

VCS® / VCS® MX Coverage Metrics User Guide

Version C-2009.06
June 2009

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

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2009 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

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

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

Destination Control Statement

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

Disclaimer

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

Registered Trademarks (®)

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

Trademarks (™)

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

Service Marks (SM)

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

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

ARM and AMBA are registered trademarks of ARM Limited.

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

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

Contents

1. Common Operations

Operations When You Compile or Simulate	1-2
Using Coverage Metrics Files and Directories	1-2
Specifying Coverage for Libraries and Cells	1-6
Filtering Constants	1-6
Naming Intermediate Data Files	1-19
Specifying Coverage Metrics Log Files	1-20
Controlling the Scope of Coverage Compilation	1-21
Using Glitch Suppression	1-34
Collecting an Execution Count	1-35
Operations When You Post-process Results	1-38
Using Compressed Files With cmView	1-38
Using Merged Intermediate Data Files	1-38
Naming the Report Files	1-40
Specifying the Test Files cmView Reads in Batch Mode	1-41
Controlling the Scope of Coverage Reports and Displays	1-43
Omitting Coverage for Default Case Statements	1-43

Test Autograding in Batch Mode	1-44
Writing Summary Information at the Top of the Report File	1-50
Reporting the Worst Coverage First	1-50
Displaying Summary Information After Writing Reports	1-51
Generating a Summary Report File	1-52
Reporting What Test File Covered the Line or Condition	1-54
Adding Condition and FSM Coverage Data to Annotated Files .	1-55
Using the Tcl Command Interface for cmView	1-57
 Merging VCS Results for Regressions	1-57
Merging Additional Tests	1-64
 Merging VCS MX Results for Regressions	1-65
Mapping Subhierarchy Coverage Between Designs in Verilog	1-69
Using Multiple -cm_map Options	1-71
Mapping Subhierarchy Coverage in VHDL	1-74
Understanding Instance-Based Mapping	1-75
Examples of Instance-Based Mapping Syntax	1-78
 2. Line Coverage	
What Is Covered in Line Coverage?	2-80
Line Coverage for Verilog	2-80
Line Coverage for VHDL	2-88
Line Coverage Glitch Suppression	2-89
Limitation on Clocks	2-91
Creating Line Coverage Reports	2-93

Annotated Source Files	2-95
The cmView.long_I File	2-105
The cmView.long_Id File	2-112
The cmView.hier_I File.....	2-113
The cmView.mod_I File	2-114
The cmView.mod_Id File	2-115
The cmView.short_I File.....	2-116
The cmView.short_Id File.....	2-118
Viewing Line Coverage with the Coverage Metrics GUI	2-119
Manipulating the Summary Pane.....	2-126
Detailed View.....	2-127
The Annotated Window	2-129
Excluding Lines From Coverage Calculations	2-136
 3. Toggle Coverage	
Supported Data Types.....	3-148
Realtime Control of Toggle Coverage	3-149
Limiting Toggle Coverage to Ports Only	3-149
Limiting Toggle Coverage to Uncovered Signals.....	3-150
Autograding Limited Toggle Coverage.....	3-152
Merging Limited Toggle Coverage Data.....	3-152
Excluding and Including Signals in Toggle Coverage	3-153
Excluding a Signal in Toggle Coverage	3-153
Including a Signal in Toggle Coverage.....	3-154
Including Part-Selects and Bit-Selects.....	3-154

Using Wildcard Characters	3-155
Specifying SystemVerilog Structures and Unions	3-155
Toggle Coverage Glitch Suppression.....	3-156
 Using Pragmas to Limit Toggle Coverage	3-157
Toggle Coverage Reports	3-158
The cmView.long_t File	3-161
The cmView.hier_t File.....	3-164
The cmView.mod_t File	3-165
The cmView.mod_td File	3-166
The cmView.short_t File.....	3-167
The cmView.short_td File.....	3-168
Displaying Port Signal Direction in the Toggle Coverage Reports 3-169	
 Viewing Toggle Coverage With the Coverage Metrics GUI.....	3-172
The Summary Pane.....	3-178
Detailed View.....	3-179
The Annotated Toggle Coverage Window	3-180
 4. Condition Coverage	
What Is Condition Coverage?	4-198
Verilog Conditional Expressions Not Monitored.....	4-201
VHDL Conditional Expressions Not Monitored	4-202
Modifying Condition Coverage	4-203
Enabling Coverage for Event Controls.....	4-206
Enabling Condition Coverage for More Operators.....	4-208

Enabling Condition Coverage in For Loops	4-209
Enabling Condition Coverage in Tasks and Functions.	4-210
Enabling Condition Coverage in Port Connection Lists	4-211
Using Sensitized Multiple Condition Coverage Vectors	4-212
Using the -cm_cond allvectors Option	4-216
Using Multiple Condition Value Vectors With Constant Filtering	4-218
Specifying Condition SOP Coverage	4-219
Enabling Bitwise Reporting for Vectors	4-223
Converting Bitwise Operators	4-225
Enabling Coverage for All Possible Combinations of Vectors . .	4-226
Disabling Vector Conditions	4-229
Excluding Conditions and Vectors From Reports	4-231
Specifying Continuous Assignment Coverage	4-235
Excluded Subexpressions	4-237
Using Condition Coverage Glitch Suppression	4-238
 Condition Coverage Reports	4-240
The cmView.long_c File	4-244
The cmView.hier_c File	4-250
The cmView.mod_c File	4-251
The cmView.mod_cd File	4-252
The cmView.short_c File	4-252
The cmView.short_cd File	4-254
 Viewing Condition Coverage with the Coverage Metrics GUI	4-255
Manipulating the Summary Pane.	4-261
Detailed View	4-262
The Annotated Window	4-263

5. FSM Coverage

Finite State Machines — FSMs	5-272
Verilog FSMs	5-273
VHDL FSMs	5-275
The Purpose of FSM Coverage	5-276
Coding a Verilog FSM	5-278
Using the Encoded FSM Style	5-278
Implementing Hot Bit or One Hot FSMs	5-286
Using Continuous Assignments for FSMs	5-289
Avoiding Substituting the Same Numeric Constant	5-290
Coding a VHDL FSM	5-290
Sequence Coverage	5-297
Controlling How VCS and VCS MX Extract FSMs	5-297
Using a Configuration File	5-298
Using Pragmas	5-306
Using Optimistic Extraction	5-311
Reporting FSM State Values Instead of Named States	5-314
Enabling Indirect Assignment to State Variables	5-315
Enabling Two-state FSMs	5-316
Enabling the Monitoring of Self Looping FSMs	5-317
Enabling X Value States	5-319
Filtering Out Transitions Caused by Specified Signals	5-321
FSM Coverage Reports	5-323
The cmView.long_f File	5-324
The cmView.long_fd File	5-330

The cmView.hier_f File	5-331
The cmView.mod_f File	5-332
The cmView.mod_fd File	5-333
The cmView.short_f File	5-334
The cmView.short_fd File	5-337
Excluding Sequences From Reports	5-341
 Viewing FSM Coverage With the Coverage Metrics GUI	5-341
 6. Path Coverage	
Path Coverage Example	6-364
Generating Path Coverage	6-366
Path Coverage Reports	6-366
The cmView.long_p File	6-368
The cmView.hier_p File	6-373
The cmView.mod_p File	6-374
The cmView.mod_pd File	6-374
The cmView.short_p File	6-375
The cmView.short_pd File	6-376
 7. Branch Coverage	
Branch Coverage Example	7-380
Branch Coverage With Unknown and High Impedance Values	7-381
Enabling Branch Coverage	7-382
For Loops and User-defined Tasks and Functions	7-382
Branch Coverage Reports	7-383

Example Branch Coverage Report	7-384
Using Pragmas to Limit Branch Coverage	7-391
Viewing Branch Coverage with the Coverage Metrics GUI.....	7-400
Excluding Branches from Coverage Calculation	7-411
Branch Coverage Limitations	7-416
8. Using the Graphical User Interface	
cmView Command-line Options	8-420
Common Operations in cmView Windows	8-427
Hiding or Showing the Tool and Status Bars	8-427
Controlling the Display of Multiple Windows in cmView.....	8-428
Displaying the Log of Error and Warning Messages	8-429
The Hierarchy Pane.....	8-430
The Summary Pane.....	8-431
The Result Files Menu.....	8-434
Opening a Design File	8-435
Adding the Results of a Test to the Total Results.....	8-437
Viewing the Incremental Coverage of an Added Test	8-441
Viewing the Differential Coverage Between Two Tests	8-441
Creating Reports	8-443
The View Menu	8-446
The Action Menu	8-448
The Options Menu	8-451
Test Grading	8-452
Using the Automatic Grading Window.....	8-459

User Preferences	8-461
The Colors Tab	8-461
The Command Tab	8-464
The Log Tab	8-465
The Weights Tab	8-466
9. Real Time Coverage	
The \$cm_coverage System Function	9-472
The \$cm_get_coverage System Function	9-476
The \$cm_get_limit System Function	9-478
Examples	9-479
Index	

1

Common Operations

There are a number of operations that affect more than one type of coverage or all types of coverage. There are two types of common operations you follow when you:

- Compile or simulate your design
- Post-process your results with cmView

The chapter includes the following topics:

- “[Operations When You Compile or Simulate](#)”
- “[Operations When You Post-process Results](#)”
- “[Merging VCS Results for Regressions](#)”
- “[Merging VCS MX Results for Regressions](#)”
- “[Mapping Subhierarchy Coverage Between Designs in Verilog](#)”

Note:

The last three topics are especially important to your understanding of coverage metrics in VCS and VCS MX.

Operations When You Compile or Simulate

There are many operations related to generating coverage metrics that you can perform while compiling or simulating your design. These include specifying coverage for libraries and cells, naming intermediate data files, using glitch suppression, and so on. You can also control the scope of the coverage compilation using either pragmas or a configuration file.

Using Coverage Metrics Files and Directories

If you use the `-cm` compile-time option, VCS and VCS MX creates the `simv.cm` directory (the coverage metrics database) in the current directory.

VCS and VCS MX both write the `db/verilog`, `db/vhdl`, `reports`, `coverage/verilog` and `coverage/vhdl` directories in the `simv.cm` directory. They also create the annotated directory in the `reports` directory.

During compilation, VCS and VCS MX write data files about the design in the `db/verilog` directory (for Verilog) or `db/vhdl` (for VHDL). They use these data files during simulation.

During simulation, VCS and VCS MX write intermediate data files (also called test files) that record the coverage results from the simulation in the `coverage/verilog` directory (for Verilog) or the

coverage/vhdl directory (for VHDL). These files have the default name, test, with various extensions for each type of coverage, for example, test.line and test.fsm.

You can instruct VCS or VCS MX to use a different name for the intermediate data files, see “[Naming Intermediate Data Files](#)”. You might want to do so if you plan multiple simulations with different stimuli and you want to see the coverage results from the different simulations.

Also, during simulation, VCS and VCS MX write the `cm.decl_info` file in either the `simv.cm/db/verilog` directory (for Verilog) or the `simv.cm/db/vhdl` directory (for VHDL). The `cmView` command needs this file to show coverage information.

The `cmView` command writes its reports in the `simv.cm/reports` directory.

The files in the `simv.cm/db/verilog` and `simv.cm/coverage/verilog` directories are compatible with all supported platforms. For example, you can have VCS compile and monitor for coverage using the Solaris platform, and therefore, write files in these directories using Solaris. You can then have `cmView` read these files to display coverage information or write reports using the Linux platform. See the release notes (available through the online HTML documentation system) for a list of supported platforms.

The `simv.cm` directory is named after the `simv` executable file. If you name the simulation executable file something else with the `-o` compile-time option, VCS/VCS MX names the coverage metrics database directory after the name you assigned to the simulation executable. For example, if you name the executable, `mysimv`, as follows:

```
vcs source.v -cm fsm -o mysimv
```

VCS creates the `mysimv.cm` directory instead of the `simv.cm` directory. The `mysimv.cm` directory will still contain the `db`, `reports`, and `coverage` directories.

You can also use the `-cm_dir directory_path_name` compile-time option and argument to specify a different name and location for the `simv.cm` directory. For example:

```
vcs -cm tgl -cm_dir /net/design1/mycm source.v
```

This command line instructs VCS to create the `mycm` directory in the `/net/design1` directory instead of the `simv.cm` directory in the current directory. The `mycm` directory will still contain the `db`, `reports`, and `coverage` directories. The `-cm_dir` compile-time option takes precedence over the `-o` compile-time option when specifying the new name for the `simv.cm` directory.

When you use the `-cm_dir` option at compile time, the name and location of this directory is hard coded into the `simv` executable, so there is no need to use the option at runtime to specify this directory.

However, this alternative name and location is not hard coded into the `cmView` executable, because `cmView` is not created at compile-time. Therefore, you need to use the `-cm_dir` option to give `cmView` the name and location of the coverage metrics database.

Using the `-cm_dir` option at runtime specifies a different location for the intermediate data files (also called test files) that VCS writes at runtime. The default location of these test files is the `coverage` subdirectory in the coverage metrics database.

If you specify a different location for the intermediate data files at runtime, you must enter the `-cm_dir` option on the `cmView` (or `vcs -cm_pp`) command line to give `cmView` location of the test files. If you have also specified an alternate name or location for the coverage metrics database, enter another `-cm_dir` option to specify its location.

To summarize when to use `-cm_dir` and for what purpose:

- At compile-time, to specify a different name or location for the coverage metrics database directory (`simv.cm`) that is created during compilation.
- If you move the coverage database directory or the binary executable to a new location after VCS creates it during compilation, use the `-cm_dir` option at runtime, to specify the path for the coverage database directory .
- If you invoke your binary executable from a different location, then use the `-cm_dir` option at runtime to specify the path for the coverage database directory.
- On the `cmView` command line, you can specify:
 - The location of the coverage metrics database (the location you had specified at compile-time)
 - The location of the intermediate data files (the location you had specified at runtime)

This means that you can use the `-cm_dir` option two times on the `cmView` command line.

The following command lines illustrate these different possibilities:

```
vcs -cm line -cm_dir /net/design1/my_cov_info source.v
```

```
simv -cm line -cm_dir /net/design1/int_dat_files  
vcs -cm_pp -cm line -cm_dir /net/design1/my_cov_info -cm_dir  
/net/design1/int_dat_files
```

Specifying Coverage for Libraries and Cells

By default, VCS does not compile the following for coverage:

- The source code in Verilog library directories
- Verilog library files
- Any module defined under the `celldefine` compiler directive

If you want this excluded code compiled for coverage, include the `-cm_libs` compile-time option along with one or both of the following arguments:

- `yv` - for compiling for coverage source code from Verilog libraries.
- `celldefine` - for compiling coverage modules defined under the `celldefine` compiler directive.

If you want to specify both arguments, include the plus delimiter (+) between the two arguments. For example:

```
vcs source.v -v mylib.v -y /net/libs/teamlib -cm fsm  
-cm_libs yv+celldefine
```

Filtering Constants

Coverage statistics can sometimes be distorted if you use certain nets as flags to deliberately stop the execution of certain lines or conditions. Using this technique to prevent the execution of these

lines in certain circumstances can give you the simulation results you want, but also lowers your line, condition, and toggle coverage percentages.

To prevent this lowering of coverage percentages, use the `-cm_noconst` compile-time option.

Use this option to instruct VCS to perform the following:

1. Recognize when a net is permanently at a constant 1 or 0 value.
2. Determine what lines and conditions will not be covered because the net is permanently at a constant 1 or 0 value.
3. Not monitor these nets, lines, or conditions during simulation so that the coverage amount is not distorted.

Note:

Constant filtering for toggle coverage is available only for Verilog-only designs.

In [Example 1-1](#) there are conditions that will never be met, lines that will never execute, and nets that will never toggle between true and false.

Example 1-1 Lines and Conditions That Can Never Be Covered

```
1 module test;
2 reg A,B,C,D,E;           net F is at 0 throughout the simulation
3 wire F;
4
5 assign F = 1'b0;
6
7 always @(A or B)
8 begin
9   C = (A && B && D);
10  E = (C || F) so F will never be true
11 if (!D)
12   C = 0;
13 else
14   C = (A || B);
15 if (!F)
16   E = 0;
17 else
18   E = 1; and this line will never execute
19 end
20
21 endmodule
```

In this example, VCS does not monitor when net F toggles or is true as a subexpression of the logical or operator || in line 10. It also does not monitor for the execution of line 18 because the condition in line 10 will never be covered, and line 18 will never be executed.

Note:

The lines, which VCS does not monitor for line coverage when you use the `-cm_noconst` option, are blocks of code controlled by `if` and `case` statements.

There are a number of ways that a net can be permanently at 1 or 0, but VCS can only recognize this in the following cases:

- When a 0 or 1 value is assigned to the net in a continuous assignment statement. For example:

```
assign sig1 = 1'b0;
```

- When a 0 or 1 value is assigned to the net in the net declaration. For example:

```
wire sig1 = 1'b0;
```

- When it is a supply0 or supply1 net.
- When VCS can determine through analysis of the code that a signal will always have a 1 or 0 value. For example:

```
assign sig1 = 1'b0;
assign sig2 = sig3 && sig1;
```

A net can be permanently at values higher than 1 in a continuous assignment statement. For example:

```
assign G = 2;
```

However, VCS does not omit monitoring for condition, line, or toggle coverage when a net is permanently at a value higher than 1.

There are a number of ways you can assign the values 0 or 1 in a continuous assignment statement. Here are a few examples:

```
assign G = 1;
assign H = 0;
assign I = 3'b0;
assign J = 1'b1;
```

No matter how you assign these 0 or 1 values, when you use the `-cm_noconst` option, VCS will not monitor for the pertinent conditions or lines.

Constant filtering does not just pertain to scalar nets. If a continuous assignment assigns a 1 or 0 value to a vector net, and you use the `-cm_noconst` option, VCS does not monitor for the condition or line coverage that cannot be covered because these vector nets have a 1 or 0 value.

If a vector net is continuously assigned a 0 or 1 value, the least significant bit is assigned either 1 or 0. All the other bits are assigned 0 and remain at 0 permanently. This means that a bit-select or part-select of a vector net can be permanently at a value and the `-cm_noconst` option can instruct VCS not to monitor for certain conditions or line execution based on the permanent values of the bit-select or part-select. For example:

```
wire [7:0] w1;
.
.
.
assign w1 = 1;

always @ (sig1
begin
if (w1[0] && sig1)           VCS does not monitor for when w1[0] is 0
.
.
.
sig2 = (w1[1] && sig1)       VCS does not monitor for when w1[1] is 1
```

If a net is continuously assigned an expression whose operands are constants or nets that are themselves continuously assigned constant values, VCS does not automatically omit monitoring for condition and line coverage. For example:

```
assign w1 = 0;           ← w2 is permanently at 0 throughout the simulation
assign w2 = ( w1 && 1);

always @ (sig1
r1 = (w2 && sig1);

but VCS monitors for when w2 is both 1 and 0
```

If there is more than one continuous assignment statement of the 1 or 0 value to a net, VCS does not omit monitoring for conditions or lines specified or controlled by the value of that net. This is true even when the multiple continuous assignments are assigned to different bits of a vector net. For example:

```
assign w1 = 1;           ← multiple assignments to the same net
assign w1 = 0;
assign w2 [1] = 0;       ← assignments to different bits
assign w2 [0] = 1;

always @ (r1
begin
r2 = (w1 && r1);
r3 = (w2 [0] && r1);
end

VCS monitors for when w2[0] is both 1 and 0
```

Treating Other Signals as Permanently at the 1 or 0 Value

You can also tell VCS or VCS MX to treat specified signals, nets and variables, as permanently at 1 or 0 constant values by specifying the list of signals in `const_file` as an argument to the `-cm_constfile` option. VCS or VCS MX will not monitor the specified signals, nets, or variables for toggle coverage (Verilog only), or for conditions that cannot be covered and lines that won't be executed. VCS or VCS MX treats these signals as permanently 1 or 0 even if they change value during the simulation.

When compiled with the `-cm_constfile` option, VCS or VCS MX treats the following as constants:

- Signals specified in `const_file`
- Constants extracted from a continuous assignment
- `supply0` or `supply1`

For example:

Example 1-2 Using the cm_constfile Option

```
module top();
  wire [1:0] a;
  wire [7:0] b;
  wire [4:0] c;
  reg d;
  assign b = {a, d, 5'b11x01};
endmodule
```

In the `const_file`, you have:

```
top.a 1'b10
```

then VCS or VCS MX evaluates the value of "b" as 8'b10x1101.

Note:

The value of the constant "a" defined, in the `const_file` is considered to evaluate the wire "b".

These constants remain local to the instance and do not propagate across the hierarchy through ports.

However, you can use the `-cm_noconst` compile-time option to propagate the local constants across the hierarchy.

const_file Syntax

You can specify a constant value to a signal, net, or variable in the following two ways:

- Specify a constant value
- Write an expression - VCS or VCS MX evaluates the expression, and assigns the obtained value to the specified constant.

Using the `const_file`, you can:

- Include single-line or multi-line comments.
- Specify a list of signals, nets, or variables and their values.

The syntax to write a `const_file`, is as follows:

- Single line comments should start with " // ". You can also comment a block, by enclosing it within "/*" and "*/".
- Specify a list of signals, nets, or variables, and their corresponding constant values.

For example:

```
top.t1.a 1
top.t1.b 4'b1010
```

You can also specify more than one signal in the same line separated by a space, as follows:

```
top.t1.a 1 top.t1.b 4'b1010
```

You can also specify signed constants, as shown in the following example:

```
top.t2.a 4'sb1111
```

In case of signed constants, the sign bit occurs if the width of the LHS signal specified in the `const_file` is greater than the width of the constant value specified.

For example:

Using wire `c`, shown in [Example 1-2](#), the following assignment in the `const_file`:

```
top.c 1'sb1
```

is equivalent to `top.c 5'b11111`.

- You can specify the value of a signal, net, or variable in the following formats:
 - Decimal
 - Binary
 - Octal
 - Hexadecimal
- Use the "U" character to concatenate two or more part selects of the same signal, as follows:

```
top.c[0:3] 4'b1111  
top.c[5:7] 3'b101
```

The above declaration, can also be written as:

```
top.c 4'b101U1111
```

If the representation is in binary format, "U" acts as a single bit. For Octal and Hexadecimal formats, "U" acts as 3 bits and 4 bits, respectively.

For example:

```
top.d 9'o3U4  
top.e 12'h5U8A
```

VCS or VCS MX expands above declarations as:

```
top.d 9'b011UUU0100  
top.e 12'b0101UUUU10001010
```

Instead of a constant value, you can also specify an expression as follows:

```
top.t1.f {4'b1010, 4'b0101} | 8'h5A
```

VCS or VCS MX evaluates the above expression and assigns the resultant to the signal, top.t1.f.

Note:

If the specified expression contains "U", VCS or VCS MX replaces "U" with "x" as in Verilog during evaluation, and the evaluation follows Verilog semantics.

The following table lists the supported operators:

Table 1-1 Supported Operators

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	Two
	/	divide	Two
	+	add	Two
	-	subtract	Two
	%	modulus	Two
	**	power (exponent)	Two
Logical	!	logical negation	One
	&&	logical and	Two
		logical or	Two
Relational	>	greater than	Two
	<	less than	Two
	>=	greater than or equal	Two
	<=	less than or equal	Two
	==	equality	Two
Equality	!=	inequality	Two
	==>	case equality	Two
	!=>	case inequality	Two
	In constfile	case equality and normal equality behave the same way	
Bitwise	~	bitwise negation	One
	&	bitwise and	Two
		bitwise or	Two
	^	bitwise xor	Two
	^~ or ~^	bitwise xnor	Two
Reduction	&	reduction and	One
	~&	reduction nand	One
		reduction or	One
	~	reduction nor	One

Table 1-1 Supported Operators (Continued)

	<code>^</code>	reduction xor	One
	<code>^~ or ~^</code>	reduction xnor	One
Shift	<code>>></code>	Right shift	Two
	<code><<</code>	Left shift	Two
	<code>>>></code>	Arithmetic Right Shift	Two
	<code><<<</code>	Arithmetic Left Shift	Two
Concatenation	<code>{ }</code>	Concatenation	Any number
Replication	<code>{ {} }</code>	Replication	Any number
Conditional	<code>?:</code>	Conditional	Three
Parentheses	<code>()</code>	Change Precedence	--

The constfile syntax is:

```

constfile_statement_declarations ::= 
    constfile_statement_declarations constfile_statement
    |
    ;
constfile_statement ::= 
    identifier constant_expression
identifier ::= 
    hierarchical_path_of_signal
constant_expression::=
    [ sign ] decimal_number
    | [ sign ] octal_number
    | [ sign ] binary_number
    | [ sign ] hex_number
    | expression_with_operator
expression_with_operator ::= 
    '(' constant_expression ')'
    | constant_expression BIN_OP constant_expression
    | '{' list_of_constant_expression '}'
    | '{' constant_expression '{' constant_expression '}' '}'
    | UNI_OP constant_expression
    |
    | constant_expression '?' constant_expression ':'
constant_expression

```

```

list_of_constant_expression ::=

    constant_expression
    | list_of_constant_expression , constant_expression

BIN_OP ::= * | / | + | - | % | ** | && | || | < | > | <= | >=
    == | != | === | !== | | | & | ^ | ~^ | ^~ | << |
    >> | <<< | >>>

UNI_OP ::= ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~

decimal_number ::=

    unsigned_number | [ size ] decimal_base unsigned_number

binary_number ::= [ size ] binary_base binary_value

octal_number ::= [ size ] octab1_base octal_value

hex_number ::= [ size ] hex_base hex_value

sign ::= +|-

size ::= non_zero_unsigned_number

non_zero_unsigned_number* ::=

    non_zero_decimal_digit { _ | decimal_digit }

unsigned_number* ::= decimal_digit { _ | decimal_digit }

binary_value* ::= binary_digit { _ | binary_digit }

octal_value* ::= octal_digit { _ | octal_digit }

hex_value* ::= hex_digit { _ | hex_digit }

decimal_base* ::= '[s|S]d' | '[s|S]D'

binary_base* ::= '[s|S]b' | '[s|S]B'

octal_base* ::= '[s|S]o' | '[s|S]O'

hex_base* ::= '[s|S]h' | '[s|S]H'

non_zero_decimal_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

decimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

binary_digit ::= U | 0 | 1

octal_digit ::= U | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hex_digit ::= U | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
            | a | b | c | d | e | f | A | B | C | D | E | F

```

Note:

Embedded spaces are illegal.

See “[Using Multiple Condition Value Vectors With Constant Filtering](#)” on page 4-218 to see how this option affects the contents of condition coverage reports.

Naming Intermediate Data Files

By default, when VCS monitors for any type of coverage, it records the results in intermediate data files named “test” with various extensions, as follows:

- `test.line` for line coverage
- `test_fsm` for FSM coverage
- `test_cond` for condition coverage
- `test_tgl` for toggle coverage
- `test_path` for path coverage
- `test_branch` for branch coverage

By default, VCS writes these files in the `./simv.cm/coverage/verilog` directory. The `cmView` command reads these files to show coverage results.

You might want to have a different name for these intermediate data files. For example, you might want to apply different stimuli in different simulations and have intermediate data files with different names so that you can compare coverage results using `cmView`. To specify a different name for the intermediate data files, you can perform either of the following:

- Include the `-cm_name` option at compile-time. For example:

```
vcs source.v -cm line -cm_name test1
```

This command line compiles into the executable the name that the executable will use when it writes the intermediate data files. After simulation, the `simv.cm/coverage/verilog` directory contains the `test1.line` file.

- Include the `-cm_name` option at runtime, for example:

```
simv -cm line -cm_name test2
```

This command line instructs the executable to name the intermediate data file, `test2.line`. Using this option at runtime overrides any name for the intermediate data file that might have been compiled into the executable by using this option at compile time.

You cannot use this option to specify a different location for these files.

Specifying Coverage Metrics Log Files

During simulation, you can use the `-cm_log` runtime option to have VCS write a log file about how it monitors for coverage. For example:

```
simv -cm fsm -cm_log run1.log
```

You can also compile the name of the log file into the executable by including the `-cm_log` option at compile time. The `-cm_log` runtime option overrides any name for the log file that was compiled in at compile time.

You can also use the `-cm_log` option on the `cmView` command line (or `vcs -cm_pp`) to specify a log file for writing reports. By default, the `cmView` log file is `cmView.log`.

Controlling the Scope of Coverage Compilation

You do not need to compile the entire design hierarchy for code coverage. Instead, you can limit the scope of coverage metrics by doing either of the following:

- Enter the `-cm_hier` compile-time option and its configuration file to specify the module definitions, instances and subhierarchies, and source files that you want VCS either to exclude from coverage or exclusively compile for coverage (see “[Using a Configuration File](#)”).
- Enter pragmas in your Verilog source code to exclude lines, source files, and module instances from coverage (see “[Using Pragmas to Exclude Verilog Lines, Signals, Source Files, and Module Instances from Coverage](#)”).
- Enter pragmas in your VHDL source code to exclude lines from coverage (see “[Using Pragmas to Exclude VHDL Lines From Coverage](#)”).

Using a Configuration File

The `-cm_hier` compile-time option takes a configuration file argument. You can use this configuration file to specify the parts of the design that you want VCS or VCS MX to:

- Compile exclusively for coverage
- Exclude from coverage

Note:

The `-cm_hier` compile-time option can also be used on the `cmView` command line (or `vcs -cm_pp`), where it controls the scope of what `cmView` displays or reports.

Coverage metrics also has another type of configuration file for specifying FSMs in a design for FSM coverage.

Statements in the configuration file begin with a plus (+) or minus (-) sign. If a statement begins with a plus sign (+), it specifies what VCS or VCS MX should compile for coverage. If a statement begins with a minus sign (-), it specifies what VCS or VCS MX should not compile for coverage.

You can use the following statements in the configuration file (however, some of these statement are supported only at compile time and are not supported in the file you submit to `cmView` with the `-cm_hier` command-line option.):

`+tree instance_name [level_number]`

VCS and VCS MX compile only the specified instance and the instances under it for coverage. These instances can be Verilog module or VHDL entity instances. VCS and VCS MX exclude all other instances from coverage.

A level number of 0 (or no level number) specifies the entire subhierarchy, 1 specifies only this instance, 2 specifies this instance and those instances directly under this instance, 3 specifies this instance and instances in the subhierarchies that are one and two levels below the specified instance. There is no limit to the integer you specify as the level number.

If the subhierarchy includes instances in the other HDL, VCS MX does not include these instances.

-tree *instance_name* [*level_number*]

VCS and VCS MX exclude this instance from coverage and other instances under it. These instances can be Verilog module or VHDL entity instances. VCS and VCS MX include all other instances in coverage.

A level number of 0 (or no level number) specifies the entire subhierarchy, 1 specifies only this instance, 2 specifies this instance and those instances directly under this instance, and so on.

If the subhierarchy includes instances in the other HDL, VCS MX does not exclude these instances.

+module *module_name* / *entity_name*

VCS and VCS MX compiles all instances of the specified Verilog module or VHDL entity definition, and excludes all other definitions under it, for coverage.

-module *module_name* / *entity_name*

VCS and VCS MX does not compile all instances of the specified Verilog module or VHDL entity definition, and includes all other definitions under it, for coverage.

+file *file_name*

VCS and VCS MX compile for coverage only the code in this file. If the file is not in the current directory, specify the path name of the file.

-file *file_name*

VCS and VCS MX exclude the code in this file from coverage. If the file is not in the current directory, specify the path name of the file.

+filelist *file_name*

VCS and VCS MX compile for coverage only the source files listed in the specified file.

-filelist *file_name*

VCS and VCS MX exclude from coverage the source files listed in the specified file.

Note:

The following statements (+library and -library) are not supported in the file you submit to cmView with the -cm_hier command-line option. However, they are supported in the configuration file you use at compile time.

+library *library_name*

VCS MX compiles for coverage all the Verilog module and VHDL entities in the specified library. VCS MX excludes all other Verilog modules and VHDL entities from coverage.

-library *library_name*

VCS MX excludes from coverage all the Verilog modules and VHDL entities in the specified library. VCS MX includes all other Verilog modules and VHDL entities in coverage.

+modulmtree *module_name* [*level_number*]

VCS and VCS MX provide coverage metrics for the specified module and for all self-instances of the module in the hierarchy below the specified module. VCS and VCS MX provide coverage metrics for the number of hierarchy levels specified in the +modulmtree statement.

This functionality is available only for Verilog.

```
-modulmtree module_name [level_number]
```

VCS and VCS MX exclude coverage metrics for the specified module and for all self-instances of the module in the hierarchy below the specified module. VCS and VCS MX exclude coverage metrics for the number of hierarchy levels specified in the `-modulmtree` statement.

This functionality is available only for Verilog.

You can also specify metric-specific inclusions or exclusions with the following configuration file syntax.

```
begin metric[+metric...] +modulmtree module_name
    [level] end
VCS and VCS MX apply inclusions only for the metric (or metrics) and hierarchy level(s) specified.
```

The following metrics are valid:

```
line | fsm | tgl | branch | cond | path
```

```
begin metric[+metric...] -modulmtree module_name
    [level] end
VCS and VCS MX apply exclusions only for the metric (or metrics) and hierarchy level(s) specified.
```

The following metrics are valid:

```
line | fsm | tgl | branch | cond | path
```

You can use the * , ? and + wildcard characters in the configuration file to specify definitions, instances, VHDL libraries, and file names.

You can add comments to this file using the convention for Verilog comments:

Type	Used for...
// comment	The end of the line comment.
code /* comment */ code	The inside of a line comment.
/* comment 1 comment 2 comment3 */	Multiple line comments.

Begin - End Blocks for Coverage Type Control

The keyword arguments to the `-cm` compile-time option (`line`, `cond`, `fsm`, `tgl`, and `path`) specify the type of coverage for which VCS and VCS MX compile. These arguments specify compiling the entire design for that type of coverage. With begin-end blocks in the `-cm_hier` file, you can specify what parts of the design VCS and VCS MX compile for each type of coverage.

Note:

When you use this feature, the argument to the `-cm` compile-time option enables the compilation of the type of coverage and the entries in the `-cm_hier` file begin-end blocks specify where VCS or VCS MX compiles for that type of coverage.

Begin-end blocks are not supported in the file passed to `cmView` by the `-cm_hier` command-line option.

Lines in the `-cm_hier` file that are not begin-end blocks must come before any begin-end block lines.

The following are examples of the usage of begin-end blocks in the `-cm_hier` file:

Example 1

```
begin fsm+line +tree TOP.D1 end
```

This line specifies that VCS and VCS MX compile only the instances in the specified subhierarchy (`TOP.D1`) for FSM coverage. This line only specifies where VCS and VCS MX compile for FSM coverage. If other types of coverage are specified with other `-cm` compile-time option arguments, VCS and VCS MX compile the entire design for those types of coverage.

Note:

VHDL hierarchical names can only contain uppercase letters. This requirement is not true for Verilog.

Example 2

```
begin line+tgl+cond -module bdev end
```

This line specifies that VCS and VCS MX do not compile the instances of the module or design entity named `bdev` for line, toggle, and condition coverage. To specify more than one type of coverage, use the plus (+) delimiter.

Example 3

```
begin cond+line -library gate end
begin tgl -tree TOP.D1.L* end
```

The first begin-end block calls for excluding all instances of the entities in the VHDL library named `gate` from condition and line coverage. The second begin-end block calls for excluding all subhierarchies where the top instance has a hierarchical name beginning with `TOP.D1.L` (some of these uppercase letters could be lowercase in the VHDL source code) from toggle coverage.

Example 4

```
+tree tb.dut1  
begin line -module bdev end
```

The first line does not specify a type of coverage. It calls for VCS or VCS MX to compile the instances in the subhierarchy with the top instance `tb.dut` for the types of coverage specified with the `-cm` compile-time option (the hierarchical name has lowercase letters so it must be a Verilog instance).

The second line is a begin-end block that tells VCS and VCS MX not to compile all instances of the definition named `bdev` for line coverage.

Using Pragmas to Exclude Verilog Lines, Signals, Source Files, and Module Instances from Coverage

Pragmas are metacomments in your source code.

VCS and VCS MX have pragmas that allow you to specify that certain lines of Verilog code, Verilog source files, and all instances of certain module definitions are not to be compiled for line, toggle, condition, FSM, or branch coverage. They work like compiler directives (see “[Using Pragmas to Limit Toggle Coverage](#)” on page [3-157](#) and “[Viewing Branch Coverage with the Coverage Metrics GUI](#)” on page [7-400](#) for more details).

These pragmas are as follows:

Pragma	Description
<code>//VCS coverage off</code>	Specifies disabling coverage for the block of source code that follows.

//VCS coverage on	Specifies re-enabling coverage for the source code that follows when a previous block of source code is disabled by the //VCS coverage off pragma.
//VCS coverage exclude_file	Specifies disabling coverage for the code in the source file that contains this pragma. Synopsys recommends entering this pragma at the beginning of the source file, before any module definitions.
//VCS coverage exclude_module	Specifies disabling coverage for all instances of the module definition that contains this pragma. Synopsys recommends entering this pragma immediately after the port declarations. This pragma does not exclude from coverage the module instances hierarchically under these instances. To do this, see " Controlling the Scope of Coverage Compilation ".

The HDL Compiler and Behavioral Compiler user can use the //synopsys translate_off directive in place of the //VCS coverage off pragma and the //synopsys translate_on directive in place of the //VCS coverage on pragma.

The //VCS coverage on pragma enables line coverage after a //synopsys translate_off directive and a //synopsys translate_off directive disables line coverage after a //VCS coverage on pragma.

Similarly, the //VCS coverage off pragma disables line coverage after a //synopsys translate_on directive and a //synopsys translate_on directive enables line coverage after a //VCS coverage off pragma.

When you disable line coverage using pragmas, cmView indicates this fact in its line coverage report. For example:

```
// Statements ignored due to pragmas : 1
// Warning: 1 pragma ignored statement(s) were covered
```

In the above example, cmView reports that in a section on a module instance, a statement in the instance was excluded from coverage by pragmas. This statement did execute during simulation, but it is not listed in the report.

Pragmas do not exclude module instances. For example:

```
module test;
reg clk, a;
// Synopsys translate_off
mod1 inst1(a,clk);
// Synopsys translate_on
.
.
.
endmodule
```

This example does not exclude `test.inst1` from coverage.

Pragma pairs for disabling and re-enabling coverage cannot be used across module boundaries or across blocks of code (such as begin-end or always blocks). [Figure 1-1](#) illustrates this point.

Figure 1-1 Using Pragmas

```

always @(posedge CLK)
begin
    casex (Control_state_next)
        //VCS coverage on
        3'b0: PC = PCmux;
        //VCS coverage off
        3'b001: begin
            //VCS coverage on
            IR = Iin;
            end
        3'b010: begin
            RS_ALU = RSbus;
            //VCS coverage off
            T_ALU = RTbus;
            end
        3'b011: begin
            MAR = ALUout;
            SMDR = RTbus;
            end
        //VCS coverage on
    default: begin
        RS_ALU = RSbus;
        SMDR = ALUout;
        end
    endcase
end

always @(Control_state_pres)
begin
    casex (Control_state_pres)
        3'b100: Wrt_en = 1'b1;
        default: Wrt_en = 1'b0;
    endcase
end
//VCS coverage off

```

The first pair is good

This pair straddles begin-end boundaries

This pair straddles always block boundaries

Using Pragmas to Exclude VHDL Lines From Coverage

Pragmas are metacomments in your source code. VCS MX has pragmas that instruct VCS MX not to compile certain lines of VHDL sequential statements or concurrent signal assignments for line and condition coverage. They also instruct VCS MX not to compile certain signal declarations for toggle coverage. These pragmas are as follows:

`--synopsys coverage_off`

Specifies disabling line and condition coverage for the sequential statements or concurrent signal assignments that follow, for line and condition coverage. Specifies disabling toggle coverage for the signal declarations that follow.

`--synopsys coverage_on`

Specifies re-enabling line and condition coverage for the sequential statements or concurrent signal assignments that follow and also specifies re-enabling toggle coverage for the signal declarations that follow, when a previous block of source code is disabled by the `-- synopsys coverage off` pragma.

The following examples use these pragmas to disable and re-enable line and condition coverage in sequential code:

Example 1

```
architecture A of E is
.
.
.
begin
p : process
begin
--vhdlcoveroff
```

```

sig1 <= 0; -- not compiled for coverage
--vhdlcoveron
if (x == '0') then
  s <= y;
end if;
end process p;
end A;

```

VCS MX does not compile the signal assignment statement to signal `sig1` for line coverage.

Example 2

```

package body mypack is
procedure vec_assn(
  signal out: bit_vector(3 downto 0);
  x : bit_vector(3 downto 0)
) is
begin
  --vhdlcoveroff
  if (x == "000") then      -- not compiled for coverage
    sequential statements -- not compiled for coverage
  .
  .
  .
  end if;
  --vhdlcoveron
  out <= x;
end vec_assn;
.
.
.

```

VCS MX does not compile the sequential statements controlled by the `if` statement in procedure `vec_assn` for line or condition coverage.

Example 3

You can also use these pragmas around signal declarations to disable and re-enable toggle coverage. For example:

```
architecture A of E is
  signal clk : bit;
  --vhdlcoveroff
  signal flag1 : integer;    --not compiled for toggle coverage
  --vhdlcoveron
  signal flag2 : integer;    --compiled for toggle coverage
begin
  .
  .
  .
end A;
```

VCS MX does not compile signal `flag1` for toggle coverage; it does compile signal `flag2` for toggle coverage.

Using Glitch Suppression

Signals often go through many oscillations before they settle down for an event. Such transitions on a signal to a value for a short period of time, or even transitioning to a value and then another value during the same simulation time step, can cause transitions or toggles on other signals and cause some lines to execute, and perhaps execute more than once, during the same time step.

These oscillations, glitches, or narrow pulses can skew the coverage results in line, condition or toggle coverage, by showing lines, conditions, or toggles that were covered only when these glitches occur and not after these oscillating signals settle down.

To prevent this, you can use the `-cm_glitch` compile-time option as follows:

```
vcs -cm line+cond+tgl -cm_glitch period
```

When you use this option at compile-time, during simulation, VCS ignores value changes and line executions that occur because there was a transition on a signal to a value for a period of time shorter than the specified period.

Specify the period with a non-negative integer. The simulation time that you specify is similar to the simulation time you specify in a delay specification (#10 for example). It is scaled based on the `time_unit` and `time_precision` arguments to the `'timescale` compiler directive.

Glitch suppression works slightly differently in line, condition, and toggle coverage (see “[Line Coverage Glitch Suppression](#)” on page [2-89](#), “[Using Condition Coverage Glitch Suppression](#)” on page [4-238](#), and “[Toggle Coverage Glitch Suppression](#)” on page [3-156](#) for details).

The `-cm_glitch` option is also a runtime option, but only works for toggle coverage.

Collecting an Execution Count

You can have cmView add the following to its report:

- In toggle coverage, not just whether a signal toggled from 0 to 1 and 1 to 0, but also the number of times it toggled.
- In FSM coverage, not just whether an FSM reached a state, had such a transition from one state to another, 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.

You enable cmView to add this information to its reports with the `-cm_count` compile-time option.

Note:

You can use the `-cm_tgl count` option at compile time, which enables `-cm_count` functionality (but only for toggle coverage).

Example 1-3 Section From a cmView.long_l Line Coverage Report

```
MODULE TB
FILE /u/SCRATCH/my_home/my_verilog.v
Line No      Hits      Block Type
11           12       INITIAL
12-16         12
17           12       IF
18           3        ELSE
19           0        IF
21           3        ELSE
22           3
24           1
24.1          1       WAIT
```

The `-cm_count` compile-time option specifies that the report will have a `Hits` column showing the number of times a line was executed. If you did not include this option, the report will have a `Coverage` column instead, which contains 1 for executed lines and 0 for unexecuted lines.

Example 1-4 Section From a cmView.long_t Toggle Coverage Report

```
MODULE top.fsm2_1
// Net Coverage
// Name   Toggled   1->0   0->1   ToggleCount
//      Yes      Yes     Yes      1
// Register Coverage
// Name   Toggled   1->0   0->1   ToggleCount
//      current[0] Yes     Yes     Yes      2
```

current [1]	Yes	Yes	Yes	2
next [0]	Yes	Yes	Yes	1
next [1]	Yes	Yes	Yes	1

Without the `-cm_count` compile-time option there would be no ToggleCount column.

Example 1-5 Section From a cmView.long_f FSM Coverage Report

FILE	expl.v	
FSM	current_state	
// state coverage results		
	start	2
	step1	2
	step2	1
// state transition coverage results		
	start->step1	2
	step1->step2	1
	step2->start	1
// sequence coverage results		
	start->step1	Covered
	step1->step2	Covered
	step2->start	Covered
	start->step1->step2	Covered
	step1->step2->start	Covered
	step2->start->step1	Covered
	start->step1->step2->start	Covered Loop
	step1->step2->start->step1	Covered Loop
	step2->start->step1->step2	Not Covered Loop

To the right of each state and transition between states, is the number of times the FSM achieved the state or transition. Without the `-cm_count` compile-time option, the report would simply say Covered or Not Covered, like it does for sequences.

Operations When You Post-process Results

There are many operations related to coverage metrics that you can perform while post-processing your design. These include naming the report files, specifying the test files cmView reads in batch mode, generating a summary report file, controlling the scope of coverage reports and displays and so on.

Using Compressed Files With cmView

To save disk space, you can compress the files in the `db/verilog` and `coverage/verilog` directories using gzip. After compression, the old files are removed by gzip and the compressed files have the `.gz` filename extension. For example, `test.line.gz` and `cm.decl_info.gz`.

If `gunzip` is set in the `$path` environment variable, cmView automatically decompresses the files when it needs to read them in order to display coverage results or write coverage reports. If `gunzip` is not set in the `$path` environment variable, cmView displays an error message when it tries to decompress the files.

If cmView finds both compressed and decompressed versions of the files in the `db/verilog` and `coverage/verilog` directories, it displays a warning message and reads the decompressed files.

Using Merged Intermediate Data Files

When you run cmView in batch mode, by default, it creates the `simv.cm/reports` directory. In this directory, cmView writes files named, `cmView.ext`, with different file name extensions for the

different types of coverage. For example, `cmView.line` and `cmView.tgl`. These files contain the merged results for each type of coverage. For example, if the `simv.cm/coverage/verilog` directory contained the following files: `test1.line`, `test2.line`, `test1.tgl`, and `test2.tgl`, the `cmView.line` file would contain the data recorded in the `test1.line` and `test2.line` files and the `cmView.tgl` file would contain the data recorded in the `test1.tgl` and `test2.tgl` files.

After `cmView` writes the `cmView` files, you can delete all the other intermediate data files in the `simv.cm/coverage/verilog` directory. The `cmView` command is still able to display coverage information and write reports from the data in the merged intermediate data files. Deleting these other files could save considerable disk space and `cmView` will work faster because it has fewer files to read.

You can use the `-cm_name` option to specify a different directory that `cmView` creates for the `cmView` files and a different name for these files. Remember that this option also names the report files. Therefore, for example if you only compiled or monitored for line coverage and entered the following command line:

```
vcs -cm_pp -cm_name mydir/suite1
```

`cmView` would write the reports for line coverage in the `mydir` directory and name them all `suite1` with various extensions for different types of line coverage reports. In this example, `cmView` would also create a directory named, `suite1`, in the `mydir` directory. In the `suite1` directory `cmView` would write the `suite1.line` file that contains all the merged line coverage data.

Once cmView has written the `suite1.line` merged intermediate data file for line coverage, you can remove the intermediate data files for line coverage from the `simv.cm/coverage/verilog` directory and just use the `suite1.line` file the next time you run cmView. By default, the cmView command looks for the files in the `simv.cm/coverage/verilog` directory so you must inform cmView of the location of the merged intermediate data files using the `-cm_dir` option. For example:

```
vcs -cm_pp -cm_dir mydir/suite1
```

In this example, cmView writes its reports in the default location, the `simv.cm/reports` directory.

Naming the Report Files

You can also use the `-cm_name` option on the cmView (or `vcs -cm_pp`) command line to specify the name and location of the report files. For example:

```
vcs -cm_pp -cm_name /net/design1/reports/test1_1
```

The `./net/design1/reports` directory contains the following files:

- `test1_1.long_1`
- `test1_1.short_1`
- `test1_1.hier_1`
- `test1_1.mod_1`

Specifying the Test Files cmView Reads in Batch Mode

By default, cmView reads all the test files (also called intermediate data files) in the `./simv.cm/coverage/verilog` (or `vhdl`) directory or all the test files in the `/coverage/verilog` (or `vhdl`) subdirectory in an alternative coverage metrics database specified by the `-cm_dir` option.

You can control what test files cmView reads in batch mode with the `-cm_tests` cmView (or `vcs -cm_pp`) command-line option. For example:

```
vcs -cm_pp -cm_tests file_name
```

You specify the test files with entries in the specified file. Make each entry on a separate line. An entry can be one of the following:

- A file name without the file name extension. If so, cmView reads all the test files for each type of coverage with the specified name in the `simv.cm/coverage/verilog` directory or the directory specified with the `-cm_dir` option.
- A path name for a file without the file name extension. If so, cmView reads all the test files for each type of coverage with the specified name in the specified directory.
- A file name with the file name extension. If so, cmView reads the specified test files in the `simv.cm/coverage/verilog` directory or the directory specified with the `-cm_dir` option.
- A path name for a file with the file name extension. If so, cmView reads the specified test file in the specified directory.

You can include the question mark ? and asterisk * single character and multi-character wild card characters in entries in the file.

[Example 1-6](#) illustrates the use of this option.

Example 1-6 Reading Test Files in Batch Mode

Consider the following command line:

```
vcs -cm_pp -cm_dir /CovTests/dir1 -cm_dir  
/CovTests/dir2 -cm_tests tests.list
```

The content of the `tests.list` file is as follows:

```
test1  
test2  
/CovTests/mytests/test0?  
/CovTests/spectest/spectest.line
```

In this case, cmView reads the following test files:

- All test files for all types of coverage with the name `test1` or `test2` in the `/CovTests/dir1` and `CovTests/dir2` directories
- All test files for all types of coverage in the `/CovTests/mytests` directory with a name that begins with `test0` followed by one additional character
- The `/CovTests/spectest/spectest.line` test file

The `-cm_tests` option is also useful for merging data from additional test files into the report files (see “[Merging Additional Tests](#)”).

