult to understand.
Encrypting	Alters the entire source description but some identifiers can be seen in generated C code	Same as VCS	Yes, the output is an ASCII file	No, the encrypted output is in a format that only VCS can read.	Yes, but only if you know the identifiers before they were encrypted
VMC	Absolute, the output is an executable binary	Some impact on small and active VMC models	No, you can only simulate a VMC model on the platform that was used to generate it.	Yes, using standard PLI access	No access into VMC models

Note:

This Appendix describes mangling and encrypting. For information on creating a VMC model see the *VMC User's Guide*, for information on simulating a VMC model see "SWIFT VMC Models and SmartModels Introduction" on page 10-2.

Encrypting Source and SDF Files

You use compiler directives and command line options to encrypt source and SDF files. VCS creates copies of your files with encrypted contents. The following sections describe how you encrypt Verilog source and SDF files:

- The section “Encrypting Specified Regions” on page B-4 describes how you can specify what part of your source description you want VCS to encrypt.
- The section “Encrypting The Entire Source Description” on page B-6 describes how you can encrypt all the contents of your source description.
- The section “Encrypting SDF Files” on page B-9 describes the source protection options for SDF files.
- The section “Specifying Encrypted Filename Extensions” on page B-10 specifies how you can override the default filename extensions for encrypted files. The default extension is .vp.
- The section “Specifying Encrypted File Locations” on page B-11 describes how you can specify the location of the encrypted files that VCS creates. The default is the same directory as the original source or SDF file.
- The section “Multiple Runs and Error Handling” on page B-11 describes how you can circumvent problems that can occur when you perform multiple encryptions of the same files or encounter error conditions.

Encrypting Specified Regions

You can control what part of your source VCS encrypts in the new files and which part remains readable. To do so you do the following:

1. Enclose the part of your source that you want VCS to encrypt between the `` `protect `` and the `` `endprotect `` compiler directives.
2. Include the `+protect` option on the `vcs` command line.

   When you run VCS with the `+protect` option:

   - VCS creates new files for the Verilog source files you specify on the `vcs` command line.
   - VCS replaces the `` `protect `` and `` `endprotect `` compiler directives with the `` `protected `` and `` `endprotected `` compiler directives and encrypts all the Verilog source description between these `` `protected `` and `` `endprotected `` compiler directives.

The new files have the same filename except that VCS appends a "`p`" to the filename extension of the new files. For example, if you run source protection for a source file named `my_design.v`, VCS names the new file with encrypted source `my_design.vp`. This is the default filename extension for the new file. You can specify a different extension.

The following is an example of a Verilog source file with a region of the source that you want protected:

```
module top( inp, outp);
input [7:0] inp;
output [7:0] outp;
reg [7:0] count;
assign outp = count;
```

```
always
begin:counter
`protect      //begin protected region
reg [7:0] int;
count = 0;
int = inp;
while (int)
     begin
     if (int [0]) count = count + 1;
          int = int >> 1;
     end
     `endprotect  //end protected region
end
endmodule
```

The following is the contents of the new file after you run source protection with the `+protect` option:

```
module top( inp, outp);
input [7:0] inp;
output [7:0] outp;
reg [7:0] count;
assign outp = count;
always
begin:counter
`ifdef VCS
     `protected
     %%AqDtf%,%,%,%,%,%,-%,%,UB@%,|%5%B%<z%,NIS%A-%,%,DH
     ?%,NIW%A-%,%,PONKB%,%4NIS%3-%,%,%,%,EB@NI-%,%,%,%,%
     RIS%,%?%,DHRIS%,%1%,%;%A-%,%,%,%,%,%,NIS%,%?%,NIS%
,%,$
     `endprotected
`endif
//end protected region
end
endmodule
```

Encrypting The Entire Source Description

You can encrypt all the modules and UDPs in the new Verilog source files that VCS creates without yourself entering `` `protect `` and `` `endprotect `` compiler directives in your source code.

To do so include the `+autoprotect`, `+auto2protect` or `+auto3protect` option on the vcs command line. The difference between these options is where VCS begins to encrypt the Verilog source description in modules and UDPs. This difference is as follows:

`+autoprotect`
: The `+autoprotect` option encrypts the module port list (or UDP terminal list) in addition to the body of the module (or UDP.)

`+auto2protect`
: The `+auto2protect` option does not encrypt the module port list (UDP terminal list). Instead encryption begins after the semicolon following the port or terminal list. This difference produces a syntactically correct module or UDP header statement.

`+auto3protect`
: The `+auto3protect` option does what the `+auto2protect` option does except that it does not encrypt parameter declarations that precede the first port declaration.

When you include the `+auto2protect` option, VirSim and third party tools are able to parse the encrypted source files without decoding the encrypted source description that follows these lists. For this reason Synopsys recommends the use of `+auto2protect` option instead of `+autoprotect` option.

When you include the `+auto2protect` option:

- VCS creates new files for the Verilog source files you specify on the vcs command line.
- VCS inserts the `` `ifdef VCS `` compiler directive on the line that follows the module or UDP header. VCS also inserts the `` `endif `` compiler directive on the line that precedes the `endmodule` keyword.
- VCS inserts the `` `protected `` compiler directive after the module header and the `` `endprotected `` compiler directive before the `endmodule` keyword. VCS encrypts the Verilog source between the `` `protected `` and `` `endprotected `` compiler directives that it inserts.

The following is an example of a Verilog source file whose entire contents you want to encrypt with source protection:

```
module top( inp, outp);
parameter size=8;
input [size-1:0] inp;
output [size-1:0] outp;
reg [7:0] count;
assign outp = count;
parameter stoptime=100;
always
begin:counter
  reg [size-1:0] int;
  count = 0;
  int = inp;
  while (int)
    begin
      if (int [0]) count = count + 1;
      int = int >> 1;
    end
  #stoptime $stop;
end
endmodule
```

The following is an example of the contents of the new source file VCS creates when you run source protection with the `+auto2protect` option:

```
module top( inp, outp);
`ifdef VCS  // VCS Release 4.1
`protected
\,MGH?JR[D?0R_D#+XJ(MQD#HgXV@ZUVI2+HT)1OS)C8#L7OA[9ge#^#5@WO0P_<
,Y)[^ZVDDCBf<EB?2(=)>S#aSR58?]Qgg6\#OOf<^&#5.+RK[6<#2&X>SZM:)F9>
VLf:FHRSd[QP=WCC\gA;=g5M=>PG5EDUaZ:#/If[CTXV9RKJNNOf]>Cfgg[4&W.f
=2FD]<,R0?@:B0R:?\4fP_dgaGgF_IB9MV#E1M?b2)Cd._<:&@,KV\a5:Q3D]CPL
[9HDe2.gQYL0;\_Y^V\a;_Ag-fP;+K\;GUU/:HXFf;gaGJ1fO8_f1M)eGF8LRRY]
CB75+Q/]R_IAP/>HS+b<XFP,-BHfcZTIG0-QILLIa1#.RbX6.K?Oc8f5]f);BW=Q
FVa@-^&@e+:K0>(?(&ZL:4Z:.F[<5J)gQ+CaA]^7\.N^/0@RRZQe-]A,Z+L>?FQG
([A0S=LXHPgN[<Dg5fQG?^6IUP7.VV3a@$
`endprotected
`endif
endmodule
```

The following is an example of the contents of the new source file VCS creates when you run source protection with the `+auto3protect` option:

```
module top( inp, outp);
parameter size=8;
`ifdef VCS  // VCS Release 4.1
`protected
\,MGH?JR[D?0R_D#+XJ(MQD#HgXV@ZUVI2+HT)1OS)C8#L7OA[9ge#^#5@WO0P_<
,Y)[^ZVDDCBf<EB?2(=)>S#aSR58?]Qgg6\#OOf<^&#5.+RK[6<#2&X>SZM:)F9>
VLf:FHRSd[QP=WCC\gA;=g5M=>PG5EDUaZ:#/If[CTXV9RKJNNOf]>Cfgg[4&W.f
=2FD]<,R0?@:B0R:?\4fP_dgaGgF_IB9MV#E1M?b2)Cd._<:&@,KV\a5:Q3D]CPL
[9HDe2.gQYL0;\_Y^V\a;_Ag-fP;+K\;GUU/:HXFf;gaGJ1fO8_f1M)eGF8LRRY]
CB75+Q/]R_IAP/>HS+b<XFP,-BHfcZTIG0-QILLIa1#.RbX6.K?Oc8f5]f);BW=Q
FVa@-^&@e+:K0>(?(&ZL:4Z:.F[<5J)gQ+CaA]^7\.N^/0@RRZQe-]A,Z+L>?FQG
([A0S=LXHPgN[<Dg5fQG?^6IUP7.VV3a@$
`endprotected
`endif
endmodule
```

Notice that parameter size is not encrypted but parameter stoptime is.