Controlling the Scope of Coverage Reports and Displays

The `cmView` (or `vcs -cm_pp`) `-cm_hier` command-line option controls the scope of what `cmView` reports or displays. This option is also a compile-time option. At compile-time, it limits the parts of the design that VCS compiles for coverage; on the `cmView` command line it limits the parts of the design that `cmView` reports on or displays.

Just like the compile-time option, when you use `-cm_hier` on the `cmView` command line, you specify a configuration file as an argument to this option. You specify the part or parts you want included or excluded from the display or reports in this configuration file. The syntax of this configuration file is described in the section, “[Using a Configuration File](#)”.

Omitting Coverage for Default Case Statements

Typical usage of default case items in case statements is to tell VCS what to do if the simulation does something that you do not expect, that is, if it does not find a case item expression that matches the case expression. For example:

```
case (r1)
1'b0 : r2=0;
1'b1 : r2=1;
default : begin
            if (f1 || f2)
                $display("one flag true");
            else
                $display("no flag true");
            $stop;
        end
    endcase
```

In this example, the default case statement displays information about the current state of the design and then halts simulation for debugging. It is not part of how the design works.

If you use default case items in this way, you might not want cmView to display or report line or condition coverage information about default case statements when they are not executed. You do not want to see a lower percentage of coverage just because VCS did not execute the default case statement.

In this case, you can tell cmView not to display or report line or condition coverage information when default case statements are not executed with the `-cm_nocasedef` option on the cmView (or `vcs -cm_pp`) command line. For example:

```
vcs -cm_pp -cm_nocasedef
```

Note:

This option does not keep cmView from displaying or reporting toggle or FSM coverage information about default case statements. It also does not keep cmView from displaying or reporting line or condition coverage information when VCS executes default case statements.

Test Autograding in Batch Mode

Another term for an intermediate data file is a test file or simply a test. Test grading is listing all tests that cmView has found and reporting on how much coverage each test provides and how much each test contributes to the total amount provided by all tests.

Note:

Test autograding is only implemented for line, condition, FSM, and toggle coverage.

You perform test grading using the test grading window in the cmView graphical user interface. Using this window you can instruct cmView to write a test grading report. Test autograding is examining a number of tests to determine the fewest number of tests that will give you a specified percentage amount of coverage. Test autograding can be done using the cmView graphical user interface, but you can also run cmView in batch mode to write a test autograding report.

You use the `-cm_autograde percentage` option and argument to the cmView (or vcs -cm_pp) command-line option to run test autograding in cmView batch mode and have cmView write the test autograding report. The reports are named `cmView.grd_c`, `cmView.grd_f`, `cmView.grd_f` and `cmView_grd_t` (depending on the type of coverage). cmView writes these files in the `simv.cm/reports` directory.

When you use this option, cmView writes no other reports.

Note:

Test autograding using the graphical user interface is done with the **Tools > Autograding** menu command which displays the cmAutoGrade dialog box. This dialog box has a field for processing time. Allowing additional processing time can improve the accuracy of the results. When you do test autograding in batch mode, there is no way to tell cmView to use more processing time.

cmView does test autograding for line, FSM, condition, and toggle coverage.

cmView can only perform batch mode test autograding for one type of coverage at a time, therefore, do not include more than one of the cond, fsm, tgl, and line arguments to the -cm option. The following are valid cmView (or vcs -cm_pp) command lines for test autograding:

vcs -cm_pp -cm line -cm_autograde 100

Specifies test autograding for 100% line coverage

vcs -cm_pp -cm cond -cm_autograde 100

Specifies test autograding for 100% condition coverage

vcs -cm_pp -cm fsm -cm_autograde 100

Specifies test autograding for 100% FSM coverage

vcs -cm_pp -cm tgl -cm_autograde 100

Specifies test autograding for 100% toggle coverage

The following is a scenario where you would use test autograding: you are running a large number of simulations to obtain a large amount of coverage. You want to continue to have a large amount of coverage but with fewer simulations. You can do this by determining which tests contribute the least to your overall coverage and which tests you can use to dispense. To determine this using test autograding, perform the following:

1. Run the simulations over again, but use the -cm_name runtime option specifying a different name with each simulation. For example:

```
simv -cm line -cm_name test0
simv -cm line -cm_name test1
simv -cm line -cm_name test2
simv -cm line -cm_name test3
simv -cm line -cm_name test4
```

```
simv -cm line -cm_name test5
```

The simv.cm/coverage/verilog directory now has test or intermediate data files from all the simulations. For example, for line coverage it now contains test0.line, test1.line, test2.line, test3.line, test4.line and test5.line. Each of these files exists from different simulations.

2. Now, run test autograding in cmView batch mode to write a test autograding report. For line coverage, enter the following:

```
vcs -cm_pp -cm line -cm_autograde 100
```

This command line instructs cmView to run test autograding in batch mode to look for the least number of tests that will result in 100% line coverage.

cmView writes the cmView.grd file in the simv.cm/reports directory (see “[The Test Autograding Reports](#)” for an example of a cmView.grd file). In this example, the results of test autograding shows that only two of the five tests are required to provide 100% line coverage.

Limiting the Processing Time

Batch mode test autograding can take a long time to process your test files. You might want cmView to limit its processing time to a few hours. Limiting processing time often provides satisfactory results, such as a report showing that only some of the test files you have are needed for the amount of coverage you want.

To limit the processing time, include the -cm_timelimit compile-time option with the maximum number of minutes you want cmView to use before writing the report. For example:

```
vcs -cm_pp -cm cond -cm_autograde 100 -cm_timelimit 240
```

Specifies test autograding for 100% condition coverage using no more than four hours.

The Test Autograding Reports

cmView writes the `cmView.grd_c`, `cmView.grd_f`, `cmView.grd_f` and `cmView_grd_t` (depending on the type of coverage) test autograding report files when you add the `-cm_autograde` option on the cmView command line (see “[Test Autograding in Batch Mode](#)”).

The `cmView.grd_I` File

```
// Synopsys, Inc.  
//  
// Generated by: cmView 6.1S  
// User: smart_user  
// Date: Thu Mar  7 20:06:28 2002  
  
// STATEMENT COVERAGE TEST GRADING  
  
Test No.    Test Name          Coverage File Name  
0           test0             simv.cm/coverage/verilog/test0  
1           test1             simv.cm/coverage/verilog/test1  
2           test2             simv.cm/coverage/verilog/test2  
3           test3             simv.cm/coverage/verilog/test3  
4           test4             simv.cm/coverage/verilog/test4  
5           test5             simv.cm/coverage/verilog/test5  
  
// Test Grading Criteria  
  
Module/Instance Coverage: Instance  
Type of Coverage: Statement  
Coverage Goal: 100  
  
// Test Grading Data  
  
// Listed in the descending order of coverage  
Test NoIncrementalDifferenceCoveredAccumulatedTest Name  
5      58.33 58.33 58.33 58.33 test5
```

```

4    41.67 91.67 50.00 100.00 test4
0    0.00  58.33 41.67 100.00 test0
1    0.00  66.67 33.33 100.00 test1
2    0.00  75.00 25.00 100.00 test2
3    0.00  83.33 16.67 100.00 test3

// Listed in the Optimal order for Coverage goal

Test No. Incremental Difference Covered Accumulated Test Name

5    58.33 58.33 58.33 58.33 test5
4    41.67 91.67 50.00 100.00 test4
0    0.00  58.33 41.67 100.00 test0
3    0.00  83.33 16.67 100.00 test3
1    0.00  66.67 33.33 100.00 test1
2    0.00  75.00 25.00 100.00 test2

```

This report includes the following columns:

Column	Description
Incremental data	The amount of additional coverage provided by a given test over the previous test.
Difference data	The difference in coverage between a given test and the accumulated total up to the previous total.
Covered data	The coverage for a given test. That is, the data reflects the total coverage for that test only.
Accumulated data	The cumulative result of all tests added in the current session up to that point in the order of tests.

In this example, we see that out of five tests, from five simulations, we only need two for 100% coverage.

Adding Simulation Time to the Autograding Report

You can add another column to the report, listing the simulation time used in the generation of the test (intermediate data file). This could tell you that a testbench that provides slightly higher coverage also uses significantly more simulation time than other testbenches.

Add this column by using the `-cm_report simtime` command-line option and keyword argument on the `cmView` (or `vcs -cm_pp`) command line.

Simulation time column



Test No.	IncrementalDifference	Covered	Accumulated	Test Name	simTime
0	50.050.0050.0050.00	test0			0
1	25.00 58.33 41.67 75.00	test1	10000		
2	16.67 75.00 33.33 91.67	test2	20000		
4	8.33 50.00 58.33 100.00	test4	40000		
5	0.00 41.67 58.33 100.00	test5	50000		
3	0.00 75.00 25.00 100.00	test3	300	00	

Writing Summary Information at the Top of the Report File

By default, `cmView` writes summary information for the entire design at the end of the report file. You can tell `cmView` to write this summary information at the beginning of the report file with the `-cm_report summary_on_top` command-line option and argument on the `cmView` (or `vcs -cm_pp`) command line.

Reporting the Worst Coverage First

`cmView` report files include a section for each module instance. By default, `cmView` writes these sections according to the design hierarchy beginning with the top-level module followed by the section for module instances in the top-level module and then down the design hierarchy to the leaf-level module instances.

You can instruct cmView to write report sections for module instances in the order of how bad the coverage was, with the section containing the instance with the worst coverage listed first in the report file. To do this, use the `-cm_report worst_first` command-line option and argument on the cmView (or vcs `-cm_pp`) command line.

Note:

The `-cm_report` option can take multiple arguments separated by plus characters (+). For example:

```
-cm_report annotate+summary_on_top+worst_first
```

Displaying Summary Information After Writing Reports

You can use the `-cm_verbose` command-line option to instruct cmView to display summary information about coverage on the screen while it is writing reports. This option takes either a 1 or 2 argument. For example:

```
vcs -cm_pp -cm_verbose 2
```

The 2 argument displays the summary information for each test file. The 1 argument displays just the Total Coverage summary.

The following is an example of this summary information when you include the 2 argument:

```
----- SOURCE COVERAGE SUMMARIES -----
./simv.cm/coverage/verilog/test1 : 296 blocks covered out of 510 total
(58.0% coverage)

./simv.cm/coverage/verilog/test4 : 247 blocks covered out of 510 total
(48.4% coverage)

./simv.cm/coverage/verilog/test6 : 354 blocks covered out of 510 total
(69.4% coverage)
```

```

----- CONDITION COVERAGE SUMMARIES -----
./simv.cm/coverage/verilog/test1 : 195 vectors out of 593 total (32.9%
coverage)

./simv.cm/coverage/verilog/test4 : 181 vectors out of 593 total (30.5%
coverage)

./simv.cm/coverage/verilog/test6 : 231 vectors out of 593 total (39.0%
coverage)

----- FSM      COVERAGE      SUMMARIES -----
./simv.cm/coverage/verilog/test1 : 28 states out of 43 total (65.1% coverage)
./simv.cm/coverage/verilog/test4 : 21 states out of 43 total (48.8% coverage)
./simv.cm/coverage/verilog/test6 : 32 states out of 43 total (74.4% coverage)

----- TOGGLE COVERAGE SUMMARIES -----
./simv.cm/coverage/verilog/test1 : 205 bits covered out of 977 total
(21.0% coverage)

./simv.cm/coverage/verilog/test4 : 210 bits covered out of 977 total
(21.5% coverage)

./simv.cm/coverage/verilog/test6 : 404 bits covered out of 977 total
(41.4% coverage)

-----
Total Coverage Summary:

Source: 354 blocks covered out of 510 total (69.4% coverage)
Toggle: 404 bits toggled out of 977 total (41.4% coverage)
Cond: 231 vectors covered out of 593 total (39.0% coverage)
Fsm: 32 states covered out of 43 total (74.4% coverage)
-----
```

Generating a Summary Report File

You can instruct cmView to write a summary file of the coverage information in its reports. You enable writing the summary file with the `-cm_report summary` command-line option and argument.

Note:

In VCS MX, do not use this feature with a VHDL-top design.

The default name of the summary file is `cmView.summary`. You can specify a different name, but not extension, with the `-cm_name` option, just like you do for all cmView report files.

The data in the file is in a wide format, so it is best printed in landscape mode. The following is an example summary file. Shown in this example, is the left side of the report followed by the right side.

module/instance name	line			conditional		
	!cov	tot	%	!cov	tot	%
* TOTAL_MODULEDEF	0	41	100.00	5	6	16.67
* top	0	9	100.00	--	--	--
* intrctr	0	12	100.00	5	6	16.67
* gzmo	0	6	100.00	--	--	--
* spdev	0	14	100.00	--	--	--

toggle !cov	state			transition			sequences				
	!cov	tot	%	!cov	tot	%	!cov	tot	%		
3	21	85.71	0	6	100.00	0	6	100.00	2	18	90.00
0	1	100.00	--	--	--	--	--	--	--	--	--
3	6	50.00	--	--	--	--	--	--	--	--	--
0	5	100.00	--	--	--	--	--	--	--	--	--
0	9	100.00	0	6	100.00	0	6	100.00	2	18	90.00

By default, the information in the report is for the module definitions in the design. You can instruct cmView to include module instance information in the summary file by including the `-cm_summary` instance command-line option and argument. In the above example, the information on left side of module `intrctr` would be as follows:

module/instance name	line			conditional		
	!cov	tot	%	!cov	tot	%
* intrctr	0	12	100.00	5	6	16.67
top.intr 1	0	12	100.00	5	6	16.67

If a module name is longer than 29 characters, an instance name is longer than 31 characters, or a value is larger than 9999999, so that it takes too much room, the remaining data in the row appears on the following line. For example:

+-----+-----+-----+
| module/instance | line | conditional |

name	!cov	tot	%	!cov	tot	%
* intrctrctr_with_a_very_long_name	0	12	100.00	5	6	16.67
top.intr_1	0	99999999	100.00	5	6	16.67

After cmView writes the summary file, it sends you an e-mail containing the contents of the file. You can instruct cmView not to do this with the `-cm_summary noemail` command-line option and argument.

You can specify both keyword arguments to the `-cm_summary` option with a plus delimiter (+). For example:

```
-cm_summary instance+noemail
```

The default subject of the e-mail message is:

`coverage_results_summary:full-path of summary file`

You can specify another subject with the `-cm_mailsub string` command-line option and argument. The character string you specify must be a continuous string with no breaks. Using quotation marks makes no difference.

Reporting What Test File Covered the Line or Condition

When you have multiple test files (intermediate results files) from multiple simulations, cmView merges the coverage results so that if something is covered in one, but not all the test files, cmView reports it as covered.

In line coverage, you can have cmView indicate in the `cmView.long_l` and `cmView.long_ld` files, which test covered which line. In condition coverage, you can have cmView indicate in its `cmView.long_c` and `cmView.long_cd` report files, which tests covered which condition.

You instruct cmView to do this with the `-cm_report testlists` command-line option and keyword argument. For example:

```
vcs -cm_pp -cm line+cond -cm_report testlists
```

See “[Reporting the Test That Caused Line Coverage](#)” on page 2-111 for an example of what cmView adds to the `cmView.long_l` file. See “[Reporting the Test That Caused Condition Coverage](#)” on page [4-248](#) for an example of what cmView adds to the `cmView.long_c` file.

By default, cmView only does this for the first three test files that it finds by alphanumeric order of the test file names. You can instruct cmView to do this for more or fewer test files by replacing the `-cm_report testlists` cmView command-line option and keyword argument with the `-cm_report_testlists_max x` command-line option and integer argument. For example:

```
vcs -cm_pp -cm line -cm_report_testlists_max 4
```

Adding Condition and FSM Coverage Data to Annotated Files

When you include the `-cm_report annotate` cmView (or `vcs -cm_pp`) command-line option and keyword argument, cmView writes annotated files in the `.simv.cm/reports/annotated`

directory (see “[Annotated Source Files](#)” on page 2-95). These files contain line coverage information. There is a file for every module instance and module definition.

If you also include the `-cm_annotate type_of_coverage cmView` (or `vcs -cm_pp`) command-line option and keyword argument, with the `-cm_report annotate cmView` command-line option and keyword argument, the annotated files also contain condition and/or FSM coverage information similar to the contents of the `cmView.long_f` and `cmView.long_c` files.

The keyword arguments to the `-cm_annotate` option are as follows:

`cond`

Specifies adding condition coverage information to the annotated files.

`fsm`

Specifies adding FSM coverage information to the annotated files.

You can use a plus (+) delimiter to specify adding both kinds of coverage information to the annotated files. For example:

```
vcs -cm_pp -cm line+cond+fsm -cm_report annotate  
-cm_annotate fsm+cond
```

or

```
cmView -b -cm line+cond+fsm -cm_report annotate -cm_annotate  
fsm+cond
```

Using the Tcl Command Interface for cmView

cmView has a Tcl command interface that allows you to create your own reports. The Tcl commands are categorized as follows:

Command Type	Commands that instruct cmView to...
Database	Perform operations on the coverage data such as read test files, map coverage from one design to another, and write reports.
Design	Report information about the design such as the number of source files, module definitions, and module instances.
Statement Coverage	Report on statement coverage such as the total number of statements and the total number of statements covered.
Condition Coverage	Report on condition coverage such as the total number of conditions and expressions in the design.
FSM Coverage	Report on FSM coverage such as the total number of FSMs states and state transitions in these FSMs.
Toggle Coverage	Report on toggle coverage such as the total number of nets and registers (in IEEE Std 1364-2001 called variables) in the design and the total number of nets and registers covered.

For more information about Tcl commands, see the chapter entitled [Tcl Commands](#).

Merging VCS Results for Regressions

cmView automatically merges the results from the intermediate data files that VCS writes during more than one simulation of your design. This section describes four methods of running regressions to produce merged coverage results. In most of these methods, you

only need to compile for coverage once and then run multiple simulations, either sequentially or in parallel. You can use either different names for intermediate data files in the same directory or create different directories for these intermediate data files and then inform cmView where to look for them.

Method 1: Build the simv executable once, then simulate three times sequentially using the same testbench, but different inputs.

There are a number of ways to use the same testbench in different simulations with different input stimulus values. One common way is to use the \$readmemb system task to read values from a file into the elements of a memory. Then for each simulation, you change the contents of the data file specified in the system task. VCS reads this data at runtime and you make no changes to the system task. Therefore, you only need to compile for coverage one time.

The values in the memory elements control the execution of your design and exercise different parts of the design during the different simulations. When the simulations are over, you can see the total and merged coverage from all three simulations.

This procedure is as follows:

1. Compile for coverage:

```
vcs -cm coverage_arguments  
      additional_options_and_source_files
```

This command line instructs VCS to compile for all types of coverage. VCS creates the `simv.cm` directory in the current directory. In this directory, it creates the `coverage`, `db`, and `reports` directories.

2. Start the first simulation, specifying a name for the intermediate data files that associates them with the first simulation:

```
simv -cm_name test1 -cm coverage_arguments  
additional_runtime_options
```

During simulation, VCS writes the `test1.line`, `test1 fsm`, `test1.cond` and `test1.tgl` intermediate data files in the `./simv.cm/coverage/verilog` directory.

3. Revise the contents of the data file in the `$readmemb` system task and then start the second simulation, specifying a name for the intermediate data files that associates them with the second simulation:

```
simv -cm_name test2 -cm coverage_arguments  
additional_runtime_options
```

During simulation, VCS writes the `test2.line`, `test2 fsm`, `test2.cond` and `test2.tgl` intermediate data files in the `./simv.cm/coverage/verilog` directory.

4. Revise the contents of the data file in the `$readmemb` system task again and then start the third simulation, specifying a name for the intermediate data files that associates them with the third simulation:

```
simv -cm_name test3 -cm coverage_arguments  
additional_runtime_options
```

During simulation, VCS writes the `test3.line`, `test3 fsm`, `test3.cond` and `test3.tgl` intermediate data files in the `./simv.cm/coverage/verilog` directory.

5. Run `cmView` in batch mode telling it to write reports with file names that indicate that they have merged data:

```
vcs -cm_pp -cm_name merged
```

cmView reads all the intermediate data files in the ./simv.cm/coverage/verilog directory and writes the report files with the merged results in the ./simv.cm/reports directory. For example: merged.hier_c, merged.hier_f, merged.hier_t, and so on.

Method 2: Build the simv executable once, then simulate three times sequentially as in the previous method, but write the intermediate data files to a common directory for your design group.

This procedure is as follows:

1. Compile for coverage:

```
vcs -cm_dir /net/design1/oursimv.cm  
      -cm coverage_arguments  
      additional_options_and_source_files
```

This command line instructs VCS to compile for all types of coverage. VCS creates the oursimv.cm directory in the /net/design1 directory, and in the oursimv.cm directory, it creates the coverage, db, and reports directories.

2. Start the first simulation, specifying both the directory that VCS created for coverage information, as done in the previous step, and a name for the intermediate data files that associates them with the first simulation:

```
simv -cm_name test1 -cm coverage_arguments  
      additional_runtime_options
```

During simulation, VCS writes the test1.line, test1.fsm, test1.cond and test1.tgl intermediate data files in the /net/design1/oursimv.cm/coverage/verilog directory.

3. Revise the contents of the data file in the \$readmemb system task and then repeat the previous simv command line changing only the name of the intermediate data files:

```
simv -cm_name test2 -cm coverage_arguments  
      additional_runtime_options
```

4. Repeat the previous step, revising the contents of the data file and the name of the intermediate data files.

```
simv -cm_name test3 -cm coverage_arguments  
      additional_runtime_options
```

5. Now run cmView in batch mode, telling it the name and location of the directory for coverage data that you specified in step 1 and to write reports with file names that indicate that they have merged data:

```
vcs -cm_pp -cm_dir /net/design1/oursimv.cm  
      -cm_name \ merged
```

Method 3: Build the simv executable once, then simulate three times in parallel.

It is possible to compile for coverage once and simulate the design multiple times in parallel using different inputs. You do this by using the \$test\$plusargs system function and the corresponding runtime options. You can use this system function to execute different \$readmemb system tasks depending on the runtime option you use for each parallel simulation.

This procedure is as follows:

1. Compile for coverage:

```
vcs -cm_dir /net/design1/oursimv.cm -cm  
      coverage_arguments  
      additional_options_and_source_files
```

2. Start the three simulations with the following command lines specifying both the directory that VCS created for coverage information, as done in the previous step, and a name for the intermediate data files that associates them with the first simulation:

```
simv -cm_name test1 -cm +one coverage_arguments  
additional_runtime_options
```

```
simv -cm_name test2 -cm +two coverage_arguments  
additional_runtime_options
```

```
simv -cm_name test3 -cm +three coverage_arguments  
additional_runtime_options
```

The +one, +two, and +three options are used for the \$test\$plusargs system tasks in the testbench.

3. Run cmView in batch mode, once again specifying the name of the directory and the name of the report files.

```
vcs -cm_pp -cm_dir /net/design1/oursimv.cm -cm_name  
merged
```

Note:

Synopsys recommends this method of parallel simulation and merging results, because it is a faster way to run coverage regressions.

Method 4: Build three different simv executables and simulate sequentially or in parallel.

To simulate sequentially, the procedure is as follows:

1. Compile three times for coverage using different testbenches, specifying an alternative name for the simv executable. By doing this, you are also specifying an alternative name for the `simv.cm` coverage metrics database:

```
vcs -o simv1 test.v -cm coverage_arguments  
additional_options_and_source_files
```

```
vcs -o simv2 test.v -cm coverage_arguments  
additional_options_and_source_files
```

```
vcs -o simv3 test.v -cm coverage_arguments  
additional_options_and_source_files
```

When VCS does these compilations for coverage, it creates three coverage metrics directories in the current directory: `simv1.cm`, `simv2.cm`, and `simv3.cm`.

Note:

In step 1, all three tests use different stimulus. However, they all use the same source files from the same directories:

- Identical hierarchy
- No differences with `+define` or ``include`
- Source files are passed identically

If some source files are passed as a library in one design and not in the other, then use `cm_libs`.

2. Simulate the executables either sequentially or in parallel. Synopsys recommends parallel simulation.
3. Run cmView in batch mode telling it the name and location of the three directories that you created and the name of the report files:

```
vcs -cm_pp -cm_dir simv1.cm -cm_dir simv2.cm \
```

```
-cm_dir simv3.cm -cm_name merged
```

The report files are in the first specified directory, in this example, .*/simv1.cm/reports*.

Merging Additional Tests

If, after you have merged test data, you want to merge further data from additional tests, you can do so by using the `-cm_tests` command-line option. For example, after you execute the following command line:

```
vcs -cm_pp -cm_dir simv1.cm -cm_dir simv2.cm \
      -cm_dir simv3.cm -cm_name merged
```

You can merge additional data into the reports in `simv1.cm/reports` by entering the following command line:

```
vcs -cm_pp -cm_dir simv1.cm -cm_tests test_files
```

Where, for example, the file named, `test_files`, contains the following:

```
simv1.cm/reports/merged/merged
simv4.cm/verilog/test
```

In this example, `simv1.cm/reports/merged/merged` contains the intermediate data files named `merged`, that were created by the previously merged step and `simv4.cm/verilog/test` contains all the intermediate data files named, `test`, which are located in the `simv4.cm/verilog` directory.

Merging VCS MX Results for Regressions

cmView automatically merges the results from the intermediate data files that VCS MX writes during more than one simulation of your design. This section describes three methods of running regressions to produce merged coverage results. In most of these methods, you only need to compile for coverage once and then run multiple simulations in these three example simulations, either sequentially or in parallel. You use either different names for intermediate data files in the same directory, or create different directories for these intermediate data files and then tell cmView where to look for them.

Method 1: Build the simv executable once, then simulate three times using the same testbench, but different inputs.

There are a number of ways to use the same testbench in different simulations with different input stimulus values. One commonly-used way is to use the `load_memory` command to read values from a file into the elements of a memory. Then, for each simulation, you change the contents of the data file specified in that command. VCS MX reads this data at runtime and you make no changes to the system task, so you only need to compile for coverage once. The values in these memory elements control the execution of your design and exercise different parts of the design during the simulations. When the simulations are complete, you can see the total and merged coverage from all the simulations.

This procedure is as follows:

1. Analyze and compile for coverage:

```
vhdlan [additional_options] source_files  
vcs -cm line [additional_compile-time_options]  
design_configuration
```

These commands tell VCS MX to analyze the design for line coverage and then compile it. At compilation time, VCS MX creates the `simv.cm` directory in the current directory and creates the `coverage`, `db`, and `reports` directories in the `simv.cm` directory.

2. Start the first simulation, specifying a name for the intermediate data files that associates them with the first simulation:

```
simv -cm_name test1 -cm line [additional_runtime_options]
```

During simulation, VCS MX writes the `test1.line` intermediate data file in the `./simv.cm/coverage/vhdl` directory.

3. Revise the contents of the data file in the `load_memory` command and then start the second simulation specifying a name for the intermediate data files that associates them with the second simulation:

```
simv -cm_name test2 -cm line [additional_runtime_options]
```

During simulation, VCS MX writes the `test2.line` intermediate data file in the `./simv.cm/coverage/vhdl` directory.

4. Revise the contents of the data file in the `load_memory` command again and then start the third simulation specifying a name for the intermediate data files that associates them with the third simulation:

```
simv -cm_name test3 -cm line [additional_runtime_options]
```

During simulation, VCS MX writes the `test3.line` intermediate data files in the `./simv.cm/coverage/vhdl` directory.

5. Run `cmView` in batch mode telling it to write reports with file names that indicate that they have merged data:

```
cmView -b -cm_name merged
```

cmView reads all the intermediate data files in the ./simv.cm/coverage/vhdl directory and writes the report files with the merged results in the ./simv.cm/reports directory. For example, merged.hier_1.

Method 2: Build the simv executable once, then simulate three times as in the previous method, but write the intermediate data files to a common directory for your design group.

This procedure is as follows:

1. Analyze and compile for coverage.

```
vhdlan [additional_options] source_files  
vcs -cm line -cm_dir /net/design1/oursim.cm  
      [additional_compile-time options]  
      design_configuration
```

These commands tell VCS MX to analyze the design for line coverage and then compile it. At compilation, VCS MX creates the oursim.cm directory in the /net/design1 directory and the coverage, db, and reports directories in the oursim.cm directory.

2. Start the first simulation specifying both the directory that VCS MX created for coverage information, as done in the previous step, and a name for the intermediate data files that associates them with the first simulation:

```
simv -cm_name test1 -cm line [additional_runtime_options]
```

During simulation, VCS MX writes the test1.line intermediate data files in the /net/design1/oursim.cm/coverage/vhdl directory.

3. Revise the contents of the data file in the `load_memory` command and then repeat the previous `simv` command line, changing only the name of the intermediate data files:

```
simv -cm_name test2 -cm line [additional_runtime_options]
```

4. Repeat the previous step, revising the contents of the data file and the name of the intermediate data files.

```
simv -cm_name test3 -cm line [additional_runtime_options]
```

5. Now, run `cmView` in batch mode, telling it the name and location of the directory for coverage data, which you specified in step 1, and to write reports with file names that indicate that they have merged data:

```
cmView -b -cm_dir /net/design1/oursim.cm -cm_name merged
```

Method 3: Build three different `simv` executables and simulate sequentially or in parallel.

To simulate sequentially, the procedure is as follows:

1. Compile three times for coverage using different testbenches, specifying an alternative name for the `simv` executable and by doing so, specifying an alternative name for the `simv.cm` coverage information directory:

```
vhdlan [additional_options] source_files
```

```
vcs -cm line -o simv1 [additional_compile-time_options]  
design_configuration
```

```
vcs -cm line -o simv2 [additional_compile-time_options]  
design_configuration
```

```
vcs -cm line -o simv3 [additional_compile-time_options]  
design_configuration
```

When VCS MX does these compilations for coverage, it creates three coverage information directories in the current directory: simv1.cm, simv2.cm, and simv3.cm.

Note:

The different testbench modules must have the same module name.

2. Simulate the executables either sequentially or in parallel. Synopsys recommends parallel simulation.
3. Run cmView in batch mode telling it the name and location of the three directories that you created and the names of the report files:

```
cmView -b -cm_dir simv1.cm -cm_dir simv2.cm  
       -cm_dir simv3.cm -cm_name merged
```

The report files are located in the first specified directory (in this example, ./simv1.cm/reports).

Mapping Subhierarchy Coverage Between Designs in Verilog

Note:

For corresponding VHDL information see “[Mapping Subhierarchy Coverage in VHDL](#)”.

You can instantiate a subhierarchy (a module instance in your design and all the module instances hierarchically under this instance) in two different designs and see the combined coverage for the subhierarchy from the simulation of both designs.

You can do this by mapping the coverage information for that subhierarchy from one design to another. This is possible even though the hierarchy above this subhierarchy in the two designs is completely different.

You can map the coverage information with the `-cm_map module_name` command-line option and argument for the cmView command (or vcs -cm_pp). You must specify the top-level module name (identifier) of the subhierarchy (not the hierarchical name of the top-level module instance in the subhierarchy).

Note:

- You can only map coverage information from one design to another in cmView's batch mode, therefore, include the `-cm_pp` command-line option and not `-cm_pp gui`.
- When you map coverage from one design to another, the source file names must be identical.
- cmView does support `-cm_map` and `-cm_tests` together, with limitation that the compile time database should be present in the same directory where the selected runtime data is present. This mean that tests given inside the `-cm_test` file should be under the directory specified by `-cm_dir` at report time (either in the Base design or Input design directories).

For example, test can be present at `foo.cm/coverage/<verilog/vhdl>` or `foo.cm/reports/<cmView/ -cm_name given directory >/` for given base / input design `foo.cm`.

Using Multiple -cm_map Options

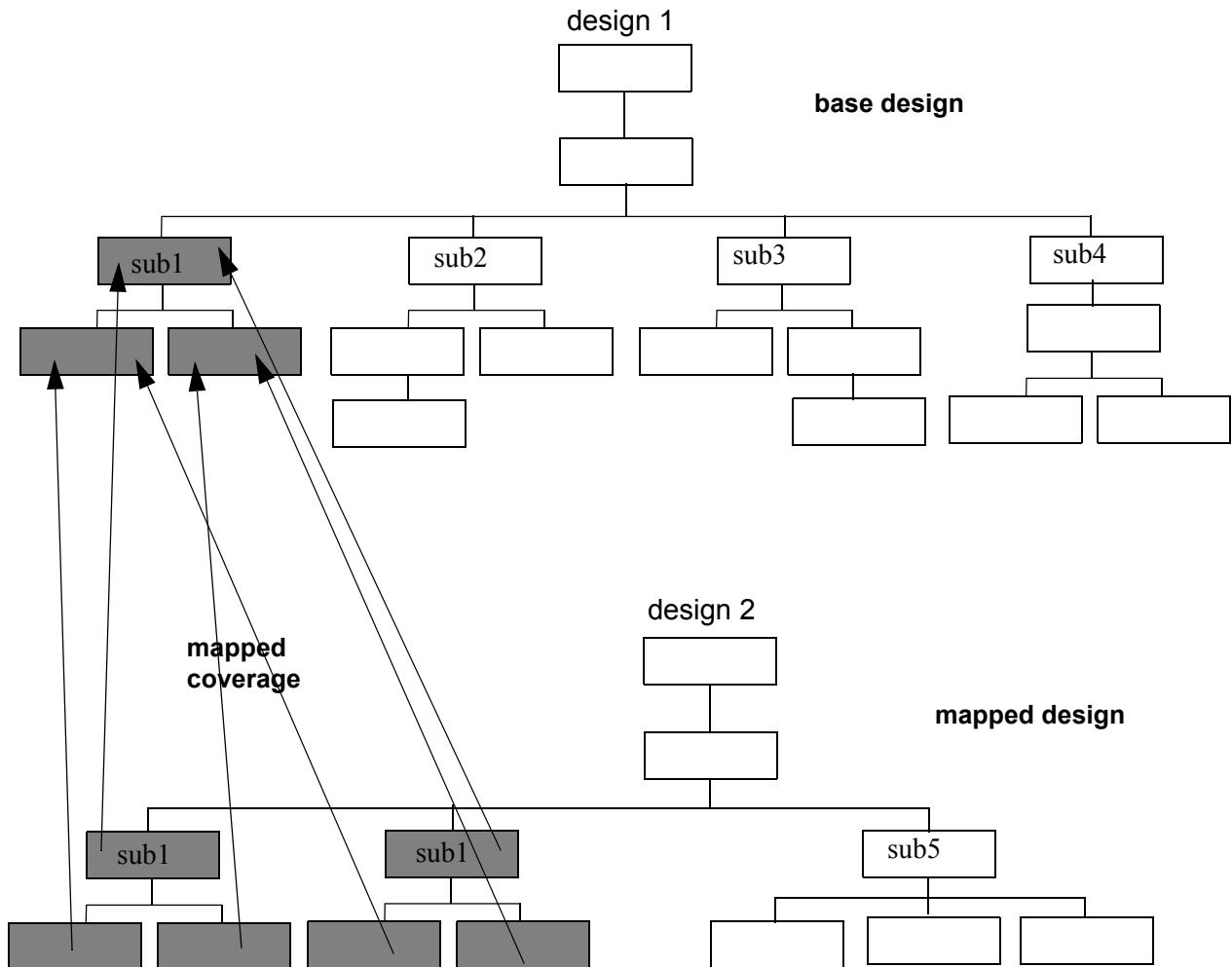
You can specify multiple `-cm_map` options as long as you also specify the respective `-cm_dir` options when using `vcs -cm_pp`. For example:

```
% vcs -cm_pp -cm_dir chip.cm -cm_dir t1.cm \
      -cm_dir t2.cm -cm_map t1 -cm_map t2
```

The first directory specified with `-cm_dir` must have both `t1` and `t2` instantiated.

In addition, you can use `vcs -cm_pp` to merge intermediate coverage data files that were generated on different platforms. In other words, the intermediate information in the coverage directories is platform-independent.

Figure 1-2 Mapping Coverage



In [Figure 1-2](#), two designs instantiate a common subhierarchy, labeled **sub1**. The code for the subhierarchy, in both designs, is in the source file named **sub1.v** and the module name (identifier) of the top-level module in the subhierarchy is **sub1**. This example shows mapping coverage information for that subhierarchy from the simulation of design 2 to the coverage information for that subhierarchy from the simulation of design 1. As illustrated in [Figure 1-2](#), there can be multiple instances of the subhierarchy in the design *from* which coverage information is mapped (from now on

simply called the mapped design), but there can only be one instance of the subhierarchy in the design *to* which the coverage information is mapped (from now on simply called the base design).

The following procedure explains how to map coverage information:

1. Compile the base design for coverage and then simulate that design while monitoring for coverage. For example:

```
cd /net/design1  
vcs -cm line sub1.v sub2.v sub3.v sub4.v main.v test.v  
simv -cm line
```

2. Compile the mapped design for coverage and then simulate that design while monitoring for coverage. For example:

```
cd /net/design2  
vcs -cm line sub1.v sub5.v main.v test.v  
simv -cm line
```

3. Run cmView specifying the name of the top-level module in the subhierarchy. Also, specify the coverage metrics database for the base design then specify the mapped design. For example:

```
vcs -cm_pp -cm_dir /net/design1/simv.cm \  
-cm_dir /net/design2/simv.cm -cm_map sub1
```

cmView writes the coverage report files in the `reports` subdirectory in the coverage metrics database for the first or base design; in this example, `/net/design1/simv.cm/reports`. By default, these report files only contain sections for the module instances in the subhierarchy. These module instances are identified by their hierarchical names in the first or base design. The coverage information in these sections is the coverage information from both designs.

If you also include the `-cm_hier` compile-time option to specify a larger subhierarchy that includes the subhierarchy for which you want mapped coverage, the resulting report files contain sections for the module instances in this larger hierarchy, but the sections for the instances in the smaller subhierarchy, that is the subhierarchy with mapped coverage, also contain coverage information from the other design.

Mapping Subhierarchy Coverage in VHDL

You can also map a VHDL subhierarchy's coverage from one design to the other using the `-cm_map` cmView command-line option.

The procedure for VHDL is as follows:

1. Compile for coverage and simulate the base design while monitoring for coverage.
2. Compile for coverage and simulate the mapped design while monitoring for coverage.
3. Run cmView specifying the subhierarchy with the `-cm_map` option. Also, the coverage metrics directories for both designs with multiple entries of the `-cm_dir` option.
4. You specify the name of the entity at the top of the subhierarchy that you use in both designs. You can specify this two ways:
 - Specify the entity name:

```
-cm_map entity_name
```

- Specify the library name and entity name:

```
-cm_map library_name.entity_name
```

An example command line for cmView is as follows:

```
cmView -b -cm_dir /net/design1/simv.cm  
-cm_dir /net/design2/simv.cm -cm_map sub1
```

cmView writes the coverage report files in the reports subdirectory in the coverage metrics database for the first or base design; in this example, /net/design1/simv.cm/reports. By default, these report files only contain sections for the entity instances in the subhierarchy. These entity instances are identified by their hierarchical names in the first or base design. The coverage information in these sections is the coverage information from both designs.

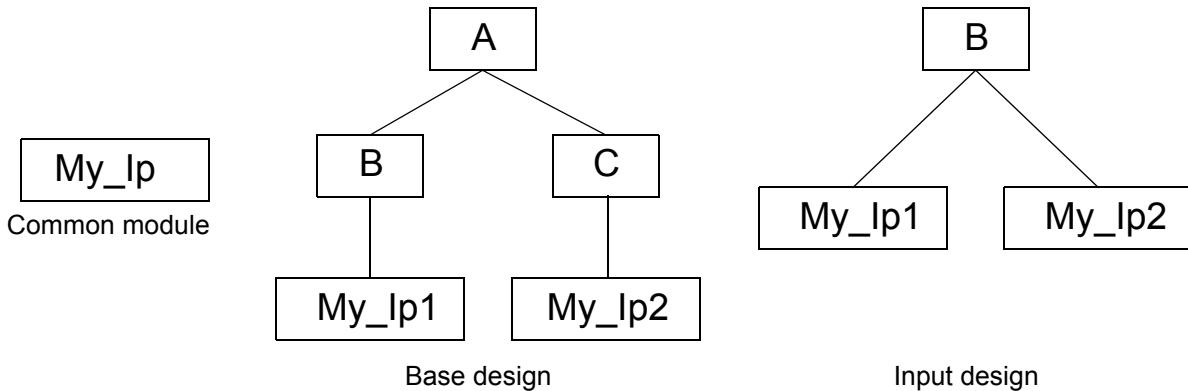
For information about using native testbench, see the *OpenVera Language Reference Manual: Native Testbench*.

Understanding Instance-Based Mapping

Instance-based mapping allows you to specify an instance in your “base design” for which you want to merge coverage data for two different designs.

Assume that you have the same module instantiated in different places in two different designs. (For example, one of the designs could be a system-level design, and the other, an RTL design.)

Consider the following illustration:



Here, you are able to map instances `B.My_Ip1` and `B.My_Ip2` to instance `A.C.My_Ip1`.

The mapping configuration file for the above illustration would contain the following information:

```
MODULE: My_Ip
INSTANCE:
  SRC: B.My_Ip1, B.My_Ip2, B.My_Ip5
  DST: A.B.My_Ip1
```

The syntax is composed of the following elements:

- **MODULE:** Specifies the name of the common module.
- **INSTANCE:** Provides source and destination information for MODULE.
- **SRC:** Represents the list of one or more module instances in the input design.

You can specify multiple instances in a comma-separated list after the `SRC` keyword. If an instance (such as `B.My_Ip5`) does not appear in the specified input design, VCS ignores that instance.

- DST: Represents a list of one or more module instances in the base design to which the source module instances are to be mapped.

Note:

Both SRC and DST can contain either a full hierarchical path or the * wildcard.

The instance-based mapping option -cm_mapfile allows you to selectively merge coverage data for two or more instances.

The following syntax shows the context for the -cm_mapfile option:

```
vcs -cm_pp -cm_dir base.cm -cm_dir input.cm -cm_mapfile map_config_file
```

Where, *map_config_file* is the mapping configuration file.

Note the following guidelines:

- You can only map instances of the same module between the input design and the base design.
- The mapping instructions in the mapping configuration file apply to all directories defined with the -cm_dir option.
- You cannot use -cm_mapfile and -cm_map together.
- The standard instance-based reports contain the merged data (unless you change the report names):
 - cmView.long_l
 - cmView.long_t
 - cmView.long_c

Examples of Instance-Based Mapping Syntax

The following examples illustrate how to use the mapping configuration file syntax.

Example 1

If you want to merge all instances of a specified module in the input design, use the * wildcard following SRC:

```
MODULE: My_Ip
INSTANCE:
    SRC: *
    DST: A.B.My_Ip1
```

This syntax merges all the instances (B.My_Ip1, B.My_Ip2) of the input design into A.B.My_Ip1.

Example 2

If you want to merge one instance of the input design into all the instances of the base design, use the * wildcard after DST:

```
MODULE: My_Ip
INSTANCE:
    SRC: B.My_Ip1
    DST: *
```

This syntax merges the data from B.My_Ip1 into both A.B.My_Ip1 and A.C.My_Ip2.

2

Line Coverage

Line coverage (or statement coverage) shows you which lines of code are exercised — and which ones are not — by your testbench during a simulation run. VCS and VCS MX generate metrics for total coverage of lines, blocks, and branches that tell you how much of your code is actually executed by the tests. They also produce annotated listings which identify the unexercised statements, therefore, giving you the information you need to write more tests to complete the coverage. However, coverage reports do not tell you how many times a line of code is executed.

This chapter describes:

- “[What Is Covered in Line Coverage?](#)”
- “[Line Coverage Glitch Suppression](#)”
- “[Creating Line Coverage Reports](#)”
- “[Viewing Line Coverage with the Coverage Metrics GUI](#)”

Note:

There are various methods used to modify both line and other types of coverage. These methods are explained in the sections: “[Filtering Constants](#)” on page 1-6, “[Using Glitch Suppression](#)” on page 1-34, and “[Omitting Coverage for Default Case Statements](#)” on page 1-43.

What Is Covered in Line Coverage?

In line coverage, VCS and VCS MX keep track of certain kinds of procedural statements (Verilog and VHDL) and statement blocks and branches (VHDL). This section discusses line coverage for both Verilog and VHDL.

Line Coverage for Verilog

VCS and VCS MX track the following types of Verilog procedural statements:

- Statements that cause a simulation event, such as a procedural assignment statement, or a system task.
- Statements that control which other statements are executed, such as a `while` or `if` statement.

Note:

By default, line coverage does not keep track of continuous assignment statements (see “[Enabling Line Coverage for Continuous Assignments](#)”).

The following code example illustrates what is covered in Verilog:

Example 2-1 Monitored Statements in Line Coverage

```
module top;
reg clk;
reg [8:0] data;
wire [8:0] results;

initial
begin
    $display("Assigning initial values");
    clk=0; data=256;
    #100 $finish;
end

always
#3 clk=~clk;

dev dev1 (results,clk,data);

endmodule

module dev (out,clock,in);
output [8:0] out;
input clock;
input [8:0] in;
reg [8:0] out;
reg en;

task splitter;
input [8:0] tin;
output [8:0] tout;
begin
    tout = tin/2;
end
endtask

always @ (posedge clock)
begin
if (in % 2 !== 0)
    $display("error cond, input not even");
```

```

else
    out = in;
if (in % 2 == 0)
en =
    1;
while (en)
forever
    case (out % 2)
        !0 : #0 $finish;
        0  : #5 splitter(out,out);
    endcase
end
endmodule

```

In [Example 2-1](#), the boldface lines contain statements that VCS keeps track of in line coverage.

VCS and VCS MX line coverage also includes information about always and initial blocks and other types of blocks of code. In line coverage, a block is a nesting level in the code, not so much a level in the hierarchy, but a level of control of the execution of the procedural code. Typically, you change the indentation of your source code for a nesting level. [Example 2-2](#) illustrates the code blocks in the source code in [Example 2-1](#). It is interrupted in a number of places to explain the blocks.

Example 2-2 Code Blocks in Line Coverage

```
module top;
reg clk;
reg [8:0] data;
wire [8:0] results;

initial
begin
$display("Assigning initial values");
clk=0; data=256;
#100 $finish;
end
```

The initial block is a distinct block of code in this top-level module. Inside the initial block is a procedural delay on the `$finish` system task. This delay and system task constitute a separate block of code, so that if simulation ends before time 100, the other statements in the initial block can be covered, just not the `$finish` system task.

Notice that the assignment statements to regs `clk` and `data` are on the same line. This illustrates the difference between line and statement coverages. In this case, one line is covered if two statements are covered.

Also, in terms of statement coverage, the procedural delay is considered a separate wait statement in addition to the `$finish` system task on that line, so the initial block contains five statements in three lines.

```

always
#3 clk=~clk;

dev dev1 (results,clk,data);

endmodule

```

The `always` block that contains an assignment statement with a procedural delay is one block instead of two. This occurs because it is always a mistake to put neither an event control (such as `@ (posedge clk)`) nor a delay specification (such as `#10`) between the `always` keyword and a procedural statement, so there is no need for the two blocks.

Similarly, the procedural delay in the `always` block is not a separate `wait` statement.

```

module dev (out,clock,in);
output [8:0] out;
input clock;
input [8:0] in;
reg [8:0] out;
reg en;
task splitter;
input [8:0] tin;
output [8:0] tout;
begin
tout = tin/2;
end
endtask

```

The procedural statements in a task definition (in this case, one statement) are a separate block.

```

always @ (posedge clock)
begin
if (in % 2 !== 0)
    $display("error cond, input not even");
else
    out = in;
if (in % 2 == 0)
    en =
        1;
while (en)
    forever
        case (out % 2)
            !0 : #0      $finish;
            0  : #5      splitter(out,out);
    endcase
end
endmodule

```

A block for this `always` block contains three statements: an `if-else`, `if`, and `while` statement.

The `$display` system task is controlled by the `if` construct of the `if-else` statement and is in a separate block.

The procedural assignment to reg `out` is a separate block because it is controlled by the `else` construct.

The procedural assignment to reg `en` is a separate block because it is controlled by the second `if` construct. This is one statement on two lines.

The `forever` statement is in a separate block because it is controlled by the `while` statement.

The `case` statement is in a separate block because it is controlled by the `forever` statement.

The case item statements are separate blocks and the procedural delays on these case item statements are considered separate `wait` statements.

There also can be blocks for missing constructs. If you use an `if` statement instead of an `if-else` statement, VCS and VCS MC consider the `else` construct to be missing. If you use a `case` statement and do not enter a default case, VCS and VCS MX consider the default case to be missing.

Enabling Line Coverage for Continuous Assignments

By default, line coverage does not include Verilog continuous assignments. You can enable line coverage for continuous assignments with the `-cm_line` `contassign` compile-time option and keyword argument. For example:

```
vcs -f design_files -cm line -cm_line contassign
```

When you include this compile-time option and keyword argument, `cmView` shows the continuous assignment statement, including any delay specification, as a block of code.

If a continuous assignment assigns a constant value, VCS and VCS MX do not compile or monitor it for line coverage. For example:

```
module dev;
  wire w1,w2,w3;
  reg r1,r2;
  .
  .
  .
```

```

assign #5 w1=1;
assign #3 w2=r1;
assign #10 w3=r1 && r2;
endmodule

```

VCS and VCS MX compile and monitor the continuous assignments to w2 and w3 for line coverage, but not to w1.

Assignment Coverage

You can have cmView report what assignment statements caused a bit in a signal to toggle from 0 to 1 and from 1 to 0. cmView reports this in the file for the module definition in the `simv.cm/reports/` annotated directory. You enable this reporting with the `-cm_line assigntgl` compile-time option and keyword argument.

Note:

This is an LCA and a Verilog-only feature. There is no similar report for VHDL.

Consider the following basic example:

```

module dev (clk1, clk2, in1, out1, out2, out3);
input clk1, clk2;
input [3:0] in1;
output [3:0] out1, out2;
output [1:0] out3;
reg [3:0] out1, out2;
reg [1:0] out3;

always @ (posedge clk1)
out1 = in1;

always @ (posedge clk2)
begin
out3 = in1[1:0];
out2 = in1;

```

```
end  
endmodule
```

There are three signals in this definition and assignment statements to these signals. Assignment coverage tells you which signals, and which bits in these signals, toggled from $0 \rightarrow 1$ and from $1 \rightarrow 0$. Toggling from X or Z $\rightarrow 1$ and from X or Z $\rightarrow 0$ are not counted in assignment coverage.

If you include the `-cm_line assigntgl` compile-time option and keyword argument, cmView indicates what assignments caused a bit to toggle in the annotated file for module dev. This file is `./simv.cm/reports/annotated/dev` (see “[Annotated File for Assignment Coverage](#)” for an example of the annotated file with assignment coverage notations for this module definition).