The following is an example of the contents of the new file VCS creates when you run source protection with the `+autoprotect` option:

```
module top
`ifdef VCS  // VCS Release 4.1
`protected
>Z.>B/)c?G.-UDVP<^?H0Yb_HSKg0>@UY0+-eY-/@YK^^Q3g+K6c\\5#ge+/8McF
WaK-/QI-?Gc^d8.))aeZ1gQBZbE_1.[U;1dVZ9b#.(A336;]&NM^GF,dR34)I]GZ
FEc-R?e1V3Mb1SUUB,J7R4[T\LZe?BDb#^@7/_5IAdM<(J.BN7\[<[^We?@JeVSf
FKC)D.#bB@C#L^I4(55SG>:SXgTL^&GCG<&&6bQO;;EG)S]2d7X6,U:,<:S:<<KM
=&<OWaR?Ef.SEQW;RKYD.F\Se&WE+ALI=PafD8b:T[4#gbK??IgJ@4c&0?#H]1:1
4HJ)e)G^U4:Y)^4@48:Gf:I6Ue)NG\JH:@fL5Q3?Gc+R)#)9f3g-cA(CM2fTf&(:
_+HVA=2@:MR&^=>5EbYF63a\/BQ71<H>\GMJe&,PIA&I_9O4fMd7#PNW1?KgK2UP
XbVJ+FB\;6IY)+QZQTBS<+f;c:YeOc7eI(G:cT#1&7E]Jda1^]]H@cNWM$
`endprotected
`endif
endmodule
```

In this file encryption begins after the module identifier instead of after the port list.

Encrypting SDF Files

To create a new SDF file with encrypted content include, on the vcs command line, the `+sdfprotect` option with the SDF filename.

For example, the following is the contents of an SDF file named `delays.sdf`, that you want to encrypt.

```
(DELAYFILE
          (DESIGN"test")
          (VENDOR"Synopsys")
          (DIVIDER.)
          (VOLTAGE:1:)
          (PROCESS"typical")
          (TEMPERATURE:1:)
          (TIMESCALE10ps)
          (CELL
            (CELLTYPE"leaf")
```

```
            (INSTANCEleaf1)
            (DELAY
            (ABSOLUTE
            (IOPATH in1 out1 (50.5))
            (IOPATH in2 out2 (51.5))
            (IOPATH in3 out3 (00.5))
            (IOPATH in4 out4 (01.5))))
          )
        )
```

The following is the contents of the delays.sdfp file that VCS creates:

```
`protected
'AtDV?SR[VNQ^[R%B7777777777?SRD^PY77777775crdc5>%B7T~Zexyx
{xp~Yt5>%B7777777777?S^A^SRE7777779>%B77777777777777?GEXT
RDD7777775cng~Ytv{5>%B7777777777?CRZ77?C^ZRDTV[R7777&'gd>%B
7777777777?TR[[%B77777777777777777777?^YDCVYTR777{rvq&>%B
77777777777?SR[VN%B77777777777?^XGVC_7~Yy&7xbc&7?%G'9%G>>%
B7777777777779%G>>%B77777777777?^XGVC_7~Yy%I7xbc%I7?''9%G>>
%B77xbc%H7?'&9%G>>>>%B7777777777>%B77777777>%B%B$
`endprotected
```

Specifying Encrypted Filename Extensions

You can specify a different extension for the new filenames by adding the new extension as a suffix to the `+protect, +auto2protect, +autoprotect`, or `+sdfprotect` options.

For example, if you include the `+protect.protected` option on the `vcs` command line, for the source file named `my_design.v`, the encrypted new filename is `my_design.v.protected`.

Specifying Encrypted File Locations

You can specify the directory where VCS writes the new and encrypted files with the `+putprotect+` option followed by the path to the directory where you want VCS to create the new files. Include this option along with the `+protect`, `+auto2protect`, `+autoprotect`, or `+sdfprotect` option on the `vcs` command line.

Make sure there are no spaces between the `+putprotect+` option and the path to this directory. For example, with the following command line:

```
vcs cache.v +auto2protect.encrypt +putprotect+/u/design/
cache
```

VCS creates the encrypted file cache.v.encrypt in the directory /u/design/cache.

Multiple Runs and Error Handling

When VCS creates encrypted files:

- If the encrypted file already exists, VCS outputs an error message and does not overwrite the existing file. This error message is: `Error: file 'filename.vp' already exists.`
- If an error condition exists in the source file, VCS continues to create the encrypted file. This file might be corrupt.

To circumvent these problems you can use the `+deleteprotected` source protection option. This option disables the check for an existing encrypted file and deletes the encrypted file if an error condition occurred in that file.

Permitting CLI/PLI Access to Encrypted Modules

The `+pli_unprotected` option can be used to disable the normal PLI/CLI protection that is placed on encrypted modules. The output files will still be encrypted, but CLI and PLI users will *not* be prevented from accessing data in the modules.

This option is used in conjunction with the `+protect`, `+autoprotect` or `+auto2protect` options. This option only affects the modules in the current file(s). Modules in files which were encrypted without using this flag are still fully protected.

This option only affects encrypted verilog files. It is ignored when encrypting SDF files.

Note: Turning off PLI/CLI protection will allow users of your modules to access object names and values, etc. In essence, the source code for your module could be substantially reconstructed using the CLI debugging commands and PLI acc routines.

Simulating Encrypted Models

When you simulate an encrypted model, information about hierarchical names and design objects is protected. This means that some CLI commands, such as those that display hierarchical names and design objects at a specified level of the hierarchy, do not work at protected levels of the design hierarchy. Protected levels are the encrypted levels.

Certain system tasks continue to work in protected levels of the design hierarchy but you need to know the hierarchical names in these protected regions and there is no way, other than from the vendor that encrypted the model, that you can obtain the hierarchical names in protected levels.

The `-Xman` compile-time option does not work in protected source descriptions.

Using the CLI

Some CLI commands whose arguments are design objects or instances do not work when these objects are in protected regions. These CLI commands are:

```
info            scope           show
```

Note:
The `info` command continues to display the simulation time.

Some CLI commands whose arguments are design objects continue to work but only if you know the hierarchical name of the object. These CLI commands are:

```
always      break       force       once
print       release     set         tbreak
```

You must know the hierarchical path names to use these commands.

Using System Tasks

The following system tasks continue to display design information from a protected region of a design:

`$display` `$write` `$monitor` `$strobe` `$gr_waves`

Note:

For `$display` the `%m` format specification does not work in protected regions

Writing PLI Applications

PLI access is restricted for encrypted modules. Any module which has any portion of it protected will cause the entire module to be deemed protected.

All `acc_next_...` type routines are blocked on protected modules. For example, `acc_next_port()` will immediately return NULL when operating on an instance who's module definition is protected.

The routine `acc_object_of_type()` can be used to determine if a module (or macromodule or primitive) is protected. Usage is:

```
if (acc_object_of_type(obj, accProtected)) {
    printf("object %s is from a protected module\n",
           acc_fetch_name(obj));
}
```

If the object passed to `acc_object_of_type()` is a design object in a module, a port or register for example, the status of the parent module will be returned.

Mangling Source Files

The purpose of mangling is to create a Verilog source file that is functionally the same as an original Verilog file or set of Verilog files, but in the new Verilog source file the identifiers (module, instance, or signal names) no longer make any sense and all the comments are removed, making it very difficult to understand the design contained in the new file.

Mangling does not change the structure of the source description, in the new mangled file the keywords and syntax remain the same as those in the original Verilog file or set of Verilog files.

One use for this new Verilog source file is that you can send the mangled file to VCS_support@synopsys.com so that our Corporate Applications Engineers can examine problems you are experiencing while maintaining the only intelligible copy of your design at your site.

The name of the mangled source file you create is tokens.v.

You mangle your source description with the `-Xmangle` or `-Xmangle=`*`number`* option. When you enter this option, VCS creates a file named `tokens.v` that contains the mangled source description.

You can substitute `-Xman` for `-Xmangle`. The argument *`number`* can be 1 or 4:

`-Xman=1`
: Randomly changes names and identifiers to provide a more secure code.

`-Xman=4`
: Preserves variable names but removes comments.

`-Xman=12`

Does the same as `-Xman=4` but also adds comments about the system tasks in the source and the source file and line numbers of the module headers.

`-Xman=28`

Does the same as `-Xman=12` but adds a statistical analysis of the design.

The following is an example of Verilog source description before mangling:

```
module demo (modulus1,cpuData);
input [7:0] modulus1;
output [255:0] cpuData;
integer cpuTmpCnt;
reg [0:34] iPb[0:10];  //incoming pbus data buffer
reg [255:0] cpuDatareg;
assign cpuData = cpuDatareg;
function [0:255] merge_word;
input [0:31] source_word;
input [0:2] word_index;
begin
end
endfunction
initial begin
cpuDatareg = 256'h0;
for(cpuTmpCnt = 0; cpuTmpCnt<8; cpuTmpCnt=cpuTmpCnt+1)
begin : assemble_incoming
reg [0:34] inData35;
inData35=iPb[cpuTmpCnt];
$display("\tiPb[%0h]=%b, %h", cpuTmpCnt,
iPb[cpuTmpCnt],iPb[cpuTmpCnt] >> 3);

cpuDatareg=merge_word(cpuData,inData35[0:31],cpuTmpCn
t);
end
end
endmodule
```

The following is the mangled code in tokens.v produced by the -Xmangle or –Xmangle=1 option:

```
‘portcoerce
‘inline
// No timescale specified
module Memoe(Remoe, Afmoe);

input [7:0] Remoe;
output [255:0] Afmoe;

integer Ifmoe;
reg [255:0] Wfmoe;

reg [0:34] Sfmoe[0:10];

assign Afmoe = Wfmoe;

function [0:255] Hgmoe;

input [0:255] Sgmoe;
input [0:31] Ehmoe;
input [0:2] Qhmoe;

reg [0:255] Sgmoe;
reg [0:31] Ehmoe;
reg [0:2] Qhmoe;
begin
end
endfunction

initial begin
Wfmoe = 256’b0;
for (Ifmoe = 0; (Ifmoe < 8); Ifmoe = (Ifmoe + 1)) begin : Bimoe

reg [0:34] Timoe;
Timoe = Sfmoe[Ifmoe];
$display("iPb[%0h]=%b, %h", Ifmoe, Sfmoe[Ifmoe],
(Sfmoe[Ifmoe] >> 3));
Wfmoe = Hgmoe(Afmoe, Timoe[0:31], Ifmoe);
end
```

```
end
endmodule
```

Notice that comments are not included in the mangled file.

The following is the mangled code in tokens.v produced by the `-Xmangle=4` option:

```
‘portcoerce
‘inline
// No timescale specified
module demo(modulus1, cpuData);

     input[7:0]    modulus1;
     output[255:0] cpuData;

     integer       cpuTmpCnt;
     reg  [255:0]  cpuDatareg;

     reg  [0:34]   iPb[0:10];

     assign cpuData = cpuDatareg;

     function [0:255] merge_word;

     input[0:255]  source_line;
     input[0:31]   source_word;
     input[0:2]    word_index;

     reg  [0:255]  source_line;
     reg  [0:31]   source_word;
     reg  [0:2]    word_index;
     begin
     end
     endfunction

     initial begin
       cpuDatareg = 256’b0;
       for (cpuTmpCnt = 0; (cpuTmpCnt < 8); cpuTmpCnt =
```

```
(cpuTmpCnt + 1))
           begin : assemble_incoming

         reg[0:34] inData35;
         inData35 = iPb[cpuTmpCnt];
         $display("iPb[%0h]=%b, %h", cpuTmpCnt,
iPb[cpuTmpCnt],
              (iPb[cpuTmpCnt] >> 3));
         cpuDatareg = merge_word(cpuData, inData35[0:31],
cpuTmpCnt);
       end
     end
endmodule
```

The comment is removed and default compiler directives and a comment about timescales is added.

The following is the mangled code in tokens.v produced by the `-Xmangle=12` option:

```
/* System Tasks:
     $display

*/

`portcoerce
`inline
/* Source file "mangle2.v", line 1 */
// No timescale specified
module demo(modulus1, cpuData);

     input[7:0]    modulus1;
     output[255:0] cpuData;

     integer       cpuTmpCnt;
     reg  [255:0]  cpuDatareg;

     reg  [0:34]   iPb[0:10];

     assign cpuData = cpuDatareg;
```

```
    function [0:255] merge_word;

    input[0:255]  source_line;
    input[0:31]   source_word;
    input[0:2]    word_index;

    reg  [0:255]  source_line;
    reg  [0:31]   source_word;
    reg  [0:2]    word_index;
    begin
    end
    endfunction

    initial begin
      cpuDatareg = 256'b0;
      for (cpuTmpCnt = 0; (cpuTmpCnt < 8); cpuTmpCnt =
(cpuTmpCnt + 1))
          begin : assemble_incoming

        reg[0:34] inData35;
        inData35 = iPb[cpuTmpCnt];
        $display("iPb[%0h]=%b, %h", cpuTmpCnt,
iPb[cpuTmpCnt],
             (iPb[cpuTmpCnt] >> 3));
        cpuDatareg = merge_word(cpuData, inData35[0:31],
cpuTmpCnt);
      end
    end
endmodule
```

Here there are additional comments about the system tasks used and the source file and line number of the module header.

The following is the mangled code in tokens.v produced by the `-Xmangle=28` option:

```
/* System Tasks:
    $display
```

```
*/

`portcoerce
`inline
/* Source file "mangle2.v", line 1 */
// No timescale specified
module demo(modulus1, cpuData);

    input[7:0]    modulus1;
    output[255:0] cpuData;

    integer       cpuTmpCnt;
    reg  [255:0]  cpuDatareg;

    reg  [0:34]   iPb[0:10];

    assign cpuData = cpuDatareg;

    function [0:255] merge_word;

    input[0:255]  source_line;
    input[0:31]   source_word;
    input[0:2]    word_index;

    reg  [0:255]  source_line;
    reg  [0:31]   source_word;
    reg  [0:2]    word_index;
    begin
    end
    endfunction

    initial begin
      cpuDatareg = 256'b0;
      for (cpuTmpCnt = 0; (cpuTmpCnt < 8); cpuTmpCnt =
(cpuTmpCnt + 1))
          begin : assemble_incoming

        reg[0:34] inData35;
        inData35 = iPb[cpuTmpCnt];
        $display("iPb[%0h]=%b, %h", cpuTmpCnt,
iPb[cpuTmpCnt],
            (iPb[cpuTmpCnt] >> 3));
```

```
         cpuDatareg = merge_word(cpuData, inData35[0:31],
cpuTmpCnt);
       end
     end
endmodule

// ===================
//  DESIGN STATISTICS
// ===================
//                                  Static<!>  Elaborated<@>
//                                     ------     ----------
// No. of module instances:                 1             1
// No. of comb. udp instances:              0             0
// No. of seq. udp instances:               0             0
// No. of gates:                            0             0
// No. of continuous assignments:           1             1
// No. of module+udp port connects:         0             0
// No. of registers (vector+scalar):        7             7
// No. of memories:                         1             1
// No. of scalar nets:                      0             0
// No. of vector nets:                      2             2
// No. of named events:                     0             0
// No. of always blocks:                    0             0
// No. of initial blocks:                   1             1
// No. of fork/join blocks:                 0             0
// No. of verilog tasks:                    0             0
// No. of verilog functions:                1             1
// No. of verilog task calls:               0             0
// No. of verilog function calls:           1             1
// No. of system task calls:                1             1
// No. of user task calls:                  0             0
// No. of system function calls:            0             0
// No. of user function calls:              0             0
// No. of hierarchical references:          0             0
// No. of concatenations:                   0             0
// No. of force/release statements:         0             0
// No. of bit selects:                      0             0
// No. of part selects:                     1             1
// No. of non-blocking assignments:         0             0
// No. of specify blocks<#>:                0             0
// No. of timing checks:                    0             0
//
```

```
// No. of top level modules:                    1
//        modules:                              1
// No. of udps:                                 0
//        seq. udps:                            0
//        comb. udps:                           0
// No. of module+udp ports:                     2
// No. of system tasks:                         1
// No. of user tasks:                           0
// No. of system functions:                     0
// No. of user functions:                       0
//
// Footnotes:
// ---------
// <!> No. of unique instances of a construct as it appears
// in the source.
// <@> No. of instances of a construct when the design //
is elaborated.
// <#> Multiple specify blocks in the SAME module are //
combined and counted as ONE block.
```

Note: If you are creating a mangled tokens.v file to send to Synopsys for technical support, we advise checking to see if you can duplicate the problem using the tokens.v file. Also please include with this file a copy of the vcs and simv command lines that duplicate the problem and any SDF files that need to be back annotated and C source files or object files for PLI applications that need to be linked to the simv executable.

Creating A Test Case

Here is a quick way to create a small test case:

1. Remove `-o` option if you are using it.

2. Add the `-Xman=4` option to your vcs command line (in addition to whatever other options you are using) and run VCS.
   VCS will not actually compile your design, instead it generates a file called tokens.v.
3. Rename the tokens.v file to tokens.orig
4. Run the following command line:

```
vcs tokens.orig
```

5. Cut out the problem module from tokens.orig and paste it into a new file, say test.v.
6. Run the following command line:

```
vcs -Xman=4 test.v -v tokens.orig
```

This will create a new tokens.v file

7. Run the following command line again:

```
vcs tokens.v
```

Do this to verify that the problem still exists.

8. Zip the tokens.v file
9. FTP the tokens.v.gz file to us:
   a. Connect to our FTP site by entering:

```
ftp ftp.synopsys.com
```

   b. At the Name prompt, enter anonymous.
      At the Password prompt, enter your e-mail address.
   c. At the `ftp>` prompt enter:

```
cd incoming
```

   d. Enter the following FTP command:

```
binary
```

e. Enter the following FTP command:

```
put tokens.v.gz
```

10. Now send e-mail to VCS_support@synopsys.com informing us that you have uploaded the test case.

Preventing Mangling of Top-Level Modules

You can prevent the mangling of top-level modules in your source description with the `-Xnomangle` option. The syntax for this option is:

```
-Xnomangle=.first | module_identifier
[, module_identifier...]
```

For example:

```
-Xnomangle=.first
```

Prevents the mangling of the first module VCS encounters in the source description.

```
-Xnomangle=top1,top2
```

Prevents the mangling of modules top1 and top2

The `-Xnomangle` option should only be used on top level modules, i.e., ones that are not called by any other module in the design being mangled. The code which performs the mangling knows how to identify module and port declarations, but not module references. Thus, if you exempt a module which is referenced by some other module, the reference will try to use the mangled name instead of the original (and fail).