Line Coverage for VHDL

In line coverage, VCS MX keeps track of the following in the VHDL source code:

- Individual statements
- Statement blocks
- Statement block type
- Branches for conditional statements

Note:

Statement blocks consist of a set of individual statements executed sequentially and always executed together in the same time step.

Concurrent signal assignments and component instantiations are not reported in the coverage reports because they are always executed.

Coverage information is not reported for design regions running in cycle mode.

Coverage information is not reported for execution lines in entity or configuration declaration statements.

VCS monitors line coverage inside for-generates, but individual instances of the generate loop are not monitored separately. Therefore, a line is reported as covered if it is covered in any iteration of the loop, and a line is reported not covered only if it is not covered in any iteration of the loop.

Line Coverage Glitch Suppression

In glitch suppression for line coverage, VCS looks for fast triggering of `always` blocks, not initial blocks, and the glitch period is an interval of simulation time in which there are one or more executions of the statements in the `always` block.

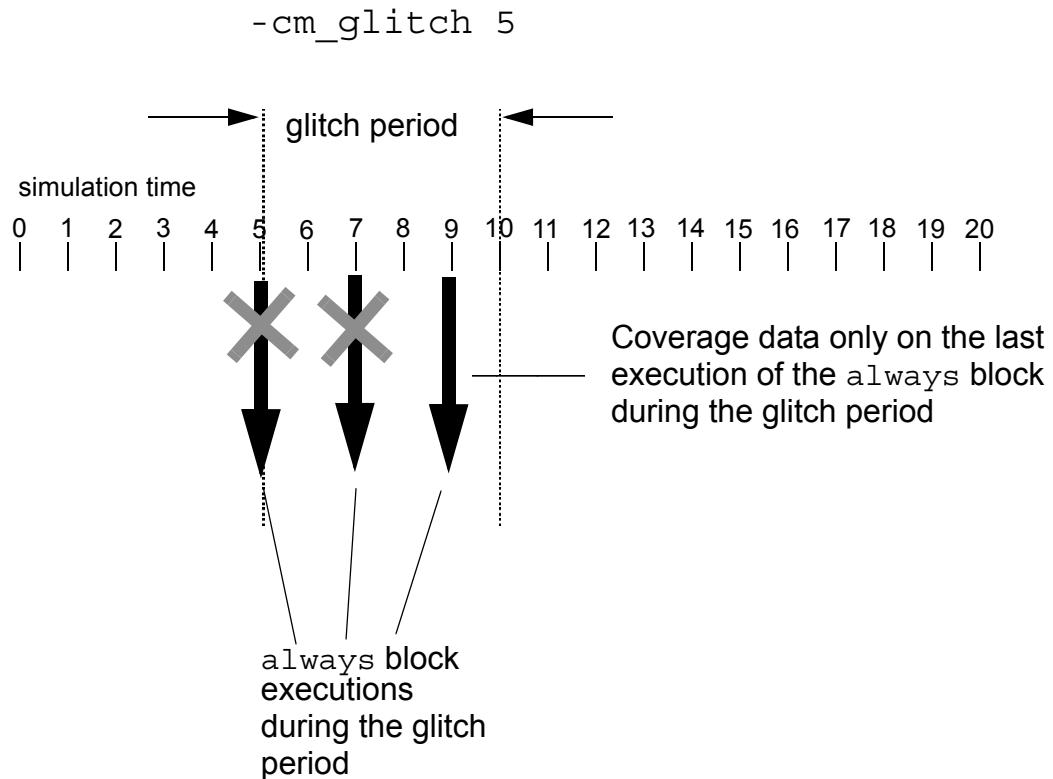
When using glitch suppression, VCS only monitors and records the results on the last execution of the `always` block during the glitch period.

The `-cm_glitch positive_integer` compile-time option specifies the simulation time period of glitch suppression (see “[Using Glitch Suppression” on page 1-34](#)).

Note:

Glitch suppression does not work for VHDL code.

Figure 2-1 Line Coverage Glitch Period



Consider the following example:

```
module dev (output1,input1,input2,input3);
  output output1;
  input input1,input2,input3;
  reg output1;
  .
  .
  .
  always @ (input1 or input2 or input3)
    if (input1 && input2)
      output1=input3;
```

```

else
    output1=~input3;
.
.
.
endmodule

```

Glitches, or narrow pulses, on the input ports of this module could cause this `always` block to trigger (or execute) several times, while little or no simulation time elapses, before the values on the input ports settle down and there is a final execution of the `always` block.

In this example, the conditional expression (`input1 && input2`) could be briefly true during one of these glitches or narrow pulses, but after settling down the expression is finally not true. If this happens, when VCS monitors instances of this module for line coverage, it sees that the lines controlled by the `if` part and the `else` part of this conditional expression, both assignment statements, are executed.

With glitch suppression, VCS does not monitor this code for line coverage during the entire length of these glitches or narrow pulses, and in this example only, VCS sees the execution of the second assignment statement in the final execution of the `always` block.

Limitation on Clocks

Line coverage interprets an event control with the `posedge` or `negedge` keywords as specifying a clock and does not suppress glitches, or narrow pulses, when you include the `-cm_glitch` compile-time option. Consider the following line numbered example:

```

1 module test;
2 reg r1;
3 initial
4 begin

```

```

5      r1=0;
6      #5 r1=1;
7      #1 r1=0;
8      #10 $finish;
9 end
10
11 dev dev1 (w1,r1);
12
13 endmodule
14
15 module dev (out,in1);
16 output out;
17 input in1;
18 reg out;
19 always @ (posedge in1 )
20 if (in1 )
21     out=in1; //Is this line covered?
22 else
23     out=~in1;
24 endmodule

```

Line coverage interprets signal `in1` in line 19 as a clock because the event control that controls the execution of the `always` block (sometimes called the sensitivity list of the `always` block) includes the keyword `posedge`.

If you entered the following command line:

```
vcs example.v -cm line -cm_glitch 1
```

You might expect line 21 not to be covered, because VCS executes it when `in1` is true, and as you can see from lines 6 and 7 that `in1` will be true for only one time unit, which is the glitch period specified on the command line.

However, line coverage does not suppress glitches on clocks, therefore, line 21 will be reported as covered even when you specify glitch suppression with the `-cm_glitch` compile-time option.

Creating Line Coverage Reports

This section describes how to use the cmView utility to write line coverage report files.

Perform the following to create line coverage report files:

1. Compile your design for line coverage. For example:

```
vcs source.v -cm line
```

or

```
vcs -cm line cfg
```

2. Instruct VCS or VCS MX to monitor for line coverage during simulation. For example:

```
simv -cm line
```

3. Instruct cmView to write line coverage reports. For example:

```
vcs -cm_pp -cm line
```

By default, cmView writes its report files in the `./simv.cm/reports` directory.

The report files created by cmView include the following:

`cmView.long_1`

A long report file containing comprehensive information about the line coverage of your design, organized by module instance (for a sample report, see “[The cmView.long_1 File](#)”).

`cmView.long_1d`

Another long report file containing comprehensive information about the line coverage of your design, organized by module definition instead of module instance (for a sample report, see “[The cmView.long_Id File](#)”).

`cmView.hier_1`

Line coverage report where coverage data is presented in subhierarchies in the design. There is a section for each module instance and the line coverage information in these sections is for the instance and all module instances hierarchically under that instance (for a sample report, see “[The cmView.hier_1 File](#)”).

`cmView.mod_1`

Line coverage report where coverage data is for module instances in the design, not subhierarchies. There is a section for each module instance and the information in these instance sections do not include data for instances hierarchically under the specified instance (for a sample report, see “[The cmView.mod_1 File](#)”).

`cmView.mod_1d`

Line coverage report, similar to the `cmView.mod_1` file, where coverage data is for module definitions instead of module instances (for a sample report, see “[The cmView.mod_1d File](#)”).

`cmView.short_1`

A short report file containing only sections for instances in which all lines were not covered. In these sections cmView only lists the uncovered lines. The report ends with summary information (for a sample report, see “[The cmView.short_1 File](#)”).

`cmView.short_1d`

Another short report file, similar to the `cmView.short_1` file, where coverage data is for module definitions instead of module instances (for a sample report, see “[The cmView.short_1d File](#)”).

Annotated Source Files

The `-cm_report annotate` or `vcs -cm_pp` option and keyword argument tells cmView to write annotated files in the `/simv.cm/reports/annotated` directory.

In the annotated source files, cmView does the following:

- Numbers each line from the module instance or module definition
- Sometimes breaks one line into two or adds extra lines

Its a good idea to review these files because cmView will report line coverage based on the line numbers in the annotated files, not the original source files. The files in the annotated directory are as follows:

- A file named after the top-level module — this file shows the lines not covered in the top-level module.
- Files named after each module instance except the top-level module — these files show the lines not covered in each instance.
- Files named after each module definition except the top-level module — these files show the lines not executed in any of the instances of the module.

You can limit this output to annotated files for definitions or instances (see “[Limiting Annotated Files to Definitions or Instances](#)”).

For the Verilog source in [Example 2-1](#), cmView wrote the following `top`, `dev`, and `top.dev1` annotated source files.

Annotated Source File `top`

```
1      module top;
2          reg clk;
```

```

3      reg [8:0] data;
4      wire [8:0] results;
5
6      initial
7      begin
8          $display("Assigning initial values");
9          clk=0; data=256;
10         ==> #100 $finish;
10.1      #(100)
10.1      ==> $finish;
11      end
12
13      always
14          #3 clk=~clk;
15
16      dev dev1 (results,clk,data);
17
18      endmodule
//-----The following is Line Coverage-----
//-----Module Coverage Summary

//          TOTAL      COVERED      PERCENT
lines        4          4          100.00
statements   6          5          83.33
blocks       3          2          66.67

ALWAYS        1          1          100.00
INITIAL       1          1          100.00
WAIT          1          0          0.00

```

Notice the following in the file:

```

10      ==> #100 $finish;
10      #(100)
10.1    ==> $finish;

```

Line 10 was not covered.

cmView sees two statements in line 10, a procedural delay that VCS interprets and a wait statement, and the \$finish system task. cmView adds line 10.1 to separate the two statements. In this example, the annotated file indicates that the delay wait statement was covered, but the system task was not. The distinction is not important and the opposite is reported in other output files of cmView. This simply indicates that the delay did not elapse, and VCS did not execute the system task.

Annotated Source File top.dev1

```
20      module dev (out,clock,in);
21          output [8:0] out;
22          input clock;
23          input [8:0] in;
24          reg [8:0] out;
25          reg en;
26
27          task splitter;
28              input [8:0] tin;
29              output [8:0] tout;
30              begin
31                  tout = tin/2;
32              end
33          endtask
34
35          always @ (posedge clock)
36          begin
37              if (in % 2 !== 0)
38                  ==> $display("error cond, input not even");
39              else
40                  out = in;
41              if (in % 2 == 0)
42                  en =
42.1          ==> MISSING_ELSE
43                  1;
44                  while (en)
45                      forever
46                          case (out % 2)
47                              !0 : #0 $finish;
48                  ==> 0 : #5 splitter(out,out);
48
48.1          #(5)
48.1          splitter(out, out);
48.2          ==> MISSING_DEFAULT
48.3          ==> WHILE_FALSE
49                  endcase
50          end
51      endmodule
//-----The following is Line Coverage-----
//-----
//          Module Coverage Summary
//          TOTAL          COVERED          PERCENT
lines          10            9            90.00
statements    13            12           92.31
blocks        11            10           90.91
ALWAYS         1             1            100.00
CASEITEM       2             2            100.00
FOREVER        1             1            100.00
IF             2             1            50.00
ELSE           1             1            100.00
MISSING_ELSE   1             0            0.00
ROUTINE        1             1            100.00
WAIT           2             2            100.00
```

WHILE	1	1	100.00
WHILE_FALSE	1	0	0.00
MISSING_DEFAULT	1	0	0.00

Notice the following in the file:

```
38      ==>     $display("error cond, input not even");
```

This line was not executed.

```
41          if (in % 2 == 0)
42          en =
42.1    ==> MISSING_ELSE
43          1;
```

cmView takes note of the fact that this source code contains an `if` statement instead of an `if-else` statement. It notes this by inserting, right after the statement controlled by the `if` construct, `==> MISSING_ELSE`. In this particular case, the assignment statement takes up two lines and cmView ignores the second line and inserts `==> MISSING_ELSE` right after the first line.

```
48      ==>     0 : #(5) splitter(out,out);
48          splitter(out, out);
48.1    ==> MISSING_DEFAULT
48.3    ==> WHILE_FALSE
```

There is a coverage problem on line 48. cmView adds the following:

- Line 48.1, separating the task enabling statement from its delay.
- Line 48.2 `==> MISSING_DEFAULT` for the missing default case item.
- Line 48.3 for `==> WHILE_FALSE`. cmView is pointing out that there is no code that VCS executed when the conditional expression in the while statement was not true.

Because there was only one instance of module `dev`, the annotated source files `top.dev1` and `dev` are identical.

Limiting Annotated Files to Definitions or Instances

You can tell cmView to omit writing annotated files for each instance, and just write annotated files for each module (or VHDL entity) definition by adding `+module` to the `annotate` keyword. For example:

```
-cm_report annotate+module
```

You can tell cmView to omit writing annotated files for each module (or VHDL entity) definition, and just write annotated files for each instance by adding `+instance` to the `annotate` keyword. For example:

```
-cm_report annotate+instance
```

Excluding Your Code from Coverage Analysis

In VCS or VCS MX, you can exclude a line or a block of code that is unreachable from being exercised in a simulation run for coverage analysis. You can do this by enclosing the code within pragmas.

When VCS or VCS MX encounters pragmas during simulation, it excludes the code within pragmas from the coverage analysis and highlights the unexercised code with either `XC` or `XX` symbol.

Usually, parts of `if-else` blocks or a few states in a case statement are not exercisable during simulation. VCS or VCS MX ignores such lines for coverage analysis and marks these lines with `XC` symbol.

The following example clearly illustrates how VCS or VCS MX ignores line number 33 for coverage analysis because it is not reachable.

When you use the compile-time option, `-cm_noconst`, VCS or VCS MX does not consider line 33 for analysis and marks it with `XC` symbol.

```
30          if(out == 'd3)
31              $display("out is 3");
32          else
33      XC>      $display("out is other than 3");
```

You can also use the pragmas `//VCS coverage off` and `//VCS coverage on` to exclude a statement, or a block of statements from coverage computation. When VCS or VCS MX encounters pragmas, it marks the lines with `xx>` symbol in the annotated file.

```
11          always @ (posedge r1)
12              //VCS coverage off
13      XX>      r2=r1;
14              //VCS coverage on
```

In case the lines within the pragmas were exercised during simulation, VCS or VCS MX marks them with `x*>` symbol in the annotated file as shown in the following example:

```
45          case (DIN)
46              4'b0001:DOUT <= 4'b0001;
47              4'b0010:DOUT <= 4'b0010;
48          //VCS coverage off
49      X*>      4'b0100:DOUT <= 4'b0011;
50      XX>      4'b1000:DOUT <= 4'b0100;
50.0      X*>      MISSING_DEFAULT
51          endcase
52          //VCS coverage on
```

However, these lines do not count for the coverage computation.

Annotated File for Assignment Coverage

The following is the annotated source file that cmView writes for the Verilog code in [Example 2-2](#). It is the result of assignment coverage (see “[Assignment Coverage](#)”).

You enable writing this file with the `-cm_line assigntgl` compile-time option and keyword argument.

Note:

The `-cm_report annotate` option and argument, on the cmView command line, does not specify writing this file.

This is a Verilog-only feature. There is no comparable report for VHDL.

We present the entire file first and then examine several lines in the file.

```
28 module dev (clk1, clk2, in1, out1, out2, out3);
29 input clk1, clk2;
30 input [3:0] in1;
31 output [3:0] out1, out2;
32 output [1:0] out3;
33 reg [3:0] out1, out2; out2(f,0)
34 (3,0) reg [1:0] out3;
35
36
37           always @ (posedge clk1)
38           out1 = in1;
39
40           always @ (posedge clk2)
41           begin
42           (3,0)           out3 = in1[1:0];
43           (f,0)           out2 = in1;
44           end
45           endmodule
//-----The following is Line Coverage-----
//-----Module Coverage Summary
//
```

	TOTAL	COVERED	PERCENT
lines	3	3	100.00

```

statements      3          3          100.00
blocks         2          2          100.00
ALWAYS          2          2          100.00

//-----Assignment Toggle Statistics-----
//-----
//           Module Coverage Summary

          TOTAL    COVERED    PERCENT
assigns        3          1          33.33
assign bits    10         4          40.00
assign bits(0->1) 10         10         100.00
assign bits(1->0) 10         4          40.00
variables      3          1          33.33
variable bits  10         4          40.00
variable bits(0->1) 10         10         100.00
variable bits(1->0) 10         4          40.00

```

Line 33 is the first line with an assignment coverage mask following it:

```

33                      reg [3:0] out1, out2;
                           out2(f,0)

```

This line indicates that these regs were declared on line 33 of the source file. The notation under the declaration, `out2(f,0)`, is called a mask. The declaration is for two four-bit regs, `out1` and `out2`. The mask indicates that for `out2` all four bits toggled from $0 \rightarrow 1$, but no bits toggle from $1 \rightarrow 0$. In the mask (or parentheses), the first value is for $0 \rightarrow 1$ transitions, and `f` is hexadecimal for the binary 1111, indicating that the $0 \rightarrow 1$ transition is covered for all four bits. The second value is for the $1 \rightarrow 0$ transitions, and the 0 indicates that none of the four bits is covered for the $1 \rightarrow 0$ transition.

Line 42 is the next line with an assignment coverage mask:

```

42          (3,0)          out3 = in1[1:0];

```

This line has this procedural assignment statement. Remember that `out3` is declared to be a two-bit reg. The mask `(3, 0)` indicates that for the $0 \rightarrow 1$ transition, both bits were covered (3 being the equivalent of the binary 11), but neither of the two bits were covered for the $1 \rightarrow 0$ transition.

Line 43 is the next line with an assignment coverage mask:

```
43          (f, 0)          out2 = in1;
```

The procedural assignment statement on line 43 executed enough times, assigning enough values, to cover all four bits of reg `out2` for the $0 \rightarrow 1$ transition, but none of the executions of this assignment statement covered any of the bits for the $1 \rightarrow 0$ transition.

What about the following line?

```
38          out1 = in1;
```

There is no assignment coverage mask between the line number and the assignment statement. By default, no mask means that all four bits of `out1` were covered for the $0 \rightarrow 1$ and the $1 \rightarrow 0$ transitions. You can specify that `cmView` include the mask at the assignment statement, specifying complete coverage, with the `astglfull` keyword argument to the `-cm_report cmView` command-line option. This keyword argument and command-line option also tells `cmView` to include another mask under the declaration for the reg.

Line coverage summary information follows.

//	Module Coverage Summary		
	TOTAL	COVERED	PERCENT
lines	3	3	100.00

statements	3	3	100.00
blocks	2	2	100.00
ALWAYS	2	2	100.00

The file ends with assignment toggle coverage summary information:

	Module Coverage Summary		
	TOTAL	COVERED	PERCENT
assigns	3	1	33.33
assign bits	10	4	40.00
assign bits(0->1)	10	10	100.00
assign bits(1->0)	10	4	40.00
variables	3	1	33.33
variable bits	10	4	40.00
variable bits(0->1)	10	10	100.00
variable bits(1->0)	10	4	40.00

This summarizes the following:

- Three signals declared in the module, `out1`, `out2`, and `out3` but only 1, `out3` was covered. All its bits toggled from 0 → 1 and 1 → 0.
- There are ten bits in these three signals; only the four for `out3` were covered.
- All of the ten bits toggled from 0 → 1.
- Only four bits toggled from 1 → 0.

If there were continuous assignments to nets of the values in these regs, the assignments, both to the entire net, and its bit, would also be shown in this information.

This information is repeated for the variables in the module. It is the same information, for both nets and variables.

The cmView.long_l File

This section shows the `cmView.long_l` file as it pertains to either Verilog or VHDL source code.

Verilog cmView.long_l File

The following is an example `cmView.long_l` file from the intermediate data files written during the simulation of the Verilog source in [Example 2-1](#). This example file is interrupted in a number of places to explain its contents.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year  
  
//          LONG SOURCE LINE COVERAGE REPORT  
  
//*****  
//          MODULE INSTANCE COVERAGE  
  
// This section contains coverage for each instance of a module  
MODULE top  
  
FILE source file path name  
  
Line No      Coverage     Block Type  
  8           1            INITIAL  
  9-10        1  
  10.1         0            WAIT  
  14.1         1            ALWAYS  
//-----
```

For the instance of the top-level module `top`, line number 8 in annotated source file `top` is the first executable line in a block of code in an initial block. cmView reports the execution of lines 9 and 10, also in the same block as line 8 (so there is no new block type listed).

cmView notes that in line 10 is another block for a delay `wait` statement and indicates that it is not covered.

cmView notes that the assignment statement in line 14 is covered.

```
//          Module Coverage Summary  
//  
//      TOTAL    COVERED    PERCENT  
lines        4         4       100.00  
statements   6         5       83.33  
blocks       3         2       66.67  
  
ALWAYS       1         1       100.00  
INITIAL      1         1       100.00  
WAIT         1         0       0.00
```

The section ends with summary information about line, statement, and block coverage and then block coverage by type.

```
/-----  
MODULE top.dev1  
  
FILE source_file_path_name  
Line No    Coverage    Block Type  
 31        1           ROUTINE  
 37        1           ALWAYS  
 38        0           IF  
 40        1           ELSE  
 41        1  
 42        1           IF  
 42.1      0           MISSING_ELSE  
 44        1  
 44.1      1           WHILE  
 46        1           FOREVER  
 47        1           CASEITEM  
 47.1      1           WAIT  
 48        1           CASEITEM  
 48.1      1           WAIT  
 48.2      0           WHILE_FALSE
```

For module instance top.dev1:

- Line number 31 is in the block type ROUTINE that cmView uses for blocks of code on task and function definitions.
- Line number 38 is a \$display system task controlled by an if statement. VCS did not execute this task.
- Line number 42.1 indicates there is no else construct following an if construct.
- Line number 48.2 indicates there is no code in the remainder of the block that VCS executes when the while condition isn't true.

```
//          Module Coverage Summary  
//  
//          TOTAL    COVERED    PERCENT  
lines        10        9        90.00  
statements   13       12        92.31  
blocks       11       10        90.91  
  
ALWAYS       1         1        100.00  
CASEITEM     2         2        100.00  
FOREVER      1         1        100.00  
IF           2         1        50.00  
ELSE          1         1        100.00  
MISSING_ELSE 1         0         0.00  
ROUTINE      1         1        100.00  
WAIT          2         2        100.00  
WHILE         1         1        100.00  
WHILE_FALSE  1         0         0.00
```

The section on the module instance ends with summary information. Block types for “missing” code are, of course, never covered and are not included in the block count for total block and total blocks covered.

```
////////////////////////////////////////////////////////////////////////  
//          Total Module Instance Coverage Summary  
//  
//          TOTAL    COVERED    PERCENT  
lines        14        13        92.86  
statements   19       17        89.47  
blocks       14       12        85.71
```

ALWAYS	2	2	100.00
CASEITEM	2	2	100.00
FOREVER	1	1	100.00
IF	2	1	50.00
ELSE	1	1	100.00
MISSING_ELSE	1	0	0.00
INITIAL	1	1	100.00
ROUTINE	1	1	100.00
WAIT	3	2	66.67
WHILE	1	1	100.00
WHILE_FALSE	1	0	0.00

The report ends with a summary of all the line coverage data for all the module instances.

VHDL cmView.long_I File

```
// Synopsys, Inc.
//
// Generated by: cmView 6.2
// User: smart user
// Date: Day Month date hour:minute:second year
```

LONG SOURCE LINE COVERAGE REPORT

```
////////////////////////////////////////////////////////////////////////
// MODULE INSTANCE COVERAGE

// This section contains coverage for each instance of a module

Test Coverage Result: Total Coverage

MODULE TOP

FILE top.vhd

Line No      Coverage      Block Type
 43          1            VHDL_PROCESS
 44          1            VHDL_WAIT
 45          1            VHDL_PROCESS
 46          1            VHDL_WAIT
 49          1            VHDL_PROCESS
 50          1            VHDL_WAIT
 51          1            VHDL_WAIT
 52          1            VHDL_PROCESS
 55          1            VHDL_PROCESS
 56          1            VHDL_PROCESS
```

```

57      1      VHDL_WAIT
58      1
62      1      VHDL_PROCESS
64      1      VHDL_CASEITEM
64.1    1
66      0      VHDL_CASEITEM
66.1    0
68      0      VHDL_CASEITEM
68.1    0
70      0      VHDL_CASEITEM
73      1
74      0      VHDL_IF
//-----
//          Module Coverage Summary

          TOTAL   COVERED   PERCENT
lines        19       15      78.95
statements   22       16      72.73
blocks       9        7       77.78

VHDL_PROCESS     4        4      100.00
VHDL_CASEITEM    1        0       0.00
VHDL_IF          1        0       0.00
VHDL_WAIT         3        3      100.00

```

The first section is for the top-level module named `top`. There are a number of executable lines in this module instance and they are listed by line number. The number 1 in the Coverage column indicates that VCS MX executed the statement.

In this section, note the following:

- In some cases, `cmView` makes one line into two, as in the case of line 64 and 64.1. In the source code, this line is as follows:

```
pass_fail <= false; report " count_max = 15 " severity
note;
```

The report statement in this case statement makes the statement a separate block of code, and `cmView` interprets the report as a separate line, line 64.1.

- There is one `case` statement in the instance and VCS MX did not execute a number of case item statements.

The section ends with summary information.

```
//-----
MODULE TOP.U_COUNTER1
FILE counter.vhd

Line No      Coverage      Block Type
 22          1            VHDL_PROCESS
 23          1            VHDL_IF
 24          1            VHDL_IF
 25          1            VHDL_IF
 26          1
 27          1            VHDL_ELSE
 32          1
 33          1            VHDL_IF
 34          1
 35          1            VHDL_ELSIF
//-----
//           Module Coverage Summary

          TOTAL    COVERED    PERCENT
lines        10        10        100.00
statements   10        10        100.00
blocks       7         7         100.00

VHDL_PROCESS     1         1        100.00
VHDL_IF          4         4        100.00
VHDL_ELSIF        1         1        100.00
VHDL_ELSE         1         1        100.00
```

The next section is for a small module instance `top.u_counter1`. In this instance, there was complete line coverage. This means that this section would not appear in the `cmView.short_1` file.

```
*****
//           Total Module Instance Coverage Summary

          TOTAL    COVERED    PERCENT
lines        39        35        89.74
statements   42        36        85.71
blocks       23        21        91.30

VHDL_PROCESS     6         6        100.00
VHDL_CASEITEM    1         0        0.00
VHDL_IF          9         8        88.89
VHDL_ELSIF        2         2        100.00
VHDL_ELSE         2         2        100.00
VHDL_WAIT         3         3        100.00
```

The report ends with summary information about the line coverage of the entire design.

Reporting the Test That Caused Line Coverage

If you include the `-cm_report testlists` cmView command-line option and keyword argument, cmView adds information about which test file (intermediate results file) caused a line to be covered in the `cmView.long_l` and `cmView.long_ld` files.

cmView does this by first listing the test files and assigning them an index number. For example:

```
//          MERGED TESTS LIST
Test No.    Test Name      Coverage File Name
0           test1          ./simv.cm/coverage/verilog/test1
1           test2          ./simv.cm/coverage/verilog/test2
2           test3          ./simv.cm/coverage/verilog/test3
```

Then, it adds a column when it reports on an instance to indicate which test file or files, by index number, caused the line to be covered. For example:

Line No	Coverage	Block Type	Test Nos
6	1	ALWAYS	0,2,3
7	1	IF	0,2,3
8	1	ELSE	2,3
11	1	ALWAYS	0,1,3
12	1	IF	0,1,3
13	1	ELSE	3
16	1	ALWAYS	0,1,2
17	1	IF	0,1,2
18	1	ELSE	2
20	1	ALWAYS	0,1,2
21	1	IF	0,1,2
22	1	ELSE	2,3
24	1	ALWAYS	1,2,3
25	1	IF	1,2,3
26	1	ELSE	2,3

By default, cmView only does this for the first three test files that it finds by alphanumeric order of the test file names. You can instruct cmView to do this for more or fewer test files by replacing the `-cm_report testlists` cmView command-line option with the `-cm_report_testlists_max int` command-line option.

Verilog:

```
vcs -cm_pp -cm line -cm_report_testlists_max 4
```

VHDL:

```
cmView -b -cm line -cm_report_testlists_max 4
```

The cmView.long_Id File

The `cmView.long_Id` file is similar to the `cmView.long_1` file except that it has separate sections for module definitions instead of module instances. If there are multiple instances of a module definition, and if in any of these instances a line is covered, then the section for the module definition reports it as covered.

The following is an excerpt from the `cmView.long_Id` that corresponds to the previous example Verilog `cmView.long_1` file:

```
/-----
MODULE dev

FILE source_file_path_name

Line No      Coverage      Block Type
 31          1            ROUTINE
 37          1            ALWAYS
 38          0            IF
 40          1            ELSE
 41          1
 42          1            IF
 42.1        0            MISSING_ELSE
```

```

44          1
44.1       1      WHILE
46          1      FOREVER
47          1      CASEITEM
47.1       1      WAIT
48          1      CASEITEM
48.1       1      WAIT
48.2       0      WHILE_FALSE

```

In this example, the coverage information is the same as it is in the section for the `top.dev1` instance in the `cmView.long_1` file because there is only one instance of this module.

The `cmView.hier_1` File

The following is the `cmView.hier_1` file generated from the intermediate data files written during the simulation of the Verilog source in [Example 2-1](#). This file contains summary information in terms of subhierarchies. Because module instance `top.dev1` is hierarchically under the top-level module `top`, the data from `top.dev1` is included in the information for top-level module `top`.

```

// Synopsys, Inc.
//
// Generated by: cmView version_number
// User: Intelligent User
// Date: Day Month date hour:minute:second year

//*****MODULE HIERARCHICAL INSTANCE COVERAGE SUMMARY

// This section summarizes coverage by providing statistics for each
// instance of a module. The statistics take into account all sub-hierarchies
// under the instance.

Test Coverage Result: Total Coverage

Module Name          Blks   Blks      Stmnts  Stmnts      Lines  Lines      (%)
                           (%)           (%)        (%)           (%)        (%)
top                  85.71  12/14     89.47  17/19     92.86  13/14
top.dev1             90.91  10/11     92.31  12/13     90.00  9/10

//*****

```

Line Coverage

```

//          Total Module Instance Coverage Summary

          TOTAL    COVERED    PERCENT
lines        14        13      92.86
statements   19        17      89.47
blocks       14        12      85.71

  ALWAYS        2         2      100.00
  CASEITEM      2         2      100.00
  FOREVER       1         1      100.00
  IF            2         1      50.00
  ELSE           1         1      100.00
  MISSING_ELSE   1         0       0.00
  INITIAL        1         1      100.00
  ROUTINE        1         1      100.00
  WAIT           3         2      66.67
  WHILE          1         1      100.00
  WHILE_FALSE    1         0       0.00

```

The cmView.mod_1 File

The following is the `cmView.mod_1` file generated from the intermediate data files written during the simulation of the Verilog source in [Example 2-1](#). This file contains summary information for module instances. Coverage data is used just for the instance and not for the instances hierarchically under these instances.

```

// Synopsys, Inc.
//
// Generated by: cmView version_number
// User: Intelligent User
// Date: Day Month date hour:minute:second year
// *****
//          MODULE INSTANCE COVERAGE SUMMARY

// This section summarizes coverage by providing statistics for each
// instance of a module. The statistics do not take into account any
// sub-hierarchy under the instance.

Test Coverage Result: Total Coverage

Module Name          Blks  Blks      Stmnts Stmnts      Lines  Lines
                      (%)   (%)      (%)   (%)      (%)   (%)
top                 66.67 2/3      83.33 5/6      100.00 4/4
top.dev1            90.91 10/11    92.31 12/13    90.00 9/10
// *****

```

```

//          Total Module Instance Coverage Summary

          TOTAL    COVERED    PERCENT
lines        14        13      92.86
statements   19        17      89.47
blocks       14        12      85.71

ALWAYS       2         2      100.00
CASEITEM     2         2      100.00
FOREVER      1         1      100.00
IF           2         1      50.00
ELSE          1         1      100.00
MISSING_ELSE 1         0       0.00
INITIAL      1         1      100.00
ROUTINE      1         1      100.00
WAIT          3         2      66.67
WHILE         1         1      100.00
WHILE_FALSE  1         0       0.00

```

The report ends with summary information on the entire design.

The cmView.mod_Id File

The `cmView.mod_Id` file is similar to the `cmView.mod_1` file except that the coverage information is organized by module definition instead of by module instance. The following `cmView.mod_Id` file corresponds to the `cmView.mod_1` file in the preceding section:

```

// Synopsys, Inc.
//
// Generated by: cmView version_number
// User: Intelligent User
// Date: Day Month date hour:minute:second year

//*****MODULE DEFINITION COVERAGE SUMMARY

// This section summarizes coverage by providing statistics for each
// module definition. The coverage is cumulative over all the instances
// of the module

```

Test Coverage Result: Total Coverage

Line Coverage

2-115

```

Module Name          Blks    Blks      Stmtns  Stmtns      Lines  Lines
                   (%)      (%)      (%)      (%)      (%)      (%)

top                66.67  2/3       83.33  5/6      100.00 4/4
dev                90.91 10/11     92.31 12/13     90.00 9/10

//*****Total Module Definition Coverage Summary

//          Total  Module  Definition  Coverage  Summary

          TOTAL   COVERED   PERCENT
lines        14       13       92.86
statements   19       17       89.47
blocks       14       12       85.71

          ALWAYS    2         2       100.00
          CASEITEM  2         2       100.00
          FOREVER   1         1       100.00
          IF         2         1       50.00
          ELSE       1         1       100.00
          MISSING_ELSE 1         0       0.00
          INITIAL   1         1       100.00
          ROUTINE   1         1       100.00
          WAIT       3         2       66.67
          WHILE      1         1       100.00
          WHILE_FALSE 1         0       0.00

```

The cmView.short_l File

The `cmView.short_l` file contains information for all module instances for which there is not 100% coverage and summary information about the entire design. The following example has explanatory comments inserted in it:

```

// Synopsys, Inc.
//
// Generated by: cmView version_number
// User: Intelligent User
// Date: Day Month date hour:minute:second year

//          SHORT SOURCE LINE COVERAGE REPORT

//*****MODULE INSTANCE COVERAGE

```

```
// This section contains coverage for each instance of a module
MODULE top
FILE source_file_path_name

Line No      Coverage      Block Type
10.1          0            WAIT
```

The report begins with what is not covered in the top-level module.

```
MODULE top.dev1
FILE /projects/CovMet/line/exp3.v

Line No      Coverage      Block Type
38           0            IF
42.1          0            MISSING_ELSE
48.2          0            WHILE_FALSE
```

Here is what is not covered in the module instance top.dev1:

```
//*****
//          Total Module Instance Coverage Summary

          TOTAL    COVERED    PERCENT
lines        14        13        92.86
statements   19        17        89.47
blocks       14        12        85.71

ALWAYS        2         2        100.00
CASEITEM      2         2        100.00
FOREVER       1         1        100.00
IF            2         1        50.00
ELSE          1         1        100.00
MISSING_ELSE 1         0        0.00
INITIAL       1         1        100.00
ROUTINE       1         1        100.00
WAIT          3         2        66.67
WHILE         1         1        100.00
WHILE_FALSE   1         0        0.00
```

The report ends with summary information about the entire design.

The cmView.short_Id File

The cmView.short_Id file is similar to the cmView.short_1 file, except that its contents is organized by module definition instead of by module instance.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year  
  
SHORT SOURCE LINE COVERAGE REPORT  
  
//*****  
// MODULE DEFINITION COVERAGE  
  
// This section contains coverage for module definitions.  
// The coverage is cumulative over all the instances of the module  
  
Test Coverage Result: Total Coverage  
  
MODULE top  
  
FILE source_file_path_name  
  
Line No      Coverage      Block Type  
 10.1        0            WAIT  
MODULE dev  
  
FILE /d1/pmcgee/projects/CovMet/line/exp3.v  
  
Line No      Coverage      Block Type  
  38          0            IF  
 42.1        0            MISSING_ELSE  
 48.2        0            WHILE_FALSE  
  
//*****  
//           Total Module Definition Coverage Summary  
  
TOTAL      COVERED      PERCENT  
lines       14          13          92.86  
statements  19          17          89.47  
blocks      14          12          85.71  
  
ALWAYS      2            2            100.00  
CASEITEM    2            2            100.00
```

FOREVER	1	1	100.00
IF	2	1	50.00
ELSE	1	1	100.00
MISSING_ELSE	1	0	0.00
INITIAL	1	1	100.00
ROUTINE	1	1	100.00
WAIT	3	2	66.67
WHILE	1	1	100.00
WHILE_FALSE	1	0	0.00

Viewing Line Coverage with the Coverage Metrics GUI

cmView is also the graphical user interface (GUI) for VCS and VCS MX coverage metrics. It can display graphical representations of the coverage information recorded by VCS and VCS MX.

Perform the following procedure to view the line coverage results VCS or VCS MX recorded for the simulation of the source code in [Example 2-1](#):

1. Enter the following command line:

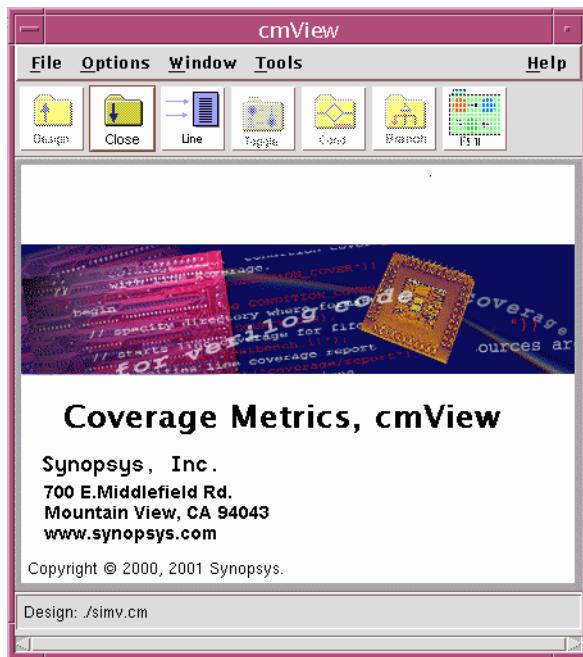
Verilog:
 vcs -cm_pp gui -cm line

VHDL:
 cmView -cm line

The above command line instructs cmView to perform the following:

- a. Open the main window (see [Figure 2-2](#)):

Figure 2-2 The Main cmView Window



- b. Read the design file to learn about the design and its hierarchy.
- c. Read the intermediate data files for line coverage.

Note:

At compile-time, if you specified a different coverage metrics database with the `-cm_dir` compile-time option, also include this option on the `cmView` (or `vcs -cm_pp gui`) command line.

4. Click the Line Coverage toolbar button (see below) to display the Statement Coverage window (see [Figure 2-3](#)):

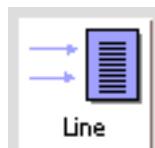
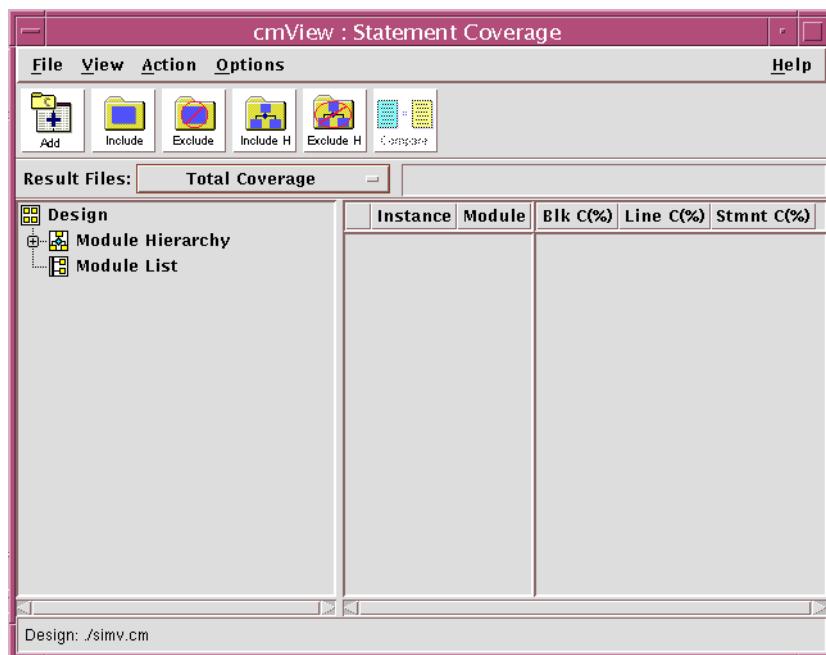


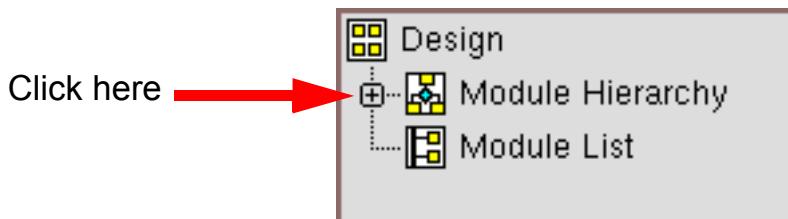
Figure 2-3 Statement Coverage Window



You use this window to view line coverage data from the intermediate data files written by VCS or VCS MX. You use the left pane to open the design hierarchy and the right pane to display coverage information.

Statement coverage and line coverage are practically identical terms (except when one statement is on two lines or two statements are on one line), therefore, VCS and VCS MX uses line and statement coverage interchangeably. This window is called the Statement Coverage window.

- By default, cmView displays coverage summary information on a per module instance basis, therefore, the Statement Coverage window needs a way to display the design hierarchy so that you can select module instances to examine their line coverage. Click the plus symbol (+) next to the icon for the Module Hierarchy.



Clicking on the plus symbol (+) displays the top-level modules (in this case one) in the design.

The screenshot shows the expanded 'Module Hierarchy' tree. The 'top' module is selected, indicated by a black border around its node. The table on the right remains the same, showing 100% coverage for the top module.

	Instance	Module	Blk C(%)
	top	top	100.00

The green results bar indicates 100% coverage.

- Click the plus symbol (+) next to the icon for the top-level module `top`. This displays module instance `top.dev1`.

The screenshot shows the expanded 'Module Hierarchy' tree with 'top.dev1' selected. The table on the right shows two rows: the top row for 'top' has 100.00% coverage, and the bottom row for 'dev1' has 77.78% coverage.

	Instance	Module	Blk C(%)
	top	top	100.00
	dev1	dev	77.78

The yellow results bar for dev1 indicates only partial coverage.

7. Click on the yellow results bar for module instance dev1.

This displays the annotated window. The following is a Verilog example in this window:

The screenshot shows the 'cmView - Statement: Instance top.dev1' window. The top pane displays the Verilog source code for module 'dev'. The bottom pane shows coverage statistics for instance 'dev1'.

Verilog Source Code (Top Pane):

```
20 module dev (out,clock,in);
21   output [8:0] out;
22   input clock;
23   input [8:0] in;
24   reg [8:0] out;
25   reg en;
26
27   task splitter;
28     input [8:0] tin;
29     output [8:0] tout;
30     begin
31       tout = tin/2;
32     end
33   endtask
34
35   initial @ (posedge clock)
36   begin
37     if (in % 2 != 0)
38       --> $display("error cond, input not even");
```

Coverage Statistics (Bottom Pane):

	TOTAL	COVERED	PERCENT
Lines	14	12	85.71
Statements	12	10	83.33
Blocks	9	7	77.78
NONE	0		
ALWAYS	1	1	100.00

In its top pane, this window displays an annotated source file for the module definition of the module instance. When it appears, uncovered lines are highlighted in red.

The following is a VHDL example:

The screenshot shows a window titled "cmView - Statement: Instance TB". The menu bar includes "File", "View", "Options", "Help", and a "Module Files" dropdown set to "tb.vhd". Below the menu is a toolbar with icons for "Top of File", "Prev Line", "Next Line", "End of File", "Go Parent", and "Go Child". The main area displays VHDL code with line numbers 59 through 77. Lines 66, 68, 70, 74, and the entire line 75 are highlighted in red, indicating they were not executed. The bottom section shows a "Tests" tab with a list of failing test cases: 66, 68, 70, and 74, each with a red background. The status bar at the bottom shows the path "/remote/dvuser/tareqa/test/counter/tb.vhd".

```
59      end process;
60      process (pass_fail)
61      begin
62          case count_max is
63              when 15 =>
64                  pass_fail <= false; report " count_max = 15 " severity note;
65              when 31 =>
66                  --> pass_fail <= false; report " count_max = 31 " severity note;
67              when 63 =>
68                  --> pass_fail <= false; report " count_max = 63 " severity note;
69              when others =>
70                  --> pass_fail <= true;
71          end case;
72
73          if pass_fail then
74              --> report "Illegal count_max value. Exiting" severity error;
75      end if;
76  end process;
77
```

Statistics | Information | Tests | Go To

66 --> pass_fail <= false; report " count_max = 31 " severity note;
68 --> pass_fail <= false; report " count_max = 63 " severity note;
70 --> pass_fail <= true;
74 --> report "Illegal count_max value. Exiting" severity error;

/remote/dvuser/tareqa/test/counter/tb.vhd

Lines Ignored by Pragmas

If a line is excluded from coverage by pragmas, it is highlighted in white in the annotated window:

```
10
11      always @ (posedge r1)
12      //synopsys translate_off
13  XX> r2=r1;
14      //synopsys translate_on
15
```

The toolbar buttons are as follows:



This button takes you to the top of the file.



This button takes you to the next uncovered line and changes the highlighting of that line to magenta.



This button takes you to the previous uncovered line.



This button takes you to the bottom of the file.



Opens another annotated window containing the module definition of the instance that is one level up in the design hierarchy.



Opens a menu of “sub-modules” — modules instantiated in the module definition displayed in the annotated window. Selecting one of these instances opens an annotated window for the instance.

Manipulating the Summary Pane

By default, the summary pane displays the following information when a Statement Coverage window is opened:

- The first column contains icons which indicate special details about the particular instance or module. These icons are described in [Table 2-1](#).

Table 2-1 Icon Indicators in the Statement Coverage Window

Icon	Description
	A green axe with a red line through it indicates that the instance contains no executable lines of code.
	A document icon indicates that an annotated window has been opened for the specified instance or module.
	A blue circle with a red line through it indicates that the instance has been excluded.

- The second column contains the instance name.
- The third column contains the module name.
- The fourth column contains the percentage coverage for total blocks.

Double-clicking on any row in the statement coverage summary pane displays an Annotated window with detailed statement coverage data for the specific instance or module. The Annotated window is described in [“The Annotated Window”](#).

Detailed View

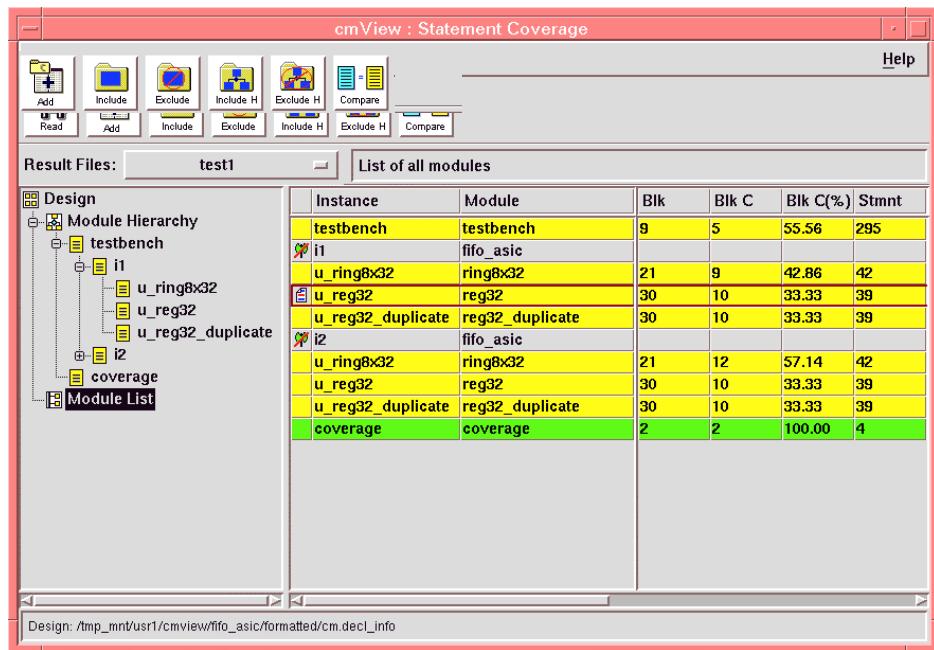
To obtain a detailed report of line or statement coverage, choose **View > Detailed**.

The detailed statistics for statement coverage are displayed as shown in [Figure 2-4](#) (Verilog) and [Figure 2-5](#) (VHDL).

In addition to the default instance and module names in the detailed report, the following information is also displayed and arranged in columns as follows:

- Total blocks
- Number of blocks covered
- Percentage of blocks covered
- Total statements
- Number of statements covered
- Percentage of statements covered

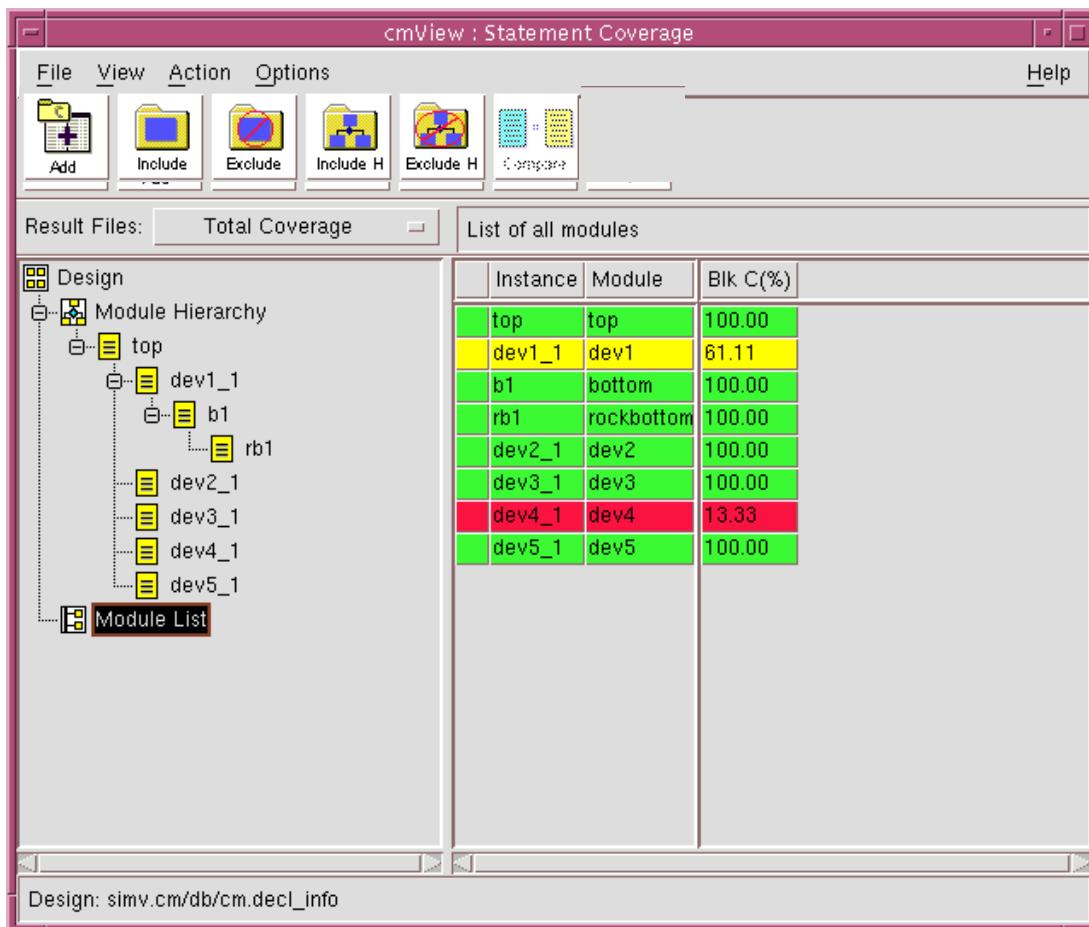
Figure 2-4 Detailed View of Coverage Statistics (Verilog)



Line Coverage

2-128

Figure 2-5 Detailed View of Coverage Statistics (VHDL)



The Annotated Window

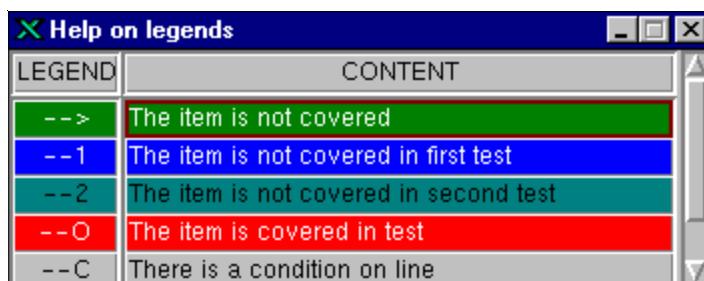
The Annotated window allows you to visually compare your original Verilog source files with copies of those files that have been annotated by VCS to show you which lines of code have not been covered.

The Annotated window displays an annotated listing of the original Verilog source file with all uncovered lines highlighted. The color used to highlight uncovered lines may be changed using the Colors tab in the User Preferences dialog box as described in “[User Preferences](#)” on page 8-461.

If the annotated listing is used for Incremental Coverage, lines covered incrementally by the first test are highlighted in red. If the annotated listing is used for Diff Coverage, lines uncovered by both tests are displayed in red. However, you can change the highlight color to display lines uncovered by the two tests in different colors by using the Colors tab in the User Preferences dialog box.

Indicators displayed in columns 9-11 to the left of the uncovered lines are explained in the legend shown in [Figure 2-6](#). Choose **Help > Marks** to display the legend at any time in the current cmView session.

Figure 2-6 Description of Uncovered Lines Indicators



Toolbar

Toolbar buttons are used to move from one uncovered line to the next. These buttons are described in [Table 2-2](#). When using these buttons to navigate through the file, the currently selected line is always displayed in magenta. You can change this color using the Colors tab in the User Preferences dialog box.

Table 2-2 Toolbar buttons in the Annotated Window

Button	Description
 Top of File	This button takes you to the top of the file.
 Next Line	This button takes you to the next uncovered line.
 Prev Line	This button takes you to the previous uncovered line.
 End of File	This button takes you to the bottom of the file.

The Summary Folder

The bottom portion of the Annotated window contains a folder with four tabs containing further details about the module as shown in [Figure 2-7](#) and [Figure 2-9](#).

The first tab in this folder is the Statistics tab. Clicking on this tab displays more detailed coverage statistics about the module, as shown in the bottom portion of [Figure 2-7](#) and [Figure 2-9](#). A complete breakdown of all the different kinds of statements and their coverage is provided, in addition to the summaries of the number of lines, statements, and blocks, in the module.

The second tab in the folder is the Information tab. Clicking on this tab displays some general information about this instance or module as shown in [Figure 2-9](#). The last row indicates the test for which this coverage data is displayed.

Figure 2-7 Annotated Source File (Verilog)

The screenshot shows the cmView software interface with the title bar "cmView – Statement: Instance testbench.i1.u_reg32". The menu bar includes "File", "View", "Options", and "Help". The toolbar contains icons for "Module Files: reg32.v" (highlighted with a red box), "Top of File", "Prev Line", "Next Line", "End of File", "Go Parent", and "Go Child". The main code editor window displays the following Verilog code:

```

1 // Eight 32-bit registers are created, which can be
2 // written and read at the same time.
3 //
4 // Inputs:      data_in      - input data for register
5 //              reg_wr_en   - if asserted, register
6 //              rd_addr     - address of register
7 //              wr_addr     - address of register
8 //              clk          - clock
9 //              reset_1     - clears register if asserted
10 // Outputs:     reg_data_out - output register state
11 //
12 'timescale 1 ns / 1 ns
13
14 'include "reg32_duplicate.v"
15
16 module reg32 (data_in, reg_wr_en, rd_addr, wr_addr,
17                 reg_data_out);
18
19     'include "reg32.vh"
20
21     input  [REG32_DATA_BUS_SIZE-1:0] data_in;
22     input  [REG32_ADDR_BUS_SIZE-1:0] rd_addr;
23     input  [REG32_ADDR_BUS_SIZE-1:0] wr_addr;
24     input                           reg_wr_en;
25
26 endmodule

```

Below the code editor is a statistics panel with tabs for "Statistics", "Information", "Tests", and "Go To". The "Statistics" tab is selected, showing the following data:

	TOTAL	COVERED	PERCENT
Lines	39	19	48.72
Statements	39	19	48.72
Blocks	30	10	33.33
ALWAYS	2	2	100.00
CASEITEM	26	6	23.08

The status bar at the bottom of the window displays "design/reg32.v".

Figure 2-8 Annotated Source File (VHDL)

The screenshot shows the 'cmView - Statement: Instance TB' application window. The menu bar includes 'File', 'View', 'Options', and 'Help'. The toolbar contains icons for 'Top of File', 'Prev Line', 'Next Line', 'End of File', 'Up Parent', and 'Go Child'. The main area displays a VHDL source file named 'tb.vhd' with line numbers 59 to 77. Lines 66, 68, 70, and 74 are highlighted in pink, while lines 66, 68, 70, and 74 in the status bar are highlighted in red. Below the code editor are tabs for 'Statistics', 'Information', 'Tests', and 'Go To', with 'Information' selected. The status bar at the bottom shows the path '/remote/dvuser/tareqa/test/counter/tb.vhd'.

```

59      end process;
60      process (pass_fail)
61      begin
62          case count_max is
63              when 15 =>
64                  pass_fail <= false; report " count_max = 15 " severity note;
65              when 31 =>
66                  pass_fail <= false; report " count_max = 31 " severity note;
67              when 63 =>
68                  pass_fail <= false; report " count_max = 63 " severity note;
69              when others =>
70                  pass_fail <= true;
71          end case;
72
73          if pass_fail then
74              --> report "Illegal count_max value. Exiting" severity error;
75      end if;
76  end process;
77

```

Figure 2-9 Information Tab on the Summary Folder in the Annotated Window

The screenshot shows the 'Information' tab in the summary folder. It displays a table with the following data:

Instance:	testbench.i1.u_reg32
Module:	reg32
Design:	/tmp_mnt/usr1/cmview/fifo_asic/formatted/cm.decl_info
Coverage:	Statement
Source Files:	design/reg32.v
Test:	test1

The third tab in the folder is the Tests tab. Clicking on this tab displays a detailed summary of all the tests read thus far, as shown

in [Figure 2-10](#). Detailed information is also provided to describe the makeup of total, incremental, and diff coverage.

Figure 2-10 Tests Tab on the Summary Folder in the Annotated Window

Tests	
Incremental & Diff	Total Coverage <i>/tmp_mnt/usr1/cmview/fifo_asic/coverage/test0</i>
Tests Added	<i>/tmp_mnt/usr1/cmview/fifo_asic/coverage/test0</i>
Tests Read	<i>/tmp_mnt/usr1/cmview/fifo_asic/coverage/test0</i>
	<i>/tmp_mnt/usr1/cmview/fifo_asic/coverage/test1</i>

The fourth tab in the folder is the Go To tab. Clicking on this tab displays a list of all the uncovered lines in the annotated file as shown in [Figure 2-11](#). Clicking any line in this list moves the annotated file in the top pane so that the specified line is visible.

Figure 2-11 Go To Tab Consolidated Listing of all Uncovered Lines

The screenshot shows the 'cmView – Statement: Instance testbench.i1.u_reg32' window. The main area displays Verilog code for a register file instance. The 'Go To' tab is selected in the bottom navigation bar, showing a list of uncovered lines from line 64.1 to 73.1. The code listing includes lines like:

```
66.1           reg_1 = data_in;
67.1           4'b1010;
67.1           reg_2 = data_in;
68.1           4'b1011;
68.1   -->     reg_3 = data_in;
69.1           4'b1100;
69.1   -->     reg_4 = data_in;
70.1           4'b1101;
70.1   -->     reg_5 = data_in;
71.1           4'b1110;
71.1   -->     reg_6 = data_in;
72.1           4'b1111;
72.1           reg_7 = data_in;
73.1           default;
73.1   -->     ;
74           end
75
76
77           always @ (negedge clk)          // generate output
78           begin
79           case (rd_addr[REG32_ADDR_BUS_SIZE-1:0])
80           3'b000:
80.1             reg_data_out = reg_0;
81           3'b001:
81.1   -->     reg_data_out = reg_1;
82           3'b010:
```

The 'Go To' tab also lists the following uncovered lines:

64.1	-->	;
68.1	-->	reg_3 = data_in;
69.1	-->	reg_4 = data_in;
70.1	-->	reg_5 = data_in;
71.1	-->	reg_6 = data_in;
73.1	-->	;

The View Menu

You can use the View menu to traverse the module hierarchy of the particular instance, if appropriate.

For example, to go to the parent module, click on the Parent button (see below), or choose **View > Open Parent Module** from the menu.



A new Annotated window with annotated listings for the parent appears. If the parent module has no executable lines, both the button and the menu item are disabled (grayed out) and are not selectable.

To go to any of the sub-modules of the current module, click on the Child button (see below), or choose **View > Open Child Module** from the menu. A list with all immediate descendants of the current module appears.



Clicking on any member in this list displays a new Annotated window with annotated listings for the sub-module. If the module has no sub-modules, or if the sub-modules contain no executable lines, both the button and the menu item are disabled (grayed out) and are not selectable.

Excluding Lines From Coverage Calculations

You can use the Annotated window to select a set of lines that you do not want included in coverage analysis. For example, you might want to exclude a set of uncovered lines in a well-tested and reused module definition in which reduced coverage should not reduce the overall coverage of a new design.

To do this using the Annotated window, select uncovered lines and then instruct cmView to recalculate the percentages of line coverage without counting these lines.

See [Figure 2-12](#) for an example.

Figure 2-12 Selecting Lines To Exclude From Coverage

The screenshot shows the Annotated window with Verilog code and coverage statistics. The code is as follows:

```
11      always @ (posedge r1)
12      begin
13      case(r1)
14      ==> 0 : r2=r1;
15      1 : r2=0;
16      ==> 1'bz : r2=1'bz;
17      ==> default : r2=1'bz;
18      endcase
19      ==> #11 $display("r1=%0d r2=%0d",r1,r2);
19      #(11)
19.1    ==> $display("r1=%0d r2=%0d", r1, r2);
20      end
21      endmodule
```

Below the code, there is a navigation bar with tabs: Statistics, Information, Tests, and Go To. The Statistics tab is selected. A table displays coverage statistics:

test	TOTAL	COVERED	PERCENT
Lines	9	6	66.67
Statements	13	9	69.23
Blocks	10	6	60.00
NONE	0		
ALWAYS	1	1	100.00

[Figure 2-12](#) illustrates an Annotated window for a module instance with only 60% line coverage for the blocks of code. Note, that the uncovered lines are in an always block. In this example, you do not want these lines to lower the coverage percentages; instead, you want to exclude them.

To do exclude these lines, perform the following:

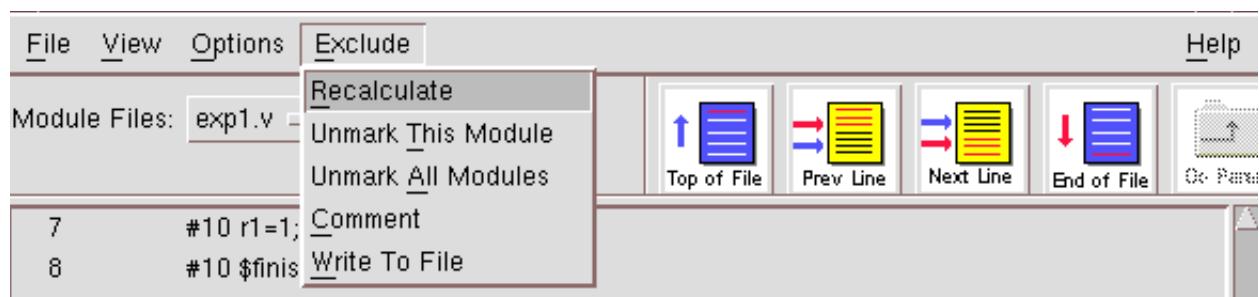
1. Click on the lines you want to exclude:

```
7      #10 r1=1;
8      #10 $finish;
9      end
10
11      always @ (posedge r1)
12      begin
13      case(r1)
14      M 0 : r2=r1;
15      1 : r2=0;
16      M 1'bz : r2=1'bz;
17      M default : r2=1'bz;
18      endcase
19      ==> #11 $display("r1=%0d r2=%0d",r1,r2);
19.1   M      $display("r1=%0d r2=%0d", r1, r2);
20      end
21      endmodule
```

Note:

After you click on an uncovered line, the ==> symbol for the line changes to M, indicating that it is marked for exclusion.

2. Choose **Exclude > Recalculate** from the menu:



The Annotated window display changes to the following:

```
11      always @ (posedge r1)
12      begin
13          case(r1)
14              X 0 : r2=r1;
15              1 : r2=0;
16              X 1'bz : r2=1'bz;
17              X default : r2=1'bz;
18          endcase
19          #11 $display("r1=%0d r2=%0d",r1,r2);
19          #(11)
20              X $display("r1=%0d r2=%0d", r1, r2);
21      end
21      endmodule
```

Statistics	Information	Tests	Go To
test	TOTAL	COVERED	PERCENT
Lines	5	5	100.00
Statements	8	8	100.00
Blocks	6	6	100.00
NONE	0		
ALWAYS	1	1	100.00

The M symbol is replaced by an x, and the lines are no longer highlighted in red. They are excluded from line coverage. The module instance now has 100% line coverage.

Including Excluded Lines

If you decide that an excluded line should not have been excluded, you can reverse the exclusion as follows:

1. Click on the excluded line:

```
19      #11 $display("r1=%0d r2=%0d",r1,r2);
19      #(11)
19.1  XM   $display("r1=%0d r2=%0d", r1, r2);
20      end
21      endmodule
```

The symbol changes from X to XM, for a marked excluded line.

2. Choose **Exclude > Recalculate** from the menu:

The symbol changes from XM to XE> and the statistics change to show the new calculations.

Using a File to Exclude Lines

You can use cmView to write a file containing the excluded lines, as indicated by the X or M symbol. When you start cmView again, the lines in this file are excluded from coverage.

To use cmView to write a file containing the excluded lines:

1. Choose **Exclude > Write To File** from the menu.

This displays the Specify File For Excluded Lines dialog box.

2. Enter a name and location for this file using the dialog box, then click OK.

If this is a new file, the process of creating this file is complete.

If this is not a new file, this opens the Specify Write Mode dialog box and displays a message similar to the following:

```
File exclude_file_path_name is Not Empty
Overwrite or Append?
```

You have a choice of overwriting the file, where the file will only have entries for what you have excluded now, or adding new entries for newly excluded files to the file.

3. Select Append or Overwrite.

Append adds new entries. Overwrite deletes previous entries and only makes entries for what you excluded now.

Appended entries for the same module instance are accumulated or merged together.

Note:

This feature works in similar ways as in toggle and branch coverage, and you can use the same file to exclude lines, signals, and branches.

In a subsequent session of the cmView GUI, you can use the `-cm_elfile` option to specify excluding the lines in the file when the session starts. For example:

```
vcs -cm_pp gui -cm line -cm_elfile exclude_file
```

The excluded lines have the X symbol and they are not counted in the coverage statistics.

The `exclude_file` can also be passed to cmView batch mode to write reports as follows:

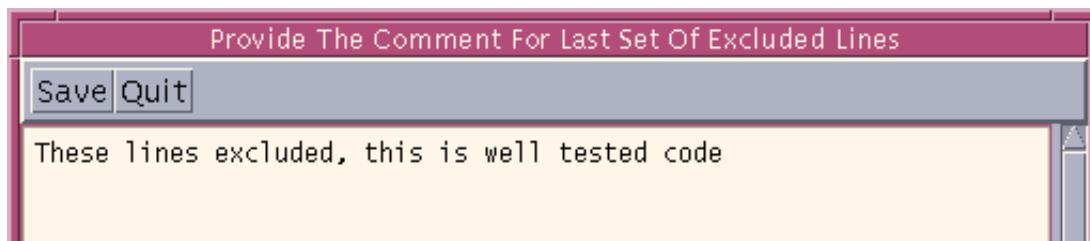
```
vcs -cm_pp -cm line -cm_elfile exclude_file
```

The excluded lines are marked with an X in the reports.

Comments in the File

If you want to include documentation, you can add comments to the output file containing excluded lines. To do so, first recalculate the exclusion of these lines, then perform the following:

1. Choose **Exclude > Comment** from the menu.
This displays the Provide The Comment For Last Set Of Excluded Lines dialog box.
2. Enter a comment, such as the following:



3. Click the Save button.
4. Now choose **Exclude > Write To File** from the menu and specify the file using the Specify File For Excluded Lines dialog box.

The following is an example of this file with comments:

```
//=====
// Lines Excluded From Coverage
// Generated By User: smart_user
// Date: Mon Jan  6 13:13:10 2003
//=====

FILE: /net/design/exp1.v
MODULE: test

COMMENT: Well tested code so we cal exclude from coverage.
LINE: 14

COMMENT: Well tested code so we cal exclude from coverage.
```

LINE: 16

COMMENT: Well tested code so we can exclude from coverage.
LINE: 17

COMMENT: Well tested code so we can exclude from coverage.
LINE: 19.1

Clearing Marked Lines

After marking a line for exclusion, you can clear it by clicking again on the line. The symbol will then change from M back to ==>. This action causes the line to be counted when you recalculate coverage.

To clear marks from multiple lines, use the following menu commands:

- **Exclude > Unmark This Module**
- **Exclude > Unmark All Modules**

This changes M symbols to ==> and MX symbols to X.

Inconsistency in the Exclude File

If the exclude file you specify with the -cm_elfile option has an entry to exclude a line from coverage, but after revising the design, the current coverage results show that the line is covered, cmView does the following depending on if you are using the batch report writing mode or GUI mode:

- In batch mode cmView writes to your screen:

```
//Error: Module version has changed since el-file was  
created  
//The excluded information is ignored
```

```
//-----  
//module exclude_file
```

cmView ignores the entry in the exclude file and the reports show the line as covered.

- In GUI mode, cmView opens the Inconsistencies Detected In Elf file dialog box to display the same message that it would display in batch report writing mode.

The dialog box has two buttons:

- Ignore
Enables you to continue the cmView GUI session. The Statement Coverage summary window for the instance does not show the covered line as excluded.
- Abort
Ends the cmView GUI session.

Excluding Covered Lines

You can exclude covered lines in the exclude file. If you do and then start cmView again, entering the `-cm_elfile` option with the exclude file containing entries for covered lines, cmView does the following depending on if you are using the batch report writing mode or GUI mode.

- In batch report writing mode, cmView reports the line as covered and puts an A symbol to the left of the line in the report, indicating that there was an attempt to exclude a covered line that was ignored. For example:

Line No	Coverage	Block Type
6	1	INITIAL
7-9	1	

	9.1	1	WAIT
A	13	1	ALWAYS
	14	1	IF
A	15	1	CASEITEM
	16	0	CASEITEM
	17	0	CASEITEM
	17.1	0	MISSING_ELSE

- In GUI mode, cmView opens the Inconsistencies Detected In Elf file dialog box to display a message similar to the following:

```
//Warning: Attempt to exclude the covered line, attempt is
ignored
//-----
---
// module line: line_number elfile: exclude_file
```

The dialog box has two buttons:

- Ignore
Enables you to continue the cmView GUI session. The Line Coverage summary window for the instance shows the A symbol to the left of the line that was covered to indicate there was an attempt to exclude it (see [Figure 2-13](#)).
- Abort
Ends the cmView GUI session.

Figure 2-13 Covered Lines Where There was an Attempt to Exclude

```
6      r1=1;
7      r2=1;
8      A  r3=1;
9      #100 $finish;
10     end
11
12     always @ (r1 or r2 or r3)
13     A  if (r1 && r2)
14         case (r3)
15         A  1 : r5 = 1;
16     ==>  0 : r5 = 0;
17     ==>  default:$display("\n\n flag problem\n\n");
17.1 ==> MISSING_ELSE
18     endcase
```

3

Toggle Coverage

Toggle coverage determines whether the scalar signals and each bit of the vector signals in your design had $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions during the last simulation.

You instruct VCS and VCS MX to compile for toggle coverage with the `-cm tg1` compile-time option and keyword argument. You specify monitoring for toggle coverage with the `-cm tg1` runtime option and keyword argument.

This chapter describes:

- “[Supported Data Types](#)”
- “[Realtime Control of Toggle Coverage](#)”
- “[Realtime Control of Toggle Coverage](#)”
- “[Limiting Toggle Coverage to Ports Only](#)”

- “Limiting Toggle Coverage to Uncovered Signals”
- “Excluding and Including Signals in Toggle Coverage”
- “Toggle Coverage Glitch Suppression”
- “Using Pragmas to Limit Toggle Coverage”
- “Toggle Coverage Reports”
- “Viewing Toggle Coverage With the Coverage Metrics GUI”

Note:

By default, toggle coverage does not display or report $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions where the signal returns to its original value during the same time step, in other words, a glitch during which zero simulation time passes.

Using glitch suppression for non-zero time glitches can alter toggle coverage and also line and condition coverage (see “[Toggle Coverage Glitch Suppression](#)” and “[Using Glitch Suppression](#)” on page 1-34).

Supported Data Types

If your design contains memories, Verilog-2001 multi-dimensional arrays (MDAs), or unpacked SystemVerilog MDAs, and you want toggle coverage for these memories and MDAs, include the `-cm_tgl mda` compile-time option and keyword argument when you compile for toggle coverage. This option and argument are not required for packed SystemVerilog MDAs.

VCS MX toggle coverage supports the following VHDL data types:

bit	bit_vector	std_logic
std_ulogic	std_logic_vector	std_ulogic_vector
signed	unsigned	

Realtime Control of Toggle Coverage

There is a realtime API for Verilog coverage. This API includes the `$cm_coverage` system function that enables you to disable and enable all types of coverage, including toggle coverage (see “[The \\$cm_coverage System Function](#)” on page 9-472).

For real-time control of VHDL toggle coverage, you can enter the following commands at the VCS MX interactive command prompt:

```
coverage -tgl off  
Disables monitoring for toggle coverage.
```

```
coverage -tgl on  
Enables monitoring for toggle coverage.
```

Limiting Toggle Coverage to Ports Only

You can instruct VCS and VCS MX to compile and monitor only the ports in your design for toggle coverage so that the coverage reports that cmView writes contain information only about the ports. These reports do not contain information about either the variables and nets that are not ports in your Verilog modules or the variables and signals declared in your VHDL architectures.

You enable this feature with the `portsonly` keyword argument to the `-cm_tgl` compile-time option. For example:

```
vcs -cm tgl -cm_tgl portsonly design.v
```

or

```
vcs -cm tgl -cm_tgl portsonly cfg
```

Note:

If you enter the `-cm_tgl portsonly` compile-time option, the cmView GUI shows the nets, signals, and variables that are not ports, as not covered. If you do not enter this option, the cmView GUI shows some or all of these nets, signals, and variables as covered. Synopsys does not recommend using this feature with the cmView GUI.

Limiting Toggle Coverage to Uncovered Signals

You might want cmView, the coverage report writing utility, and GUI for coverage, to report only the untoggled signals (those that did not have both a $0 \rightarrow 1$ and a $1 \rightarrow 0$ transition). You can then view the results, revise your testbench, recompile, resimulate, and rerun cmView, again and again, until cmView reports no uncovered signals.

You do this with the `-cm_tgl_uncovf` option, which is both a cmView (or `vcs -cm_pp`) command-line option and a VCS or VCS MX runtime option.

Note:

This feature works only with Verilog code.

The procedure to limit toggle coverage to uncovered signals is as follows:

1. Compile and simulate your design using the `-cm tgl` compile-time and runtime options so that VCS or VCS MX writes test files for toggle coverage.
2. Run cmView with the `-cm_tgl_uncovf` option, specifying a file to which cmView writes the uncovered signals. For example:

```
vcs -cm_pp -cm tgl -cm_tgl_uncovf uncovfile
```

In this example, VCS writes a file called `uncovfile` in the current directory. This file lists the uncovered signals in a format that VCS and VCS MX can read.

3. Run the simulation again, specifying the `uncovfile` file with the `-cm_tgl_uncovf` runtime option. For example:

```
simv -cm tgl -cm_tgl_uncovf uncovfile
```

In this example, during simulation, VCS monitors only the uncovered signals, and the test files that VCS writes contain toggle coverage information about only these signals.

4. Run cmView again with the `-cm_tgl_uncovf` option and review the coverage results.
5. Revise your testbench code to generate more toggle coverage.
6. Compile and simulate the design again with the `-cm tgl` compile-time and runtime options, and the `-cm_tgl_uncovf` runtime option, specifying the same file.
7. Run cmView again with the `-cm_tgl_uncovf` option and review the coverage results.

Repeat steps 5, 6, and 7 until there are no more uncovered signals.

Autograding Limited Toggle Coverage

Each set of test files generated with the same argument to the `-cm_tgl_uncovf` runtime option can be autograded together (see “[Test Autograding in Batch Mode](#)” on page 1-44).

Test files generated with different file arguments to the `-cm_tgl_uncovf` runtime option cannot be meaningfully autograded against each other.

Data from tests that monitor only 20% of the signals in a design cannot be compared for coverage to tests that monitor all of the signals.

Note:

The design does not have to be recompiled for the next set of test files. As long as the design hasn't changed, no recompilation is necessary.

Merging Limited Toggle Coverage Data

Test files generated with different file arguments to the `-cm_tgl_uncovf` runtime option can be merged together. This means all test data from all sets of test files can be merged to see the total overall coverage from all sets.

Not all `test.tgl` files need to be kept. Each set of test files can be merged into a single `*.tgl` file after autograding. The `*.tgl` files for each set can be merged to get the overall coverage for all sets without retaining the file for each individual test.

Excluding and Including Signals in Toggle Coverage

To exclude or include signals in toggle coverage, you must use the `-cm_tgl` option at both compile time and runtime.

You can modify the default behavior of toggle coverage by using the `+node` and `-node` configuration file entries in the file passed to the `-cm_hier` compile-time option.

These entries work in conjunction with other entries in the configuration file to exclude or include toggle and other types of coverage, such as the following:

<code>+file</code>	<code>-file</code>
<code>+filelist</code>	<code>-filelist</code>
<code>+library</code>	<code>-library</code>
<code>+module</code>	<code>-module</code>
<code>+tree</code>	<code>-tree</code>

The `-node` and `+node` entries work for both Verilog and VHDL signals, nets, and variables.

Excluding a Signal in Toggle Coverage

To exclude a signal, enter the `-node` entry in the `-cm_hier` file, followed by the signal's hierarchical name. For example:

```
+module dev
-node top.dev1.w2
```

In this example, all instances of module `dev` are compiled for coverage. In one such instance, `top.dev1`, signal `w2` is excluded from toggle coverage.

Including a Signal in Toggle Coverage

To include a signal, enter the `+node` entry in the `-cm_hier` file, followed by the signal's hierarchical name. For example:

```
-module dev
+node top.dev1.w2
```

In this example, all instances of module `dev` are excluded from various types of coverage, with one exception. Instance `top.dev1` is compiled for toggle coverage but only for signal `w2`.

Including Part-Selects and Bit-Selects

For Verilog nets and variables, you can include a part-select or a bit-select for a net or variable. For example:

```
-module test
+node test.w1[2:1]
+node test.w2[1]
+node test.l1[2:1]
+node test.l2[1]
```

Note:

Make sure that there are no spaces between the signal name and the bit or bits of the bit-select or part-select.

Using Wildcard Characters

You can use the asterisk (*) and question mark (?) wildcard characters in these -node and +node entries. For example:

```
-module test
+node test.w*
+node test.logic?
```

The asterisk(*) can represent multiple characters, while the question mark (?) represents a single character.

Specifying SystemVerilog Structures and Unions

You can specify an instance of a structure or a union, but not a member of the structure or union. For example, refer to the following source code:

```
module test;
typedef struct {
    logic log1;
    bit bit1;
} control;
control ctl1;
endmodule
```

The following configuration file entry is valid:

```
-module test
+node test.ctl1
```

The following configuration file entry is *not* valid:

```
-module test
+node test.ctl1.log1
```

Toggle Coverage Glitch Suppression

The `-cm_glitch positive_integer` compile-time option specifies the simulation time period of glitch suppression (see “[Using Glitch Suppression](#)” on page 1-34).

Unlike glitch suppression in line and condition coverage, glitch suppression in toggle coverage is not focused on rapidly triggering always blocks. Glitch suppression in toggle coverage indicates that VCS and VCS MX do not monitor or record any value change on a signal that lasts fewer simulation time units than the simulation time specified with the `-cm_glitch` compile-time option. It does not matter whether these short interval changes, or glitches, are caused by statements in an always or initial block, continuous assignments, or connections to a gate, primitive, or user-defined primitive (UDP).

Another difference between glitch suppression for toggle coverage and glitch suppression for line or condition coverage, is that there is a `-cm_glitch` runtime option for toggle coverage. This runtime option overrides the glitch period for toggle coverage specified at compile-time. This runtime option does not work for either line or condition coverage.

Glitch suppression in toggle coverage is not necessary for zero time glitches, that is transitions from $0 \rightarrow 1$ and then $1 \rightarrow 0$, or from $1 \rightarrow 0$ and then $0 \rightarrow 1$, during the same simulation time step. VCS never monitors or records these transitions for toggle coverage.

Note:

This feature works only with Verilog code.

Using Pragmas to Limit Toggle Coverage

To disable toggle coverage for a variable or a net, enter the pragma before the variable or net declaration. [Figure 3-1](#) shows how this works:

Example 3-1 Pragmas for Toggle Coverage

```
module test;

//VCS coverage off
reg r1;
//VCS coverage on
reg r2,r3;
//VCS coverage off
wire w1;
//VCS coverage on
wire w2,w3;

initial
begin
#1 r1=1; // pragma'd out
    r2=1;
    r3=1;
#1 r1=0;
    r2=0;
    r3=0;
#1 r1=1;
    r2=1;
    r3=1;
#1 r2=0;
#1 r2=1;
#100 $finish;
end

assign w1=r1; // pragma'd out
assign w2=r2;
assign w3=r3;

endmodule
```

[Figure 3-1](#) contains pragmas before the reg `r1` and wire `w1` declarations, in order to exclude them from the toggle coverage.

`cmView` only reports on toggle coverage for reg `r2` and `r3` and wire `w2` and `w3`. It does not report on reg `r1` or wire `w1`.

Note:

This feature works only with Verilog code.

If you enter any of these pragmas in your source code, but some time later want VCS or VCS MX to ignore these pragmas, enter the `-cm_ignorepragmas` compile-time option.

For information about using pragmas in general, see “[Using Pragmas to Exclude VHDL Lines From Coverage](#)” on page 1-32.

Toggle Coverage Reports

`cmView` writes its report files in the `./simv.cm/reports` directory.

The report files that `cmView` writes are as follows:

`cmView.long_t`

A long detailed report file, organized by module instance, containing comprehensive information about the toggle coverage of your design (see “[The `cmView.long_t` File](#)” for a sample report).

`cmView.long_td`

A long detailed report file, similar to the `cmView.long_t` file, but organized by module definition instead of module instance.

`cmView.hier_t`

A toggle coverage report where coverage data is organized by subhierarchies in the design. There is a section for each module instance and the information in these sections is used for the instance and for all module instances hierarchically under that instance (see “[The cmView.hier_t File](#)” for a sample report).

`cmView.mod_t`

A toggle coverage report where coverage data is for module instances in the design, not subhierarchies. There is a section for each module instance and the information in these instance sections do not include data for instances hierarchically under the specified instance (see “[The cmView.mod_t File](#)” for a sample report).

`cmView.mod_td`

A toggle coverage report, similar to the `cmView.mod_t` file, but organized by module definition instead of module instance (see “[The cmView.mod_td File](#)” for a sample report).

`cmView.short_t`

A short report file containing sections for instances in which there is not 100% toggle coverage. In these sections, cmView lists only signals that were not covered. The report ends with summary information (see “[The cmView.short_t File](#)” for a sample report).

`cmView.short_td`

A short report file, similar to the `cmView.short_t` file, but organized by module definition instead of by module instance (see “[The cmView.short_td File](#)” for a sample report).

To understand the contents of these report files, consider the following small Verilog example and view its toggle coverage:

Example 3-2 Example For Toggle Coverage

```
module test;
```

```

reg r1;
reg [7:0] r2;
wire w1;

dut dut1 (w1,r1);

initial
begin
r1=0;
r2=8'b00000000;
#100 $finish;
end

always
#10 r1=~r1;

always
#25 r2=r2+1;

endmodule

module dut (out,in);
output out;
input in;
reg dutr1;

always @ in
dutr1=in;

assign out=dutr1;
endmodule

```

In [Figure 3-2](#), the top-level module, test, instantiates module dut. The top-level module, test, contains two registers, r1 and r2, and one net, w1. The module instance dut1 of module dut contains one register dutr1 and two ports in and out.

In the top-level module, test, reg r1 will have ten 0 → 1 or 1 → 0 transitions and vector reg r2 will increment three times.

In module instance `dut1`, all the signals will have ten $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions.

The `cmView.long_t` File

The following is the `cmView.long_t` file that VCS generates for [Figure 3-2](#). This example file is interrupted in a number of places to explain its contents.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year  
  
// LONG TOGGLE COVERAGE REPORT  
  
//*****  
// MODULE INSTANCE COVERAGE  
  
// This section contains coverage for each instance of a module  
MODULE test  
  
// Net Coverage  
// Name Toggled 1->0 0->1  
w1 Yes  
  
// Register Coverage  
// Name Toggled 1->0 0->1  
r2[0] Yes  
r2[1] No No Yes  
r2[7:2] No No No  
r1 Yes
```

The report begins with a section on the top-level module, `test`. Note that for VHDL coverage, the heading "Net Coverage" would be "Port Coverage," and the heading "Register Coverage" would be "Signal Coverage."

The preceding report snippet says:

- The one net, `w1`, toggled, which is to say it had both $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions.
- Register `r1` also toggled.
- The toggle coverage for vector reg `r2` is different, depending on the bit: `r2 [0]` toggled, there was only a $0 \rightarrow 1$ transition in `r2 [1]`, and no transitions on the higher bits. This is what we expected because reg `r2` incremented up to a value of three.

```
//-----  
//          Module Coverage Summary  
  
//          TOTAL    COVERED    PERCENT  
regs           2        1        50.00  
reg bits       9        2        22.22  
reg bits(0->1) 9        3        33.33  
reg bits(1->0) 9        2        22.22  
  
nets           1        1        100.00  
net bits       1        1        100.00  
net bits(0->1) 1        1        100.00  
net bits(1->0) 1        1        100.00
```

The section on the top-level module, `test`, concludes with summary information. Note that for VHDL coverage, all "regs" and "reg bits" would instead be "ports" and "port bits," and that all "nets" and "net bits" would be "signals" and "signal bits." In this example, 50% of the regs toggled (there are only two regs and only one of them, `r2`, toggled). On a per bit basis 33.33% had a $0 \rightarrow 1$ transition, but only 22.22% had a $1 \rightarrow 0$ transition.

```
MODULE test.dut1  
  
//          Net Coverage  
//  Name      Toggled   1->0   0->1  
in         Yes  
out        Yes  
  
//          Register Coverage  
//  Name      Toggled   1->0   0->1  
dutr1      Yes
```

```

//-----
//          Module Coverage Summary
//
//          TOTAL    COVERED    PERCENT
regs           1        1      100.00
reg bits       1        1      100.00
reg bits(0->1) 1        1      100.00
reg bits(1->0) 1        1      100.00

nets           2        2      100.00
net bits       2        2      100.00
net bits(0->1) 2        2      100.00
net bits(1->0) 2        2      100.00

```

The next section is used for module instance `dut1`. Again, for VHDL coverage, all references to "nets" would be replaced with "signals," and all references to "regs" would be replaced with "ports." As expected, there is complete toggle coverage for the signals in this instance.

```

//*****
//          Total Module Instance Coverage Summary
//
//          TOTAL    COVERED    PERCENT
regs           3        2      66.67
reg bits       10       3      30.00
reg bits(0->1) 10       4      40.00
reg bits(1->0) 10       3      30.00

nets           3        3      100.00
net bits       3        3      100.00
net bits(0->1) 3        3      100.00
net bits(1->0) 3        3      100.00

```

The report ends with summary information for the entire example, with summary information about all the registers and nets and all of their bits. For VHDL, the summary information would cover information for ports and signals.

The cmView.hier_t File

The following is the `cmView.hier_t` file. This file contains summary information in terms of subhierarchies. As the module instance, `test.dut1`, is hierarchically under the top-level module, `test`, the data from `test.dut1` is included with the data for the top-level module, `test`.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE HIERARCHICAL INSTANCE COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// instance of a module. The statistics take into account all sub-hierarchies  
// under the instance.  
  
Test Coverage Result: Total Coverage  


| Module Name    | NetBits<br>(%) | NetBits | RegBits<br>(%) | RegBits |
|----------------|----------------|---------|----------------|---------|
| regs           | 3              | 2       | 66.67          |         |
| reg bits       | 10             | 3       | 30.00          |         |
| reg bits(0->1) | 10             | 4       | 40.00          |         |
| reg bits(1->0) | 10             | 3       | 30.00          |         |
| nets           | 3              | 3       | 100.00         |         |
| net bits       | 3              | 3       | 100.00         |         |
| net bits(0->1) | 3              | 3       | 100.00         |         |
| net bits(1->0) | 3              | 3       | 100.00         |         |

  
//*****  
// Total Module Instance Coverage Summary  


|                | TOTAL | COVERED | PERCENT |
|----------------|-------|---------|---------|
| regs           | 3     | 2       | 66.67   |
| reg bits       | 10    | 3       | 30.00   |
| reg bits(0->1) | 10    | 4       | 40.00   |
| reg bits(1->0) | 10    | 3       | 30.00   |
| nets           | 3     | 3       | 100.00  |
| net bits       | 3     | 3       | 100.00  |
| net bits(0->1) | 3     | 3       | 100.00  |
| net bits(1->0) | 3     | 3       | 100.00  |


```

The cmView.mod_t File

The following is the `cmView.mod_t` file. This file contains summary information for module instances alone, without data for lower-level instances.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE INSTANCE COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// instance of a module. The statistics do not take into account any  
// sub-hierarchy under the instance.  
  
Test Coverage Result: Total Coverage  


| Module Name | NetBits<br>(%) | NetBits | RegBits<br>(%) | RegBits |
|-------------|----------------|---------|----------------|---------|
| test        | 100.00         | 1/1     | 22.22          | 2/9     |
| test.dut1   | 100.00         | 2/2     | 100.00         | 1/1     |

  
//*****  
// Total Module Instance Coverage Summary  


|                | TOTAL | COVERED | PERCENT |
|----------------|-------|---------|---------|
| regs           | 3     | 2       | 66.67   |
| reg bits       | 10    | 3       | 30.00   |
| reg bits(0->1) | 10    | 4       | 40.00   |
| reg bits(1->0) | 10    | 3       | 30.00   |
| <br>nets       | 3     | 3       | 100.00  |
| net bits       | 3     | 3       | 100.00  |
| net bits(0->1) | 3     | 3       | 100.00  |
| net bits(1->0) | 3     | 3       | 100.00  |


```

The cmView.mod_td File

The cmView.mod_td file is similar to the cmView.mod_t file, except that the coverage information is organized by module definition instead of by module instance.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE DEFINITION COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// module definition. The coverage is cumulative over all the instances  
// of the module  
  
Test Coverage Result: Total Coverage  


| Module Name | NetBits<br>(%) | NetBits<br>(%) | RegBits<br>(%) | RegBits<br>(%) |
|-------------|----------------|----------------|----------------|----------------|
| test        | 100.00         | 1/1            | 22.22          | 2/9            |
| dut         | 100.00         | 2/2            | 100.00         | 1/1            |

  
//*****  
// Total Module Definition Coverage Summary  


|                | TOTAL | COVERED | PERCENT |
|----------------|-------|---------|---------|
| regs           | 3     | 2       | 66.67   |
| reg bits       | 10    | 3       | 30.00   |
| reg bits(0->1) | 10    | 4       | 40.00   |
| reg bits(1->0) | 10    | 3       | 30.00   |
| <br>           |       |         |         |
| nets           | 3     | 3       | 100.00  |
| net bits       | 3     | 3       | 100.00  |
| net bits(0->1) | 3     | 3       | 100.00  |
| net bits(1->0) | 3     | 3       | 100.00  |


```

The cmView.short_t File

The `cmView.short_t` file contains summary information for all module instances that do not have 100% coverage, and summary information about the entire design.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year  
  
//  
// SHORT TOGGLE COVERAGE REPORT  
  
//*****  
//  
// MODULE INSTANCE COVERAGE  
  
// This section contains coverage for each instance of a module  
MODULE test  
  
//  
// Register Coverage  
  
// Name 1->0 0->1  
r2 [1] No Yes  
r2 [7:2] No No
```

Since only the top-level module, `test`, does not have 100% toggle coverage, the report contains only this data. The report includes only those bits that do not have complete toggle coverage.

The report also includes summary information for the entire design.

```
//*****  
//  
// Total Module Instance Coverage Summary  
  
//  
//

|                | TOTAL | COVERED | PERCENT |
|----------------|-------|---------|---------|
| regs           | 3     | 2       | 66.67   |
| reg bits       | 10    | 3       | 30.00   |
| reg bits(0->1) | 10    | 4       | 40.00   |
| reg bits(1->0) | 10    | 3       | 30.00   |


```

nets	3	3	100.00
net bits	3	3	100.00
net bits(0->1)	3	3	100.00
net bits(1->0)	3	3	100.00

The cmView.short_td File

The cmView.short_td file is similar to the cmView.short_t file, except that it is organized by module definition instead of by module instance.