Index

. (period) CLI command 5-3

Symbols

3-28
- 3-28
-a filename 4-14, 4-33
-as assembler 3-69
-ASFLAGS 3-69
-assert 3-46, 15-90
-B 3-81
-C 3-70, 3-72
-c 3-70
-CC 3-71
-cc 3-71
-CFLAGS 3-71
-cm 15-97
-cm_assert_cov 15-99
-cm_assert_cov_cover 15-99
-cm_assert_dir 15-97, 15-100
-cm_assert_grade_instances 15-100
-cm_assert_grade_modules 15-100
-cm_assert_hier 15-97
-cm_assert_map 15-100
-cm_assert_merge 15-101
-cm_assert_name 15-97
-cm_assert_report 15-101
-cm_fsmopt allowTemp 3-43
-cm_fsmopt optimist 3-43
-cm_fsmopt report2StateFsms 3-43
-cm_fsmopt reportvalues 3-44
-cm_fsmopt reportWait 3-44
-cm_fsmopt reportXassign 3-44
-cm_fsmresetfilter 3-44
-cm_line contassign 3-45
-cm_opfile 3-45
-cm_scope 3-46
-cpp 3-71
-debug 3-46
-debug_all 3-46
-doc 3-26
-e 14-44, 15-99
-e name_for_main 3-60, 7-34
-E program runtime option 4-23
-F filename 3-58
-f filename 1-15, 3-57
-file 3-58
-gen_asm 3-69
-gen_c 3-71
-gen_obj 3-80
-grw 4-18
-gui 3-47, 4-15

-h 3-26
-help 3-26
-I 2-98, 3-49
-i 4-16, 5-9
-i filename 3-38, 4-10
-ID 3-69
-jnumber_of_CPUs 3-25, 3-71
-k 4-10, 4-16, 5-9
-l 4-16
-l filename 1-15, 1-19, 3-79, 4-14, 4-33
-ld linker 3-70
-LDFLAGS 3-70
-line 1-15, 3-33
-lmc-hm 3-79
-lmc-swift 3-65, 10-15
-lmc-swift-template 3-65, 10-4
-lname 3-70
-load 3-61, 7-33
-Mdefine 3-23
-Mdelete 3-23
-Mdirectory 3-24
-Mfilename 3-24
-Minclude 3-24
-Mldcmd 3-24
-Mloadlist 3-25
-Mmakefile 3-25
-Mmakeprogram 3-25
-Mrelative_path 3-25
-Msrclist 3-26
-Mupdate 3-3, 3-23
-ntb 3-28
-ntb_opts 3-28
-ntb_sfname 3-30
-ntb_vipext 3-30
-ntb_vl 3-30
-o name 3-80
-O number 3-72
-O0 3-72
-ova_cov 3-38, 4-6, 14-19, 14-23
-ova_cov_db 4-6, 14-23, 14-45
-ova_cov_events 14-45
-ova_cov_grade_instances 14-45
-ova_cov_grade_modules 14-45
-ova_cov_hier 3-39, 14-19
-ova_cov_map 14-46
-ova_cov_merge 14-46
-ova_cov_name 4-6, 14-23
-ova_cov_report 14-46
-ova_debug 3-39, 14-19
-ova_debug_vpd 3-39, 14-19
-ova_dir 14-20
-ova_enable_diag 3-39, 14-20
-ova_file 3-39, 14-20
-ova_filter 4-5, 14-22
-ova_filter_past 3-39, 14-20
-ova_inline 3-39, 14-21
-ova_max_fail 4-6, 14-22
-ova_max_success 4-6, 14-22
-ova_name 4-5
-ova_quiet 4-4, 14-21
-ova_report 4-4, 14-21
-ova_simend_max_fail 4-6, 14-22
-ova_success 14-22
-ova_verbose 4-5, 14-21
-ovac 3-38, 14-19
-override_timescale 3-76
-P pli.tab 3-60
-parameters 3-18, 3-75
-platform 3-80
-PP 2-98, 3-47, 3-49
-pvalue 3-18, 3-75
-q 3-65, 4-14
-R 1-16, 3-38
-RI 3-47, 3-49
-RIG 3-48
-RPP 3-50

-S 3-69
-s 1-16, 3-38, 4-23
-sv_pragma 3-27, 15-95
-sysc 3-76
-syslibs libs 3-70
-u 3-79
-ucli 3-47, 4-10, 4-16
-V 3-66, 4-15
-v 3-21
-vcd filename 4-19
-vera 1-17, 3-40
-vera_dbind 3-40
-Vt 3-66
-Xman 3-73, B-13, B-15
-Xmangle 3-73
-Xnoman 3-74
-Xnomangle 3-74
-y 1-17, 3-21
$ token 15-45
$assert_monitor 2-103, 15-116
$assert_monitor_off 2-104, 15-116
$assert_monitor_on 2-104, 15-116
$assertkill 2-91
$assertoff 2-91
$asserton 2-91
$async$and$array 2-114
$bitstoreal 2-105
$countdrivers 2-117
$countones 15-120
$deposit 2-117
$disable_warnings 2-111
$display 2-106
$display statement 14-78
$dist_exponential 2-115
$dist_normal 2-115
$dist_poisson 2-115
$dist_uniform 2-115
$dumpall 2-92
$dumpchange 2-93
$dumpfile 2-92
$dumpflush 2-92
$dumplimit 2-92
$dumpoff 2-92
$dumpon 2-92
$dumpports 2-95
$dumpportsall 2-96
$dumpportsflush 2-96
$dumpportslimit 2-97
$dumpportsoff 2-95
$dumpportson 2-96
$dumpvars 2-93, 3-3
$enable_warnings 2-111
$error 2-90
$fatal 2-90
$fclose 2-107
$fdisplay 2-107
$fell 15-50
$ferror 2-56, 2-107
$fflush 2-53, 2-93, 2-107
$fflushall 2-93
$fgetc 2-49, 2-107
$fgets 2-52, 2-107
$finish 2-111
$fmonitor 2-107
$fopen 2-43, 2-107
$fread 2-47, 2-108
$fscanf 2-45, 2-108
$fseek 2-55, 2-108
$fstobe 2-108
$ftell 2-53, 2-108
$fwrite 2-108
$getpattern 2-118
$gr_waves 2-94
$hold 2-112
$info 2-90
$isunknown 15-120

$itor 2-105
$log 2-105
$lsi_dumpports 2-94, 2-121–2-126
$monitor 2-106
$monitoroff 2-106
$monitoron 2-106
$nolog 2-105
$onehot 15-120
$onehot0 15-120
$ova_current_time 14-79
$ova_global_time_unit 14-79
$ova_severity_action 14-80
$ova_start 14-81
$ova_start levels 14-80
$ova_start_time 14-79
$ova_stop 14-82
$ova_stop levels 14-79
$ovadumpoff 14-80
$ovadumpon 14-81
$past 15-62
$period 2-112
$printtimescale 2-110
$q_add 2-114
$q_exam 2-114
$q_full 2-114
$q_initialize 2-114
$q_remove 2-114
$random 2-116, 2-120
$read_lib_saif 2-129
$read_rtl_saif 2-129
$readmemb 2-109
$readmemh 2-109
$realtime 2-115
$realtobits 2-105
$recovery 2-112
$recrem 2-112
$removal 2-113
$reset 2-116
$reset_count 2-116
$reset_value 2-116
$restart 2-118, 4-29
$rewind 2-55
$root 15-75
$rose 15-49
$rtoi 2-106
$save 2-118
$sdf_annotate 2-117
$set_gate_level_monitoring 2-128
$set_toggle_region 2-127
$setup 2-113
$setuphold 2-113
$sformat 2-48
$skew 2-113
$sreadmemb 2-109
$sreadmemh 2-110
$sscanf 2-46
$stable 15-50
$stime 2-115
$stop 2-111
$strobe 2-106
$swrite 2-48
$sync$nor$plane 2-114
$system 2-104
$systemf 2-104
$test$plusargs 2-116
$time 2-115
$timeformat 2-110
$toggle_report 2-128
$toggle_reset 2-128
$toggle_start 2-127
$toggle_stop 2-127
$ungetc 2-50, 2-109
$value$plusargs 4-31
$warning 2-90
$width 2-114
$write 2-106