```

// Synopsys, Inc.
//
// Generated by: cmView version_number
// User: Intelligent User
// Date: Day Month date hour:minute:second year

//          SHORT TOGGLE COVERAGE REPORT

//*****MODULE DEFINITION COVERAGE*****
// This section contains coverage for module definitions.
// The coverage is cumulative over all the instances of the module

MODULE test

//                                     Net Coverage
// Name                                1->0  0->1
//                                     Register Coverage
// Name                               1->0  0->1
// r2[1]                               No    Yes
// r2[7:2]                             No    No

//*****TOTAL MODULE DEFINITION COVERAGE SUMMARY*****
//          TOTAL   COVERED   PERCENT
regs        3       2        66.67
reg bits   10      3        30.00
reg bits(0->1) 10      4        40.00

```

reg bits(1->0)	10	3	30.00
nets	3	3	100.00
net bits	3	3	100.00
net bits(0->1)	3	3	100.00
net bits(1->0)	3	3	100.00

Displaying Port Signal Direction in the Toggle Coverage Reports

To report the direction of port signals in the toggle coverage reports, use the following command-line options:

```
vcs -cm tgl dff.v -R
vcs -cm_pp -cm tgl -cm_report portdir
```

Consider the following design:

```
module top;
reg reset;
reg clk;
wire q;
reg data;

dff d(reset, clk, data, q);

initial
$monitor($time, "\n q : %d", q);

initial begin
reset = 1;
clk = 0;
data = 0;
#1 reset = 0;
#10 data = 1;
#10 data = 0;
#50 $finish ;
end
```

```

always
#5 clk = ~clk;
endmodule

moduledff(reset, clk, data, q);
input reset;
input clk;
input data;
output q;
reg q;

always @ (posedge clk or negedge reset)
if (~reset)
    q = 0;
else
    q = data;
endmodule

```

In the above example, for module `dff`, the signals `reset`, `clk`, and `data` are of type `input`, and signal `q` is of type `output`.

When you use the `portdir` keyword argument with the `-cm_report` option for the above design, the reports include a **Direction** column:

```

// This section contains coverage for each instance of a
module

MODULE top

//                                     Net Coverage

// Name      Toggled   1->0     0->1      Direction
q          No        No       No

//                                     Register Coverage

// Name      Toggled   1->0     0->1      Direction
reset     No        Yes      No
clk       Yes

```

Toggle Coverage

3-170

```

data      Yes

-----
//          Module Coverage Summary

//          TOTAL COVERED    PERCENT
regs        3      1      33.33
reg bits   3      1      33.33
reg bits(0->1) 3      2      66.67
reg bits(1->0) 3      2      66.67

nets       1      0      0.00
net bits   1      0      0.00
net bits(0->1) 1      0      0.00
net bits(1->0) 1      0      0.00

MODULE top.d
//          Net Coverage

// Name    Toggled    1->0    0->1    Direction
reset    No         Yes     No      INPUT
clk      Yes        -       -      INPUT
data     Yes        -       -      INPUT

//          Register Coverage

// Name    Toggled    1->0    0->1    Direction
q        No         No     No      OUTPUT

//-----
//          Module Coverage Summary

//          TOTAL COVERED    PERCENT
regs        1      0      0.00
reg bits   1      0      0.00
reg bits(0->1) 1      0      0.00
reg bits(1->0) 1      0      0.00

nets       3      1      33.33
net bits   3      1      33.33
net bits(0->1) 3      2      66.67

```

```
net bits(1->0)      3      2          66.67
```

For local signals, there is no direction, therefore, the `Direction` column is empty.

Viewing Toggle Coverage With the Coverage Metrics GUI

`cmView` is also the graphical user interface (GUI) for VCS and VCS MX coverage metrics. It displays graphical representations of the coverage information recorded by VCS and VCS MX.

Perform the following to view the toggle coverage results VCS or VCS MX records for the simulation of the source code in [Figure 3-2](#):

1. Enter the following command line:

For Verilog-only and Verilog-top:

```
vcs -cm_pp gui -cm tgl
```

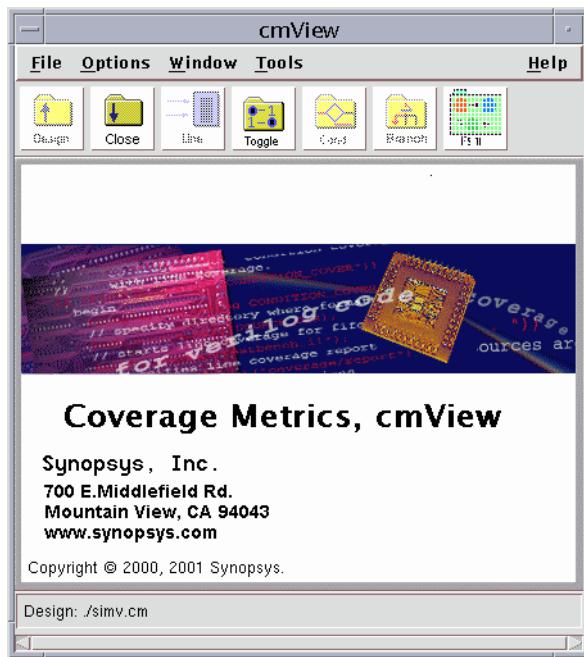
For VHDL-only and VHDL-top:

```
cmView -cm tgl
```

The command line instructs `cmView` to perform the following:

- a. Open the main window for `cmView` (see [Figure 3-1](#)).

Figure 3-1 The Main cmView Window



b. Read the design file in the following two locations:

`./simv.cm/db/verilog/cm.decl_info`

`./simv.cm/db/vhdl/cm.decl_info`

cmView reads this file to learn about the design and its hierarchy. Read the intermediate data files for toggle coverage in the following locations:

`simv.cm/coverage/verilog`

`simv.cm/coverage/vhdl`

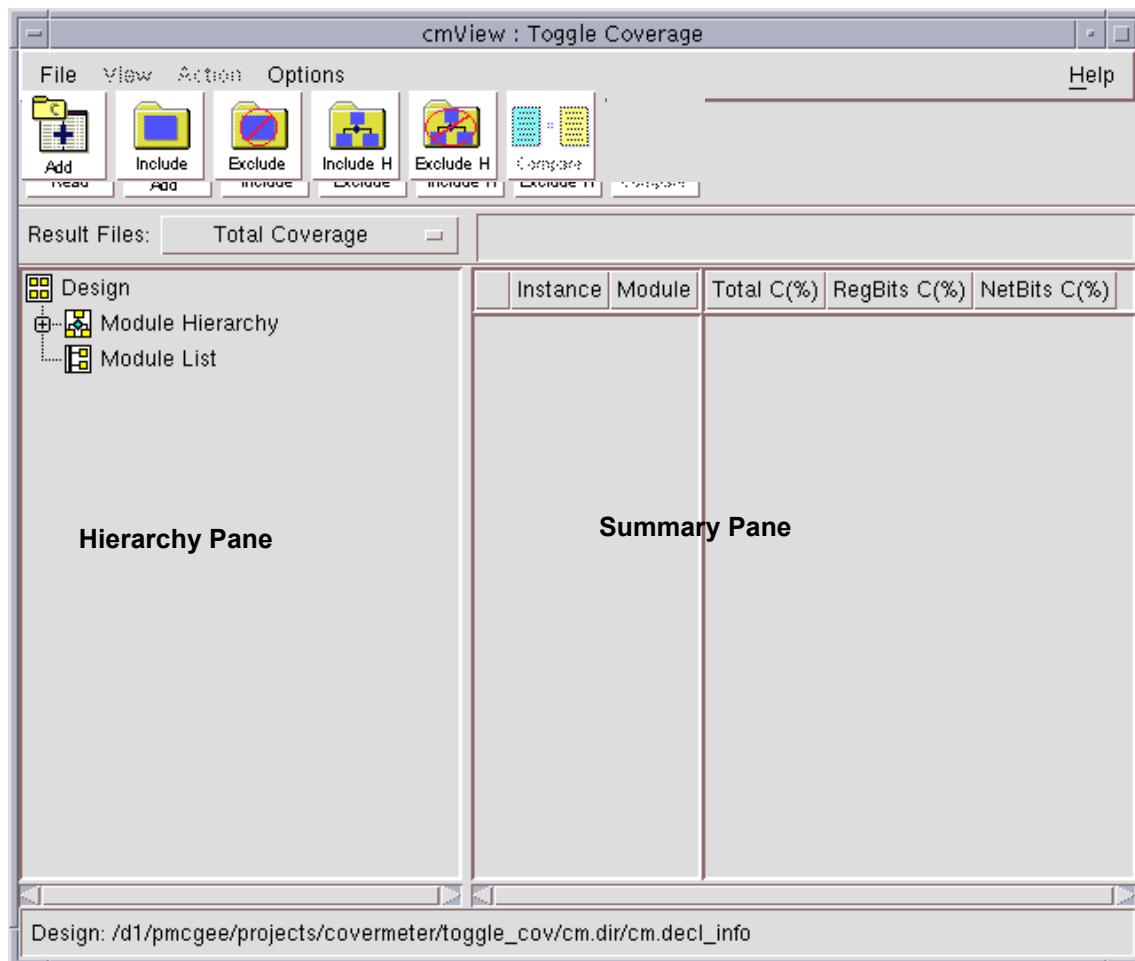
Note:

If at compile-time you specify a different coverage metrics database with the `-cm_dir` compile-time option, you need to include the `-cm_dir` option on the `cmView` (or `vcs -cm_pp gui`) command line.

3. Click the Toggle Coverage toolbar button (see below) to display the Toggle Coverage window (see [Figure 3-2](#)):

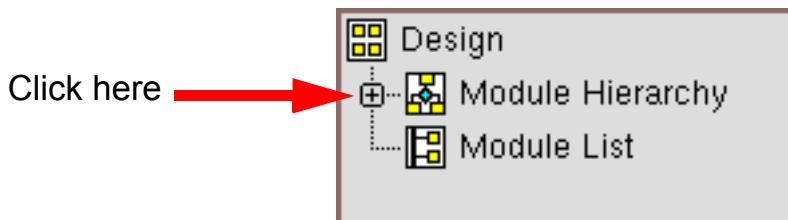


Figure 3-2 Toggle Coverage Window



You use this window to view toggle coverage data from the intermediate data files written by VCS or VCS MX. The left pane is used for opening the design hierarchy; the right pane, called the summary pane, is used for displaying a summary of coverage information.

4. By default, cmView displays coverage summary information on a per module instance basis, therefore, the Toggle Coverage window needs a way to display the design hierarchy so that you can select module instances to examine their toggle coverage. Click the plus symbol (+) next to the icon for the Module Hierarchy.

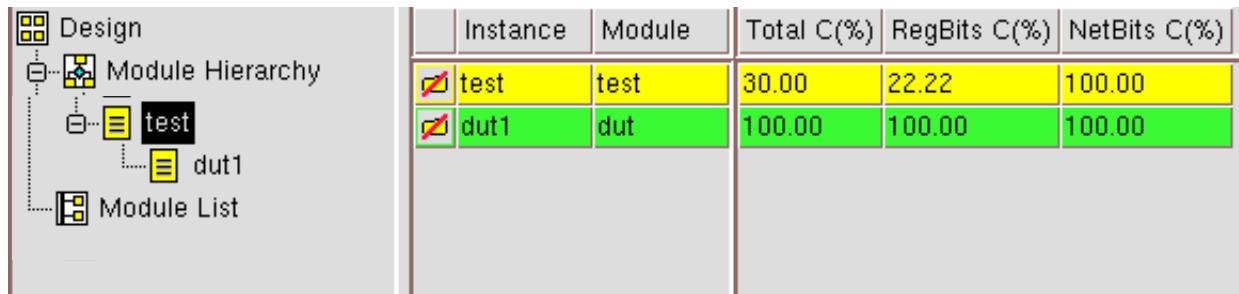


Clicking on the plus symbol (+) displays the top-level modules in the design. In this case, there is only one top-level module, test.

	Instance	Module	Total C(%)	RegBits C(%)	NetBits C(%)
test	test	test	30.00	22.22	100.00

By default, the yellow results bar indicates coverage between 20%-80%.

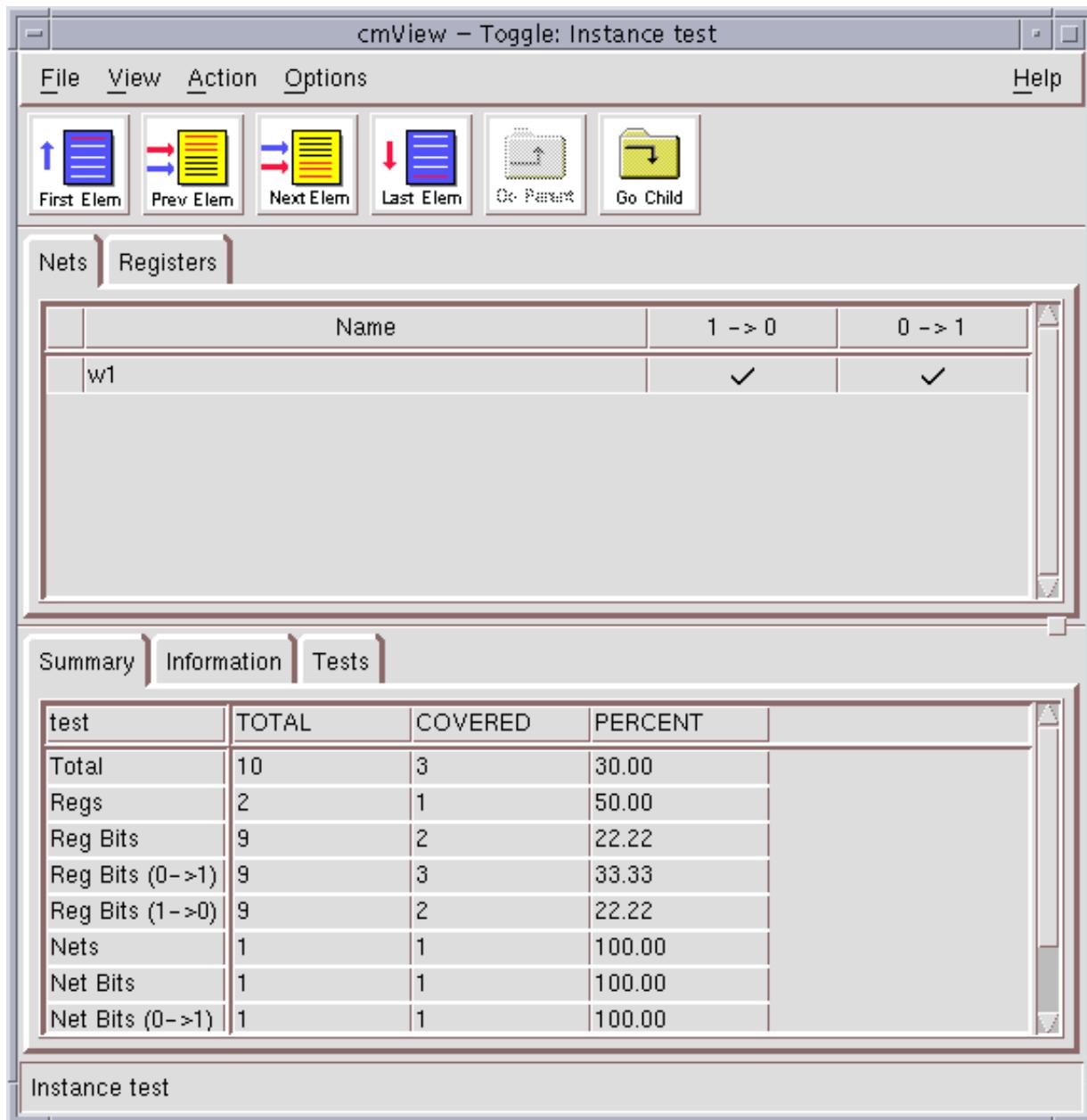
5. Click the plus symbol (+) next to the icon for the top-level module. This displays all the module instances of that module. In this case, there is only one instance of the test module, test.dut1).



The green results bar for dut1 indicates 80%-100% coverage.

6. Click on the yellow results bar for the module instance, test, to display the Annotated Toggle Coverage window (see [Figure 3-3](#)).

Figure 3-3 Annotated Toggle Coverage Window



The Summary Pane

By default, the summary pane displays the following information

when a Toggle Coverage window opens:

- The first column is a place holder.
- The second column contains the instance name.
- The third column contains the module name.
- The fourth column contains the total percentage.
- The fifth column contains the percentage coverage for total registers in the instance.
- The sixth column contains the percentage coverage for total nets in the instance.

Double-clicking on any row in the toggle coverage summary pane displays an Annotated Toggle Coverage window with detailed toggle coverage data for the specific instance or module (see “[The Annotated Toggle Coverage Window](#)” for a description of the Annotated Toggle Coverage window).

Detailed View

To obtain a detailed report of toggle coverage, choose **View > Detailed** from the menu. The detailed statistics for toggle coverage are displayed as shown in [Figure 3-4](#).

In addition to the default instance and module names, the following information is also displayed and arranged in columns as follows:

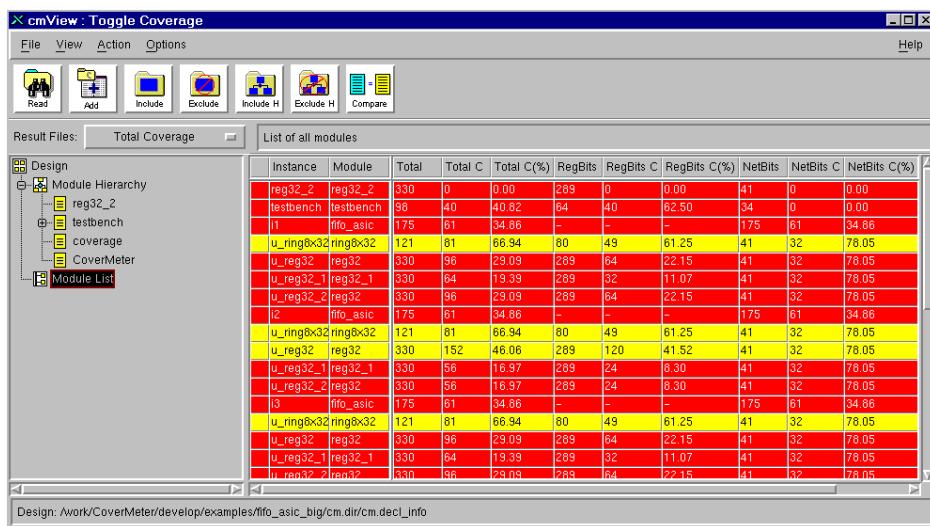
- Total registers
- Number of registers covered
- Percentage of registers covered

- Total nets
- Number of nets covered
- Percentage of nets covered

Note:

For VHDL coverage, this information also includes statistics for ports and signals.

Figure 3-4 Detailed View of Statistics for Toggle Coverage



The Annotated Toggle Coverage Window

The Annotated Toggle Coverage window allows you to closely examine the coverage for specific signals in an instance.

This window contains the following:

- Command menus
- Toolbar buttons

- The Signal Listing Pane
- The Summary Pane

The Command Menus

The Annotated Toggle Coverage window has the following four menus:

- **File** menu - contains the following two commands:
 - Report - opens the Create Toggle Annotated Report dialog box that you use to tell cmView to write coverage report files for the instance.
 - Close - closes the Annotated Toggle Coverage window.
- **View** menu - contains the following two commands:
 - Open parent module - changes the display to show you the Annotated Toggle Coverage window for the parent instance; the instance one level up in the design hierarchy. The Go Parent toolbar button has the same function (see “[The Toolbar Buttons](#)”).
 - Open child module - changes the display to show you the Annotated Toggle Coverage window for the child instance; the instance one level down in the design hierarchy. If there is more than one child instance, a dialog box appears for you to select the child instance that you want to view. The Go Child toolbar button has the same function (see “[The Toolbar Buttons](#)”).
- **Options** menu - contains the following three commands:

Toolbar

Enables and disables the display of the toolbar.

- Status Bar
Enables and disables the display of the status bar. The status bar is located under the Summary pane. It shows the hierarchical name of the current instance.
- Preferences - opens the User Preferences dialog box, where you can change the color code of the displayed items (see “[User Preferences](#)” on page 8-461).
- **Exclude** menu - contains the following five commands:
 - Recalculate - tells cmView to recalculate coverage after you mark a signal or signals for exclusion from coverage analysis (see “[Excluding Signals or Bits of Signals From Coverage Calculation](#)”).
 - Unmark This Module - tells cmView to unmark for exclusion all signals in the instance (see “[Excluding Signals or Bits of Signals From Coverage Calculation](#)” and “[Clearing Marked Signals](#)” on page 3-193).
 - Unmark All Modules - tells cmView to unmark for exclusion all signals in the design (see “[Excluding Signals or Bits of Signals From Coverage Calculation](#)” and “[Clearing Marked Signals](#)”).
 - Comment - tells cmView to open a dialog box for you to write a comment in the file that you will use to exclude signals in a subsequent cmView session (see “[Excluding Signals or Bits of Signals From Coverage Calculation](#)”, “[Using a File to Exclude Signals](#)”, and “[Comments in the File](#)”).
 - Write To File - tells cmView to write the marked signals in a file that you will use to exclude these signals from coverage in a subsequent session cmView session (see “[Excluding Signals or Bits of Signals From Coverage Calculation](#)” and “[Using a File to Exclude Signals](#)”).

The Toolbar Buttons

You use toolbar buttons to move from one uncovered signal to the next (see [Table 3-1](#) for a description of each button). When using these buttons to navigate through signals, cmView displays the currently selected signal in black with reverse highlighting. You can change this color from the Colors tab of the User Preferences dialog box (see “[User Preferences](#)” on page [8-461](#)).

Table 3-1 Toolbar Buttons in the Annotated Toggle Coverage Window

Button	Description
 First Elem	This button takes you to the first uncovered signal, or partially uncovered signal, in the instance.
 Next Elem	This button takes you to the next uncovered signal, or partially uncovered signal, in the instance.
 Prev Elem	This button takes you to the previous uncovered signal, or partially uncovered signal, in the instance.
 Last Elem	This button takes you to the last uncovered signal, or partially uncovered signal, in the instance.
 Go Parent	Changes the display to show you the toggle annotated coverage window for the parent instance.
 Go Child	Changes the display to show you the toggle annotated coverage window for the child instance. If there is more than one child instance, a dialog box appears for you to select the instance you want to view.

The Signals Listing Pane

The top half of the Annotated Toggle Coverage window includes the Signals Listing pane, which contains one tab for each register or net (Verilog), logic or bit data types (SystemVerilog), or signal and port (VHDL) that pertain to a specific instance or module. This pane

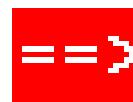
contains a tabulation of all $0 \rightarrow 1$ and $1 \rightarrow 0$ value transitions for all nets, registers, logics, bits, signals, or ports in the selected instance or module.

Figure 3-5 The Signals Listing Pane

	Name	$1 \rightarrow 0$	$0 \rightarrow 1$	
	out1[1:0]	✓	✓	[+]
==>	out1[5:2]	✗	✗	[+]
	out1[7:6]	✓	✓	[+]
	out2	✓	✓	
==>	out3	✗	✗	
	in1[1:0]	✓	✓	[+]
==>	in1[5:2]	✗	✗	[+]
	in1[7:6]	✓	✓	[+]
	in2	✓	✓	
==>	in3	✗	✗	

If some of the bits in a vector signal are covered, and others are not, the pane separates the covered bits from the uncovered bits.

In the first column, the following symbol indicates that the signal, or the bits of the vector signal, are not covered:



This symbol is used for excluding the uncovered signal or uncovered bits from the coverage data (see “[Excluding Signals or Bits of Signals From Coverage Calculation](#)”).

The second column lists the signals in the instance. cmView highlights the uncovered signals in red. If a vector signal is partially covered, the uncovered bits are highlighted in red.

The third column indicates if there was a $0 \rightarrow 1$ transition in the signal or the bits of a partially covered vector signal:

 Indicates that this transition occurred during simulation.

 Indicates that this transition did not occur during simulation.

The fourth column indicates if a $1 \rightarrow 0$ transition exists in the signal or the bits of a partially covered vector signal:

 Indicates that this transition occurred during simulation.

 Indicates that this transition did not occur during simulation.

The fifth and last column contains the following symbol for vector signals:

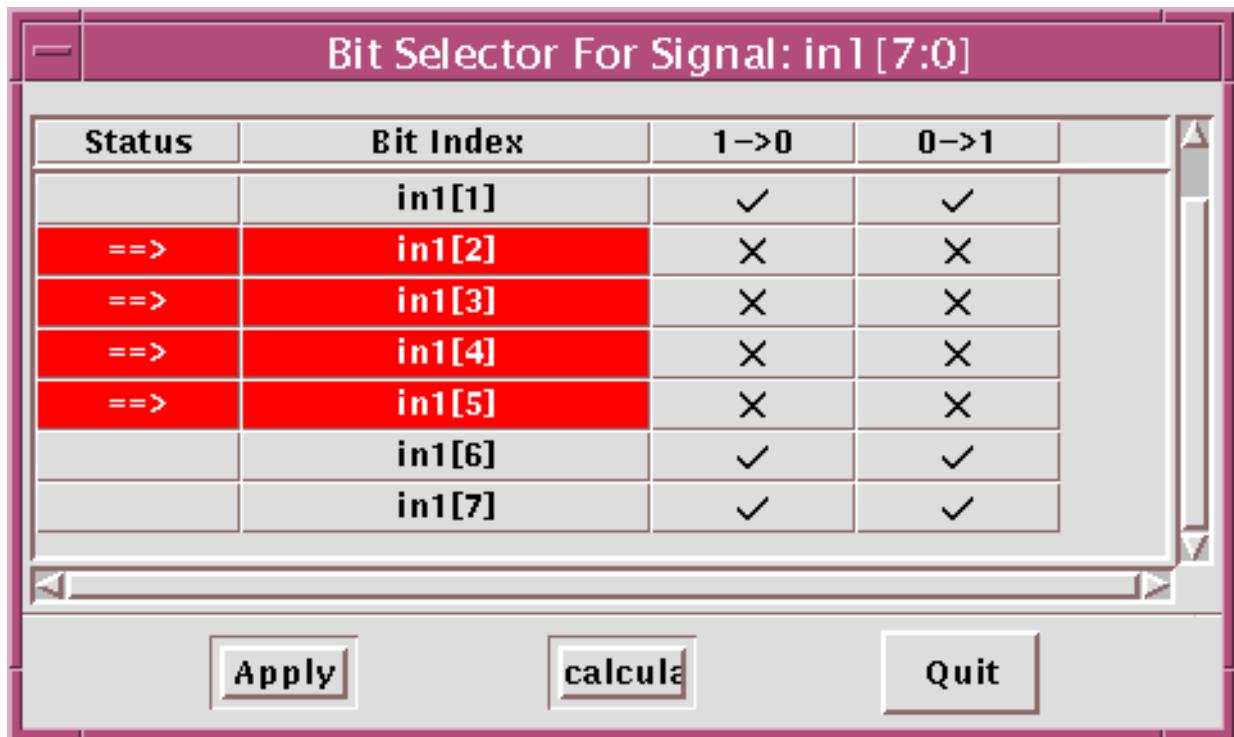


Note:

The fifth column is not supported for VHDL code.

Clicking on this symbol opens the Bit Selector For Signal:
signal_name dialog box.

Figure 3-6 The Bit Selector For Signal: signal_name Dialog Box



This dialog box provide you with coverage information for each bit in the vector signal. You can use this dialog box to exclude bits in the vector signal from the coverage calculation (see “[Excluding Individual Bits from Coverage Calculation](#)” for more information on using this dialog box).

The bottom half of the Annotated Toggle Coverage window contains a pane with three tabs that provide additional information about the module:

The screenshot shows the 'Summary' tab selected in the top navigation bar. Below it is a table with columns: test, TOTAL, COVERED, and PERCENT. The table data is as follows:

test	TOTAL	COVERED	PERCENT
Total	10	3	30.00
Regs	2	1	50.00
Reg Bits	9	2	22.22
Reg Bits (0->1)	9	3	33.33
Reg Bits (1->0)	9	2	22.22
Nets	1	1	100.00
Net Bits	1	1	100.00
Net Bits (0->1)	1	1	100.00

Below the table, a message says 'Instance test'.

The first tab in this pane is the Summary tab. Click this tab to display a summary of coverage for all nets and registers in the specified instance or module.

The second tab in the pane is the Information tab. Click this tab to display some general information about this instance or module, as shown in [Figure 3-7](#).

Figure 3-7 The Information Tab

Summary		Information	Tests
Instance:	testbench.i1		
Module:	fifo_asic		
Design:	/tmp_mnt/usr1/cmview/fifo_asic/formatted/cm.decl_info		
Coverage:	Toggle		
Test:	test2		

The third tab in the pane is the Tests tab. Click this tab to display a detailed summary of all the tests read so far, as shown in [Figure 3-8](#). Detailed information is also provided to describe the makeup of both total and incremental coverage.

Figure 3-8 The Tests Tab

Summary		Information	Tests
Incremental & Diff	Total Coverage		
	/tmp_mnt/usr1/cmview/fifo_asic/coverage/test0		
Tests Added	/tmp_mnt/usr1/cmview/fifo_asic/coverage/test1		
	/tmp_mnt/usr1/cmview/fifo_asic/coverage/test0		
Tests Read	/tmp_mnt/usr1/cmview/fifo_asic/coverage/test0		
	/tmp_mnt/usr1/cmview/fifo_asic/coverage/test1		
	/tmp_mnt/usr1/cmview/fifo_asic/coverage/test2		

Excluding Signals or Bits of Signals From Coverage Calculation

To exclude a signal from toggle coverage:

1. Choose the tab for the type of signal in the Signals Listing pane.

2. Click the  symbol next to the uncovered signal name, or the series of bits that are uncovered in a partially covered vector signal. This selects the signal or bits, changing the line to black with reverse highlighting (dark background, white text) and changes the symbol to "M" (for marked). 

3. Choose **Exclude > Recalculate** from the menu.

Selecting this command performs the following:

- Changes the M symbol  to an X symbol (for excluded) .
- Changes the  or  symbols in the columns for $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions to the following symbol:



- Changes the results in the Summary tab.

Excluding Individual Bits from Coverage Calculation

You use the Bit Select For Signal: *signal_name* dialog box to exclude individual bits of a vector signal from the coverage calculation (see [Figure 3-6](#)).

Note:

You can exclude only uncovered individual bits using this dialog box.

To exclude a bit or bits:

1. Click on the symbol in the fifth column of the Signal Listing pane for the vector signal. This opens the Bit Select For Signal: *signal_name* dialog box.

2. Click the symbol for the bit in the Status column of the dialog box. This selects the bit, changing its line to black with reverse highlighting and changes the symbol to M (for marked):



3. Click the Recalculate button in the dialog box.

Clicking on this button causes changes in the dialog box and in the Annotated Toggle Coverage window. The changes in the dialog box are as follows:

- Changes the M symbol to an X symbol (for excluded) .
- Changes the symbol in the columns for $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions to the following symbol:

The changes in the Annotated Toggle Coverage window are as follows:

- There is a new line for the bit in the Signals Listing pane. The left column for this bit has the symbol, indicating that it is excluded from coverage. The columns for $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions in this line have following symbol:

- The results change in the Summary tab in the Summary window.

You cannot do any new operations in the Annotated Toggle Coverage window until you close the Bit Select For Signal: *signal_name* dialog box.

4. Click the Apply  button to close the dialog box and instruct cmView to save the changes that you made using this dialog box, in the Annotated Toggle Coverage window.

or

Click the Quit  button to close the dialog box and discard the changes you made using this dialog box, in the Annotated Toggle Coverage window.

Using a File to Exclude Signals

You can use cmView to write a file containing the signals that are marked for exclusion with the X  or M  symbol. After this, whenever you start cmView and specify this file, the signals in this file will be excluded from coverage.

To use cmView to write a file containing the excluded signals:

1. Choose **Exclude > Write To File** from the menu.
This displays the Specify File For Excluded Toggles dialog box.
2. Choose the desired directory and enter the name of the file in this dialog box.

- If this is a new file, the process of creating this file is complete.
- If this is not a new file, the Specify Write Mode dialog box appears, displaying a message similar to the following:

File *exclude_file_path_name* is Not Empty
Overwrite or Append?

3. Click either the Append  button or the OverWrite  button.

Append adds new entries, while OverWrite deletes previous entries and only creates entries for what you have excluded.

Appended entries for the same module instance are accumulated or merged together.

Note:

This feature works in similar ways in line and branch coverage. You can use the same file to exclude lines, signals, and branches.

4. In a subsequent session of the cmView GUI, you can use the `-cm_elfile` option to specify the file, containing the excluded signals, when the session starts. For example:

```
vcs -cm_pp gui -cm tgl -cm_elfile exclude_file
```

The excluded signals are marked with the X symbol and are not counted in the coverage statistics.

Note:

The `exclude_file` can also be passed to cmView batch mode to write reports. For example:

```
vcs -cm_pp -cm tgl -cm_elfile exclude_file
```

The excluded signals are marked with an X in the reports.

Comments in the File

If you want to include documentation, you can add comments to the output file containing excluded signals. To do this, first recalculate the exclusion of these signals, then perform the following:

1. Choose **Exclude > Comment** from the menu.
This opens the Provide The Comment For Last Set Of Excluded Toggles dialog box.
2. Enter a comment.
3. Click the Save button.
4. Now, choose **Exclude > Write To File** from the menu and specify the file using the Specify File For Excluded Toggles dialog box.

Clearing Marked Signals

After marking a signal for exclusion, you can clear it by clicking the M  symbol. The symbol then changes back to ''.

If a signal is already excluded, as indicated by the X  symbol, clicking the symbol changes it to the XM .

When you choose **Exclude > Recalculate** from the menu, the statistics change to show that it is not covered, and the signal's symbol changes to .

To clear the M  and XM  symbols from multiple signals, use either of the following menu commands:

- **Exclude → Unmark This Module**
- **Exclude → Unmark All Modules**

This changes M  symbols to  and XM  symbols to X .

Inconsistency in the Exclude File

You may sometimes exclude a particular signal from coverage in the exclude file and then revise the design in such a manner that the signal is covered in the next simulation. In this case, if you specify the exclude file in the `-cm_elfile` option while running the second simulation, cmView does the following:

- If you are using the batch report writing mode, cmView ignores the entry in the exclude file and reports the signal as covered. cmView displays the following message on your screen:

```
//Warning: Attempt to exclude the covered tgl, attempt
ignored
//The excluded information is ignored
//-----
//data_type signal_name exclude_file
```

- If you are using GUI mode, cmView opens the Inconsistencies Detected In Elf file dialog box to display the same message that it would display in batch report writing mode.

This dialog box has two buttons:

- Ignore - enables you to continue the cmView GUI session. The Signals Listing pane for the instance does not show the covered signal as excluded.
- Abort - ends the cmView GUI session.

Excluding Covered Signals

You can exclude covered signals in the exclude file. In this case, if you restart cmView and specify the exclude file in `-cm_elfile` option, cmView does the following:

- If you are using batch report writing mode, cmView reports the signal as covered and puts an A symbol to the left of the signal, indicating that it has ignored an attempt to exclude the covered signal. For example:

	Register Coverage
//	
// Name	Toggled 1->0 0->1
A r1	Yes
A r2	Yes
A r3	Yes

- If you are using GUI mode, cmView opens the Inconsistencies Detected In Elf file dialog box and displays the following message:

```
//Warning: Attempt to exclude the covered tgl, attempt
is ignored
//-----
-----  

// data_type signal_name elfile: exclude_file
```

The dialog box has two buttons:

- Ignore - enables you to continue the cmView GUI session. The Signals Listing pane for the instance shows the A symbol to the left of the signal that you attempted to exclude (see [Figure 3-9](#)).
- Abort - ends the cmView GUI session.

Figure 3-9 Attempted Exclusion Signals

	Name	1 → 0	0 → 1
A	r1	✓	✓
A	r2	✓	✓
A	r3	✓	✓

4

Condition Coverage

Condition coverage monitors whether certain expressions and subexpressions in your code evaluate to true or false. You can use arguments to the `-cm_cond` compile-time option to modify how VCS or VCS MX compiles your design for condition coverage.

This chapter describes the following:

- “[What Is Condition Coverage?](#)”
- “[Modifying Condition Coverage](#)”
- “[Condition Coverage Reports](#)”
- “[Viewing Condition Coverage with the Coverage Metrics GUI](#)”

What Is Condition Coverage?

Condition coverage monitors whether certain expressions and subexpressions in your code evaluate to true or false. By default, these expressions and subexpressions are as follows:

- The conditional expression used with the conditional operator ? : in a continuous or procedural assignment statement. For example, note the following continuous assignment statement for Verilog:

```
assign w1 = r1 ^~ r2 ? r2 : r3;
```

The conditional expression is `r1 ^~ r2` and the truth or falsity of this expression are conditions for condition coverage.

- The conditional expression used in conditional concurrent signal assignment or selected concurrent signal assignment, as shown in either of the following VHDL examples:

```
c <= '0' when (a='0' and b= '0') else
      '1' when (a='1' and b= '1'); -- conditional concurrent
                                -- signal assignment
```

or

```
with (a and b or d) select
      c <= '0' when '0',
      '1' when '1',
      'X' when others;
```

In the previous example, the conditional expressions are (`a='0'` and `b= '0'`) and (`a='1'` and `b= '1'`) for selected concurrent signal assignment, and (`a and b or d`) for selected concurrent signal assignment.

- Subexpressions that are the operands to the logical AND `&&` or logical OR `||` operators in the conditional expression used with the conditional operator `? :` in a continuous or procedural assignment statement. For example, in the following Verilog procedural assignment statement:

```
r8 = (r1 == r2) && r3 ? r4 : r6;
```

The conditional expression is `(r1 == r2) && r3` and the operands of the logical AND `&&` are the subexpressions `(r1 == r2)` and `r3`. The conditions for condition coverage are the truth or falsity of the conditional expression and these two subexpressions.

In the case of VHDL, expression `(a and b or d)` consists of two subexpressions: `(a and b)` (`expr1 or d`). The conditions for condition coverage are the truth or falsity of the whole expression and these two subexpressions.

- Subexpressions that are the operands to the logical AND `&&` or logical OR `||` operators in the conditional expression in an `if` statement. For example, in the following Verilog `if` statement:

```
if ((r1 ^ (!r2)) && (r3 == r4))
begin
  .
  .
  .
end
```

The subexpressions that are the operands of the logical and operator `&&` are `(r1 ^ (!r2))` and `(r3 == r4)` and the conditions for condition coverage are the truth or falsity of these VHDL subexpressions.

```
if ( A = '0') and ( B = '0') then
  . . .
```

```
end if;
```

- Subexpressions that are the operands to the logical AND `&&` or logical OR `||` operators in an expression in a continuous or procedural assignment statement. For example, in the following Verilog continuous assignment statement:

```
assign w2 = r5 || r6;
```

Subexpressions `r5` and `r6` are operands of the logical OR `||` operator and their truth or falsity are conditions in condition coverage.

For another example, in the following Verilog procedural assignment statement:

```
r6 = (r6 != r7) || r8;
```

Subexpressions `(r6 != r7)` and `r8` are operands of the logical OR `||` operator and their truth or falsity are conditions in condition coverage.

- Concurrent or sequential assignment statements in VHDL. For example:

```
d <= (mba < mbb) and ( mbb <=mbc) ;
```

- Sequential assignment statements in VHDL. For example:

```
a <= (b /= c) or d;      ( a, b must be type of BOOLEAN)
```

Note:

The `-cm_cond` compile-time option has arguments that can increase the number of subexpressions in condition coverage (see “[Enabling Condition Coverage for More Operators](#)”).

Verilog Conditional Expressions Not Monitored

Not all occurrences of conditional expressions with the right operators result in conditions for condition coverage. The following are cases where there are conditional expressions, but VCS and VCS MX do not monitor for condition coverage:

- In terminal connection lists in task-enabling statements:

```
#1 t1(r11 && r12,r12,r3);
```

- In instance connection lists:

```
dev d1(r11 && r12);
```

- In PLI routine calls:

```
$pli((r11 && r12);
```

- In `for` loops.

```
for (i=0;i<10;i=i+1)
    if(r1 && r2)
        r3=r1;
    else
        r3=r2;
```

In this example, VCS and VCS MX do not monitor the conditional expression `(r1 && r2)` for condition coverage.

By default, VCS and VCS MX do not monitor for conditional expressions in `for` loops (see “[Enabling Condition Coverage in For Loops](#)” for information on enabling this coverage).

- In user-defined tasks and functions:

```
task mytask;
    input in;
```

```

output out;
begin
  if (r1 && r2)
    out=in;
  else
    out=~in;
end
endtask

```

By default, VCS and VCS MX do not monitor for conditional expressions in user-defined tasks and functions (see “[Enabling Condition Coverage in Tasks and Functions](#)” for information on enabling this coverage).

VHDL Conditional Expressions Not Monitored

In the following instances, VCS MX cannot monitor for condition coverage and cmView cannot report condition coverage; no warning or error message appears.

- Conditions in sequential templates. For example:

```

if(clk'event and clk='1') then
  .
  .
  .

```

- Conditions with a vector of variable size, though the size is known through further evaluation. For example:

```

variable N : integer;
DATA (N to N+3) <= DATA2 (7-N downto 4-N) AND DATA3 (7-N
downto 4-N);

```

- Conditions within a generate block.
- Conditions within subprogram body.

- Conditions within an entity declaration region. For example, a subprogram defined in an entity.
 - Conditions within a package body. For example, a subprogram defined in a package body.
-

Modifying Condition Coverage

The `-cm_cond` compile-time option accepts arguments that can add conditions to condition coverage and change the information in the report files. The arguments which you can specify are as follows:

`full`

Specifies the following:

- Logical and non-logical conditions — the subexpressions of all operators, not just logical AND `&&` and logical OR `||`, are conditions for condition coverage (see “[Enabling Condition Coverage for More Operators](#)”).
- Multiple conditions — condition coverage reports show vectors containing multiple condition values. Each vector contains a value for each subexpression of the larger expression (see “[Enabling Coverage for All Possible Combinations of Vectors](#)”).
- Event control conditions (*Note: Does not apply to VHDL*) — the signals in the sensitivity list of an always are also conditions for condition coverage.
- Full vectors — the reports show all possible vectors of multiple condition values, not just the sensitized multiple condition value vectors (see “[Using Sensitized Multiple Condition Coverage Vectors](#)”).

Note:

In Verilog condition coverage, conditions with more than ten operands result in a warning message. VCS and VCS MX replace the full argument with the `allops` argument.

In VHDL condition coverage, conditions with more than six operands are reported in `std` format of sensitized vectors. VCS MX displays a warning message when it switches to the `std` format.

`std`

Specifies the following:

- Logical conditions — the subexpressions of just the logical AND `&&` and logical OR `||` operator are conditions for condition coverage.
- Multiple conditions
- Sensitized vectors or sensitized conditions — the only vectors are those where only one subexpression makes the conditional expression true or false, and one more vector to indicate if the conditional expression was either true or false, depending on the operator (see “[Using Sensitized Multiple Condition Coverage Vectors](#)”).

`basic`

Specifies the following:

- Logical conditions
- No multiple conditions — subexpression values are reported on separate lines (see “[Disabling Vector Conditions](#)”).

`sop` (*Note: Does not apply to VHDL*)

Specifies an alternative to sensitized vectors called condition SOP coverage, where a conditional expression is broken up into max terms and min terms. Each term is then marked as covered and uncovered (see “[Specifying Condition SOP Coverage](#)”).

`for` (*Note: Does not apply to VHDL*)

Enables compiling and monitoring conditional expressions in `for` loops (see “[Enabling Condition Coverage in For Loops](#)”).

`tf` (*Note: Does not apply to VHDL*)

Enables compiling and monitoring conditional expressions in user-defined tasks and functions (see “[Enabling Condition Coverage in Tasks and Functions](#)”).

`event` (*Note: Does not apply to VHDL*)

Specifies that the signals in event controls, in the sensitivity list position of an always block, are also conditions for condition coverage (see “[Enabling Coverage for Event Controls](#)”).

`allops`

Specifies that the subexpressions of all operators, not just logical AND `&&` and logical OR `||`, in conditional expressions are conditions for condition coverage.

`ports` (*Note: Does not apply to VHDL*)

Enables monitoring conditional expressions in a Verilog module instance port connection list (see “[Enabling Condition Coverage in Port Connection Lists](#)”).

`allvectors` (*Note: Does not apply to VHDL*)

Reports all possible vectors for an expression. If this option is not used, then VCS or VCS MX prints only the sensitized vectors (see “[Using the -cm_cond allvectors Option](#)”).

`scalarbitwise`

Reports sensitized vectors for bitwise expressions, having bitwise operators, and single-bit or one-bit operands . Use `allvectors` argument along with `scalarbitwise` to report all possible vectors for a bitwise expression.

You can specify more than one argument. If you do, use the plus delimiter (+) between arguments. For example:

```
-cm_cond basic+allops
```

Do not use the `full`, `std`, and `basic` arguments together.

The `std` argument specifies the default settings for condition coverage.

The different ways to modify both condition coverage and other types of coverage are explained in the sections: “[Filtering Constants](#)” on page 1-6, “[Using Glitch Suppression](#)” on page 1-34“[Omitting Coverage for Default Case Statements](#)” on page 1-43.

The following sections elaborate on the use of the `-cm_cond` compile-time option and explain other options that you can use to modify condition coverage.

Enabling Coverage for Event Controls

Note:

This feature does not apply to VHDL.

Verilog event controls can have subexpressions. For example:

```
@ (r1 or r2)
```

The subexpressions in this event control are `r1` and `r2`.

You can use the `event` argument to the `-cm_cond` compile-time option to make the occurrence of the transition specified by these subexpressions into a condition in conditional coverage. For example:

```
vcs source.v -cm cond -cm_cond event
```

The event control must meet the following requirements for its subexpressions to become conditions:

- The event control must be in the “sensitivity list” position, immediately following the `always` keyword for an `always` block.
- The event control expression list must contain more than one signal and also contain the `or` keyword or a comma.
- The event control must be explicit. This feature does not work for Verilog 2001 implicit event controls. For example:

```
always @*
```

VCS and VCS MX monitor each of the signals in this list as conditions for condition coverage. The resulting report appears similar to the following:

LINE 18					
STATEMENT	always @ (in1 or in2 or in3)	-1-	-2-	-2-	
EXPRESSION	-1-	-2-	-3-		
	changed	-	-		Covered
	-	changed	-		Covered
	-	-	changed		Covered

Enabling Condition Coverage for More Operators

The truth or falsity of subexpressions, that are the operands of the logical AND operator `&&` and the logical OR operator `||`, in conditional expressions and in assignment statements are, by default, conditions for condition coverage. To do this for subexpressions that are the operands of other operators with the `full` or `allops` arguments to the `-cm_cond` compile-time option, use one of the following command lines:

```
vcs source.v -cm cond -cm_cond full
```

or

```
vcs source.v -cm cond -cm_cond allops
```

[Table 4-1](#) lists the Verilog operators whose subexpression operands you can include in this expanded condition coverage. [Table 4-2](#) lists the VHDL operators whose subexpression operands you can include in this expanded condition coverage.

Table 4-1 Additional Verilog Operators For Condition Coverage

Operator	Description	Type
<code>==</code>	Logical equality	Binary
<code>!=</code>	Logical inequality	Binary
<code>&</code>	Bit-wise and	Binary
<code> </code>	Bit-wise inclusive or	Binary
<code>^</code>	Bit-wise xor	Binary
<code>^~</code> or <code>~^</code>	Bit-wise xnor	Binary
<code>&</code>	Reduction and	Unary
<code>~&</code>	Reduction nand	Unary
<code> </code>	Reduction or	Unary
<code>~ </code>	Reduction nor	Unary
<code>^</code>	Reduction xor	Unary

Table 4-1 Additional Verilog Operators For Condition Coverage

Operator	Description	Type
\sim^{\wedge}	Reduction xnor	Unary
@	Event Control	Unary

Table 4-2 Additional VHDL Operators for Condition Coverage

Operator	Description	Type
=	Equality	Binary
/=	Non-equality	Binary
<	Less than	Binary
<=	Less than or equal to	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary

The condition coverage reports would now also include something similar to the following:

LINE	22
STATEMENT	if ((in1 & in2 & in3))
	-1- -2- -3-
EXPRESSION	-1- -2- -3-
	0 1 1 Covered
	1 0 1 Not Covered
	1 1 0 Not Covered

The subexpressions of operators other than the logical AND `&&` and logical OR `||` are now conditions for condition coverage.

Enabling Condition Coverage in For Loops

By default, VCS and VCS MX do not monitor for conditional expressions in `for` loops. For example:

```
always @ r1
begin:named
```

```
integer i;
for (i=0;i<10;i=i+1)
  if(r1 && r2)
    r3=r1;
  else
    r3=r2;
end
```

In this example, VCS and VCS MX do not monitor the conditional expression (`r1 && r2`) for condition coverage.

This default behavior occurs because `for` loops typically exist in testbenches. You can instruct VCS or VCS MX to compile and monitor for conditional expressions in `for` loops with the `-cm_cond` `for` compile-time option and keyword argument. For example:

```
vcs source.v -cm cond -cm_cond for
```

Note:

This feature does not apply to VHDL.

Enabling Condition Coverage in Tasks and Functions

By default, VCS and VCS MX do not monitor conditions in user-defined tasks and functions. For example:

```
module test;
reg r1,r2,r3,r4,r5,r6;
.
.
.
task mytask;
  input in;
  output out;
  reg tr1,tr2,tr3;
begin
```

```

if (r1 && r2)
    out=in;
else
    if(tr1 || tr2)
        tr3=~in;
end
endtask
.
.
.
endmodule

```

VCS and VCS MX do not monitor the `(r1 && r2)` conditions or the `(tr1 || tr2)` conditions in this task.

You can instruct VCS or VCS MX to compile and monitor these conditions with the `-cm_cond_tf` compile-time option and keyword argument. For example:

```
vcs source.v -cm cond -cm_cond_tf
```

or

```
vcs cfg -cm cond -cm_cond_tf
```

VCS and VCS MX can monitor conditions in tasks (functions) independent of the operands in the task (function) being declared locally or in the module that contains the task (function).

Enabling Condition Coverage in Port Connection Lists

By default, VCS and VCS MX do not monitor conditional expressions in a port connection list in a Verilog module instantiation statement. For example:

```
module test;
```

```

.
.
.
dev d1 (.q(eIpu), .d(eIpurb), .en (rIpu || dIp || wIp),
.clk(cclk));
.
.
.
endmodule

```

By default, VCS and VCS MX do not monitor the `rIpu`, `dIp`, or `wIp` conditions. You can instruct VCS and VCS MX to monitor these expressions with the `ports` keyword argument to the `-cm_cond` compile-time option. For example:

```
vcs exp1.v -cm cond -cm_cond ports
```

or

```
vcs cfg -cm cond -cm_cond ports
```

The `cmView.long_c` file will now contain the following:

LINE	4	INSTANCE	PORTS	dev	d1((rIpu dIp wIp))	
				-1--	-2-	-3-
EXPRESSION		-1-	-2-	-3-		
		0	0	0		Covered
		0	0	1		Covered
		0	1	0		Covered
		1	0	0		Covered

Using Sensitized Multiple Condition Coverage Vectors

By default, VCS and VCS MX compile and monitor for sensitized multiple condition coverage vectors.

Sensitized multiple condition coverage vectors enable you to see if the execution of a block of code is sensitive to all the subexpressions in a conditional expression. Consider the following Verilog `if` statement:

```
if (in1 && in2 && && in3 && in4)
begin
.
.
.
end
```

VCS and VCS MX do not execute the begin-end block if any of the signals `in1`, `in2`, `in3` or `in4` are false. Because the operator in this expression is the logical AND operator `&&`, you would want to determine if:

- During simulation, each of these signals were the only instance that was false, and thus the sole cause of not executing the block.
- All of the signals were true, and therefore the block did execute.

```
if (in1 and in2 and in3 and in4) then
.....
end if;
```

The signals `in1`, `in2`, `in3` or `in4` are expected to be of type Boolean only.

To determine if the block was sensitive to all four signals, you can use sensitized condition coverage. With this coverage VCS and VCS MX look for the following patterns or vectors of values for these signals:

in1	in2	in3	in4
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0
1	1	1	1

These patterns become conditions in condition coverage, therefore instead of looking for eight conditions, that is the truth or falsity of all four signals, VCS and VCS MX look for five conditions, one condition where all of the subexpression signals are true, and four conditions where only one of them is false.

Alternatively, if the operator in this expression were the logical OR operator `||`, you would want to determine if:

- During simulation, each of these signals was the only one that was true, and the sole cause of executing the block.
- All of the signals were false and, therefore the block did not execute.

When cmView reports condition coverage, it duplicates these vectors of values and informs you of whether or not these vectors were covered. For example:

```

LINE      6
STATEMENT    if ((in1 && in2 && in3 && in4))
                  -1-      -2-      -3-      -4-
EXPRESSION    -1-      -2-      -3-      -4-

```

0	1	1	1	Not Covered
1	0	1	1	Not Covered
1	1	0	1	Not Covered
1	1	1	0	Not Covered
1	1	1	1	Not Covered

You can enable sensitized condition coverage with the `std` argument to the `-cm_cond` compile-time option:

```
vcs source.v -cm cond -cm_cond std
```

or

```
vcs cfg -cm cond -cm_cond std
```

However, you do not need to explicitly enable it. By default, VCS and VCS MX generate sensitized condition coverage. If you specify the `full` or `basic` arguments, you will not see sensitized condition coverage.

VHDL Data Type Difference

The VHDL hardware description language specifies that if the Boolean data type is the operand to the right of the AND, OR, NAND, and NOR operators, VCS MX evaluates it only if the evaluation of the left operand does not determine the value of the expression.

Therefore, for the expression:

A and B and C and D

If A and B has the Boolean data type, the sensitized vectors are as follows:

- 0 - - - The leftmost operand determines the value of the expression, which is 0. Therefore, VCS MX does not evaluate the operands to the right.
- 1 0 - - The leftmost operand enables the evaluation of the first right operand. The first right operand makes the expression 0. Therefore, VCS MX does not evaluate the other right operands.
- 1 1 1 0 The previous three left operands enable the evaluation of the last right operand. The last right operand makes the expression 0.
- 1 1 1 1 All the operands make the expression 1.

Using the `-cm_cond allvectors` Option

Note:

This feature does not apply to VHDL.

By default, VCS and VCS MX report only the sensitized vectors for any expression in your design.

```
module test;
reg a, b;
wire c;
reg [3:0] e,f;
wire [3:0] g;
assign c = a || b;
assign g = e |f;
endmodule
```

For example, if you compile the above example with the `-cm cond` option, VCS or VCS MX reports only the sensitized vectors for the expression "assign c = a || b", as follows:

LINE	6
STATEMENT	$c = (a \mid\mid b)$
	1 2
EXPRESSION	-1- -2-
	0 0 Not Covered
	0 1 Not Covered
	1 0 Not Covered

To view all possible vectors for the expression “`assign c = a || b`”, use the compile-time option `-cm_cond allvectors` along with `-cm cond`, as follows:

```
vcs source.v -cm cond -cm_cond allvectors
```

or

```
vcs -cm cond -cm_cond allvectors cfg
```

The report generated with the `-cm_cond allvectors` option is as follows:

LINE	6
STATEMENT	$c = (a \mid\mid b)$
	1 2
EXPRESSION	-1- -2-
	0 0 Not Covered
	0 1 Not Covered
	1 0 Not Covered
	1 1 Not Covered

Using Multiple Condition Value Vectors With Constant Filtering

When you use constant filtering, condition coverage reports do not include condition value vectors in which one value in the vector cannot be covered. The vectors it does display are not sensitized, neither are all possible vectors, except those filtered out by the `-cm_noconst` compile-time option.

In [Example 1-1](#), a certain condition, net F having a true value, cannot be covered. If we also used the `-cm_constfile` file to specify that signal test.D is permanently at 0 (see [“Treating Other Signals as Permanently at the 1 or 0 Value” on page 1-12](#) “Treating Other Signals as Permanently at the 1 or 0 Value” on page 1-12) the `cmView.long_c` file would contain the following in the case of Verilog coverage:

```
LINE      9
STATEMENT   C = (A && B && D)
             1     2     3
EXPRESSION  -1-    -2-    -3-
             0     1     0 | Not Covered
             1     0     0 | Not Covered
             1     1     0 | Not Covered

LINE      10
STATEMENT  E = (C || F)
             1     2
EXPRESSION -1-    -2-
             0     0 | Not Covered
             1     0 | Not Covered
```

For line 9, there are no vectors where signal `D` has a value of 1. Neither is there a vector where all the subexpressions have a value of 0, therefore, we are not seeing all possible vectors, only those that are filtered out. The three vectors for line 9 are the same vectors you would see if you replaced `D` in line 9 with its permanent value, 0.

Similarly for line 10, there are no vectors where `F` has a value of 1. The vectors are those you would see if you replaced `F` in line 10 with 0.

Specifying Condition SOP Coverage

Condition SOP coverage is an alternative to sensitized vectors in condition coverage. In condition SOP coverage, a conditional expression is broken up in terms of max terms and min terms. Condition SOP coverage employs Boolean logic.

Note:

This feature does not apply to VHDL coverage.

To enable condition SOP (Sum of Products) coverage, use the `-cm_cond` compile-time option and the `sop` keyword argument. For example:

```
vcs -f filename -cm cond -cm_cond sop
```

To understand how condition SOP coverage works, consider the following conditional expression:

```
((a == 2) || b && c) || (d && e)
```

VCS breaks this expression up into the smallest non-Boolean subexpressions (terminals in Boolean logic):

((a == 2) || (b && c) || (d && e))

Next, VCS/VCS MX finds a reduced set of controlling vectors. VCS/VCS MX looks for two types of controlling vectors: those that control whether the entire conditional expression evaluates to true, and those that control whether it evaluates to false.

First, VCS/VCS MX looks for vectors controlling whether the expression evaluates to true. It finds the minimal Sum Of Products (SOP) using these terminals and then extracts the vectors where each product terminal has a value of 1. The SOP format is:

(a == 2) + b.c + d.e

Therefore, VCS/VCS MX monitors for the following vectors:

(a == 2)
(b && c)
(d && e)

Next, VCS/VCS MX looks for the vectors that control whether the conditional expression evaluates to false. It finds the minimal Product Of Sums (POS) using these terminals and then extracts the vectors where each product terminal has a value of 0. The POS format is:

((a == 2) + c + e) . ((a == 2) + b + e) . ((a == 2) + c + d) . ((a == 2) + b + d)

Therefore, VCS/VCS MX also monitors for the following vectors:

```
! ((a == 2) || c || e)
! ((a == 2) || b || e)
! ((a == 2) || c || d)
! ((a == 2) || b || d)
```

Note:

The limit for SOP terms is 100. You cannot change this limit. If the condition crosses the limit of 100 terms, VCS and VCS MX switch to the normal sensitized vector scheme, which is the default for condition coverage.

Pragmas for Condition SOP Coverage

VCS and VCS MX have Pragmas for condition SOP coverage. To use these pragmas, you must enable condition coverage with the `-cm cond` compile-time option and keyword argument.

You use these pragmas instead of the `-cm cond sop` compile-time option and keyword argument. If you also include the `-cm cond sop` compile-time option and keyword argument, VCS and VCS MX ignore these pragmas.

```
// VCS sop_coverage_on start
    Specifies compiling the code that follows for condition SOP
    coverage.

//VCS sop_coverage_on end
    Specifies an end to the code that VCS or VCS MX compiles for
    condition SOP coverage.

//VCS sop_coverage_off start
    Also specifies an end to the code that VCS or VCS MX compiles
    for condition SOP coverage.

//VCS sop_coverage_off end
    Specifies resuming to compile the code that follows for condition
    SOP coverage.
```

Note:

The //vcs coverage on and //vcs coverage off pragmas have a higher priority and override these pragmas for condition SOP coverage.

Condition SOP Coverage Warning and Error Conditions

If the number of maxterms (pos) or minterms (sop) exceed the maximum limit, a warning or error message is displayed.

VCS or VCS MX displays a warning if you enabled the SOP for the expression with the -cm_cond sop compile-time option and keyword argument.

VCS or VCS MX displays an error if you enabled the SOP for the expression with the // vcs sop_coverage_on start pragma.

Note:

The pragma has higher priority over the compile-time option. When an expression exceeds the maximum number of maxterms/minterms, and the line is contained within pragmas and also had condition SOP enabled by the -cm_cond sop compile-time option and keyword argument, it will still be an error. This is the only exception to “If you also include the -cm_cond sop compile-time option and keyword argument, VCS and VCS MX ignores these pragmas.” in the section “[Pragmas for Condition SOP Coverage](#)”.

Enabling Bitwise Reporting for Vectors

You can have cmView report for vector signals that are conditions the coverage of each bit, arranged in a vector format. You enable this with the `allops` and `sop` arguments to the `-cm_cond` compile-time option.

Note:

This feature does not apply to VHDL.

The following example applies to Verilog:

```
vcs -f source_files -cm cond -cm_cond allops+sop
```

Note:

The `sop` argument also enables converting bitwise operators (see “[Converting Bitwise Operators](#)”).

Consider the following Verilog code example:

```
module test;
reg [2:0] r1, r2, r3;
.
.
.
always @ (r1 or r2)
r3 = (r1 & r2);
```

By default, (and using the `allops` argument) cmView reports condition coverage for the conditions in the assignment statement as follows:

```
LINE    17
STATEMENT    r3 = (r1 & r2)
              1-    2-
```

EXPRESSION	-1-	-2-	
BIT 0	0	1	Covered
	1	0	Covered
	1	1	Covered
BIT 1	0	1	Not Covered
	1	0	Covered
	1	1	Covered
BIT 2	0	1	Not Covered
	1	0	Not Covered
	1	1	Not Covered

Coverage is presented for each bit, one at a time.

If you include the `sop` argument, cmView reports the condition coverage as follows:

```
LINE    17
STATEMENT   r3 = ((r1 & r2))

MINTERM r1 & r2          | Covered 011
MAXTERM ~r1                | Covered 111
MAXTERM ~r2                | Covered 111
```

In this example:

- The first line specifies that the expression `r1 & r2` is a minterm, because it assigns a true value to reg `r3`. The expression was true, and therefore covered. The `011` indicates that the leftmost bits of `r1` and `r2` were never 1, but the center and rightmost bits did have a value of 1 at one time during the simulation.
- The second line shows expression `~r1`; a tilde `~` is added to show that the report is about the inverse or 0 values of `r1`. It is a maxterm because it assigns a 0 value to the corresponding bits in `r3`. It was covered in the sense that `r1` had a false value during simulation. The `111` indicates that all three bits had a 0 value at one time during simulation.

- The third line indicates the same thing about $r2$.

Converting Bitwise Operators

When VCS/VCS MX reads conditional expressions that contain the \wedge bitwise XOR and $\sim \wedge$ bitwise XNOR operators, you can instruct VCS/VCS MX to reduce the expression to negation and logical AND or OR. To do this, use the `sop` argument to the `-cm_cond` compile-time option, as follows:

Note:

This feature does not apply to VHDL.

The following example applies to Verilog:

```
vcs -f source_files -cm cond -cm_cond allops+sop
```

Specifying the `sop` argument adds the following coverage data to the condition coverage report:

LINE 18	<pre>STATEMENT if (((r1 ^ r2)))}</pre>
	 Additional coverage data

Enabling Coverage for All Possible Combinations of Vectors

Sensitized condition coverage does not look for all combinations of values of the subexpressions in a conditional expression, it only looks for the conditions where the following code is sensitive to each subexpression. You might want to see if you have coverage for all combinations of values in these condition vectors.

To do this, use the `full` argument to the `-cm_cond` compile-time option, as follows:

```
vcs source.v -cm cond -cm_cond full
```

or

```
vcs -cm cond -cm_cond full cfg
```

Note:

The `full` argument also specifies additional conditions that are the subexpressions of other operators (see “[Enabling Condition Coverage for More Operators](#)”).

Consider the following example, which is the same example as shown in the section, “[Using Sensitized Multiple Condition Coverage Vectors](#)”:

Verilog:

```
if (in1 && in2 && in3 && in4)
    begin
        .
        .
        .
    end
```

With the `full` argument to the `-cm_cond` compile-time option, VCS and VCS MX look for the following conditions:

in1	in2	in3	in4
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Therefore, VCS and VCS MX look for, and cmView reports on, 16 conditions, these 16 vectors. The report appears similar to the following:

LINE	6	STATEMENT	if ((r1 && r2 && r3 && r4))	1-	2-	3-	4-	
EXPRESSION				-1-	-2-	-3-	-4-	
				0	0	0	0	Not Covered
				0	0	0	1	Not Covered
				0	0	1	0	Not Covered
				0	0	1	1	Not Covered
				0	1	0	0	Not Covered

0	1	0	1	Not Covered
0	1	1	0	Not Covered
0	1	1	1	Not Covered
1	0	0	0	Not Covered
1	0	0	1	Not Covered
1	0	1	0	Not Covered
1	0	1	1	Not Covered
1	1	0	0	Not Covered
1	1	0	1	Not Covered
1	1	1	0	Not Covered
1	1	1	1	Not Covered

As previously mentioned, the full argument makes the subexpressions of other operators conditions in condition coverage. This indicates that cmView reports all possible vectors, not just the sensitized ones.

For the VHDL Boolean data type, the report would look as follows:

LINE	6	STATEMENT	if ((r1 and r2 and r3 and r4))			
			1-	2-	3-	4-
EXPRESSION		-1-	-2-	-3-	-4-	
		0	-	-	-	Not Covered
		1	0	-	-	Not Covered
		1	1	0	-	Not Covered
		1	1	1	0	Not Covered
		1	1	1	1	Not Covered

This is because for the Boolean data type, if evaluating the left operand determines the value of the expression, the operand to the right is never evaluated.

Disabling Vector Conditions

If you do not want VCS or VCS MX to look for vectors of conditions, or cmView to report them, you can specify the `basic` argument to the `-cm_cond` compile-time option:

```
vcs source.v -cm cond -cm_cond basic
```

or

```
vcs -cm cond -cm_cond basic cfg
```

The following example is the same example as shown in the sections, “[Using Sensitized Multiple Condition Coverage Vectors](#)” and “[Enabling Coverage for All Possible Combinations of Vectors](#)”:

```
if (in1 && in2 && in3 && in4)
begin
.
.
.
end
```

In this example, if you specify the `basic` argument to the `-cm_cond` compile-time option, the Verilog report appears as follows:

LINE	6	STATEMENT	if ((in1 && in2 && in3 && in4))				
			1-	2-	3-	4-	
EXPRESSION			-1-	-2-	-3-	-4-	
			0	-	-	-	Not Covered
			1	-	-	-	Not Covered
			-	0	-	-	Not Covered
			-	1	-	-	Not Covered
			-	-	0	-	Not Covered
			-	-	1	-	Not Covered
			-	-	-	0	Not Covered

- - - 1 | Not Covered

Note:

The basic argument also specifies that only the subexpressions of the logical AND operator `&&` and the logical OR operator `||`, are conditions, therefore, if you use this argument and want other conditions, also include the `allops` argument.

```
if (in1 AND in2 AND in3 AND in4)
.
.
.
```

The report for VHDL appears as follows:

LINE	6				
STATEMENT	if ((in1 and in2 and in3 and in4))	1-	2-	3-	4-
EXPRESSION	-1-	-2-	-3-	-4-	
	0	-	-	-	Not Covered
	1	-	-	-	Not Covered
	-	0	-	-	Not Covered
	-	1	-	-	Not Covered
	-	-	0	-	Not Covered
	-	-	1	-	Not Covered
	-	-	-	0	Not Covered
	-	-	-	1	Not Covered

Note:

The basic argument also specifies that only the subexpressions of the logical AND operator `&&` and the logical OR operator `||`, are conditions, therefore, if you use this argument and want other conditions, also include the `allops` argument.

The report is different for the Boolean data type.

LINE	6				
STATEMENT	if ((in1 and in2 and in3 and in4))	1-	2-	3-	4-
EXPRESSION	-1-	-2-	-3-	-4-	

0	-	-	-	Not Covered
1	-	-	-	Not Covered
1	0	-	-	Not Covered
1	1	-	-	Not Covered
1	1	0	-	Not Covered
1	1	1	1	Not Covered

This is different because the VHDL language specifies that VCS MX cannot evaluate a Boolean operand to the right of the logical AND operator `&&` unless the evaluation of the Boolean operator to the left of the AND operator is evaluated first.

Excluding Conditions and Vectors From Reports

With multiple cmView batch runs, you can exclude from the coverage reports, any conditions and vectors that you specify in a file. You can edit a file that cmView writes that lists all the conditions and vectors in the design or write your own file. The procedure is as follows:

1. Include the `-cm_report cond_ids` command-line option and argument when you run cmView in batch mode the first time. For example:

Verilog: `vcs -cm_pp -cm cond -cm_report cond_ids`

VHDL: `cmView -cm -b cond -cm_report cond_ids`

This instructs cmView to include ID numbers for conditions in its reports.

For example, each condition expression has an ID number in the following section on module instance `top.dev1` (Verilog) in the `cmView.long_c` file:

```

MODULE top.dev1

FILE /net/design/dev.v
-----

LINE    22
CONDITION_ID 1
STATEMENT   if ((in1 && in2 && in3 && in4))
             -1-  -2-  -3-  -4-
EXPRESSION  -1-  -2-  -3-  -4-
              0    1    1    1 | Not Covered
              1    0    1    1 | Not Covered
              1    1    0    1 | Not Covered
              1    1    1    0 | Covered
              1    1    1    1 | Covered

LINE    24
CONDITION_ID 2
STATEMENT   r1 = (in1 && in2)
             -1-  -2-
EXPRESSION  -1-  -2-
              0    1  | Covered
              1    0  | Covered
              1    1  | Covered

```

The conditional expression on line 22 has an ID of 1 and there are five vectors, or lines, of coverage values. The expression with the logical and operator on line 24 has an ID of 2 and there are three vectors, or lines, of coverage values.

Note:

In the condition coverage report files, the numbering of the conditions restart with every instance.

Using the `-cm_report cond_ids` command-line option does not eliminate the necessity of using compile-time and runtime options that modify condition coverage. For example, you need the `-cm_cond allops` or `-cm_cond full` compile-time option and keyword arguments to have the expressions that are operands of the bitwise and (`&`) or bitwise or (`|`) reported as conditional expressions.

`cmView` also writes the `cm_excludedCond_generated` file in the `simv.cm/db/verilog` directory (for VHDL, it writes it in the `simv.cm/db/vhdl` directory). The following is an example of its contents for instance `top.dev1`:

```
Instance top.dev1
# Condition 1
# Vector 1
# Vector 2
# Vector 3
# Vector 4

#Condition 2
# Vector 1
# Vector 2
# Vector 3
```

The section on instance `top.dev1` begins with a line beginning with the keyword `Instance` followed by the instance hierarchical name.

As shown in the previous example of the `cmView.long_c` file, there are two conditional expressions in the instance, they have the IDs 1 and 2.

2. Edit the `cm_excludedCond_generated` file.

To exclude a condition from coverage, remove the pound (#) character from the condition line and do not remove this character from any of the lines for the vectors.

To exclude one or more vectors for a condition from coverage, remove the pound (#) character from the condition line and the lines for the vectors.

The following is an example of an edited `cm_excludeCond_generated` file:

```
Instance top.dev1
Condition 1
Vector 1
Vector 2
# Vector 3
# Vector 4

Condition 2
# Vector 1
# Vector 2
# Vector 3
```

This file excludes the first two vectors of condition 1 and all of condition 2.

3. Run `cmView` in batch mode again, specifying the file with the `-cm_exclude` command-line option. For example:

```
Verilog: vcs -cm_pp -cm cond -cm_exclude simv.cm/db/
verilog/cm_excludeCond_generated
```

```
VHDL: cmView -b -cm cond -cm_exclude simv.cm/db/vhdl/
cm_excludeCond_generated
```

Note:

If you add new conditions to your source code, you need to start this process over again.

Specifying Continuous Assignment Coverage

The conditional expressions in Verilog continuous assignments are conditions for condition coverage. For example, if a module definition contains the following continuous assignment:

```
assign w1 = r1 ? (r2 ? (r3 ? 2'b11 : 2'b01) : 2'b10) : 2'b00;
```

Each of the conditional expressions, in this case signals `r1`, `r2`, and `r3`, are conditional expressions for condition coverage, and `cmView` reports them as follows:

```
MODULE test

FILE /net/design1/branch.v
-----

LINE    11
STATEMENT   w1 = (r1 ? (r2 ? (r3 ? 2'b11 : 2'b1) : 2'b10)
: 2'b0)
           1-
EXPRESSION   -1-
              0 | Not Covered
              1 | Covered

LINE    11
STATEMENT   w1 = (r1 ? (r2 ? (r3 ? 2'b11 : 2'b1) : 2'b10)
: 2'b0)
           1-
EXPRESSION   -1-
              0 | Not Covered
              1 | Covered
```

```

LINE    11
STATEMENT   w1 = (r1 ? (r2 ? (r3 ? 2'b11 : 2'b1) : 2'b10)
: 2'b0)
               1-
EXPRESSION   -1-
              0  | Not Covered
              1  | Covered

```

//

// Module Coverage Summary

	TOTAL	COVERED	PERCENT
// conditions	6	3	50.00
// logical	6	3	50.00

You might not want to see which conditional expressions are covered but rather what values, possible branches, were assigned to signal w1. To have cmView report this instead, include the **-cm_cond_branch** command-line option on the cmView (or vcs -cm_pp) command line. For example:

vcs -cm_pp -cm cond -cm_cond_branch

cmView reports the following:

```

MODULE test

FILE  /net/design1/branch.v
-----

LINE    11
STATEMENT   w1 = (r1 ? (r2 ? (r3 ? 2'b11 : 2'b1) : 2'b10)
: 2'b0)
               --1--  --2--  --3--  --4--
BRANCH      -1-    2'b11  | Covered
BRANCH      -2-    2'b1   | Not Covered
BRANCH      -3-    2'b10  | Not Covered

```

```

BRANCH      -4-      2'b0 | Not Covered
//-----


//          Module Coverage Summary

//          TOTAL      COVERED      PERCENT
// conditions      0          0        100.00
// branches        4          1        25.00

//-----
```

Excluded Subexpressions

The truth or falsity of subexpressions with explicit X or Z value operands cannot be conditions for condition coverage.

Note:

This does not apply to VHDL.

For example, in Verilog:

```
if (a == 'bx) && (b == 'bz) )
```

The truth or falsity of subexpression operands of the operators in [Table 4-3](#) are not conditions in condition coverage even when you include the `full` or `allops` arguments to the `-cm_cond` compile-time option.

Table 4-3 Operators Excluded From Condition Coverage

Operator	Description	Type
<code>==</code>	Case equality	Binary
<code>!=</code>	Case inequality	Binary
<code>~</code>	Bit-wise negation	Unary
<code>!</code>	Logical negation	Unary

Using Condition Coverage Glitch Suppression

The `-cm_glitch positive_integer` compile-time option specifies the simulation time period of glitch suppression (see [“Using Glitch Suppression” on page 1-34](#) “Using Glitch Suppression” on page 1-34).

In glitch suppression for condition coverage, like glitch suppression for line coverage (see [“Line Coverage Glitch Suppression” on page 2-89](#)), VCS/VCS MX looks for fast triggering of always blocks, not initial blocks, and the glitch period is an interval of simulation time in which there are one or more executions of the statements in the always block.

Note:

Glitch suppression does not work for VHDL code.

When you use glitch suppression, VCS/VCS MX only monitors and records the results on the last execution of the always block during the glitch period.

Consider the following example:

```
module dev (out,in1,in2,in3,in4);
output out;
input in1,in2,in3,in4;
reg out;

.

.

.

always @ (in1 or in2 or in3)
if (in4)
    out=in1 && in2 && in3;
else
    out=~in4;
.

.

.

endmodule
```

Glitches or narrow pulses on the input ports could cause the always block to execute several times, while little or no simulation time elapses, before the values on the input ports settle and there is a final execution of the always block.

In this example, the default condition coverage is on the assignment statement that assigns an expression with the logical AND `&&` operator:

```
out=in1 && in2 && in3;
```

These glitches, although momentary, by default would show a high amount of coverage. For example:

LINE	29
STATEMENT	out = (in1 && in2 && in3)
	-1- -2- -3-
EXPRESSION	-1- -2- -3-
	0 1 1 Covered

1	0	1	Covered
1	1	0	Covered
1	1	1	Covered

Including the `-cm_glitch` compile-time option, specifying a short simulation time interval, but longer than the glitches or narrow pulses, changes the coverage to the following:

LINE	29	STATEMENT	out = (in1 && in2 && in3)	-1-	-2-	-3-	
EXPRESSION				-1-	-2-	-3-	
				0	1	1	Not Covered
				1	0	1	Not Covered
				1	1	0	Not Covered
				1	1	1	Not Covered

Condition Coverage Reports

cmView writes condition coverage report files when the following conditions are met:

- You compile your design for condition coverage. For example:

```
vcs source.v -cm cond
```

or

```
vcs -cm cond cfg
```

- You instruct VCS or VCS MX to monitor for condition coverage during simulation. For example:

```
simv -cm cond
```

- You instruct cmView to write condition coverage reports. For example:

```
vcs -cm_pp -cm cond
```

cmView writes its report files in the `./simv.cm/reports` directory.

The report files, which cmView creates regarding condition coverage, are as follows:

`cmView.long_c`

A long report file, organized by module instance, containing comprehensive information about the condition coverage of the design.

`cmView.long_cd`

Another long report file, similar to the `cmView.long_c` file, but organized by module definition instead of by module instance.

`cmView.hier_c`

Summary condition coverage report where coverage data is presented in subhierarchies in the design. For each instance, includes the data for each specific instance and all instances hierarchically under the instance.

`cmView.mod_c`

A summary condition coverage report file where coverage data is for module instances in the design, not subhierarchies. For each instance the data is for that instance alone.

`cmView.mod_cd`

Another summary condition coverage report file, similar to the `cmView.mod_c` file, but organized by module definition instead of by module instance.

`cmView.short_c`

A short report file containing only sections for module instances where there is not 100% condition coverage. `cmView` only lists the conditions that were not covered. The report ends with summary information.

`cmView.short_cd`

Another short report file, similar to the `cmView.short_c` file, but organized by module definition instead of by module instance.

To understand the contents of these report files, consider the following Verilog example and its condition coverage in the sections that follow. This example is taken from the `simv.cm/reports/` annotated directory. In this directory, `cmView` writes annotated module definitions for the module instances in the design. The copies of the source file are named after their corresponding instance name.

Example 4-1 Example for Condition Coverage

```
1 module test;
2 reg r1,r2,r3,r4,r6,r7,r8,r12;
3 reg [1:0] r5,r9,r10,r11;
4 wire w1,w2, w3;
5 initial
6 begin
7 #10 r1 = 1;
8     r5 = 2'b00;
9 #10 r2 = 0;
.
.
.
21 #10 $finish;
22 end
23
24 assign w1 = r1 ^~ r2 ? r2 : r3;
25
26 assign w2 = r5 || r6;
27
```

```

28 always @ (posedge r1 or r2)
29 begin
30     r8 = (r1 == r2) && r3 ? r4 : r6;
31
32     if ((r1 ^ (!r2)) && (r3 == r4))
33         begin
34             r6 = (r6 != r7) && r8;
35         end
36     else
37         begin
38             r6 = 1;
39         end
40     if (^r5)
41         begin
42             r9= r10 | r11;
43         end
44     else
45         begin
46             r6= ^r10;
47         end
48 end
49 dev dev1 (r6,r7,r8,r12,w3);
50 endmodule
51
52 module dev(in1,in2,in3,in4,out1);
53 input in1,in2,in3,in4;
54 output out1;
55 reg[1:0] devr1;
56 reg devr2, devr3;
57
58 always @ (in1 or in2)
59 begin
60     devr1 = in1 ? {in2,in3} : {in3,in4};
61     while (((~^devr1) != devr2) || !devr3)
62         begin
63             devr3=1;
64         end
65     if(in1 && in2 && in3 && in4)
66         begin
67             devr2=0;
68         end
69 end

```

```
70 endmodule
```

The cmView.long_c File

The `cmView.long_c` file that `cmView` writes for the source code in [Example 4-1](#) is as follows (this example file is interrupted in a number of places to explain its contents):

Note:

This design was not compiled with the `-cm_cond` compile-time option.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version number  
// User: user name  
// Date: Day Month date hour:minute:second year  
  
//                                     LONG CONDITION COVERAGE REPORT  
  
//*****  
// MODULE INSTANCE COVERAGE  
  
// This section contains coverage for each instance of a module  
  
The report begins with a section on the top-level module named test.  
  
MODULE test  
  
FILE source file pathname of the module definition  
-----  
-----  
LINE 30  
STATEMENT   r8 = (((r1 == r2) && r3) ? r4 : r6  
                  -----1----- 2-  
EXPRESSION    -1-    -2-  
                0        1    | NotCovered
```

1	0	Not Covered
1	1	Not Covered

In line 30, the truth or falsity of the conditional expression `((r1 == r2) && r3)` (labeled 1) in a procedural assignment statement is a condition. This conditional expression uses the logical AND `&&` operator, so the truth or falsity of its subexpressions are also conditions.

```
LINE    32
STATEMENT   if (((((r1 ^ (!r2))) && ((r3 == r4))))
               -----1----- -----2-----
EXPRESSION   -1-   -2-
              0     1 | Not Covered
              1     0 | Not Covered
              1     1 | Not Covered
```

In a conditional statement (`if` statement), the truth or falsity of the subexpressions of the conditional expression can be conditions for condition coverage. By default, they are conditions if they are subexpressions of the logical AND `&&` operator.

The following is an example of sensitized condition coverage, the default format of condition coverage reports. In this example, there are vectors of values. The values for `(r1 ^ (!r2))` (labeled 1) and `(r3 == r4)` (labeled 2) are on vector lines together. The vectors you see are the sensitized vectors which you see on the vectors when the following occurs:

- Subexpression 1 alone has a 0 value and thus prevents the conditional expression from evaluating to true. This condition did not happen.
- Subexpression 2 alone has a 0 value and thus prevents the conditional expression from evaluating to true. This condition did not happen.

- Both subexpressions had a value of 1 enabling the conditional expression to evaluate to true. This condition also did not happen.

```
LINE    24
STATEMENT   w1 = ((r1 ~^ r2)) ? r2 : r3
               -----1-----
EXPRESSION   -1-
              0 | Covered
              1 | Covered
```

In this example, the truth or falsity of the conditional expressions for the conditional operator in a continuous assignment statement, are conditions for condition coverage.

```
LINE    26
STATEMENT   w2 = (r5 || r6)
               1-      2-
EXPRESSION   -1-      -2-
              0      1 | Not Covered
              1      0 | Not Covered
//-----
//          Module Coverage Summary
//          TOTAL    COVERED    PERCENT
//  conditions        11        2        18.18
//  logical           11        2        18.18
```

The section on the top-level module ends with summary information. This report is used for default condition coverage where the only conditions are the truth or falsity of conditional expressions and subexpressions of logical operators in these expressions. These conditions are all classified as logical.

The following is a section on module instance, `top.dev1`.

```
//-----
MODULE test.dev1
FILE source file pathname of the module definition
-----
LINE    60
```

```

STATEMENT    devr1 = in1 ? ({in2, in3}) : ({in3, in4})
              -1-
EXPRESSION   -1-
              0  | Not Covered
              1  | Not Covered

LINE      61
STATEMENT  while (((((~^devr1) != devr2)) || (!devr3)))
              -----1-----  -----2-----
EXPRESSION   -1-  -2-
              0    0  | Covered
              0    1  | Not Covered
              1    0  | Not Covered

LINE      65
STATEMENT  if ((in1 && in2 && in3 && in4))
              -1-  -2-  -3-  -4-
EXPRESSION   -1-  -2-  -3-  -4-
              0    1    1    1  | Not Covered
              1    0    1    1  | Not Covered
              1    1    0    1  | Not Covered
              1    1    1    0  | Not Covered
              1    1    1    1  | Not Covered

```

The following is another example of sensitized condition coverage. In this expression, each signal is a subexpression. We see vectors of values with a value for each signal in all the vectors. We do not see all the possible vectors, we only see:

- The vectors where the conditional expression was sensitive to each signal, that is the signal alone prevented the conditional expression from evaluating to true.
- A final vector that shows whether the conditional expression was ever true.

```

//-----
//          Module Coverage Summary
//          TOTAL    COVERED    PERCENT
//  conditions        10        1        10.00
//  logical           10        1        10.00
//-----

```

The section on the instance ends with summary information.

```
//*****  
//          Total Module Instance Coverage Summary  
//  
//      conditions      TOTAL    COVERED    PERCENT  
//      logical        21        3          14.29  
//
```

The report ends with summary information for the entire design.

Reporting the Test That Caused Condition Coverage

If you include the `-cm_report testlists` cmView command-line option and keyword argument, cmView adds information about which test file (intermediate results file) caused a condition to be covered in the `cmView.long_c` and `cmView.long_cd` files.

cmView does this by listing the test files and assigning them an index number. For example:

```
//          MERGED TESTS LIST  
Test No.   Test Name           Coverage File Name  
  
0          test1              ./simv.cm/coverage/verilog/test1  
1          test2              ./simv.cm/coverage/verilog/test2  
2          test3              ./simv.cm/coverage/verilog/test3
```

Then, cmView adds a column when it reports on an instance to indicate which test file or files, by index number, caused the condition to be covered. For example:

LINE	6			
STATEMENT	if ((in1 in5))			
	-1-	-2-		
EXPRESSION	-1-	-2-	Test Nos.	
	0	0	Covered	2,3
	0	1	Covered	0
	1	0	Covered	2,3

LINE 11				
STATEMENT	if ((in2 in5))	-1-	-2-	
EXPRESSION		-1-	-2-	Test Nos.
		0	0	Covered 3
		0	1	Covered 0,1
		1	0	Covered 3
LINE 16				
STATEMENT	if ((in3 in5))	-1-	-2-	
EXPRESSION		-1-	-2-	Test Nos.
		0	0	Covered 2
		0	1	Covered 0,1
		1	0	Covered 2
LINE 20				
STATEMENT	if ((in4 in5))	-1-	-2-	
EXPRESSION		-1-	-2-	Test Nos.
		0	0	Covered 2,3
		0	1	Covered 0,1
		1	0	Covered 2,3
LINE 24				
STATEMENT	if ((in6 in5))	-1-	-2-	
EXPRESSION		-1-	-2-	Test Nos.
		0	0	Covered 2,3
		0	1	Covered 1
		1	0	Covered 2,3

By default, cmView only does this for the first three test files that it finds by alphanumeric order of the test file names. You can instruct cmView to do this for more or fewer test files by replacing the `-cm_report testlists` cmView (or `vcs -cm_pp`) command-line option and keyword argument with the `-cm_report_testlists_max int` command-line option and integer argument. For example:

```
vcs -cm_pp -cm cond -cm_report_testlists_max 4
```

The cmView.hier_c File

The cmView.hier_c file that cmView writes for the source code shown in [Example 4-1](#), is as follows:

```
// Synopsys, Inc.  
//  
// Generated by: cmView version number  
// User: user name  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE HIERARCHICAL INSTANCE COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// instance of a module. The statistics take into account all sub-hierarchies  
// under the instance.  
  
Test Coverage Result: Total Coverage  
  
Module Name           Logical  Logical      Non-logical Non-logical  
Events   Events          (%)        (%)  
test            26.09    6/23     100.00    0/0      100.00 0/0  
test.dev1        0.00    0/10     100.00    0/0      100.00 0/0  
  
//*****  
//      Total Module Instance Coverage Summary  
  
//           TOTAL    COVERED    PERCENT  
//   conditions      23        6        26.09  
//   logical         23        6        26.09
```

In this report, the information for each instance includes the instances under them in the hierarchy. Therefore, the information for the top-level module is the same for the entire design.

Logical refers to subexpressions of the logical operators.
Non-logical refers to other subexpressions. Events are not implemented.

The cmView.mod_c File

The `cmView.mod_c` file that `cmView` writes for the source code shown in [Example 4-1](#), is similar to the `cmView.hier_c` file, except that the summary data for an instance exists only for that instance and not the subhierarchy under the instance.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version number  
// User: user name  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE INSTANCE COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// instance of a module. The statistics do not take into account any  
// subhierarchy under the instance.  
  
Test Coverage Result: Total Coverage  


| Module Name<br>Events | Logical<br>(%) | Logical<br>(%) | Non-logical<br>(%) | Non-logical<br>(%) |
|-----------------------|----------------|----------------|--------------------|--------------------|
| test                  | 46.15          | 6/13           | 100.00             | 0/0                |
| test.dev1             | 0.00           | 0/10           | 100.00             | 0/0                |

  
//*****  
// Total Module Instance Coverage Summary  


|               | TOTAL | COVERED | PERCENT |
|---------------|-------|---------|---------|
| // conditions | 23    | 6       | 26.09   |
| // logical    | 23    | 6       | 26.09   |


```

In this report the information for an instance is only for the instance.

The cmView.mod_cd File

The cmView.mod_cd file is similar to the cmView.mod_c file, except that the information is organized by module definition instead of by module instance. The information for a module definition is cumulative for all instances of the module.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version number  
// User: user name  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE DEFINITION COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// module definition. The coverage is cumulative over all the instances  
// of the module  
  
Test Coverage Result: Total Coverage  


| Module Name<br>Events | Logical<br>Events | Logical<br>(%) | Non-logical<br>Events | Non-logical<br>(%) |
|-----------------------|-------------------|----------------|-----------------------|--------------------|
| test                  | 46.15             | 6/13           | 100.00                | 0/0                |
| dev                   | 0.00              | 0/10           | 100.00                | 0/0                |

  
//*****  
// Total Module Definition Coverage Summary  


| conditions | TOTAL | COVERED | PERCENT |
|------------|-------|---------|---------|
| logical    | 23    | 6       | 26.09   |


```

The cmView.short_c File

The cmView.short_c file reports only on conditions that are not covered.

```
// Synopsys, Inc.
```

Condition Coverage

```

//  

// Generated by: cmView 6.1Beta  

// User: pmcgee  

// Date: Wed Oct 17 16:16:38 2001

```

SHORT CONDITION COVERAGE REPORT

```

//*****  

// MODULE INSTANCE COVERAGE

```

// This section contains coverage for each instance of a module

Test Coverage Result: Total Coverage

MODULE test

FILE *source file pathname of the module definition*

LINE 30					
STATEMENT	r8 = ((r1 == r2) && r3) ? r4 : r6				
		-----1-----	2-		
EXPRESSION	-1- -2-				
	1 0	Not Covered			

LINE 32					
STATEMENT	if (((r1 ^ (!r2)) && ((r3 == r4))))				
		-----1-----	-----2-----		
EXPRESSION	-1- -2-				
	0 1	Not Covered			
	1 0	Not Covered			
	1 1	Not Covered			

LINE 26					
STATEMENT	w2 = (r5 r6)				
		1-	2-		
EXPRESSION	-1- -2-				
	0 0	Not Covered			
	0 1	Not Covered			
	1 0	Not Covered			

MODULE test.dev1

FILE *source file pathname of the module definition*

LINE 60					
STATEMENT	devr1 = in1 ? ({in2, in3}) : ({in3, in4})				
		-1-			
EXPRESSION	-1-				
	0	Not Covered			
	1	Not Covered			

```

LINE   61
STATEMENT  while (((((~^devr1) != devr2)) || (!devr3))))
-----1-----2-----
EXPRESSION -1-    -2-
      0      0 | Not Covered
      0      1 | Not Covered
      1      0 | Not Covered

LINE   65
STATEMENT  if ((in1 && in2 && in3 && in4))
-----1-  -2-  -3-  -4-
EXPRESSION -1-    -2-    -3-    -4-
      0      1      1      1 | Not Covered
      1      0      1      1 | Not Covered
      1      1      0      1 | Not Covered
      1      1      1      0 | Not Covered
      1      1      1      1 | Not Covered

//*****Total Module Instance Coverage Summary*****
//          Total Module Instance Coverage Summary
//          TOTAL    COVERED    PERCENT
// conditions        23        6        26.09
// logical           23        6        26.09

```

The cmView.short_cd File

The `cmView.short_cd` file is similar to the `cmView.short_c` file, except that the information is organized by module definition instead of by module instance. The following is an excerpt from that file:

```

MODULE dev

FILE source file pathname of the module definition
-----

LINE   60
STATEMENT  devr1 = in1 ? ({in2, in3}) : ({in3, in4})
-----1-
EXPRESSION -1-
      0 | Not Covered
      1 | Not Covered

LINE   61
STATEMENT  while (((((~^devr1) != devr2)) || (!devr3))))

```

Viewing Condition Coverage with the Coverage Metrics GUI

cmView is also the graphical user interface (GUI) for VCS and VCS MX coverage metrics. It displays graphical representations of the coverage information recorded by VCS and VCS MX. Perform the following procedure to see the condition coverage results which VCS and VCS MX recorded for the simulation of the source code in [Example 4-1](#):

1. Enter the following command line:

Verilog:

```
cs -cm_pp gui -cm cond
```

VHDL:

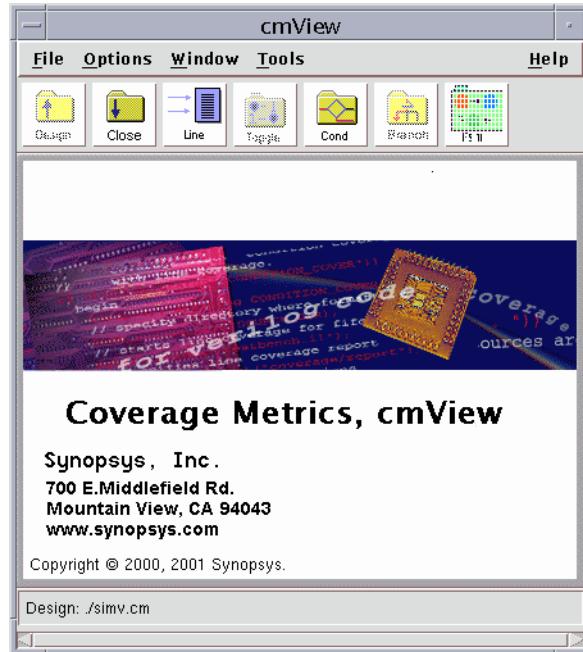
```
cmView -cm cond
```

Note:

At compile-time, if you specified a different coverage metrics database with the `-cm_dir` compile-time option, also include the option on the cmView (or `vcs -cm_pp gui`) command line.

The command line opens the main window for cmView (see [Figure 4-1](#)).

Figure 4-1 cmView Main Window



This command line also instructs cmView to read the following design files:

- Verilog: ./simv.cm/db/verilog/cm.decl_info
- VHDL: /simv.cm/db/vhdl/cm.decl_info

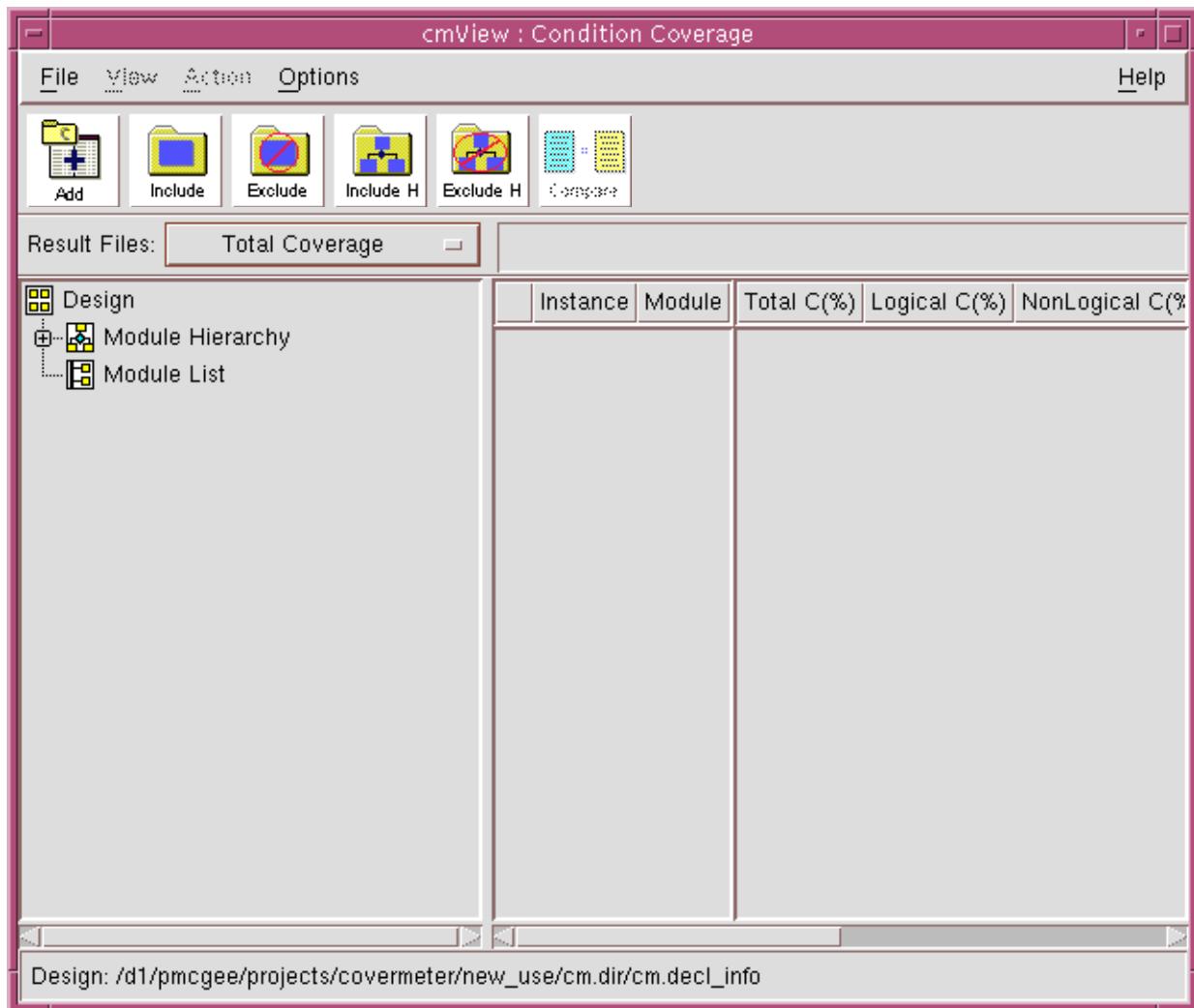
cmView reads this file to learn about the design and its hierarchy. Read the intermediate data files for condition coverage in one of the following directories:

- Verilog: simv.cm/coverage/verilog
- VHDL: simv.cm/coverage/vhdl

2. Click the Condition Coverage toolbar button.

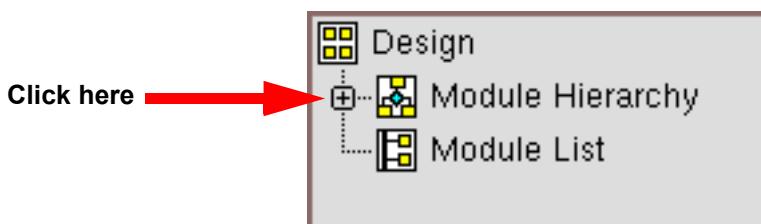


This opens the Condition Coverage window.



Use this window to view condition coverage data from the intermediate data files written by VCS and VCS MX. The left pane is used for opening the design hierarchy, the right pane is used for displaying coverage information.

3. By default, cmView displays coverage summary information on a per module instance basis. Therefore, the Condition Coverage window needs a way to display the design hierarchy so you can select module instances to examine their condition coverage. Click the plus symbol (+) next to the icon for the Module Hierarchy.



This displays the top-level modules (in this case one) in the design.

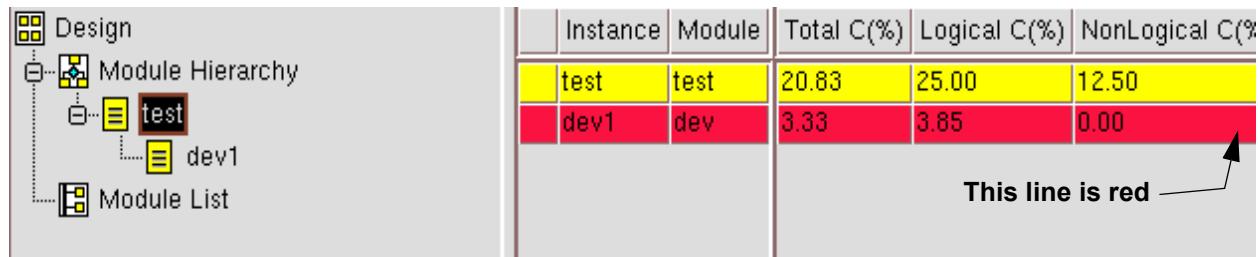
	Instance	Module	Total C(%)	Logical C(%)	NonLogical C(%)
	test	test	20.83	25.00	12.50

A screenshot of the cmView application window showing the results of a condition coverage analysis. The left pane shows the design hierarchy with 'test' selected under 'Module Hierarchy'. The right pane displays a table of coverage results. The table has columns for Instance, Module, Total C(%), Logical C(%), and NonLogical C(%). The row for 'test' is highlighted in yellow. An annotation 'This line is yellow' with an arrow points to the yellow row.

By default, the yellow results bar indicates coverage between 20%-80%. Logical refers to conditions that are the truth or falsity of subexpressions that are operands of the logical AND && or logical OR || operators. Nonlogical refers to conditions derived from other operators as shown in [Table 4-1](#).

4. Click the plus symbol (+) next to the icon for the top-level module `top`.

This displays module instance `test.dev1`.



The red results bar indicated coverage less than 20%.

5. Click on the yellow results bar for the top-level module `test`.

This displays the condition coverage annotated window for the module instance.

The screenshot shows the 'cmView - Condition: Instance test' window. The top menu bar includes File, View, Options, and Help. The toolbar contains icons for Top of File, Prev Line, Next Line, End of File, Go Parent, and Go Child. The main pane displays Verilog code with red horizontal bars under certain lines, indicating incomplete condition coverage. The code is as follows:

```

20      r11 = 2'b11;
21      #10 $finish;
22      end
23
24  -->C assign w1 = r1 ^~ r2 ? r2 : r3;
25
26  -->C assign w2 = r5 || r6;
27
28      always @ (posedge r1 or r2)
29      begin
30  -->C r8 = (r1 == r2) && r3 ? r4 : r6;
31
32  -->C if ((r1 ^ (r2)) && (r3 == r4))
33      begin
34          r6 = (r6 != r7) && r8;
35      end
36      else
37      begin
38          r6 = 1;

```

The bottom pane shows a table of condition coverage statistics:

test	TOTAL	COVERED	PERCENT
Total	48	10	20.83
Logical	32	8	25.00
NonLogical	16	2	12.50
Events	0		

The status bar at the bottom indicates 'for_doc.v'.

The top pane displays the annotated module definition for the instance. Red lines contain subexpressions whose conditions were not completely met. They are also indicated by the `-->C` text symbol to the right of the line number.

The text symbol `-->C` is used for lines with subexpressions whose conditions are completely met.

Manipulating the Summary Pane

By default, the summary pane displays the following information when a Condition Coverage window is opened:

- The first column contains icons which indicate special details about the particular instance or module. These icons are described in [Table 4-4](#).

Table 4-4 Icon Indicators in the Condition Coverage Window

Icon	Description
	A green axe with a red line through it indicates that the instance contains no conditions.
	A document icon indicates that an Annotated window has been opened for the specified instance or module.
	A blue circle with a red line through it indicates that the instance has been excluded.
	A yellow folder with a red line through it indicates that the instance has not been compiled for coverage.
	<ul style="list-style-type: none">• The second column contains the instance name.• The third column contains the module name.• The fourth column contains the total percentage coverage for all conditions.• The fifth column contains the percentage coverage for logical conditions.• The sixth column contains the percentage coverage for non-logical conditions.• The seventh column contains the percentage coverage for events.

Double-clicking on any row in the condition coverage summary pane displays an Annotated window with detailed condition coverage data for the specific instance or module (see “[The Annotated Window](#)”).

Detailed View

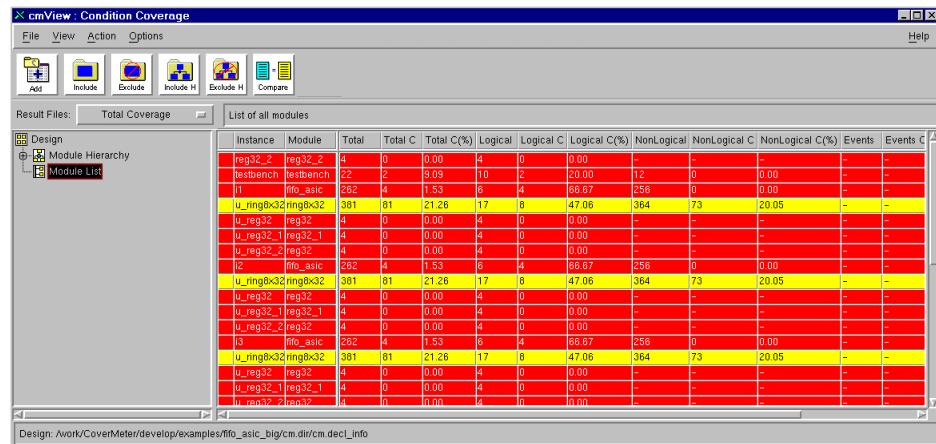
To obtain a detailed report of statement coverage, choose **View > Detailed**.

The detailed statistics for condition coverage are displayed in [Figure 4-2](#).

In addition to the default instance and module names in the detailed report, the following information is also displayed and arranged in columns as follows:

- Total logical conditions
- Total logical conditions covered
- Percentage of logical conditions covered
- Total non-logical conditions
- Total non-logical conditions covered
- Percentage of non-logical conditions covered
- Total events
- Number of events covered
- Percentage of events covered

Figure 4-2 Detailed View of Statistics for Condition Coverage



The Annotated Window

The Annotated window allows you to visually compare your original Verilog source files with copies of those files that have been annotated by VCS or VCS MX to illustrate in the most direct fashion which conditions have not been covered.

The Annotated File

The Annotated window displays an annotated listing of the original Verilog source file with all uncovered conditions highlighted in red. You can change the color used to highlight uncovered conditions using the Colors tab in the User Preferences dialog box (see “[The Colors Tab](#)” on page 8-461). “[The Colors Tab](#)” on page 8-461

If the annotated listing is used for incremental coverage, conditions covered incrementally by the first test are highlighted in red. If the annotated listing is used for Diff Coverage, conditions uncovered by

both tests are displayed in red. You can change the highlight color to display conditions uncovered by the two tests in different colors, if you wish, by using the Colors tab in the User Preferences dialog box.