$writememb 2-110
$writememh 2-110
%= 15-28
%CELL 7-13, 7-17
  use of 8-35
%TASK 7-14
&= 15-28
**NC 5-16
*= 15-28
+2state 3-32, 13-27
+acc+level_number 3-36, 7-22
+ad 3-74
+allmtm 3-51, 3-54, 8-57
+applylearn 3-36, 7-25–7-29
+auto2protect 3-73, B-6
+auto3protect 3-73, B-6
+autoprotect 3-73, B-6
+cfgfile 3-50
+charge_decay 3-52
+cli+level_number 1-14, 3-33
+cliecho 4-10
+cliedit 3-34
+csdf+precomp+dir 3-55, 8-30
+csdf+precomp+ext 3-55, 8-30
+csdf+precompile 3-54, 8-29
+define+macro=value 1-15, 3-79
+delay_mode_distributed 3-52, 8-22
+delay_mode_path 3-52, 8-22
+delay_mode_unit 3-52, 8-22
+delay_mode_zero 3-52, 8-22
+deleteprotected 3-73, B-11
+incdir 1-15, 3-21
+libext 3-22
+liborder 3-22
+librescan 3-22
+libverbose 3-22, 3-65
+lint 3-65
+maxdelays 3-53, 3-55, 4-20, 8-56, 8-57, 10-16, 10-21
+memcbk 3-78
+memopt 3-13, 3-77
+mindelays 3-53, 3-55, 4-20, 8-56, 8-57, 10-16, 10-21
+multisource_int_delays 3-53, 3-54, 8-6, 8-35
+nbaopt 3-53
+neg_tchk 3-63, 8-58, 9-12, 9-19
+no_notifier 3-62, 4-12, 9-12
+no_pulse_msg 4-15
+no_tchk_msg 3-62, 4-13, 9-13
+nocelldefinepli+0 3-67
+nocelldefinepli+1 3-67
+nocelldefinepli+2 3-67
+noerrorIOPCWM 3-77
+noinitnegedge 3-32, 13-27, 13-28
+nolibcell 3-67
+nospecify 1-16, 3-61, 9-13
+notimingcheck 1-16, 1-19, 3-61, 4-13, 4-33, 9-13
+ntb_cache_dir 4-2
+ntb_debug_on_error 4-3
+ntb_enable_solver_trace 4-3
+ntb_enable_solver_trace_on_failure 4-3
+ntb_enable_solver_trace_on_failure=value 4-3
+ntb_engable_solver_trace 4-3
+ntb_exit_on_error 4-3
+ntb_load 4-3
+ntb_random_seed 4-3
+ntb_solver_mode 4-4
+ntb_solver_mode=value 4-4
+ntb_stop_on_error 4-4
+NTC2 3-63, 9-18
+old_ntc 3-63
+oldsdf 3-55, 8-26
+optconfigfile 3-31, 3-33, 6-2, 6-4
+overlap 3-63, 9-22

+override_model_delays 4-24, 10-16, 10-21
+pathpulse 3-61
+pli_unprotected 3-73, B-12
+plusarg_ignore 3-58
+plusarg_save 3-58
+plus-options 4-24
+prof 3-56
+protect file_suffix 3-73
+pulse_e/number 3-59, 8-6, 8-8, 8-9, 8-12, 8-17, 8-18
+pulse_int_e 3-59, 8-7, 8-8, 8-10, 8-12
+pulse_int_r 3-59, 8-7, 8-8, 8-10, 8-12
+pulse_on_detect 3-59, 8-19
+pulse_on_event 3-59, 8-18
+pulse_r/number 3-59, 8-6, 8-8, 8-9, 8-12, 8-17, 8-18
+putprotect+target_dir 3-73, B-11
+race 3-37
+race=all 3-37, 5-22
+race_maxvecsize 3-37, 5-12
+racecd 3-37
+rad 2-43, 3-31, 6-1
+sdf_nocheck_celltype 3-55, 8-38
+sdfprotect B-9
+sdfprotect file_suffix 3-73
+sdfverbose 4-15
+sim 3-48
+spl_read 3-77
+systemverilogext 3-30
+timopt 8-59
+transport_int_delays 3-54, 8-7, 8-10, 8-12
+transport_path_delays 3-54, 8-7, 8-9, 8-12
+typdelays 3-53, 3-55, 4-21, 8-56, 8-57, 10-16, 10-21
+v2k 1-17, 3-76
+vc 3-62
+vcdfile 3-50
+vcs+boundscheck 3-20, 3-77
+vcs+dumpoff+t+ht 4-19
+vcs+dumpon+t+ht 4-19
+vcs+dumpvarsoff 4-19, 4-33
+vcs+finish 4-14, 4-30
+vcs+flush+all 4-22
+vcs+flush+dump 4-22
+vcs+flush+fopen 4-22
+vcs+flush+log 4-22
+vcs+grwavesoff 4-18
+vcs+ignorestop 4-24, 4-33
+vcs+initmem 3-31
+vcs+initreg 3-31
+vcs+learn+pli 1-20, 4-23, 7-25–7-29
+vcs+lic+vcsi 3-68, 4-22
+vcs+lic+wait 3-69, 4-22
+vcs+mipd+noalias 4-24
+vcs+mipdexpand 3-56
+vcs+nostdout 4-15
+vcs+stop 4-13, 4-30
+vcsi+lic+vcs 3-69, 4-22
+vcsi+lic+wait 3-69
+vera_load 4-10
+vera_mload 4-10
+vera_stop_on_error 4-4
+verilog1995ext 3-31
+verilog2001ext 3-31
+vissymbols 3-57
+vpdfile 3-49, 3-51
+vpdfileswitchsize 3-50
+vpi 3-60
+vslogfile 3-48, 3-50
+vslogfilesim 3-48
+warn 2-121, 3-66
+warn2val 3-32, 13-28
/= 15-28
<<<= 15-28
<<= 15-28
-= 15-28

=+ 15-28
>>= 15-28
>>>= 15-28
? CLI command 5-3
^= 15-28
|= 15-28
‘accelerate 2-89
‘autoexpand_vectornets 2-89
‘celldefine 2-80
‘default_nettype 2-81
‘default_rswitch_strength 2-89
‘default_switch_strength 2-89
‘default_trireg_strength 2-89
‘define 2-81
‘delay_mode_distributed 2-84
‘delay_mode_path 2-84
‘delay_mode_unit 2-84
‘delay_mode_zero 2-84
‘disable_portfaults 2-89
‘else 2-81
‘elseif 2-82
‘endcelldefine 2-81
‘endif 2-82
‘endprotect 2-85
‘endprotected 2-86
‘endrace 2-83
‘expand_vectornets 2-89
‘ifdef 2-82
‘ifndef 2-83
‘include 2-87
‘line 2-88
‘noaccelerate_ignored 2-89
‘noexpand_vectornets 2-89
‘noportcoerce 2-15, 2-86
‘noremove_gatenames 2-89
‘noremove_netnames 2-89
‘nosuppress_faults 2-89
‘nounconnected_drive 2-88
‘portcoerce 2-86
‘protect 2-86
‘protected 2-86
‘race 2-83
‘remove_gatenames 2-89
‘remove_netnames 2-89
‘resetall 2-81
‘suppress_faults 2-89
‘timescale 2-87
‘unconnected_drive 2-88
‘undef 2-83
‘uselib 2-87
‘vcs_mipdexpand 2-85
‘vcs_mipdnoexpand 2-85

Numerics

+2state 3-32, 13-27

A

-a filename 4-14, 4-33
+acc+level_number 3-36, 7-22
ACC capabilities 7-27
  cbk 7-15, 7-19
  cbka 7-15
  frc 7-15, 7-19
  gate 7-15
  mip 7-15
  mipb 7-15
  mp 7-15
  prx 7-15
  r 7-14, 7-18
  rw 7-15, 7-18
  s 7-15
  specifying 7-12–7-20
  tchk 7-15
ACC capabilities for SDF back annotation 8-31–8-35
acc_handle_simulated_net() PLI routine 4-24

‘accelerate 2-89
action blocks 15-67
action_block 14-78
+ad 3-74
alias CLI command 5-4
+allmtm 3-51, 3-54, 8-57
always CLI command 5-4
always_comb block 15-32
always_ff block 15-36
always_latch block 15-35
and operator 15-50
anding sequences 15-50
annotation
  overhead 8-35
+applylearn 3-36, 7-25–7-29
args PLI Specificaction 7-10
arrays
  indexing and slicing 15-14
-as assembler 3-69
-ASFLAGS options 3-69
assembly code generation
  assembling by hand 3-70
  passing options to the assembler 3-69
  specifying 3-69
  specifying another assembler 3-69
-assert 3-27, 3-46, 4-7, 15-90
assert OVA directive 14-3
assert statements 15-63–15-64
$assert_monitor 2-103, 15-116
$assert_monitor_off 2-104, 15-116
$assert_monitor_on 2-104, 15-116
Assertion failure, displaying message 14-78
assertion files 14-3
assertion pragmas 14-62
assertions 14-4
assertions, monitoring 14-73
$assertkill 2-91
$assertoff 2-91
$asserton 2-91
$async$and$array 2-114
+auto2protect 3-73, B-6
+auto3protect 3-73, B-6
‘autoexpand_vectornets 2-89
+autoprotect 3-73, B-6

B

-B 3-81
behavioral drivers 15-17
benefits of OpenVera Assertions 14-2
bit data type 15-3
$bitstoreal 2-105
break CLI command 5-4
building OVA post-processor 14-25
byte data type 15-3

C

-C 3-70, 3-72
-c 3-70
C code generating
  halt before compiling the generated C code 3-72
  passing options to the compiler 3-71
  specifying 3-71
  specifying another compiler 3-71
  suppressing optimization for faster compilation 3-72
C compiler, environment variable specifying the 1-11
call PLI specification 7-9
cbk ACC capability 7-15, 7-19
cbka ACC capability 7-15
-CC 3-71
-cc 3-71
%CELL
  use of 8-35
‘celldefine 2-80
+cfgfile 3-50

-CFLAGS 3-71
char data type 15-3
+charge_decay 3-52
check argument to -ntb_opts 3-29
check PLI specification 7-9
CLI
  summary 5-3
+cli+level_number 1-14, 3-33
CLI commands, OVAPP 14-30
CLI task invocation 14-79
cli_0, OVAPP CLI command 14-30
+cliecho 4-10
+cliedit 3-34
clock signals 8-59–8-63
-cm 3-40, 4-9, 4-11, 15-97
-cm_assert_cov 15-99
-cm_assert_cov_cover 15-99
-cm_assert_dir 4-9, 15-97, 15-100
-cm_assert_grade_instances 15-100
-cm_assert_grade_modules 15-100
-cm_assert_hier 3-41, 15-97
-cm_assert_map 15-100
-cm_assert_merge 15-101
-cm_assert_name 15-97
-cm_assert_report 15-101
-cm_cond 3-41
-cm_constfile 3-42
-cm_dir 3-43, 4-12
-cm_fsmcfg 3-43
-cm_fsmopt allowTmp 3-43
-cm_fsmopt optimist 3-43
-cm_fsmopt report2StateFsms 3-43
-cm_fsmopt reportvalues 3-44
-cm_fsmopt reportWait 3-44
-cm_fsmopt reportXassign 3-44
-cm_fsmresetfilter 3-44
-cm_glitch 4-12
-cm_hier 3-44
-cm_ignorepragmas 3-44
-cm_libs 3-44
-cm_line contassign 3-45
-cm_log 4-12
-cm_name 3-45, 4-12
-cm_noconst 3-45
-cm_opfile 3-45
-cm_pp 3-45
-cm_scope 3-46
-cm_tglfile 3-46, 4-12
cnt.txp 14-4
compiler directives 2-80–2-89
compile-time options 3-21–3-81
compiling
  incremental compilation 3-2–3-6
    shared 3-4–3-6
    triggering 3-3
  verbose messages 3-66
concurrent assertions 15-42
Configuration file
  Command line, loading from 3-48
consecutive repetition 15-47
context-dependent pragmas 14-62
continue CLI command 5-4
control tasks, OVA debug 14-80
$countdrivers 2-117
cover OVA directive 14-3
cover statements 15-64–15-67
Coverage 14-41
coverage expressions 14-3
coverage, testing 14-6
-cpp 3-71
+csdf+precomp+dir 3-55, 8-30
+csdf+precomp+ext 3-55, 8-30
+csdf+precompile 3-54, 8-29

D

daidir directory 3-6

.daidir extension 3-6
data PLI specification 7-9
Data Type Mapping File
  VCS/SystemC cosimulation interface 16-22
-debug 3-46, 3-47
Debug Control Tasks,OVA 14-80
-debug_all 3-28, 3-46, 3-47
debugging
  capability 5-2
debugging, OVA 14-36
‘default_nettype 2-81
‘default_rswitch_strength 2-89
‘default_switch_strength 2-89
‘default_trireg_strength 2-89
‘define 2-81
define CLI command 5-4
+define+macro=value 1-15, 3-79
delay values
  back annotating to your design 2-117
+delay_mode_distributed 3-52, 8-22
‘delay_mode_distributed 2-84
+delay_mode_path 3-52, 8-22
‘delay_mode_path 2-84
+delay_mode_unit 3-52, 8-22
‘delay_mode_unit 2-84
+delay_mode_zero 3-52, 8-22
‘delay_mode_zero 2-84
delete CLI command 5-4
+deleteprotected 3-73, B-11
dep_check argument to -ntb_opts 3-29
$deposit 2-117
Direct Access Interface directory 3-6
directive, assert 14-3
directive, cover 14-3
directory
  .daidir 3-6
‘disable_portfaults 2-89
$disable_warnings 2-111
Disk space
  temporary 3-84
$display statement 14-78
$display 2-106
DISPLAY_VCS_HOME 1-11
$dist_exponential 2-115
$dist_normal 2-115
$dist_poisson 2-115
$dist_uniform 2-115
DKI Communication 16-13
DKI communication 16-3
do while statement 15-32
-doc 3-26
dump files 14-35
$dumpall 2-92
$dumpchange 2-93
$dumpfile 2-92
$dumpflush 2-92
dumping signals automatically 14-36
$dumplimit 2-92
$dumpoff 2-92
$dumpon 2-92
$dumpports 2-95
$dumpportsall 2-96
$dumpportsflush 2-96
$dumpportslimit 2-97
$dumpportsoff 2-95
$dumpportson 2-96
$dumpvars 2-93, 3-3

E

-e 14-44, 15-99
-e name_for_main 3-60, 7-34
-E program 4-23
edge-triggered flip-flops, using 2-16
‘else 2-81
‘elseif 2-82
$enable_warnings 2-111

enabling
  only where used in the last simulation 7-27
‘endcelldefine 2-81
ended method 15-53
‘endif 2-82
‘endprotect 2-85
‘endprotected 2-86
‘endrace 2-83
enum construct 15-5
enumerations 15-5
Environment variables 1-10–1-12
$error 2-90
error messaages, OVA linter option 14-9
error messaages, OVA MR linter option 14-16
Event coverage, testing 14-6
example
  temporal assertion file 14-4
‘expand_vectornets 2-89
expressions 14-4

F

-F filename 3-58
-f filename 1-15, 3-57
facilities, test 14-2
$fatal 2-90
$fclose 2-107
$fdisplay 2-107
$ferror 2-56, 2-107
$fflush 2-53, 2-93, 2-107
$fflushall 2-93
$fgetc 2-49, 2-107
$fgets 2-107
-file 3-58
file, report 14-40
Files
  tokens.v 3-74
  vcs.key 5-9
files, temporal assertion 14-3
$finish 2-111
finish CLI command 5-4
flow 14-7
flow of OVAPP 14-24
$fmonitor 2-107
$fopen 2-43, 2-107
force CLI command 5-4
frc ACC capability 7-15, 7-19
$fread 2-47, 2-108
$fscanf 2-45, 2-108
$fseek 2-55, 2-108
$fstobe 2-108
$ftell 2-53, 2-108
full_case 14-61, 14-62
Functional Coverage 14-41
functions
  void 15-41
functions and system tasks, OVA 14-71
functions in SystemVerilog 15-38
$fwrite 2-108

G

gate ACC capability 7-15
-gen_asm 3-69
-gen_c 3-71
-gen_obj 3-80
$getpattern 2-118
gmake 1-10
$gr_waves 2-94
-grw 4-18
-gui 3-47, 4-2, 4-15

H

-h 3-26
-help 3-26
help CLI command 5-4
$hold 2-112

I

-I 2-98, 3-49
-i 4-2, 4-16, 5-9
-i filename 3-38, 4-10
-ID 3-69
‘ifdef 2-82
‘ifndef 2-83
-ignore 3-27, 15-89
Ignoring Calls and License Checking 4-17
implications 15-59
implicit .* connections 15-79
implicit .name connections 15-79
+incdir 1-15, 3-21
‘include 2-87
Incremental Compilation 3-2–3-6, 3-23–3-26
$info 2-90
info CLI command 5-4
-ignore 3-27, 15-89
inlining
  context-dependent pragmas 14-62
inout ports 15-17
input ports
  valid data types 15-77
int data type 15-3
Interactive Debugging
  example 5-8
Interfaces 15-81–15-88
interfaces
  functions in 15-87
  methods 15-87
  modports 15-85
introducing OpenVera Assertions 14-2
IOPATH entries in SDF files 8-35
$itor 2-105