Buttons are used to move from one uncovered conditional line to the next (see [Table 4-5](#) for a description of these buttons). When using these buttons to navigate through the file, the currently selected condition is always displayed in magenta. You can change this color using the Colors tab in the User Preferences dialog box.

Figure 4-3 Annotated Conditions in Verilog Source File

The Summary Folder

The bottom portion of the Annotated window contains a folder with five tabs containing further details about the module (see [Figure 4-3](#)).

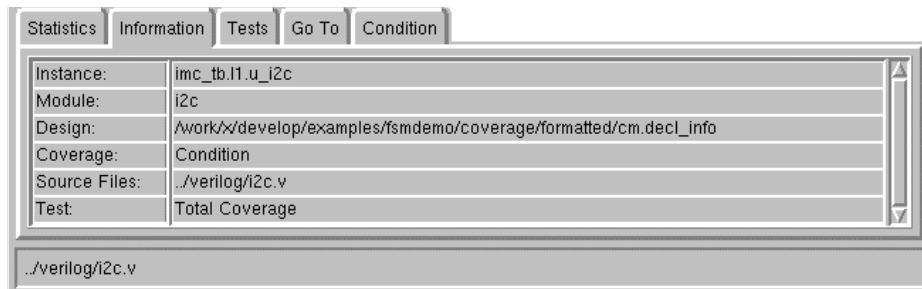
The first tab in this folder is the Statistics tab. Click this tab to display more detailed coverage statistics about the module (see the bottom portion of [Figure 4-3](#)). A complete breakdown of all the different kinds of conditions and their coverage is provided in addition to the summaries of the number of logical, and non-logical conditions in the module.

Table 4-5 Toolbar Buttons in the Annotated Window

Button	Description
 Top of File	This button takes you to the top of the file.
 Next Line	This button takes you to the next uncovered conditional line.
 Prev Line	This button takes you to the previous uncovered conditional line.
 End of File	This button takes you to the bottom of the file.

The second tab in the folder is the Information tab. Click this tab to display general information about this instance or module (see [Figure 4-4](#)). The last row indicates the test for which this coverage data is displayed.

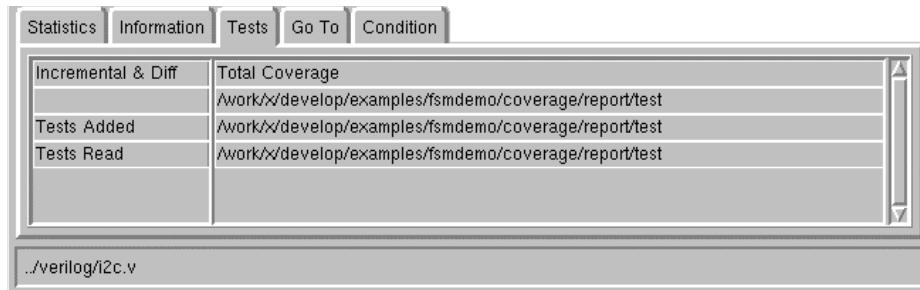
Figure 4-4 Information Tab on Summary Folder in the Annotated Window



The third tab in the folder is the Tests tab. Click this tab to display a detailed summary of all the tests read thus far (see [Figure 4-5](#)). Detailed information is also provided to describe the makeup of total, incremental, and diff coverage.

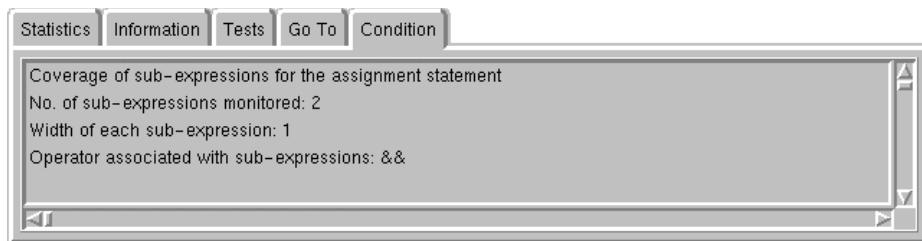
The fourth tab in the folder is the Go To tab. Click this tab to display a list of all uncovered conditional lines in the annotated file (see [Figure 4-7](#)). Click any line in this list to position the annotated file in the top pane in order to display the specified line.

Figure 4-5 Tests Tab on the Summary Folder in the Annotated Window



The fifth tab in the folder is the Condition tab. Click this tab to display details of all subexpression coverage for the selected line (see [Figure 4-6](#)).

Figure 4-6 Condition Tab on the Summary Folder in the Annotated Window



The View Menu

You can use this menu to traverse the module hierarchy of the particular instance, if appropriate.

For example, click the Parent button to go to the parent module or choose **View > Open Parent** from the menu.

A new Annotated window with annotated listings for the parent appears. If the parent module has no executable lines, both the button and the menu item are disabled (grayed out) and are not selectable.

Click the Child button to go to any of the sub-modules of the current module or choose **View > Open Child** from the menu.

A list with all immediate descendants of the current module appears.

Click any member in this list to display a new Annotated window with annotated listings for the submodule. If the module has no submodules, or if the submodules contain no executable lines, both the button and the menu item are disabled (grayed out) and are not selectable.

Figure 4-7 Go To Tab Listing of all Uncovered Conditions

Condition Coverage

4-270

5

FSM Coverage

In hardware, a Finite State Machine (FSM) is some sequential logic that outputs a current state and some combinational logic that outputs the next state. When VCS and VCS MX compile your design for FSM coverage, they identify a group of statements in the source code to be an FSM and to keep track of the states and transitions that occur in the FSM during simulation.

This chapter describes the following:

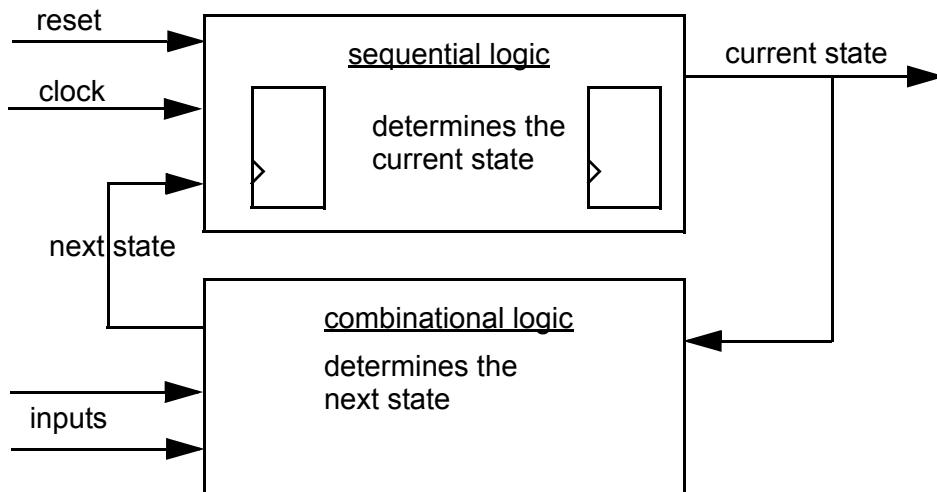
- “Finite State Machines — FSMs”
- “Coding a Verilog FSM”
- “Coding a VHDL FSM”
- “Sequence Coverage”
- “Controlling How VCS and VCS MX Extract FSMs”

- “[FSM Coverage Reports](#)”
- “[Viewing FSM Coverage With the Coverage Metrics GUI](#)”

Finite State Machines — FSMs

In hardware, a Finite State Machine (FSM) is some sequential logic that outputs a current state, and some combinational logic that outputs the next state. The sequential logic is driven by the next state signal as well as clock and reset signals. The combinational logic is driven by the current state and the inputs to the FSM.

Figure 5-1 Finite State Machine



Verilog FSMs

In Verilog, a group of statements can be used to describe a higher level of abstraction of an FSM. VCS and VCS MX treat a group of statements as an FSM if it meets the following criteria:

- The group of statements must contain a procedural assignment statement to a vector variable to assign the current state of the FSM.

The group can also contain one of the following (though this is not required):

- A procedural assignment to assign the next state to another variable
- A concurrent assignment statement to assign the next state to a net

Note:

By default, the variable that holds the current state of the FSM must be directly assigned either a numerical constant or the value of a variable that holds the next state of the FSM. You can use the `-cm_fsmopt allowTmp` compile-time option and keyword argument to enable FSM extraction in cases where there is indirect assignment to the variable that holds the current state (see “[Enabling Indirect Assignment to State Variables](#)”).

By default, FSMs must have more than two states. However, you can use the `-cm_fsmopt report2StateFsms` compile-time option and keyword argument to enable two-state FSM extraction (see “[Enabling Two-state FSMs](#)”).

- The values that the statements in the group assign to a signal are the states of the FSM, and must be of an identifiable set of parameters, numeric constants, or text macros (where the macro text that VCS/VCS MX substitutes for the macro name is a numeric constant). The states can also be enumerated types in SystemVerilog.

Note:

By default, VCS and VCS MX do not extract an FSM with an X value state. However, you can use the `-cm_fsmopt reportXassign` compile-time option and keyword argument to enable this (see “[Enabling X Value States](#)”).

- In the group, a dependency exists between the current value assigned and the next value assigned.

When you write such a group of statements, it is important to know if the statements made the assignments to all possible states of the FSM and all possible transitions between states.

VCS and VCS MX can “extract” the FSM from your design. In other words, identifying the group of statements as an FSM, enabling the tracking of current states of the FSM, and enabling cmView to report or display state and state transition coverage information. For FSM coverage you can perform the following:

- Instruct VCS and VCS MX to automatically extract all the identified FSMs in your design.
- Use a configuration file to limit the VCS/VCS MX FSM extraction to certain module definitions and limit VCS/VCS MX to keeping track of only certain states and transitions that are specified in the configuration file.

- Use compiler directives in your code to make sure that VCS/VCS MX extracts an FSM for certain group of statements.

Note:

This does not apply to VHDL.

VCS/VCS MX does not automatically extract an FSM when:

- The group of statements exists in a user-defined task (in VHDL, a function or procedure), including a SystemVerilog global task (it will extract if the group is in a user-defined function, including a SystemVerilog global function).
- There are less than three possible states.

VHDL FSMs

VCS MX automatically extracts FSMs from VHDL code when it sees the following:

- The state values are stored in enumerated, std_logic_vector, bit_vector, or integer data types.
- The state values of the FSM are either an enumerated type, VHDL constant, or literal constant. No expressions or function calls on the right side of assignment statements assigning the next state of the FSM.
- The code determines the next state of the FSM using a `case` statement. VCS MX does not extract the FSM if there is an `if` statement for determining the next state.
- The code for the FSM is entirely within a procedure or process or using one combinational process to determine the next state of the FSM and a sequential process to set the next state on the clock edge.

VCS MX cannot automatically extract the following types of FSMs:

- One-hot or hot-bit FSMs
- FSMs using conditional assignment statements

The Purpose of FSM Coverage

FSM coverage provides you with information about your design, including the FSMs in your design (see “[Coding a Verilog FSM](#)”) that you cannot learn from other types of coverage.

Verilog

[Example 5-1](#) is a Verilog module definition that contains statements that function as an FSM:

Example 5-1 Verilog Module Definition Containing an FSM

```
module dev (clk,in,state);
  input clk,in;
  output [1:0] state;

  reg [1:0] state,next;
  parameter idle = 2'b00,
            first = 2'b01,
            second = 2'b10,
            third = 2'b11;

  initial
  begin
    state=idle;
    next=idle;
  end

  always @ in
  begin
    next = state; // by default hold
```

```

case (state)
    idle      : if (in) next = first;
    first     : if (in) next = second;
    second    : if (in) next = third;
    third     : if (in) next = idle
endcase
end

always @ (posedge clk)
state=next;

endmodule

```

The FSM has four states: idle, first, second and third. It is possible for a testbench to apply stimulus such that line coverage indicates that VCS/VCS MX executed all the executable lines, but FSM coverage indicates that there never was a transition from the third to the idle state.

VHDL

[Example 5-2](#) illustrates a VHDL architecture that models an FSM:

Example 5-2 VHDL Architecture Modeling an FSM

```

architecture exfsmarch of exfsm is
    type my_fsm_states is ( idle, first, second, third );

    signal curr_state : my_fsm_states := idle;
    signal next_state : my_fsm_states := idle;

begin

    my_fsm : process(insig)
    begin
        next_state <= curr_state;
        case curr_state is
            when idle => if (insig = '1') then next_state <= first;
        end if;
    end

```

```

        when first => if (insig = '1') then next_state <= second;
    end if;
        when second => if (insig = '0') then next_state <= third;
    end if;
        when third => if (insig = '1') then next_state <= idle;
    end if;
    end case;
end process;

advance_fsm : process
begin
    wait until clk'event and clk = '1';
    curr_state <= next_state;
end process;

end exfsmarch;

```

Coding a Verilog FSM

There are a wide range of coding styles for FSMs. The FSM can consist of a continuous assignment of the next state value to a net using conditional operators along with procedural assignment statements in an always block to transfer (assign) the next state value of the net to the reg that holds the current state. More typically the FSM consists of procedural statements inside an always block including procedural assignments to one or more regs of current state and next state values controlled by `case`, `while`, or `if-else` statements.

This section includes various groups of statements which VCS/VCS MX automatically extracts as an FSM.

Using the Encoded FSM Style

The encoded FSM style does not require that:

- Only one bit, in the vector reg that contains the current state, be true
- You use the entire bit width of the reg for the state

In [Example 5-3](#), the FSM has four possible states:

- NO_ONES
- ONE_ONE
- TWO_ONES
- AT_LEAST_THREE_ONES

Each state is delineated in a parameter declaration.

Example 5-3 FSM with States Delineated in a Parameter Declaration

```
module enum2_V(signal, clock, detect);
  input signal, clock;
  output detect;
  reg detect;

  parameter [1:0]
    NO_ONES = 2'h0,
    ONE_ONE = 2'h1,
    TWO_ONES = 2'h2,
    AT_LEAST_THREE_ONES = 2'h3;

  // Declare current state and next state variables.
  reg [1:0] cs, ns;

  always @ (cs or signal)
  begin
    detect = 0;// default value
    if (signal == 0)
      ns = NO_ONES;
    else
      case (cs)
        NO_ONES: ns = ONE_ONE;
```

```

ONE_ONE: ns = TWO_ONES;
TWO_ONES,
AT_LEAST_THREE_ONES:
begin
    ns = AT_LEAST_THREE_ONES;
    detect = 1;
end
default: ns = NO_ONES;
endcase
end

always @ (posedge clock) begin
    cs = ns;
end

endmodule

```

In [Example 5-3](#), the case expression is the reg that holds the current state of the FSM. The case item expressions are states of the FSM. When the current state of the FSM is that in the case item expression the case item statements specify the next state of the FSM.

You can use signed values for the states of an FSM, as shown in [Example 5-4](#):

Example 5-4 FSM With Signed Values

```

module finite_state (clk, in, state);

input clk, in;
output state;

reg signed [1:0] state, next;

parameter signed idle=2'sb00, // 0
            first=2'sb01, // +1
            second=2'sb10, // -2
            third=2'sb11; // -1

initial begin

```

```

state=idle;
next=idle;
end

always @ in
begin
    next=state;
    case (state)
        idle : if(in) next=first;
        first : if(in) next=second;
        second: if(in) next=third;
        third : if(in) next=idle;
    endcase
end

always @ (posedge clk)
    state=next;
endmodule

```

In the FSM in [Example 5-5](#), the four possible states are A, B, C and D. They are delineated in text macros specified in 'define compiler directives. In these 'define compiler directives, the macro text that VCS/VCS MX substitutes for the macro names A, B, C and D are numeric constants 2'b00, 2'b01, 2'b10 and 2'b11. The 'define compiler directive text macros must substitute a numeric constant for the macro name.

Example 5-5 FSM With States Delineated with Text Macros

```

#define A 2'b00
#define B 2'b01
#define C 2'b10
#define D 2'b11

module counter_top_fsm(clock, reset, count,
mode, countA, countB);

input clock, reset;
input [3:0] countA, countB;

```

```

output [3:0] count;
output [1:0] mode;

reg [3:0] count;
reg [1:0] mode, mode_next;
reg [1:0] top_state, top_state_next;

always @ top_state
begin
case (top_state)
'A : begin
    top_state_next = 'B ;
    mode_next = 1;
end
'B : begin
    top_state_next = 'C;
    mode_next = 2;
end
'C : begin
    top_state_next = 'D;
    mode_next = 3;
end
'D : begin
    top_state_next = 'A;
    mode_next = 0;
end
endcase
end

```

```

always @(posedge clock)
begin
if (!reset)
begin
    count = 4'b0000;
    mode = 0;
    top_state_next = 'A;
    top_state = 'A;
end
else
begin

```

```

        mode = mode_next;
        top_state = top_state_next;
        count = (mode == 0) ? countA : countB;
    end
end
endmodule

```

[Example 5-6](#) shows an FSM that has no reg or wire to hold the next state of the FSM. VCS/VCS MX can still identify and extract this FSM.

Example 5-6 FSM With No Next State Signal

```

module no_NS (en,clock,state);
input en,clock;
output [1:0] state;
reg [1:0] state;

initial
state=0;

always @ (posedge clock)
case (state)
  0: if (en == 1) state = 1;
  1: state = en == 0 ? 0: 2;
  2: state = en == 0 ? 1 : 0;
endcase

endmodule

```

In [Example 5-6](#) the clock controls when the next state is assigned to the state reg. The group of statements uses an expression that uses the conditional operator to assign the next state to the state reg.

VCS/VCS MX can identify and extract an FSM even when the state value propagates from the next state signal to the current state signal through another module instance as shown in [Example 5-7](#).

Note:

This feature does not apply to VHDL.

Example 5-7 State Value Propagating Through Another Instance

```
module fsm_mod (in);
  input [1:0] in;
  parameter
    start=2'b11, step1=2'b10, step2=2'b01, finish=2'b00;

  wire [1:0] current;

  reg [1:0] next;

  always @ (in
  begin
    case (current)
      start : next = step1;
      step1 : next = step2;
      step2 : next = finish;
      finish : next = start;
      default: next = step1;
    endcase
  end

  connector ctl (current,next);
endmodule

module connector (out,in);
  output [1:0] out;
  input [1:0] in;
  reg [1:0] out;

  always @ (in
  #5 out = in;

endmodule
```

VCS/VCS MX looks for a way for the state value to propagate from the next state signal to the current state signal. In this example, VCS/VCS MX determines that these two signals both connect to an instance of the `connector` module, and in this module exists a direct connection of the input and output ports. Therefore, VCS/VCS MX extracts the FSM in module `fsm_mod`.

VCS/VCS MX does not search for this propagation path through intermediate signals. If the `connector` module was defined as follows:

```
module connector (out,in);
output [1:0] out;
input [1:0] in;
reg [1:0] r1;

assign out = r1;

always @ in
#5 r1 = in;

endmodule
```

VCS/VCS MX does not search for the path `in` → `r1` → `out` and, therefore, VCS/VCS MX does not automatically extract the FSM.

You can use a configuration file or pragmas to instruct VCS/VCS MX to extract the FSM (see “[Using a Configuration File](#)” and “[Using Pragmas](#)”).

Implementing Hot Bit or One Hot FSMs

Hot bit or One Hot FSMs are usually implemented with case statements. The case expression is 1'b1, and the case item expression is a specific bit of the reg that holds the current state of the FSM.

Note:

Hot bit and One Hot FSMs are not applicable to VHDL.

Example 5-8 Hot Bit FSM

```
module prep3(clk, rst, in, out);
    input clk, rst;
    input [7:0] in;
    output [7:0] out;

    parameter [2:0]
        START = 0 ,
        SA = 1 ,
        SB = 2 ,
        SC = 3 ,
        SD = 4 ,
        SE = 5 ,
        SF = 6 ,
        SG = 7 ;

    reg [7:0] state;
    reg [7:0] next_state;
    reg [7:0] out, next_out;
    always @ (in or state) begin
        // default values
        next_state = 8'b0;
        next_out = 8'bx;

        case (1'b1)
            state[START]:
                if (in == 8'h3c) begin
                    next_state[SA] = 1'b1 ;
```

```

        next_out = 8'h82 ;
    end
else begin
    next_state[START] = 1'b1 ;
    next_out = 8'h00 ;
end
state[SA] :
case (in)
8'h2a:
begin
next_state[SC] = 1'b1 ;
next_out = 8'h40 ;
end
8'h1f:
begin
next_state[SB] = 1'b1 ;
next_out = 8'h20 ;
end
default:
begin
next_state[SA] = 1'b1 ;
next_out = 8'h04 ;
end
endcase

state[SB] :
if (in == 8'haa) begin
next_state[SE] = 1'b1 ;
next_out = 8'h11 ;
end
else begin
next_state[SF] = 1'b1 ;
next_out = 8'h30 ;
end
state[SC] :
begin
next_state[SD] = 1'b1 ;
next_out = 8'h08 ;
end
state[SD] :
begin
next_state[SG] = 1'b1 ;

```

```

        next_out = 8'h80 ;
    end
state[SE] :
begin
    next_state[START] = 1'b1 ;
    next_out = 8'h40 ;
end
state[SF] :
begin
    next_state[SG] = 1'b1 ;
    next_out = 8'h02 ;
end
state[SG] :
begin
    next_state[START] = 1'b1 ;
    next_out = 8'h01 ;
end
endcase
end

// build the state flip-flops
always @ (posedge clk or negedge rst)

begin
    if (!rst) begin
        state <= #1 8'b0 ;
        state[START] <= #2 1'b1 ;
    end
    else
        state <= #1 next_state ;
end

// build the output flip-flops
always @ (posedge clk or negedge rst)
begin
    if (!rst) out <= #1 8'b0 ;
    else out <= #1 next_out ;
end

endmodule

```

[Example 5-8](#) contains case statements, one nested inside the other. The outer case statement is what makes this FSM a hot bit FSM. In the outer case statement:

- The case expression is `1'b1`.
- The case item expressions are `state[START]`, `state[SA]`, `state[SB]`, `state[SC]`, `state[SD]`, `state[SE]`, `state[SF]`, and `state[SG]`. These expressions specify specific bits of a reg named `state` that holds the current state of the FSM. The parameter declaration in [Example 5-8](#) specifies parameters for bit numbers of that reg. More typically parameter declarations specify the states of the FSM.
- The case item statements are begin-end blocks of statements, including the nested case statement that control the assignment of the next state of the FSM to the reg named `next_state` that holds this next state.

The nature of the hot bit FSM is that only one bit of the reg that holds the current state of the FSM can be true, and in this example, that is also true of the reg that holds the next state.

Using Continuous Assignments for FSMs

Not all FSMs consist entirely of procedural statements in always blocks. In this example, the next state signal is a wire and the next state is assigned using a continuous assignment statement. An always block changes the current state contained in a reg.

Example 5-9 Continuous Assignment Statement FSM

```
module M (x,clock);  
  
input x,clock;  
reg [1:0] state;
```

```

wire [1:0] next;

always @ (posedge clock)
state = next;

assign next = state == 0 ?
              (x == 1 ? 1 : 0) :
              state == 1 ?
              (x == 0 ? 0 : 2) :
              state == 2 ?
              (x == 0 ? 1 : 0) :
              2'b00;
endmodule

```

In [Example 5-9](#), the states are 0, 1, and 2. The reg named `state` holds the current state and the wire named `next` holds the next state.

Avoiding Substituting the Same Numeric Constant

When coding an FSM, avoid substituting the same numeric constant for more than one macro name for a state in multiple `'define` compiler directives, otherwise, this can cause VCS/VCS MX to confuse one state for another. For example:

```

#define first_state 0
#define prime_state 0

```

Coding a VHDL FSM

This section contains examples of coding a VHDL FSM that VCS MX automatically extracts. In [Example 5-10](#), the states of the FSM are stored in an integer data type.

Example 5-10 Integer Data Type FSM

```
entity E is
  port (
    clk : in bit);
end E;

architecture A of E is
  signal cs : integer := 0;
  signal ns : integer := 0;
begin

  process(cs)
  begin
    case cs is
      when 0 => ns <= 1;
      when 1 => ns <= 3;
      when 3 => ns <= 1;
      when others => ns <= 0;
    end case;
  end process;

  process(clk)
  begin
    if (clk'event and clk = '1') then
      cs <= ns;
    end if;
  end process;

end A;
```

In [Example 5-10](#), the next state of the FSM, stored in signal `ns`, is determined by one process and the current state, stored in signal `cs`, is determined in another process.

[Example 5-11](#), stores the states in the `bit_vector` data type. The next state is assigned to selected bits.

Example 5-11 The Next State is Assigned to Selected Bit

```
entity E is
```

```

port (
    clk : in bit);
end E;

architecture A of E is
    signal cs : bit_vector(1 downto 0) := "00";
    signal ns : bit_vector(1 downto 0) := "00";
begin

process(cs)
begin
    case cs is
        when "00" => ns(0) <= '1';
        when "01" => ns(1) <= '1';
        when "11" => ns(1) <= '0';
        when others => ns <= "00";
    end case;
end process;

process(clk)
begin
    if (clk'event and clk = '1') then
        cs <= ns;
    end if;
end process;

end A;

```

In [Example 5-12](#), the states are in an enumerated type.

Example 5-12 States in Enumerated Type

```

entity E is
    port (
        clk : in bit);
end E;

architecture A of E is
    type STATES is (S0, S1, S2, S3, S4);
    constant ss0 : STATES := S0;

```

```

constant ss1 : STATES := S1;
constant ss3 : STATES := S3;
signal cs : STATES := ss0;
signal ns : STATES := ss0;
begin

process(cs)
begin
    case cs is
        when ss0 => ns <= ss1;
        when ss1 => ns <= ss3;
        when ss3 => ns <= ss1;
        when others => ns <= ss0;
    end case;
end process;

process(clk)
begin
    if (clk'event and clk = '1') then
        cs <= ns;
    end if;
end process;

end A;

```

An FSM cannot use a concurrent signal assignment statement unless it is also a selected signal assignment as shown in [Example 5-13](#).

Example 5-13 FSM Using a Concurrent Selected Signal Assignment Statement

```

entity E is
    port (
        clk : in bit);
end E;

architecture A of E is
    signal cs : integer := 0;
    signal ns : integer := 0;

```

```

begin

    with cs select
        ns <= 1 when 0,
            3 when 1,
            1 when 3,
            0 when others;

    process(clk)
    begin
        if (clk'event and clk = '1') then
            cs <= ns;
        end if;
    end process;

end A;

```

In [Example 5-14](#) there is no next state signal.

Example 5-14 FSM With No Next State Signal

```

entity E is
    port (
        rst : in bit;
        clk : in bit);
end E;

architecture A of E is
begin

    process(clk,rst)
        variable cs : integer := 0;
    begin
        if rst = '0' then
            cs := 0;
        elsif clk'event and clk = '1' then
            case cs is
                when 0 => cs := 1;
                when 1 => cs := 3;
                when 3 => cs := 1;
                when others => cs := 0;
            end case;
        end if;
    end process;

```

```

        end case;
    end if;
end process;

end A;
```

In [Example 5-15](#), the states are in the std_logic data type.

Example 5-15 FSM Using the std_logic Data Type

```

library ieee;
use ieee.std_logic_1164.all;

entity E is
    port (
        sync1 : in std_logic;
        sync2 : in std_logic;
        reset : in std_logic;
        clk : in std_logic);
end E;

architecture A of E is
    signal cs : std_logic_vector(1 downto 0);
    signal ns : std_logic_vector(1 downto 0);
begin

process(cs, sync1, sync2)
begin
    case cs is
        when "00" =>
            if(sync1 = '1') then
                if(sync2 = '1') then
                    ns <= "01";
                else
                    ns <= "10";
                end if;
            else
                ns <= cs;
            end if;

        when "01" =>
```

```

        if(sync1 = '0') then
            ns <= "00";
        elsif (sync2 = '0') then
            ns <= "10";
        else
            ns <= cs;
        end if;

when "10" =>
    if(sync1 = '0') then
        if(sync2 = '0') then
            ns <= "00";
        else
            ns <= cs;
        end if;
    else
        if(sync2 = '0') then
            ns <= cs;
        else
            ns <= cs;
        end if;
    end if;
when others => ns <= "00";
end case;
end process;

process(clk, reset)
begin
    if reset = '0' then
        cs <= "00";
    elsif (clk'event and clk = '1') then
        cs <= ns;
    end if;
end process;

end A;

```

Sequence Coverage

You can instruct VCS and VCS MX to compile for and monitor sequence coverage with the `-cm_fsmopt sequence` compile-time option and keyword argument.

With sequence coverage, you not only see the states that were covered, but the sequences of states that were covered during simulation.

Controlling How VCS and VCS MX Extract FSMs

When VCS and VCS MX compile your design for FSM coverage, they extract FSMs from your source code. Extracting FSMs means identifying a group of statements to be an FSM so that VCS/VCS MX is ready to keep track of the states and transitions that occur in the FSM during simulation.

By default, VCS and VCS MX automatically extract all the FSMs that it can identify in all the module definitions and VHDL architectures in your design.

However, you can limit the extraction of FSMs to a part or parts of the design hierarchy that you specify with the `-cm_hier` compile-time option.

You can also specify the FSMs in a module definition by using one of the following:

- A configuration file
- Pragmas in your code (does not apply to VHDL)

Using a Configuration File

A configuration file enables you to specify:

- The FSMs that VCS or VCS MX extracts from a module or entity definition
- Which states and which transitions between states that VCS or VCS MX keeps track of in the FSMs
- The maximum number of sequences that VCS or VCS MX keeps track of in any of the modules or design entities in your design, and specify the maximum length of any sequence that VCS or VCS MX keeps track of (see “[Specifying the Maximum for Sequences](#)”)

When VCS/VCS MX compiles your Verilog design for coverage, it creates the `simv.cm` directory in the current directory, and the `db` directory in the `simv.cm` directory. If you compile your design for FSM coverage, VCS/VCS MX writes the `cm fsm_config.generated` file in the `db` directory. You can either use this file, editing it to suit your needs, or start a new file as your configuration file.

When VCS MX compiles your VHDL design, it creates the `simv.cm/db/vhdl` directory and writes the `cm fsm_config.generated` file in this directory. Similar to VCS, you can either use this file, editing it to suit your needs, or start a new file as your configuration file.

You write a separate section in the configuration file for each module definition or design entity definition for which you want to specify the FSMs that VCS or VCS MX extracts.

VCS/VCS MX expects the entries in the FSM configuration file to be written in a particular order. It expects the design level entries at the top, followed by module level entries, and then the FSM level entries at the bottom.

The syntax of a configuration file section is as follows:

```
SEQ_NUMBER_MAX = integer
SEQ_LENGTH_MAX = integer
MODULE = module_identifier
FSMS = RESTRICT | EXCLUDE
FSMS = START_STATE_DFLT
CURRENT = reg_identifier
NEXT = net_or_reg_identifier
STATES = list_of_states
STATES_X = list_of_states
STATES_NEVER = list_of_states
START_STATE = state
TRANSITIONS = list_of_transitions
TRANSITIONS_X = list_of_transitions
TRANSITIONS_NEVER = list_of_transitions
SEQ_REQUIRE = pattern
SEQ_EXCLUDE = pattern
```

The following tables describe the FSM configuration file entries:

Design Level Entries	Description
SEQ_NUMBER_MAX = <i>integer</i>	Specifies the maximum number of sequences in any module or design entity that VCS or VCS MX maintains. The integer value must be non-negative.
SEQ_LENGTH_MAX = <i>integer</i>	Specifies the length of the longest sequence that VCS or VCS MX maintains. The integer value must be non-negative.

Module Level Entries	Description
Verilog: MODULE = <i>module_identifier</i>	Specifies a module definition. VCS or VCS MX extracts FSMs from all instances of this module definition. This line is always required in a section.
VHDL: MODULE = <i>E</i> or MODULE = <i>lib.E</i>	For VHDL designs, specify the entity name as MODULE = <i>Ent_name</i> , or as MODULE = <i>Library_name.Ent_name</i> .
FSMS = RESTRICT	Specifies that VCS or VCS MX only extracts the FSMs specified in a line that begins with the keyword CURRENT. This line is optional and specifies the default condition.
FSMS = EXCLUDE	Specifies that VCS or VCS MX extracts all the FSMs in the module or design entity definition except those specified in a line that begins with the keyword CURRENT. This line is optional.
FSMS = START_STATE_DFLT	Specifies that VCS or VCS MX only maintains the sequences that begin with the state that has the lowest value. This line is also optional. You can enter an FSMS = RESTRICT or FSMS = EXCLUDE line along with an FSMS = START_STATE_DFLT line.

FSM Level Entries	Description
CURRENT = <i>reg_identifier</i>	Specifies the Verilog variable or VHDL signal that holds the current state of the FSM. This line is always required in a section. If you want to restrict extraction to, or exclude extraction from, more than one FSM in the module or design entity definition, enter this line for each FSM.

(continued)

<code>NEXT = net_or_reg_identifier</code>	Specifies the wire or reg that holds the next state of the FSM. This line is required if the FSM has a reg or wire that holds the next state of the FSM (see Example 5-6). Like with the CURRENT line, if you want to restrict extraction to, or exclude extraction from, more than one FSM in the module or design entity definition, enter this line for each FSM.
<code>STATES = list_of_states</code>	Specifies a list of states, or the values of states, separated by commas. Specifying these states ensures that VCS or VCS MX keeps track of transitions to these states. This list is not restrictive; if the FSM transitions to other states VCS or VCS MX also keeps track of these transitions and reports these unlisted states by value. This line is required in a section.
<code>STATES_X = list_of_states</code>	Specifies a list of states, or the values of states, that you want VCS or VCS MX to ignore. Separate the states by commas. VCS or VCS MX does not keep track of these states, transitions to and from these states, or sequences that include these states. This line is optional.
<code>STATES_NEVER = list_of_states</code>	Specifies a list of states, or the values of states, that you want VCS or VCS MX to report as <code>ILLEGAL</code> in the report files if there is a transition or sequence involving these states or values. This line is also optional.
<code>START_STATE = state</code>	Specifies a start state for the FSM such as a reset state. This line is optional. When you include it VCS or VCS MX only keeps track of sequences that begin and end with this start state.
<code>TRANSITIONS = list_of_transitions</code>	Specifies a list of transitions, separated by commas. You can specify state names or state values. Like the STATES line, specifying these transitions ensures that VCS or VCS MX keeps track of them. This list is not restrictive; if the FSM makes other transitions, VCS or VCS MX also keeps track of them. This line is optional.

(continued)

TRANSITIONS_X = <i>list_of_transitions</i>	Specifies a list of transitions that you want VCS or VCS MX to ignore. You can specify transitions by state names or state values. Separate the transitions by commas. This line is optional.
TRANSITIONS_NEVER = <i>list_of_transitions</i>	Specifies a list of transitions that you want VCS or VCS MX to report as <code>ILLEGAL</code> in the report files if there is a transition or sequence involving these states or values. You can specify transitions by state names or state values. Separate the transitions by commas. This line is also optional.
SEQ_REQUIRE = <i>pattern</i>	Specifies a pattern of transitions. cmView reports only sequences that contain the pattern, see " Sequence Filtering in Reports ".
SEQ_EXCLUDE = <i>pattern</i>	Specifies a pattern of transitions. cmView does not report sequences that contain the pattern, see " Sequence Filtering in Reports ".

As shown in [Example 5-16](#) and [Example 5-17](#), if you have more states or transitions than you want on a line, you can enter a line break and enter more states or transitions on the following line.

Example 5-16 Verilog Configuration File

```
MODULE = fsmmod

CURRENT = CS
NEXT = NS
STATES = ZERO,ONE,TWO,THREE,FOUR,FIVE
START_STATE = ZERO
TRANSITIONS = ZERO->ONE, ONE->TWO, TWO->THREE,
THREE->FOUR, FOUR->FIVE, FIVE->ZERO

CURRENT = current_state
NEXT = next_state
STATES = step1, step2, step3
```

Example 5-17 VHDL Configuration File

```
MODULE=DEFAULT.E A
CURRENT=CS
NEXT=NS
```

```
STATES=s0,s1,s3
TRANSITIONS=s0->s1,
s1->s0,
s1->s3,
s3->s0,
s3->s1
```

The TRANSITIONS Line

The TRANSITIONS line is optional. The syntax for the transition line is as follows:

```
TRANSITIONS = state -> state, ...
```

The TRANSITIONS line can list one or more transitions from a state, to a state, with these transitions separated by commas. Enter the characters "->" between states to specify a transition between these states.

You can add a second line to the TRANSITIONS line to list more transitions.

The syntax for a TRANSITIONS_X or TRANSITIONS_NEVER line is similar.

Specifying the Configuration File

VCS or VCS MX looks for a configuration file named `cm.fsm_config` in the current directory. You can specify a configuration file with a different name and location with an argument to the `-cm_fsmcfg` compile-time option. For example:

```
vcs -cm fsm -cm_fsmcfg myconfig.txt source.v
```

or

```
vcs -cm fsm -cm_fsmcfg myconfig.txt cfg
```

Sequence Filtering in Reports

You use the SEQ_REQUIRE and SEQ_EXCLUDE commands in the FSM configuration file for filtering sequences in report files. You can use either command separately or you can use them together to specify what sequences you want reported in the report file. The SEQ_REQUIRE command is used for specifying what must be included in a sequence for cmView to report the sequence. The SEQ_EXCLUDE command is used for specifying what cannot be included in a sequence for cmView to report it.

The arguments to these commands are patterns for transitions and can be any of the following:

- A state
- A transition between states
- A sequence of states of any length

You can use the wildcard character * in any transition or sequence to specify a transition from/to any state.

The following are examples of these commands and arguments:

```
SEQ_REQUIRE=state2->state3
```

In this example, cmView only reports sequences that include the transition from state2 to state3.

```
SEQ_EXCLUDE=*->state2
```

In this example, cmView does not report any sequence that includes a transition from any other state to state2.

```
SEQ_REQUIRE=state4  
SEQ_EXCLUDE=state4->state5
```

In this example, cmView only reports sequences that include state4, but does not report sequences that include a transition from state4 to state5.

Specifying the Maximum for Sequences

There are commands that you can enter at the beginning of the configuration file that specify how VCS and VCS MX compile and monitor the sequences in the FSMs in all the module or design entity definitions in your design. These commands must come before any section for a module or design entity definition. These commands are as follows:

`SEQ_NUMBER_MAX=integer`

Specifies the maximum number of sequences in any module or design entity that VCS or VCS MX maintains. The integer value must be non-negative.

`SEQ_LENGTH_MAX=integer`

Specifies the length of the longest sequence that VCS or VCS MX maintains. The integer value must be non-negative.

In a mixed HDL design, if you enter these commands in both the Verilog and VHDL FSM configuration files, and the integer values are not the same, VCS MX uses the higher value and displays a warning message.

Using Pragmas

A pragma is a metacomment, something entered in a code comment, that is a message for the software that reads the comment. Pragmas are sometimes called compiler directives even though they do not have the syntax of a compiler directive defined in the IEEE Std 1364 or SystemVerilog.

You can use pragmas to specify an FSM that VCS might not automatically extract.

Note:

Pragmas do not apply to VHDL.

With pragmas, you can inform VCS/VCS MX about the following FSM matter:

- The vector signal, part-select of a vector signal, or concatenation of signals that hold the current state of the FSM.
- The vector signal that holds the next state of the FSM, unless the FSM does not use a signal for the next state (in which case VCS displays a warning and assumes that the current state and next state are in the same signal, part-select of a signal, or concatenation of signals).
- The possible states of the FSM that are specified in a parameter declaration. When using pragmas to specify an FSM, there must be a parameter declaration to specify the possible states of the FSM.

Specifying the Signal That Holds the Current State

You use the following pragma to identify the vector signal that holds the current state of the FSM:

```
/* VCS state_vector signal_name */
```

VCS and `state_vector` are required keywords. You must enter this pragma inside the module definition where the signal is declared.

You also must use a pragma to specify an enumeration name for the FSM. This enumeration name is also specified for the next state and the possible states, associating them with each other as part of the same FSM. There are two ways you can do this:

- Use the same pragma:

```
/* VCS state_vector signal_name enum enumeration_name */
```

- Use a separate pragma in the signal's declaration:

```
/* VCS state_vector signal_name */  
reg [7:0] /* VCS enum enumeration_name */ signal_name;
```

In either case, `enum` is a required keyword. If using the separate pragma, `VCS` is also a required keyword. Also, when using a separate pragma, enter the pragma immediately after the bit range of the signal.

Specifying the Part-Select that Holds the Current State

You can specify that a part-select of a vector signal holds the current state of the FSM. Normally, when `cmView` displays or reports FSM coverage data, it names the FSM after the signal that holds the

current state. In your FSM, if a part-select holds the current state, you must also specify a name for the FSM that cmView can use. The FSM name is not the same as the enumeration name.

You can specify the part-select with the following pragma:

```
/* VCS state_vector signal_name[n:n] FSM_name enum  
   enumeration_name */
```

Specifying the Concatenation that Holds the Current State

Like specifying a part-select, you can specify a concatenation of signals to hold the current state. When you do this, you also need to specify an FSM name and an enumeration name:

```
/* VCS state_vector {signal_name, signal_name, ...} FSM_name  
   enum enumeration_name */
```

The concatenation is of the entire signals. You cannot include bit-selects or part-selects of signals.

Specifying the Signal that Holds the Next State

You also specify the signal that holds the next state of the FSM with the pragma that specifies the enumeration name:

```
reg [7:0] /* VCS enum enumeration_name */ signal_name
```

If, and only if, the FSM does not have a signal for the next state, you can omit this pragma.

Specifying the Current and Next State Signals in the Same Declaration

If you use the pragma for specifying the enumeration name in a declaration of multiple signals, VCS/VCS MX assumes that the first signal following the pragma holds the current state and the next signal holds the next state. For example:

```
/* VCS state_vector cs */
reg [1:0] /* VCS enum myFSM */ cs, ns, nonstate;
```

In this example, VCS/VCS MX assumes that signal `cs` holds the current state and signal `ns` holds the next state. It assumes nothing about signal `nonstate`.

Specifying the Possible States of the FSM

You can also specify the possible states of the FSM with the pragma that specifies the enumeration name:

```
parameter /* VCS enum enumeration_name */
  S0 = 0,
  s1 = 1,
  s2 = 2,
  s3 = 3;
```

Enter this pragma immediately after the keyword `parameter`, unless you specify a bit width for the parameters. If you do specify a bit width, enter the pragma immediately after the bit width:

```
parameter [1:0] /* VCS enum enumeration_name */
  S0 = 0,
  s1 = 1,
  s2 = 2,
  s3 = 3;
```

Pragmas in One Line Comments

These pragmas work in both block comments, between the /* and */ character strings, and in one line comments, following the // character string. For example:

```
parameter [1:0] // VCS enum enumeration_name
  s0 = 0,
  s1 = 1,
  s2 = 2,
  s3 = 3;
```

Specifying FSM With Pragmas - an Example

```
module m3;
    reg[31:0] cs;
    reg[31:0] /* VCS enum MY_FSM */ ns; // Signal ns holds the next state
    reg[31:0] clk;
    reg[31:0] rst; // signal cs holds the current state
    // VCS state_vector cs enum MY_FSM
    parameter // VCS enum MY_FSM
        p1=10,
        p2=11,
        p3=12; // p1, p2, and p3 are possible states of the FSM
endmodule // m3
```

Using Optimistic Extraction

You can instruct VCS or VCS MX to identify illegal transitions when it extracts an FSM from the source code. You do this with the `-cm_fsmopt` optimist compile-time option and keyword argument. The following code example illustrates optimistic extraction:

```
always @ ns
    cs <= ns;
always @ cs
begin
    if (reset)
        cs = 1; // 0->1, 2->1, 3->1
    case(cs)
        0: ns=1; // 0->1
        1: ns=2; // 1->2
        2: ns=3; // 2->3
```

```

        default: ns=0; // 3->0
    endcase
end

```

When VCS or VCS MX extracts this FSM, it determines that the possible states are 0, 1, 2, and 3. In this example, there is a case item for the 0 to 1, 1 to 2, and 2 to 3 transitions, but no case item for the 3 to 0 transition. The default case item makes the 3 to 0 transition possible.

When there is a default case item like this, the default behavior is to determine that a transition from any other state to 0 is legal. When you use the `-cm_fsmopt` optimist compile-time option and keyword argument, VCS or VCS MX determines that only a 3 to 0 transition is legal, and if there is a 2 to 0 or 1 to 0 transition, cmView reports the transition as illegal.

In `casex` statements, this option optimistically reduces the number of legal transitions, as shown in the following example:

```

always @ ns
    cs <= ns;
        // list of states 0,1,2,3,4,5

always @ cs
casex(cs)
    4'b0000: ns=4'b0001; // 0->1
    4'b00x0: ns=4'b0011; // 2->3
    4'b001x: ns=4'b0010; // 3->2
    4'b0x11: ns=4'b0101; // none, 3 is used
    default: ns=4; // 1->4, 5->4
endcase

```

When you use this option and keyword, VCS and VCS MX perform the following:

1. Determine from analysis of the assignment statements that the possible states of this FSM are 0, 1, 2, 3, 4, and 5.
2. Analyze the first case item and determine that the 0 to 1 transition is legal.

```
4'b0000: ns=4'b0001; // 0->1
```

3. Analyze the second case item:

```
4'b00x0: ns=4'b0011; // 2->3
```

In this example, the possible values of the case expression are 0 and 2. VCS and VCS MX have already found a case expression with a value of 0 in the first case item, therefore, for this case item to ever execute, the case expression must have a value of 2. Consequently, VCS and VCS MX determine that the 2 to 3 transition is legal.

4. Analyze the third case item:

```
4'b001x: ns=4'b0010; // 3->2
```

The possible values of the case expression are 2 and 3. If the case expression had a value of 2, it would never execute, therefore, the case expression must have a value of 3. Consequently, VCS and VCS MX determine that the 3 to 2 transition is legal.

5. Analyze the fourth case item:

```
4'b0x11: ns=4'b0101; // none, 3 is used
```

A case expression with a value of 3 is in a previous case item so it would never execute if it had a value of 3. The only other possible value of the case expression is 7, and 7 is not one of the possible states of the FSM, therefore, this case item is not used in determining a legal transition.

6. Analyze the default case item:

```
default: ns=4; // 1->4, 5->4
```

VCS and VCS MX ignore a transition from 4 to 4. There have been no transitions from the other possible states 1 and 5 used so far, therefore, VCS or VCS MX determine from this case item that the 1 to 4 and 5 to 4 transitions are legal.

Reporting FSM State Values Instead of Named States

There are two ways to express an FSM:

- With procedural assignments to the entire reg (variable) that holds the current state and sometimes to the reg that holds the next state of the FSM. It could also be a continuous assignment to net that holds the next state.
- With procedural assignments to individual bits of the reg that holds the current or next state of the FSM.

VCS/VCS MX coverage metrics identifies the states of the FSM by analyzing assignment statements. If VCS/VCS MX finds parameter names for bit numbers in these assignment statements, and the FSM is expressed as assignments to bits, like the Hot Bit or One Hot FSM in [Example 5-8](#), by default, it uses these parameter names as the states of the FSM. In the simulation of [Example 5-8](#), by default,

VCS/VCS MX monitors for, and cmView displays or reports transitions of START to SA or SE to START instead of reporting the value transitions of the reg that holds the current state.

If you want VCS/VCS MX to monitor the value of the reg that holds the current state, and cmView to display and report these values, instead of the parameters like START and SA, enter the `-cm_fsmopt reportvalues` compile-time option and keyword argument.

In [Example 5-8](#), using this compile-time option with the START to SA transition would be reported as '`h1`' to '`h2`' and the SE to START transition would be reported as a '`h20`' to '`h1`'.

This feature does not work for VHDL coverage metrics.

Enabling Indirect Assignment to State Variables

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. You can allow FSM extraction when there is indirect assignment to the variable that holds the current state with the `-cm_fsmopt allowTmp` compile-time option and keyword argument.

Without this option and argument, VCS/VCS MX does not extract the FSM in [Example 5-18](#):

Example 5-18 Indirect Assignment to State Variables

```
module fsm;
    reg [3:0] ns;
    wire [3:0] cs;
    Connect con(cs,ns);
    always @ cs
```

```

casex(cs)
  4'b0000: ns=4'b0001; // 0?1
  4'b00x0: ns=4'b0011; // 2?3
  4'b001x: ns=4'b0010; // 3?2
  4'b0x11: ns=4'b0101; //none
  default: ns=4; // 1?4, 5?4
endcase // casex(cs)
endmodule // fsm

module Connect(cs,ns);
  input [3:0] ns;
  output [3:0] cs;
  reg [3:0] temp; // this is the temp variable
  assign cs = temp;
  always @ ns
    begin #1 temp = ns;
    end
endmodule // Connect

```

In this FSM, the value of the signal that holds the next state of the FSM is assigned to signal `temp`, which is continuously assigned to the signal that holds the current state. You can instruct VCS/VCS MX to extract the FSM despite this indirect assignment by using the `-cm_fsmopt allowTmp` compile-time option and keyword argument.

This feature does not work in VHDL coverage metrics.

Enabling Two-state FSMs

By default, FSMs must have more than two states. You can tell VCS/VCS MX to extract two-state FSMs, scalar signals for the current and next state, with the `-cm_fsmopt report2StateFsms` compile-time option and keyword argument.

Without this option and argument, VCS/VCS MX does not extract the FSM in the following code:

```
module fsm(in);
    input in;
    reg cs, ns;
    parameter zero=1'b0,
              one =1'b1;
    always @ in
        begin
            case(cs)
                zero : ns=one;
                one : ns=zero;
            endcase // case(cs)
        end
    always @ ns
        cs <= ns;
endmodule // fsm
```

This feature does not work in VHDL coverage metrics.

Enabling the Monitoring of Self Looping FSMs

VCS/VCS MX can extract an FSM from code in which the signal that holds the current state is assigned its current value. However, by default ,VCS/VCS MX cannot monitor and have cmView report “transitions” in which it is assigned its current value. You can enable this monitoring with the `-cm_fsmopt reportWait` compile-time option and keyword argument.