J

-jnumber_of_CPUs 3-25, 3-71

K

-k 4-2, 4-10, 4-16, 5-9

L

-l 4-2, 4-16
-l filename 1-15, 1-19, 3-79, 4-14, 4-33
-ld linker 3-70
-LDFLAGS options 3-70
+libext 3-22
+liborder 3-22
+librescan 3-22
+libverbose 3-22, 3-65
-line 1-15, 3-33
‘line 2-88
line CLI command 5-4
linking 3-70
  linking a specified library to the executable 3-70
  linking by hand 3-70
  passing options to the linker 3-70
  specifying another linker 3-70
+lint 3-65
Lint, using 3-16–3-18
linter rules, OVA code checking 14-9
linter rules, OVA MR code checking 14-16
list CLI command 5-4
-lmc-hm 3-79
-lmc-swift 3-65, 10-15
-lmc-swift-template 3-65, 10-4
-lname 3-70
-load 3-61, 7-33
$log 2-105
log file, environment variable specifying the 1-11
log files
  specifying compilation log file 1-15, 3-79
  specifying with a system task 2-105
logic data type 15-3

longint data type 15-3
$lsi_dumpports 2-94, 2-121–2-126

M

Main Window
  example 11-4
  overview 11-2
make 1-10
make program 3-25
-Marchive 3-23
maxargs PLI specification 7-10
+maxdelays 3-53, 3-55, 4-20, 8-56, 8-57, 10-16, 10-21
-Mdefine 3-23
-Mdelete 3-23
-Mdir 3-4
-Mdirectory 3-24
+memcbk 3-78
+memopt 3-13, 3-77
memories
  sparse memory models 2-73
memory size limits 2-72
message, assertion failure 14-78
methods 15-87
-Mfilename 3-24
minargs PLI specification 7-10
-Minclude 3-24
+mindelays 3-53, 3-55, 4-20, 8-56, 8-57, 10-16, 10-21
mip ACC capability 7-15
mipb ACC capability 7-15
misc PLI specification 7-9
-Mldcmd 3-24
-Mlib 3-4, 3-24, 3-25
-Mloadlist 3-25
-Mmakefile 3-25
-Mmakeprogram 3-25
modports 15-85
module path delays
  disabling for an instance 8-58
  suppressing 1-16, 3-61
    in specific module instances 8-58
$monitor 2-106
Monitoring assertions 14-73
$monitoroff 2-106
$monitoron 2-106
mp ACC capability 7-15
-Mrelative_path 3-25
-Msrclist 3-26
multi-dimensional arrays 15-13
multiple OVAPP post-processing sessions 14-31
+multisource_int_delays 3-53, 3-54, 8-6, 8-35
-Mupdate 3-3, 3-23

N

native code generating
  specifying 3-80
+nbaopt 3-53
**NC 5-16
+neg_tchk 3-63, 9-12, 9-19
next CLI command 5-5
no_case 14-62
no_file_by_file_pp argument to -ntb_opts 3-29
+no_identifier 4-12
+no_notifier 3-62, 9-12
+no_pulse_msg 4-15
+no_tchk_msg 3-62, 4-13, 9-13
‘noaccelerate_ignored 2-89
+nocelldefinepli+1 3-67
nocelldefinepli PLI specification 7-10
+nocelldefinepli+0 3-67
+nocelldefinepli+2 3-67
‘noexpand_vectornets 2-89
+noinitnegedge 3-32, 13-27, 13-28
+nolibcell 3-67

$nolog 2-105
‘noportcoerce 2-15, 2-86
‘noremove_gatenames 2-89
‘noremove_netnames 2-89
+nospecify 1-16, 3-61, 9-13
‘nosuppress_faults 2-89
-notice 3-65
+notimingcheck 1-16, 1-19, 3-61, 4-13, 4-33, 9-13
‘nounconnected_drive 2-88
-ntb 3-28
+ntb_cache_dir 4-2
-ntb_cmp 3-28
+ntb_debug_on_error 4-3
-ntb_define 3-28
+ntb_enable_solver_trace_on_failure 4-3
+ntb_engable_solver_trace 4-3
+ntb_exit_on_error 4-3
-ntb_filext 3-28
-ntb_incdir 3-28
+ntb_load 4-3
-ntb_noshell 3-28
-ntb_opts 3-28
  print_deps 3-29
+ntb_random_seed 4-3
-ntb_sfname 3-30
-ntb_shell_only 3-30
-ntb_sname 3-30
+ntb_solver_mode 4-4
-ntb_spath 3-30
+ntb_stop_on_error 4-4
-ntb_vipext 3-30
-ntb_vl 3-30
+NTC2 3-63, 9-18

O

-o name 3-80
-O number 3-72
-O0 3-72
oformat% CLI command 5-5
+old_ntc 3-63
+oldsdf 3-55, 8-26
once CLI command 5-5
OpenVera Assertions
  benefits 14-2
  flow 14-7
  introduction 14-2
  overview 14-3
operating system commands, executing 2-104
+optconfigfile 3-31, 3-33, 6-2, 6-4
or operator 15-51
oring sequences 15-51
output ports
  valid data types 15-77
OVA cover directive 14-3
OVA debug control tasks 14-80
OVA linter 14-8
OVA post-processing 14-23
OVA post-processor
  building 14-25
  running 14-25
OVA system tasks and functions 14-71
OVA user action function 14-85
OVA, assert directive 14-3
OVA, see OpenVera Assertions
OVA, Verilog parameters in 14-65
-ova_cov 3-38, 4-6, 14-19, 14-23
-ova_cov_cover 14-45
-ova_cov_db 4-6, 14-23, 14-45
-ova_cov_events 3-39, 14-19, 14-45
-ova_cov_grade_instances 14-45
-ova_cov_grade_modules 14-45
-ova_cov_hier 3-39, 14-19
-ova_cov_map 14-46
-ova_cov_merge 14-46
-ova_cov_name 4-6, 14-23
-ova_cov_report 14-46

$ova_current_time 14-79
-ova_debug 3-39, 14-19
-ova_debug_vpd 3-39, 14-19
-ova_dir 14-20
-ova_enable_diag 3-39, 14-20
-ova_file 3-39, 14-20
-ova_filter 4-5, 14-22
-ova_filter_past 3-39, 14-20
$ova_global_time_unit 14-79
-ova_inline 3-39, 14-21
-ova_lint 3-39, 14-8
-ova_lint_magellan 3-39, 14-16
-ova_max_fail 4-6, 14-22
-ova_max_success 4-6, 14-22
-ova_name 4-5
-ova_quiet 4-4, 14-21
-ova_report 4-4, 14-21
$ova_severity_action 14-80
-ova_simend_max_fail 4-6, 14-22
$ova_start 14-81
$ova_start levels 14-80
$ova_start_time 14-79
$ova_stop 14-82
$ova_stop levels 14-79
-ova_success 4-6, 14-22
ova_trace_off assertion_hierarchical_name, OVAPP CLI command 14-31
ova_trace_off instance_hierarchical_name assertion_name time time, OVAPP CLI command 14-31
ova_trace_on assertion_hierarchical_name, OVAPP CLI command 14-31
ova_trace_on instance_hierarchical_name assertion_name time, OVAPP CLI command 14-31
-ova_verbose 4-5, 14-21
-ova 3-38, 14-19
$ovadumpoff 14-80
$ovadumpon 14-81
OVAPP CLI commands 14-30
OVAPP Flow 14-24
+overlap 3-63, 9-22
+override_model_delays 4-24, 10-16, 10-21
-override_timescale 3-76

P

-P pli.tab 3-60, 7-20
packed arrays 15-12
packed keyword 15-9
packed structure 15-9
parallel compilation 3-25, 3-71
parallel_case 14-62
parameter expansion, Verilog with OVA 14-66
-parameters 3-18, 3-75
+pathpulse 3-61
PATHPULSE$ specparam, enabling 3-61
$period 2-112
persistent PLI specification 7-10
-platform 3-80
PLI specifications
  args 7-10
  call 7-9
  check 7-9
  data 7-9
  maxargs 7-10
  minargs 7-10
  misc 7-9
  nocelldefinepli 7-10
  persistent 7-10
  size 7-9
PLI table file 3-56, 7-7–7-20
+pli_unprotected 3-73, B-12
pli.tab file 7-7–7-20
+plusarg_ignore 3-58
+plusarg_save 3-58
plusargs, checking for on the simv command line 2-116

+plus-options 4-24
port coercion 2-14, A-7
port collapsing 2-14, A-7
Port Mapping File
  VCS/SystemC cosimulation interface 16-21
‘portcoerce 2-86
post-processing OVA 14-23
post-processing sessions, multiple OVAPP 14-31
-PP 2-98, 3-47, 3-49
pp_fastforward time, OVAPP CLI command 14-30
pp_rewind time, OVAPP CLI command 14-31
pragmas 14-62
  for SystemVerilog assertions 15-95
print CLI command 5-5
print_deps argument to -ntb_opts 3-29
$printtimescale 2-110
priority case statements 15-31
priority if statements 15-30
+prof 3-56
properties 15-57–15-62
  formal arguments 15-58
  implications 15-59
  inverting 15-61
  past value 15-62
+protect file_suffix 3-73
‘protect 2-86
‘protected 2-86
prx ACC capability 7-15
+pulse_e/number 3-59, 8-6, 8-8, 8-9, 8-12, 8-17, 8-18
+pulse_int_e 3-59, 8-7, 8-8, 8-10, 8-12
+pulse_int_r 3-59, 8-7, 8-8, 8-10, 8-12
+pulse_on_detect 3-59, 8-19
+pulse_on_event 3-59, 8-18
+pulse_r/number 3-59, 8-6, 8-8, 8-9, 8-12, 8-17, 8-18
pulses
  filtering out narrow pulses 3-59
    and flag as error 3-59
+putprotect+target_dir 3-73, B-11
-pvalue 3-18, 3-75

Q

-q 3-65, 4-14
$q_add 2-114
$q_exam 2-114
$q_full 2-114
$q_initialize 2-114
$q_remove 2-114

R

-R 1-16, 3-38
r ACC capability 7-14, 7-18
+race 3-37
‘race 2-83
race conditions A-1
  avoiding 2-2–2-7
  continuous assignment evaluations 2-5
  in counting events 2-6
  in flip-flops 2-4
  setting a value twice at the same time 2-3
  time zero 2-6
  using and setting a value at the same time 2-2
+race=all 3-37, 5-22
+race_maxvecsize 3-37, 5-12
+racecd 3-37
+rad 2-43, 3-31, 6-1
$random 2-116, 2-120
$read_lib_saif 2-129
$read_rtl_saif 2-129
$readmemb 2-109
$readmemh 2-109
$realtime 2-115
$realtobits 2-105

recommendation messages, OVA linter option 14-9
recommendation messages, OVA MR linter option 14-16
$recovery 2-112
$recrem 2-112
release CLI command 5-5
‘remove_gatenames 2-89
‘remove_netnames 2-89
repetition
  consecutive 15-47
report file 14-40
$reset 2-116
$reset_count 2-116
$reset_value 2-116
‘resetall 2-81
resets, synchronous or asynchronous, using 2-16
resetting
  keeping track of the number of resets 2-116
  passing a value from before to after a reset 2-116
  resetting VCS to simulation time 0 2-116
$restart 2-118, 4-29
results 14-40, 14-41, 14-47
$rewind 2-55
-RI 3-47, 3-49
-RIG 3-48
-RPP 3-50
$rtoi 2-106
rules, OVA linter option 14-9, 14-16
running OVA post-processor 14-25
runtime libraries, environment variable specifying the 1-12
runtime options 4-2–4-25
rw ACC capability 7-15, 7-18

S

-S 3-69
-s 1-16, 3-38, 4-23
s ACC capability 7-15
$save 2-118
scalarizing vector nets in VCD files 12-2
scope CLI command 5-5
$sdf_annotate 2-117
+sdf_nocheck_celltype 3-55, 8-38
+sdfprotect B-9
+sdfprotect file_suffix 3-73
+sdfverbose 4-15
sequences 15-43–15-54
  anding 15-50
  conditions for 15-52
  end point 15-53
  formal arguments 15-45
  oring 15-51
  repetition 15-46
  specifying a clock 15-49
  specifying a range of clock ticks 15-45
  unconditionally extending 15-46
sequential devices
  inferring 2-18–2-22, 8-59–8-63
sequential devices, using 2-16
set CLI command 5-5
$set_gate_level_monitoring 2-128
$set_toggle_region 2-127
$setup 2-113
$setuphold 2-113
$sformat 2-48
shortint data type 15-3
show CLI command 5-5
signals, dumping automatically 14-36
Signed Arithmetic Extensions 2-29–2-43
+sim 3-48
simulation state
  saving 2-118
simv executable file 1-13
size PLI specification 7-9
$skew 2-113

slicing arrays 15-14
source CLI command 5-6
sparse memory models 2-73
specify blocks
  disabling for an instance 8-58
  suppressing 1-16, 3-61
    in specific module instances 8-58
specifying libraries on the link line 3-70
$sreadmemb 2-109
$sreadmemh 2-110
$sscanf 2-46
$stimen 2-115
$stop 2-111
$strobe 2-106
struct construct 15-8
structural drivers 15-17
structures 15-8
‘suppress_faults 2-89
-sv_pragma 3-27, 15-95
-sverilog 3-26, 15-88, 15-96
SWIFT SmartModels
  generating a template 3-65
$swrite 2-48
$sync$nor$plane 2-114
-sysc 3-76, 16-11
syscan utility 16-6–16-9
-syslib libs 3-70
$system 2-104
system tasks 2-90–2-120
  IEEE standard system tasks not implemented 2-118
  non-standard system tasks not implemented 2-119
  Verilog-XL system tasks not implemented 2-119
System tasks and functions, OVA 14-71
SystemC
  cosimulating with Verilog 16-1
SYSTEMC_OVERRIDE environment variable 16-28
$systemf 2-104
SystemVerilog assertions 15-41–15-74
SystemVerilog functions 15-38
SystemVerilog tasks 15-36
+systemverilogext 3-30

T

task invocation from CLI 14-79
tasks in SystemVerilog 15-36
tb_timescale argument to -ntb_opts 3-29
tbreak CLI command 5-6
tchk ACC capability 7-15
temporal assertion files 14-3
temporal assertions 14-4
temporal expressions 14-4
test facilities 14-2
$test$plusargs 2-116
testing event coverage 14-6
throughout operator 15-52
$time 2-115
$timeformat 2-110
-timescale 3-75
‘timescale 2-87
timing check system tasks
  disabling
    in specific module instances 8-58
timing check system tasks, disabling 1-16, 3-61
timing checks
  disabling for an instance 8-58
Timopt
  the timing optimizer 8-59–8-63
+timopt 8-59
TMPDIR 1-11
$toggle_report 2-128
$toggle_reset 2-128
$toggle_start 2-127
tokens.v file 3-74, 15-123
trace CLI command 5-6

transparent latches, using 2-16
+transport_int_delays 3-54, 8-7, 8-10, 8-12
+transport_path_delays 3-54, 8-7, 8-9, 8-12
+typdelays 3-53, 3-55, 4-21, 8-56, 8-57, 10-16, 10-21
typedef construct 15-4, 15-9

U

-u 3-79
-ucli 3-47, 4-2, 4-10, 4-16
unaccelerated
  definitions and declarations 2-17
  structural instance declarations 2-17
unalias CLI command 5-6
‘unconnected_drive 2-88
‘undef 2-83
undefine CLI command 5-6
$ungetc 2-50, 2-109
unions 15-8
unique case statements 15-30
unique if statements 15-29
uniquifying identifier codes in VCD files 12-3
unpacked arrays 15-12
unpacked structure 15-9
upper case characters, changing all identifiers to 3-79
upscope CLI command 5-7
use_sigprop 3-29
use_sigprop argument to -ntb_opts 3-29
-use_vpiobj 3-61, 7-32
‘uselib 2-87
User Action Function, OVA 14-85
user defined data types 15-4
utility, vcsplit 12-23

V

-V 3-66, 4-15
-v 3-21
+v2k 1-17, 3-76
$value$plusargs 4-31
variables
  writing to 15-16
+vc 3-62
vcat utility 12-13
-vcd filename 4-19
-vcd2vpd 3-51
+vcdfile 3-50
vcdiff utility 12-5
  syntax 12-6
vcdpost utility 12-2
  syntax 12-4
VCS
  predefined text macro 2-82
vcs command line 1-13
+vcs+boundscheck 3-20, 3-77
+vcs+dumpoff+t+ht 4-19
+vcs+dumpon+t+ht 4-19
+vcs+dumpvarsoff 4-19, 4-33
+vcs+finish 4-14, 4-30
+vcs+flush+all 4-22
+vcs+flush+dump 4-22
+vcs+flush+fopen 4-22
+vcs+flush+log 4-22
+vcs+grwavesoff 4-18
+vcs+ignorestop 4-24, 4-33
+vcs+initmem 3-31
+vcs+initreg 3-31
+vcs+learn+pli 1-20, 4-23, 7-25–7-29
+vcs+lic+vcsi 4-22
+vcs+lic+wait 3-69, 4-22
+vcs+mipd+noalias 4-24
+vcs+mipdexpand 3-56
+vcs+nostdout 4-15
+vcs+stop 4-13, 4-30
VCS/SystemC cosimulation interface 16-1–16-29

compiling for using 16-11, 16-17
supported port data types 16-5
VCS_CC 1-11
VCS_COM 1-11
VCS_LOG 1-11
‘vcs_mipdexpand 2-85
‘vcs_mipdnoexpand 2-85
VCS_RUNTIME 1-12
VCS_SWIFT_NOTES 1-12
+vcsi+lic+vcs 4-22
+vcsi+lic+wait 3-69
vcs.key file 5-9
vcsplit utility 12-23
-vera 1-17, 3-40
-vera_dbind 3-40
+vera_load 4-10
+vera_mload 4-10
vera_portname argument to -ntb_opts 3-29
verify 14-40, 14-41, 14-47
Verilog parameter expansion 14-66
Verilog parameters in OVA 14-65
+verilog1995ext 3-31
+verilog2001ext 3-31
violation windows
using multiple non-overlapping 9-22–9-27
VirSim 14-47
+vissymbols 3-57
vlogan utility 16-14–16-16
VMC 4-43
void functions 15-41
VPD
Command line options
Ignore $vcdplus calls in code 4-17
VPD files 2-97
-vpd2vcd 3-51
+vpdfile 3-49, 3-51
+vpdfileswitchsize 3-50
+vpi 3-60
+vslogfile 3-48, 3-50
+vslogfilesim 3-48
-Vt 3-66

W

+warn 2-121, 3-66
+warn2val 3-32, 13-28
$warning 2-90
warning messages, OVA linter option 14-9
warning messages, OVA MR linter option 14-16
waveform dump files 14-35
Waveform Window
overview 11-8, 11-9
$width 2-114
wrapper for VCS/SystemC cosimulation
instantiating in SystemC 16-16
instantiating in Verilog 16-9
SystemC in Verilog 16-6
Verilog in SystemC 16-14
$write 2-106
$writememb 2-110
$writememh 2-110

X

-Xman 3-73, B-13, B-15
-Xmangle 3-73
-Xnoman 3-74
-Xnomangle 3-74

Y

-y 1-17, 3-21