Without this option and argument, VCS cannot monitor certain transitions in [Example 5-19](#):

Example 5-19 Monitoring Self Looping FSMs

```
module fsm(sig1,sig2);
    input sig1,sig2;
```

```

reg [2:0] cs,ns;
parameter zero = 3'b000,
          first = 3'b001,
          second= 3'b010,
          third = 3'b011,
          fourth = 3'b100;
always @ (cs or sig1)
begin
  case (cs)
    zero : begin
      if (~sig1)
        ns = zero; // self loop
      else
        ns = first;
    end
    first :
      ns = second;
    second :
      ns = third;
    third :
      ns = fourth;
    fourth :
      if (~sig2)
        ns = zero;
      else
        ns = fourth; // self loop
  endcase // case(cs)
end
always @ sig2
begin
  cs <= ns;
end
endmodule // fsm

```

In this FSM, the current state signal is assigned the value of the next state signal, and in some cases, the next state signal is assigned the value of the current state signal, such as when the current state signal value is zero or fourth.

Note:

There is a performance cost to using this feature. That is why it is not the default behavior.

This feature does not work in VHDL coverage metrics.

Enabling X Value States

You may want to assign an X value to the signal that holds the current state to indicate a problem in the logic of the FSM. By default, VCS/VCS MX does not extract FSMs that have X values. However, you can enable this extraction with the `-cm_fsmopt reportXassign` compile-time option and keyword argument.

Without this option and argument, VCS does not extract the FSM in [Example 5-20](#):

Example 5-20 Enabling X Value States

```
module fsm (sig) ;
    input sig;
    reg [10:0] ns;
    reg [10:0] cs;
    parameter IDLE = 0,
              STATE1 = 1,
              STATE2 = 2,
              STATE3 = 3;

    always @ cs
        begin
            case(cs)
                IDLE :
                    ns = STATE1;
                STATE1 :
                    ns = STATE2;
                STATE2 :
                    ns = STATE3;
```

```

STATE3 :
    ns = IDLE;
default:
    ns = 4'bxxxx; // the whole value
endcase // case(cs)
end
always @ sig
begin
    cs <= ns;
end
endmodule // fsm

module fsm1 (sig,reset) ;
    input sig,reset;
    reg [10:0] ns;
    reg [10:0] cs;
    parameter IDLE = 0,
                STATE1 = 1,
                STATE2 = 2,
                STATE3 = 3;

    always @ cs
    begin
        if(reset)
            cs=IDLE;
        else begin
            case(cs)
                IDLE :
                    ns = STATE1;
                STATE1 :
                    ns = STATE2;
                STATE2 :
                    ns = STATE3;
                STATE3 :
                    ns = IDLE;
                default:
                    ns = 4'b000x; // just 1 bit is x
            endcase // case(cs)
        end // else: !if(reset)
    end
    always @ sig
    begin

```

```
    cs <= ns;
end
endmodule // fsm1
```

cmView reports transitions to and from X as follows:

```
IDLE->X
X->STATE2
```

This feature does not work in VHDL coverage metrics.

Filtering Out Transitions Caused by Specified Signals

You can filter out transitions in assignment statements controlled by `if` statements where the conditional expression (following the `if` keyword) is a signal you specify. You might want to do this, for example, to a reset signal. This filtering out can be used for the specified signal in any module definition or in the module definition you specify. You can also specify the FSM and whether the signal is true or false.

To do this, use the `-cm_fsmresetfilter` compile-time option, and in the file that you specify, include what you want to filter. The following is an example of the contents of this file:

```
signal=reset case=TRUE
module=abc signal=rst case=FALSE
module=xyz fsm=state signal=more_rst CASE=TRUE
module=ABC fsm=STATE signal=reset case=NONE.
```

The first line does not specify a module, therefore, it applies to all signals named `reset` in any module definition. The filtering out applies when that signal is true.

The second line begins with a specified module, named abc. It applies to the signal named rst when that signal is false.

The third line is used for module xyz. It applies to assignments in FSM named state, to the signal named, more_rst, when that signal is true.

The fourth line applies to the module named ABC, the FSM named STATE, the signal named reset, and any type of transition on that signal. The statement case=NONE specifies not considering the value of the signal.

Note:

This applies only to assignments controlled by the if statement. If you are using an if-else statement, it does not filter out transitions controlled by the else part. Therefore, for example, if the file contains the following entry:

```
signal=reset case=TRUE
```

In the following FSM:

```
always@ns
cs <= ns;

always@cs
begin
    if (reset)
        cs = 1;
    else
        cs <= ns;
    case(cs)
        0: ns=1;
        1: ns=2;
        2: ns=3;
        default: ns=0;
    endcase
```

```
end
```

The explicit assignment of the 1 value to the signal that holds the current state is filtered out because it is controlled by the `if` part of the `if-else` statement, but neither of the assignments of the next state to the current state are filtered out, including the one controlled by the `else` part.

FSM Coverage Reports

`cmView` writes its report files in the `./simv.cm/reports` directory, and writes FSM coverage in the following report files:

`cmView.long_f`

A long report file, organized by module instance, containing comprehensive information about the FSM coverage of your design.

`cmView.long_fd`

Another long report file, similar to the `cmView.long_f` file, but organized by module definition instead of by module instance.

`cmView.hier_f`

Summary FSM coverage information for subhierarchies. For each instance, the information is specific to that instance and the instances hierarchically below that instance.

`cmView.mod_f`

A summary FSM coverage report file for module instances. For each instance, the information is specific to that instance alone.

`cmView.mod_fd`

Another summary FSM coverage report file, similar to the `cmView.mod_f` file, but organized by module definition instead of by module instance.

`cmView.short_f`

A short report file containing only sections for instances with states that were not covered and summary information about the FSM coverage of your design.

`cmView.short_fd`

Another short report file, similar to the `cmView.short_f` file, but organized by module definition instead of by module instance.

The following sections contain examples of FSM coverage reports. They are interrupted in a number of places to explain their content.

The `cmView.long_f` File

```
// Synopsys, Inc.  
//  
// Generated by: cmView version number  
// User: user name  
// Date: Day Month date hour:minute:second year  
  
//                                     LONG FSM COVERAGE REPORT  
//*****  
//***** MODULE INSTANCE COVERAGE  
  
// This section contains coverage for each instance of a module  
  
Test Coverage Result: Total Coverage  
  
MODULE top fsm1_1  
  
FILE      example.v  
FSM       cs  
// state coverage results  
'h0          | Covered
```

'h1	Covered
'h2	Covered
'h3	Covered
// state transition coverage results	
'h0->'h1	Covered
'h1->'h2	Covered
'h2->'h3	Covered
'h3->'h0	Covered
// sequence coverage results	
'h0->'h1	Covered
'h1->'h2	Covered
'h2->'h3	Covered
'h3->'h0	Covered
'h0->'h1->'h2	Covered
'h1->'h2->'h3	Covered
'h2->'h3->'h0	Covered
'h3->'h0->'h1	Covered
'h0->'h1->'h2->'h3	Covered
'h1->'h2->'h3->'h0	Covered
'h2->'h3->'h0->'h1	Covered
'h3->'h0->'h1->'h2	Covered
'h0->'h1->'h2->'h3->'h0	Covered Loop
'h1->'h2->'h3->'h0->'h1	Covered Loop
'h2->'h3->'h0->'h1->'h2	Covered Loop
'h3->'h0->'h1->'h2->'h3	Covered Loop

The following section references a module instance named, `top fsm1_1`. In the module definition for this instance, a signal named `cs` holds the current state of an FSM in that module definition. It reports that the FSM reached all its possible states: 0, 1, 2, and 3. VCS/VCS MX also analyzed the source code for the FSM and determined all the possible transitions in the FSM. `cmView` reports that all these possible transitions did occur and also reports on the sequences of states.

```
//-----
//                               Single FSM Coverage Summary

          TOTAL    COVERED    PERCENT
States        4         4      100.00
Transitions   4         4      100.00
Sequences    16        16     100.00
```

The following displays summary information about the FSM:

```
//-----
```

```
//          Module Coverage Summary

          TOTAL    COVERED    PERCENT
Fsms        1         1      100.00
States      4         4      100.00
Transitions 4         4      100.00
Sequences   16        16     100.00
```

The following displays summary information about the FSM coverage for the module instance `top.fsm1_1`:

```
MODULE top.fsm2_1

FILE      example.v
FSM       current
// state coverage results
  'h1           | Covered
  'h2           | Covered
// state transition coverage results
  'h1->'h2    | Covered
  'h2->'h1    | Covered
// sequence coverage results
  'h1->'h2    | Covered
  'h2->'h1    | Covered
  'h1->'h2->'h1 | Covered Loop
  'h2->'h1->'h2 | Covered Loop
```

The following section references a module instance named, `top.fsm2_1`. In the module definition for this instance, there exists an FSM where the current state of the FSM is stored in a signal named `current`. The report shows that all possible states and transitions occurred.

```
-----//
//          Single FSM Coverage Summary

          TOTAL    COVERED    PERCENT
States      2         2      100.00
Transitions 2         2      100.00
Sequences   4         4      100.00
```

The following displays summary information for the FSM:

```
-----//
//          Module Coverage Summary
```

	TOTAL	COVERED	PERCENT
Fsms	1	1	100.00
States	2	2	100.00
Transitions	2	2	100.00
Sequences	4	4	100.00

The following displays summary information for the module instance:

```
MODULE top.fsm3_1

FILE      example.v
FSM        fsm1_current_state
// state coverage results
    fsm1_start          | Covered
    fsm1_step1          | Covered
    fsm1_step2          | Not Covered
// state transition coverage results
    fsm1_start->fsm1_step1 | Covered
    fsm1_step1->fsm1_step2 | Not Covered
    fsm1_step2->fsm1_start | Not Covered
// sequence coverage results
    fsm1_start->fsm1_step1 | Covered
    fsm1_step1->fsm1_step2 | Not Covered
    fsm1_step2->fsm1_start | Not Covered
    fsm1_start->fsm1_step1->fsm1_step2 | Not Covered
    fsm1_step1->fsm1_step2->fsm1_start | Not Covered
    fsm1_step2->fsm1_start->fsm1_step1 | Not Covered
    fsm1_start->fsm1_step1->fsm1_step2->fsm1_start | Not Covered Loop
    fsm1_step1->fsm1_step2->fsm1_start->fsm1_step1 | Not Covered Loop
    fsm1_step2->fsm1_start->fsm1_step1->fsm1_step2 | Not Covered Loop
```

The following section references a module instance named, `top.fsm3_1`. The section begins with coverage information for an FSM where the current state is stored in a signal named `fsm1_current_state`. Not all possible states or transitions were covered.

```
//-----
//          Single FSM Coverage Summary

TOTAL      COVERED      PERCENT
States     3           2           66.67
Transitions 3           1           33.33
Sequences   9           1           11.11
```

The following displays summary information for the FSM:

FSM	fsm2_current_state	
// state coverage results		
fsm2_start		Covered
fsm2_step1		Covered
fsm2_step2		Not Covered
// state transition coverage results		
fsm2_start->fsm2_step1		Covered
fsm2_step1->fsm2_step2		Not Covered
fsm2_step2->fsm2_start		Not Covered
// sequence coverage results		
fsm2_start->fsm2_step1		Covered
fsm2_step1->fsm2_step2		Not Covered
fsm2_step2->fsm2_start		Not Covered
fsm2_start->fsm2_step1->fsm2_step2		Not Covered
fsm2_step1->fsm2_step2->fsm2_start		Not Covered
fsm2_step2->fsm2_start->fsm2_step1		Not Covered
fsm2_start->fsm2_step1->fsm2_step2->fsm2_start		Not Covered Loop
fsm2_step1->fsm2_step2->fsm2_start->fsm2_step1		Not Covered Loop
fsm2_step2->fsm2_start->fsm2_step1->fsm2_step2		Not Covered Loop

The following displays FSM coverage information about another FSM in the module definition. In this FSM, the current state is stored in a signal named `fsm2_current_state`. Not all possible states or transitions are covered.

----- // Single FSM Coverage Summary			
	TOTAL	COVERED	PERCENT
States	6	4	66.67
Transitions	6	2	33.33
Sequences	9	1	11.11

The following displays summary information about the FSM:

FSM	fsm3_current_state	
// state coverage results		
fsm3_start		Covered
fsm3_step1		Covered
fsm3_step2		Not Covered
// state transition coverage results		
fsm3_start->fsm3_step1		Covered
fsm3_step1->fsm3_step2		Not Covered
fsm3_step2->fsm3_start		Not Covered
// sequence coverage results		
fsm3_start->fsm3_step1		Covered
fsm3_step1->fsm3_step2		Not Covered
fsm3_step2->fsm3_start		Not Covered
fsm3_start->fsm3_step1->fsm3_step2		Not Covered
fsm3_step1->fsm3_step2->fsm3_start		Not Covered
fsm3_step2->fsm3_start->fsm3_step1		Not Covered
fsm3_start->fsm3_step1->fsm3_step2->fsm3_start		Not Covered Loop

```

fsm3_step1->fsm3_step2->fsm3_start->fsm3_step1 | Not Covered Loop
fsm3_step2->fsm3_start->fsm3_step1->fsm3_step2 | Not Covered Loop
//-----

```

The following displays FSM coverage information about another FSM in the module definition. In this FSM, the current state is stored in a signal named `fsm3_current_state`. Not all possible states or transitions are covered.

```

//          Single FSM Coverage Summary

          TOTAL    COVERED    PERCENT
States        9         6       66.67
Transitions   9         3       33.33
Sequences    9         1       11.11

```

The following displays summary information about the FSM:

```

//          Module Coverage Summary

          TOTAL    COVERED    PERCENT
Fsms          3         0       0.00
States        9         6       66.67
Transitions   9         3       33.33
Sequences    27        3       11.11

```

The following displays summary information about the module instance:

```

//*****
//          Total Coverage Summary

          TOTAL    COVERED    PERCENT
Fsms          5         2       40.00
States        15        12      80.00
Transitions   15        9       60.00
Sequences    47        23      48.94

```

The following displays summary information about the entire design:

The cmView.long_fd File

The `cmView.long_fd` file is similar to the `cmView.long_f` file, except that the coverage information in it is organized into sections for module definitions instead of module instances. If there are multiple instances of a module definition, and if a state, transition, or sequence of an FSM occurred in any of these instances, then this file reports them as covered. The following is an excerpt from the corresponding `cmView.long_fd` file for the `cmView.long_f` file in “[The cmView.long_f File](#)”:

```
MODULE fsm2

FILE      example.v
FSM       current
// state coverage results
    'h1           | Covered
    'h2           | Covered
// state transition coverage results
    'h1->'h2    | Covered
    'h2->'h1    | Covered
// sequence coverage results
    'h1->'h2    | Covered
    'h2->'h1    | Covered
    'h1->'h2->'h1 | Covered Loop
    'h2->'h1->'h2 | Covered Loop
//-----//
//          Single FSM Coverage Summary

          TOTAL   COVERED   PERCENT
States        2        2        100.00
Transitions   2        2        100.00
Sequences     4        4        100.00
//-----//

//-----//
//          Module Coverage Summary

          TOTAL   COVERED   PERCENT
Fsms         1        1        100.00
States        2        2        100.00
Transitions   2        2        100.00
Sequences     4        4        100.00
```

The cmView.hier_f File

The cmView.hier_f file contains summary information for module instances in which the coverage data for a higher level module instance includes the coverage data for module instances in the subhierarchy under them.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version number  
// User: user name  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE HIERARCHICAL INSTANCE COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// instance of a module. The statistics take into account all sub-hierarchies  
// under the instance.  
  
Test Coverage Result: Total Coverage  


| Module Name | States (%) | States (4/4) | Trans. (%) | Trans. (4/4) | Seq. (%) | Seq. (16/16) |
|-------------|------------|--------------|------------|--------------|----------|--------------|
| top_fsm1_1  | 100.00     | 4/4          | 100.00     | 4/4          | 100.00   | 16/16        |
| top_fsm2_1  | 100.00     | 2/2          | 100.00     | 2/2          | 100.00   | 4/4          |
| top_fsm3_1  | 66.67      | 6/9          | 33.33      | 3/9          | 11.11    | 3/27         |

  
//*****  
// Total Module Instance Coverage Summary  


|             | TOTAL | COVERED | PERCENT |
|-------------|-------|---------|---------|
| Fsms        | 5     | 2       | 40.00   |
| States      | 15    | 12      | 80.00   |
| Transitions | 15    | 9       | 60.00   |
| Sequences   | 47    | 23      | 48.94   |


```

The cmView.mod_f File

The cmView.mod_f file is similar to the cmView.hier_f file, except that the coverage data for a module instance is used for that module instance alone and not the instances hierarchically under it.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version number  
// User: user name  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE INSTANCE COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// instance of a module. The statistics do not take into account any  
// sub-hierarchy under the instance.  
  
Test Coverage Result: Total Coverage  


| Module Name | States (%) | States (%) | Trans. (%) | Trans. (%) | Seq. (%) | Seq. (%) |
|-------------|------------|------------|------------|------------|----------|----------|
| top_fsm1_1  | 100.00     | 4/4        | 100.00     | 4/4        | 100.00   | 16/16    |
| top_fsm2_1  | 100.00     | 2/2        | 100.00     | 2/2        | 100.00   | 4/4      |
| top_fsm3_1  | 66.67      | 6/9        | 33.33      | 3/9        | 11.11    | 3/27     |

  
//*****  
// Total Module Instance Coverage Summary  


|             | TOTAL | COVERED | PERCENT |
|-------------|-------|---------|---------|
| Fsms        | 5     | 2       | 40.00   |
| States      | 15    | 12      | 80.00   |
| Transitions | 15    | 9       | 60.00   |
| Sequences   | 47    | 23      | 48.94   |


```

The cmView.mod_fd File

The cmView.mod_fd file is similar to the cmView.mod_f file, except that the coverage data reported is used for module definitions, not module instances. If there are multiple instances of a module, this report contains the cumulative coverage of all the instances.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version number  
// User: user name  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE DEFINITION COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// module definition. The coverage is cumulative over all the instances  
// of the module  
  
Test Coverage Result: Total Coverage  
  


| Module Name | States (%) | States (%) | Trans. (%) | Trans. (%) | Seq. (%) | Seq.  |
|-------------|------------|------------|------------|------------|----------|-------|
| fsm1        | 100.00     | 4/4        | 100.00     | 4/4        | 100.00   | 16/16 |
| fsm2        | 100.00     | 2/2        | 100.00     | 2/2        | 100.00   | 4/4   |
| fsm3        | 66.67      | 6/9        | 33.33      | 3/9        | 11.11    | 3/27  |

  
//*****  
// Total Module Definition Coverage Summary  
  


|             | TOTAL | COVERED | PERCENT |
|-------------|-------|---------|---------|
| Fsms        | 5     | 2       | 40.00   |
| States      | 15    | 12      | 80.00   |
| Transitions | 15    | 9       | 60.00   |
| Sequences   | 47    | 23      | 48.94   |


```

The cmView.short_f File

The cmView.short_f file provides summary information on instances that have 100% FSM coverage and provides detailed information for instances that do not have 100% coverage.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version number  
// User: user name  
// Date: Day Month date hour:minute:second year
```

```
// Synopsys, Inc.  
//  
// Generated by: cmView 6.1Beta  
// User: pmcgee  
// Date: Wed Oct 17 15:00:18 2001
```

SHORT FSM COVERAGE REPORT

```
*****  
// MODULE INSTANCE COVERAGE
```

```
// This section contains coverage for each instance of a module
```

```
Test Coverage Result: Total Coverage
```

```
MODULE top fsm1_1
```

```
FILE example.v  
FSM cs  
// state coverage results  
// state transition coverage results  
// sequence coverage results  
-----  
// Single FSM Coverage Summary
```

	TOTAL	COVERED	PERCENT
States	-	-	
Transitions	-	-	
Sequences	16	16	100.00

```
-----  
-----
```

```

//          Module Coverage Summary

          TOTAL    COVERED    PERCENT
Fsms        1         1      100.00
States       4         4      100.00
Transitions  4         4      100.00
Sequences    16        16     100.00

//-----
MODULE top.fsm2_1

FILE      example.v
FSM       current
// state coverage results
// state transition coverage results
// sequence coverage results
//-----  

//          Single FSM Coverage Summary

          TOTAL    COVERED    PERCENT
States      -         -      100.00
Transitions -         -      100.00
Sequences   4         4      100.00
//-----  

//-----  

//          Module Coverage Summary

          TOTAL    COVERED    PERCENT
Fsms        1         1      100.00
States       2         2      100.00
Transitions 2         2      100.00
Sequences   4         4      100.00

//-----  

//-----  

MODULE top.fsm3_1

FILE      example.v
FSM       fsm1_current_state
// state coverage results
          fsm1_step2           | Not Covered
// state transition coverage results
          fsm1_step1->fsm1_step2 | Not Covered
          fsm1_step2->fsm1_start | Not Covered
// sequence coverage results
          fsm1_step1->fsm1_step2 | Not Covered
          fsm1_step2->fsm1_start | Not Covered
          fsm1_start->fsm1_step1->fsm1_step2 | Not Covered

```

```

        fsm1_step1->fsm1_step2->fsm1_start | Not Covered
        fsm1_step2->fsm1_start->fsm1_step1 | Not Covered
fsm1_start->fsm1_step1->fsm1_step2->fsm1_start | Not Covered Loop
fsm1_step1->fsm1_step2->fsm1_start->fsm1_step1 | Not Covered Loop
fsm1_step2->fsm1_start->fsm1_step1->fsm1_step2 | Not Covered Loop
//-----
//          Single FSM Coverage Summary

          TOTAL    COVERED    PERCENT
States           1        0       0.00
Transitions      2        0       0.00
Sequences        9        1      11.11
//-----

          FSM      fsm2_current_state
// state coverage results
        fsm2_step2 | Not Covered
// state transition coverage results
        fsm2_step1->fsm2_step2 | Not Covered
        fsm2_step2->fsm2_start | Not Covered
// sequence coverage results
        fsm2_step1->fsm2_step2 | Not Covered
        fsm2_step2->fsm2_start | Not Covered
        fsm2_start->fsm2_step1->fsm2_step2 | Not Covered
        fsm2_step1->fsm2_step2->fsm2_start | Not Covered
        fsm2_step2->fsm2_start->fsm2_step1 | Not Covered
        fsm2_start->fsm2_step1->fsm2_step2->fsm2_start | Not Covered Loop
        fsm2_step1->fsm2_step2->fsm2_start->fsm2_step1 | Not Covered Loop
        fsm2_step2->fsm2_start->fsm2_step1->fsm2_step2 | Not Covered Loop
//-----
//          Single FSM Coverage Summary

          TOTAL    COVERED    PERCENT
States           2        0       0.00
Transitions      4        0       0.00
Sequences        9        1      11.11
//-----

          FSM      fsm3_current_state
// state coverage results
        fsm3_step2 | Not Covered
// state transition coverage results
        fsm3_step1->fsm3_step2 | Not Covered
        fsm3_step2->fsm3_start | Not Covered
// sequence coverage results
        fsm3_step1->fsm3_step2 | Not Covered
        fsm3_step2->fsm3_start | Not Covered
        fsm3_start->fsm3_step1->fsm3_step2 | Not Covered
        fsm3_step1->fsm3_step2->fsm3_start | Not Covered
        fsm3_step2->fsm3_start->fsm3_step1 | Not Covered
        fsm3_start->fsm3_step1->fsm3_step2->fsm3_start | Not Covered Loop
        fsm3_step1->fsm3_step2->fsm3_start->fsm3_step1 | Not Covered Loop
        fsm3_step2->fsm3_start->fsm3_step1->fsm3_step2 | Not Covered Loop
//-----
//          Single FSM Coverage Summmary

```

```

                TOTAL    COVERED    PERCENT
States          3        0        0.00
Transitions     6        0        0.00
Sequences       9        1       11.11
//-----
//-----  

//           Module Coverage Summary  

                TOTAL    COVERED    PERCENT
Fsms           3        0        0.00
States          9        6       66.67
Transitions     9        3       33.33
Sequences       27       3       11.11
//-----  

//*****  

//           Total Module Instance Coverage Summary  

                TOTAL    COVERED    PERCENT
Fsms           5        2       40.00
States          15       12      80.00
Transitions     15       9       60.00
Sequences       47       23      48.94

```

The cmView.short_fd File

The `cmView.short_fd` file is similar to the `cmView.short_f` file, except that the information in it is organized by sections for module definitions instead of module instances.

```

// Synopsys, Inc.
//
// Generated by: cmView version number
// User: user name
// Date: Day Month date hour:minute:second year

```

SHORT FSM COVERAGE REPORT

```

//*****  

//           MODULE DEFINITION COVERAGE

```

```
// This section contains coverage for module definitions.  
// The coverage is cumulative over all the instances of the module
```

```
Test Coverage Result: Total Coverage
```

```
MODULE fsm1
```

```
FILE      example.v  
FSM       cs  
// state coverage results  
// state transition coverage results  
// sequence coverage results  
//-----  
//           Single FSM Coverage Summary  
  
          TOTAL    COVERED    PERCENT  
States        -  
Transitions   -  
Sequences     16        16        100.00  
//-----  
//-----  
//           Module Coverage Summary  
  
          TOTAL    COVERED    PERCENT  
Fsms         1          1          100.00  
States        4          4          100.00  
Transitions   4          4          100.00  
Sequences     16        16        100.00
```

```
//-----
```

```
MODULE fsm2
```

```
FILE      example.v  
FSM       current  
// state coverage results  
// state transition coverage results  
// sequence coverage results  
//-----  
//           Single FSM Coverage Summary  
  
          TOTAL    COVERED    PERCENT  
States        -  
Transitions   -  
Sequences     4          4          100.00  
//-----  
//-----  
//           Module Coverage Summary
```

	TOTAL	COVERED	PERCENT
Fsms	1	1	100.00
States	2	2	100.00
Transitions	2	2	100.00
Sequences	4	4	100.00

//-----

MODULE fsm3

FILE example.v	
FSM fsm1_current_state	
// state coverage results	
fsm1_step2	Not Covered
// state transition coverage results	
fsm1_step1->fsm1_step2	Not Covered
fsm1_step2->fsm1_start	Not Covered
// sequence coverage results	
fsm1_step1->fsm1_step2	Not Covered
fsm1_step2->fsm1_start	Not Covered
fsm1_start->fsm1_step1->fsm1_step2	Not Covered
fsm1_step1->fsm1_step2->fsm1_start	Not Covered
fsm1_step2->fsm1_start->fsm1_step1	Not Covered
fsm1_start->fsm1_step1->fsm1_step2->fsm1_start	Not Covered Loop
fsm1_step1->fsm1_step2->fsm1_start->fsm1_step1	Not Covered Loop
fsm1_step2->fsm1_start->fsm1_step1->fsm1_step2	Not Covered Loop

//-----

// Single FSM Coverage Summary

	TOTAL	COVERED	PERCENT
States	1	0	0.00
Transitions	2	0	0.00
Sequences	9	1	11.11

//-----

FILE example.v	
FSM fsm2_current_state	
// state coverage results	
fsm2_step2	Not Covered
// state transition coverage results	
fsm2_step1->fsm2_step2	Not Covered
fsm2_step2->fsm2_start	Not Covered
// sequence coverage results	
fsm2_step1->fsm2_step2	Not Covered
fsm2_step2->fsm2_start	Not Covered
fsm2_start->fsm2_step1->fsm2_step2	Not Covered
fsm2_step1->fsm2_step2->fsm2_start	Not Covered
fsm2_step2->fsm2_start->fsm2_step1	Not Covered
fsm2_start->fsm2_step1->fsm2_step2->fsm2_start	Not Covered Loop
fsm2_step1->fsm2_step2->fsm2_start->fsm2_step1	Not Covered Loop
fsm2_step2->fsm2_start->fsm2_step1->fsm2_step2	Not Covered Loop

//-----

// Single FSM Coverage Summary

```

                TOTAL    COVERED    PERCENT
States          2        0        0.00
Transitions     4        0        0.00
Sequences       9        1        11.11
//-----

                FSM      fsm3_current_state
// state coverage results
            fsm3_step2 | Not Covered
// state transition coverage results
            fsm3_step1->fsm3_step2 | Not Covered
            fsm3_step2->fsm3_start | Not Covered
// sequence coverage results
            fsm3_step1->fsm3_step2 | Not Covered
            fsm3_step2->fsm3_start | Not Covered
            fsm3_start->fsm3_step1->fsm3_step2 | Not Covered
            fsm3_step1->fsm3_step2->fsm3_start | Not Covered
            fsm3_step2->fsm3_start->fsm3_step1 | Not Covered
            fsm3_start->fsm3_step1->fsm3_step2->fsm3_start | Not Covered Loop
            fsm3_step1->fsm3_step2->fsm3_start->fsm3_step1 | Not Covered Loop
            fsm3_step2->fsm3_start->fsm3_step1->fsm3_step2 | Not Covered Loop
//-----
//                               Single FSM Coverage Summary

                TOTAL    COVERED    PERCENT
States          3        0        0.00
Transitions     6        0        0.00
Sequences       9        1        11.11
//-----

//-----
```

```

//                               Module Coverage Summary

                TOTAL    COVERED    PERCENT
Fsms            3        0        0.00
States          9        6        66.67
Transitions     9        3        33.33
Sequences       27       3        11.11
//-----
```

```

//-----
```

```

//*****
```

```

//                               Total Module Definition Coverage Summary

                TOTAL    COVERED    PERCENT
Fsms            5        2        40.00
States          15       12       80.00
Transitions     15       9        60.00
Sequences       47       23       48.94
//-----
```

FSM Coverage

Excluding Sequences From Reports

cmView reporting sequences can make the report files very lengthy. If you are not interested in seeing sequences in reports, use the `cmView -cm_report` command-line option with the `disable_sequence` or `disable_sequence_loop` arguments. For example:

Verilog: `cs -cm_pp -cm_report disable_sequence_loop`

VHDL: `cmView -cmreport disable_sequence_loop`

The arguments specify the following:

`disable_sequence`

cmView does not list any sequences in the report.

`disable_sequence_loop`

cmView does not list any sequence that starts and ends with the same state.

Viewing FSM Coverage With the Coverage Metrics GUI

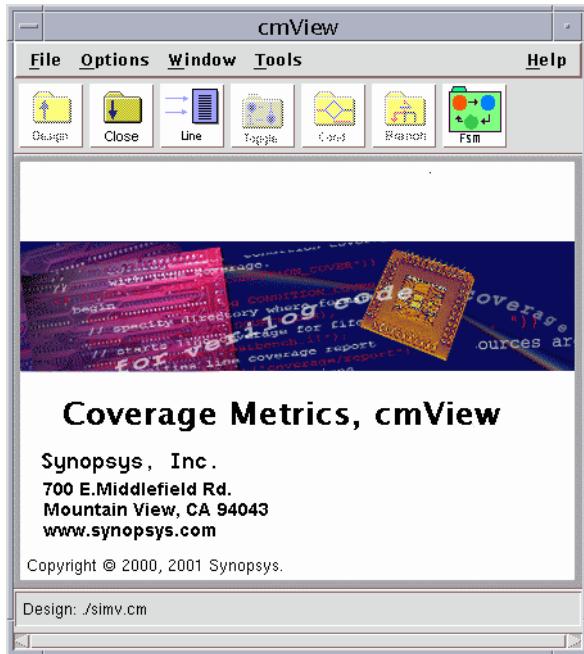
cmView can start the graphical user interface (GUI) for VCS/VCS MX coverage metrics, and you use it to see graphical representations of coverage information recorded by VCS and VCS MX. To do so, perform the following:

1. Enter the following command line:

Verilog: `cs -cm_pp -cm fsm`

VHDL: `cmView -cm fsm`

This command line displays the main window for cmView.



This command line also instruct cmView to read the design file:

- Verilog: ./simv.cm/db/verilog/cm.decl_info
- VHDL: /simv.cm/db/vhdl/cm.decl_info

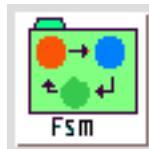
cmView reads this file to learn about the design and its hierarchy. You can read the intermediate data files for FSM coverage in one of the following directories:

- Verilog: simv.cm/coverage/verilog
- VHDL: simv.cm/coverage/vhdl

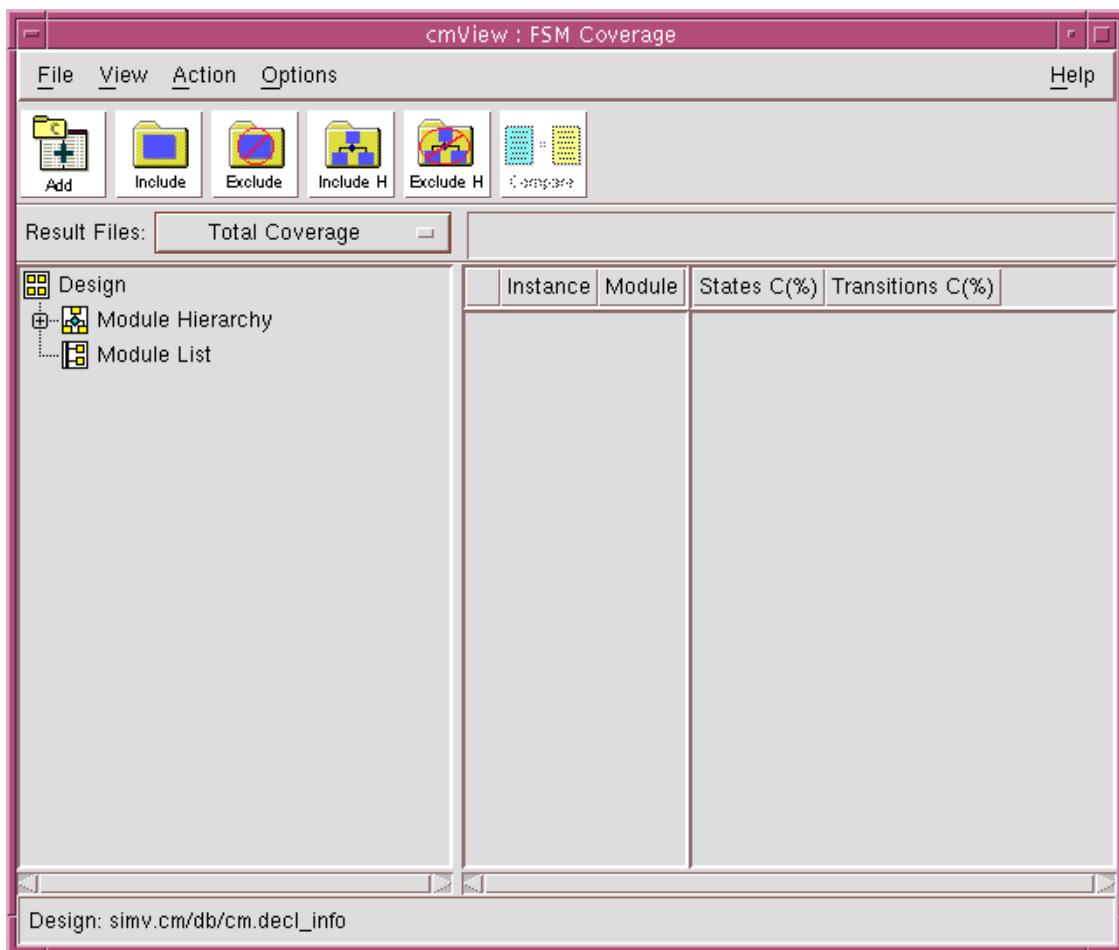
Note:

At compile time, if you specified a different coverage metrics database with the `-cm_dir` compile-time option, also include the option on the cmView (or vcs `-cm_pp` gui) command line.

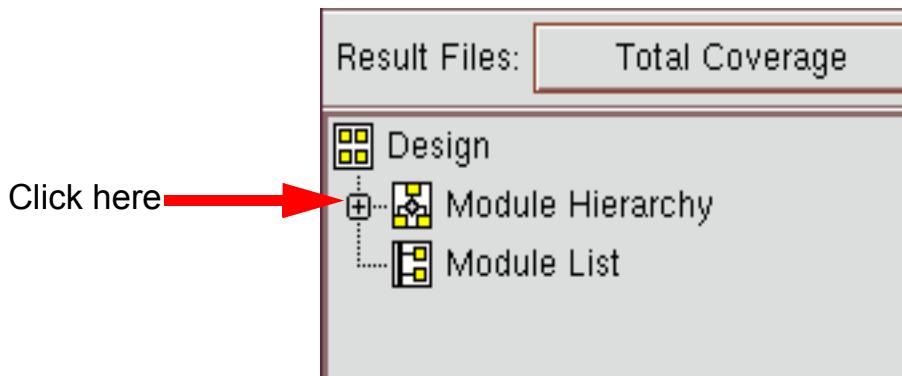
2. Click the FSM Coverage toolbar button:



This displays the FSM Coverage window:



3. By default, cmView displays coverage summary information on a per module instance basis, therefore, the FSM Coverage window needs a way to display the design hierarchy so that you can select module instances to examine their FSM coverage. Click the plus symbol (+) next to the icon for the Module Hierarchy.



This displays the top-level modules (in this case one) in the design. In this case, there were no FSMs in the top-level module.

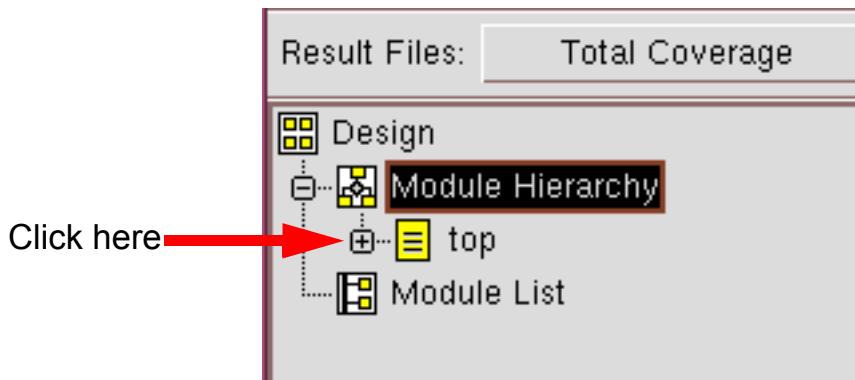
A screenshot of the cmView software interface showing the "Hierarchy Pane" and "Summary Pane".

Hierarchy Pane: Shows a tree structure under "Design". The "Module Hierarchy" node is expanded, revealing a single child node named "top". Other nodes like "Module List" are also present.

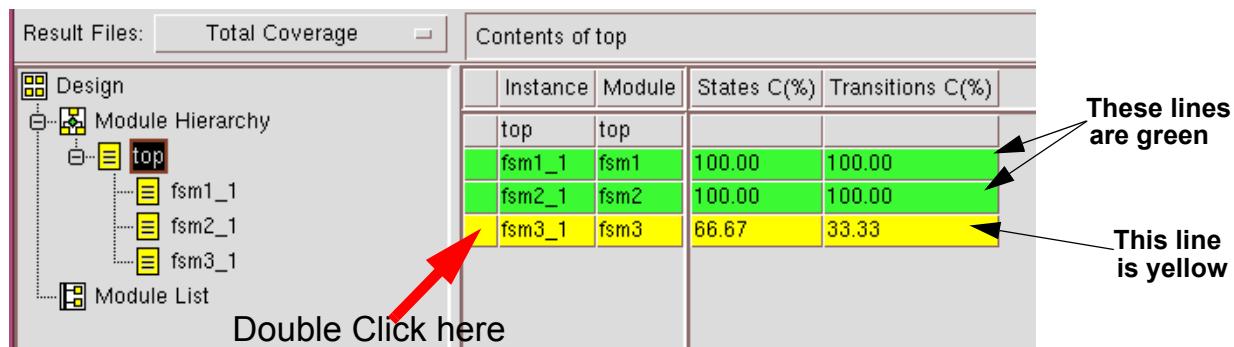
Summary Pane: Shows a table titled "Top-Level Instances".

Instance	Module	States C(%)	Transitions C(%)
top	top		

- Click the (+) plus symbol next to the icon for the top-level module, in this case, top.

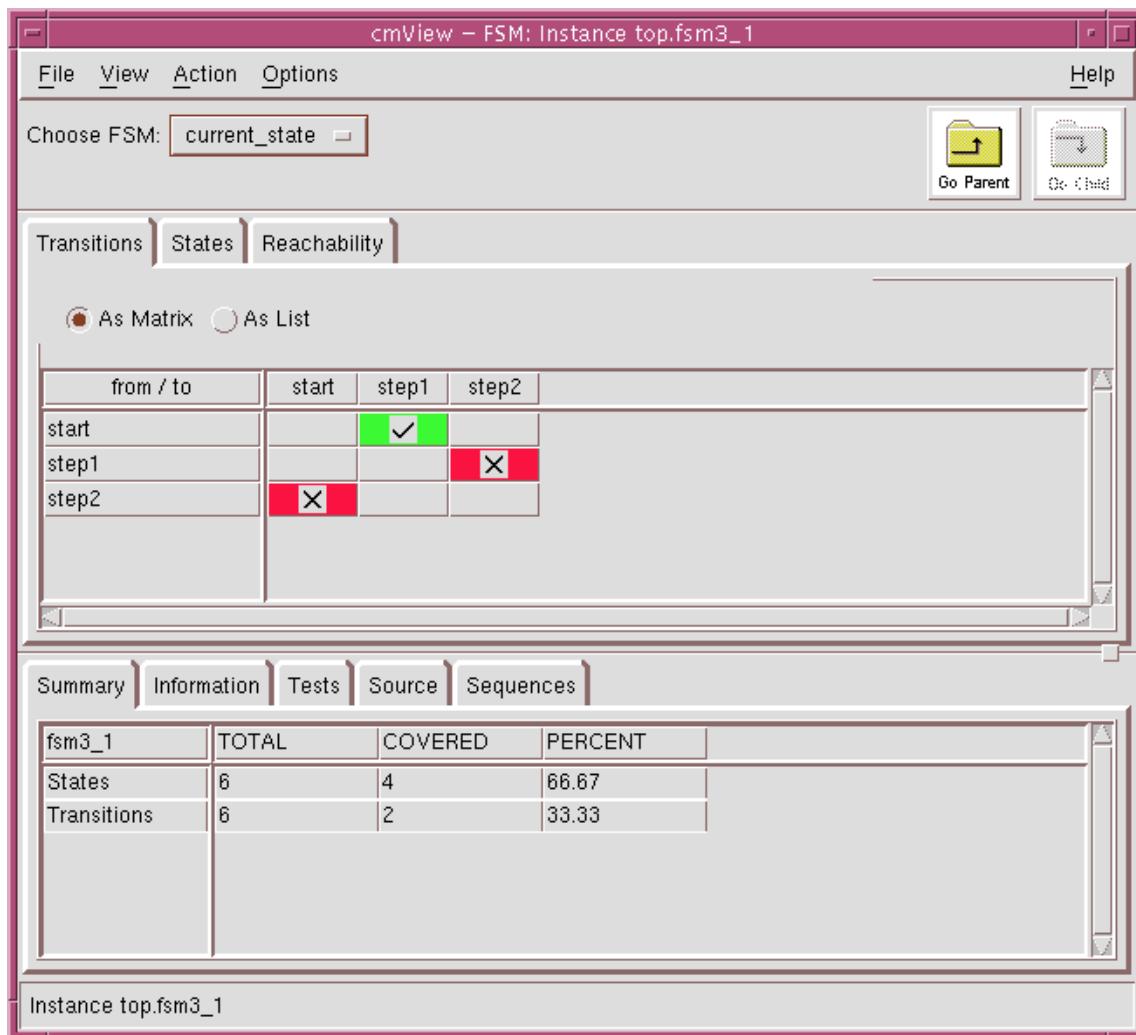


This displays the module instances under the top-level module. The Summary pane contains FSM coverage summary information. By default, a green results line indicates 80-100% coverage, and a yellow results line indicates 20-80% coverage.



In this case, there is 100% FSM coverage for module instances `fsm1_1` and `fsm2_1`. By default, the green results line indicates 80-100% coverage. Module instance `fsm3_1` has limited results.

To view additional FSM coverage information about this instance, double-click anywhere on the line to display the FSM Summary window.



The FSM Summary window appears with the data for an FSM in the instance. The “Choose FSM” drop down list, in the top left corner of the window displays this FSM. The information displayed in the tabs of the FSM Summary window pertain to this FSM. You can change the selected FSM using this drop down list.

The Information and Tests tabs also appear in the summary windows for other types of coverage, but the Transitions, States, Reachability, Source, and Sequences tabs are unique to the FSM Summary window.

The Transitions Tab

The Transitions tab has two views controlled by the radio buttons “As List” and “As Matrix”.

Figure 5-2 The Transitions Tab As List View

FromState -> ToState	Is Covered
start -> step1	✓
step1 -> start	✗
step2 -> start	✗

For the selected FSM, the “As List” view shows the possible transitions between all of the specified or determined states in the FSM and whether the current state of the FSM made any of these transitions (that is, whether the transition is covered).

Specified states are specified in the configuration file. If you do not use a configuration file, these states are determined by VCS and VCS MX when they analyze the Verilog code of the FSM.

By default, the line for a covered transition is green, and displays a check mark (see [Figure 5-2](#)).

By default, the line for an uncovered transition is red, and displays an X (see [Figure 5-2](#)).

Double-click a line in the Transitions tab to bring the Source tab to the top in the lower pane. Scroll through that pane to the statement that assigns the “To State” (the state after the transition occurred) to the reg or wire that holds the next state of the FSM. If there is no such statement, the Source tab scrolls to the procedural statement that assigns the “FromState” to the reg that holds the current state of the FSM.

Double-click on the line again and the Source tab scrolls to the statement that assigns the “ToState” (the state to which the transition occurred) to the reg or wire that holds the next state of the FSM. If there is no such statement, the Source tab scrolls to the procedural statement that assigns the “ToState” to the reg that holds the current state of the FSM.

Figure 5-3 The Transitions Tab as Matrix View

from / to	start	step1	step2
start		✓	
step1	X		
step2	X		

In the “As Matrix” view, the “from” states are listed down the left side, and the “to” states are listed across the top of the matrix.

In this matrix, a check mark indicates a covered transition (see [Figure 5-3](#)). By default, these boxes are green.

An X box indicates an uncovered transition ([Figure 5-3](#)). By default, these boxes are red.

An empty box indicates a transition that was either not specified in the configuration file or you did not specify a configuration file and VCS/VCS MX did not find the transition possible when it extracted the FSM (see [Figure 5-3](#)).

Note:

The boxes in the matrix, where the “from” and “to” states are the same state, are always blank.

VCS/VCS MX finds a transition possible if it sees, in the group of statements that make up the FSM, assignments of the “from” state and the “to” state to the next state reg or wire (or to the current state reg), and if the Verilog constructs that control these assignments make the transition possible.

Example 5-21 Possible Transitions FSM

```
initial
current_state=start;

always @ in
begin
  case (current_state)
    start : next_state=step1;
    step1 : next_state=step2;
    step2 : next_state=start;
  endcase
  #2 current_state=next_state;
end
```

In [Example 5-21](#), the FSM uses a case statement in which the case item statements are procedural assignments of states `start`, `step1`, and `step2` to the `next_state` reg.

There is no case item statement for assigning `step2` to the `next_state` reg when the case item is `start`, so the `start` to `step2` transition is not possible.

Similarly, there is no case item statement for assigning `step1` to the `next_state` reg when the case item is `step2`, therefore, the `step2` to `step1` transition is impossible.

Therefore, in [Figure 5-3](#):

- The transitions from state `start` to state `start`, from `step1` to `step1` and from `step2` to `step2` are blank because VCS/VCS MX does not keep track of return to the same state transitions.
- The transition from `step1` to `start` was either specified in the configuration file or found to be possible by VCS/VCS MX, but this transition did not occur during simulation.
- The transition from `step2` to `start` was either specified in the configuration file or found to be possible by VCS/VCS MX, and this transition did occur.
- The transitions from `start` to `step2`, and from `step2` to `step1` are blank because VCS/VCS MX found them to be impossible.

Double-click on a box in the Transition tab to bring the Source tab to the top in the lower pane. Scroll through that pane to the statement that assigns the “from” (the state from which the transition occurred) to the reg or wire that holds the next state of the FSM. If there is no such statement, the Source tab scrolls to the procedural statement that assigns the “from” to the reg that holds the current state of the FSM.

Double-click on the box again and the Source tab scrolls to the statement that assigns the “to” (the state to which the transition occurred) to the reg or wire that holds the next state of the FSM. If there is no such statement, the Source tab scrolls to the procedural statement that assigns the “to” to the reg that holds the current state of the FSM.

The States Tab

The States tab lists the states of the FSM: the states specified in the configuration file, or, if you did not use a configuration file, the states of the FSM that VCS/VCS MX were able to determine from looking at the assignment statements in the group of statements that make up the FSM.

A line for a covered state displays a check mark (see [Figure 5-4](#)). By default, this line is green.

A line for an uncovered state displays an X (see [Figure 5-4](#)). By default, this line is red.

Figure 5-4 The States Tab

The screenshot shows a software interface with a tab labeled "States". Below the tab is a table with two columns: "Name" and "Is Covered". The table has four rows. The first row, "start", has a green background and a checked checkbox in the "Is Covered" column. The next two rows, "step1" and "step2", have red backgrounds and unchecked checkboxes.

Name	Is Covered
start	<input checked="" type="checkbox"/>
step1	<input type="checkbox"/>
step2	<input type="checkbox"/>

The Reachability Tab

The Reachability tab shows you whether an FSM could and did reach from one state to another, either directly or through intervening states.

VCS/VCS MX determines whether an FSM could reach from one state to another by analyzing the Verilog code of the FSM before simulation starts. For example, you can write an FSM that transitions from state 1 to 2 to 4 or from 1 to 3 to 5. VCS/VCS MX can determine from an analysis of the code that the FSM can reach from 1 to 5, but can never reach from 2 to 5. This information is called static information or static analysis, because it is determined before simulation starts and is not determined by simulation results.

VCS/VCS MX keeps track of whether the FSM did reach from one state to another including the sequence of intervening states between these states. This information is called dynamic information because it is determined from simulation results.

Like the Transitions tab, the Reachability tab has two views controlled by the radio buttons “As List” and “As Matrix”. Both views are ways of showing you two states and whether it is possible for the FSM to eventually reach one state from another.

Double-click on a pair of states in this tab to invoke the Sequences tab that shows you the different sequences of states that the FSM can follow to reach from one state to the other state.

The Reachability tab has a filter for limiting the display of sequences in the Sequences tab to those that include or do not include a state, or a transition between states, that you specify.

Figure 5-5 The Reachability Tab As List View

FromState -> ToState	Is Covered
start -> start	✓
start -> step1	✓
start -> step2	✓
step1 -> start	✓
step1 -> step1	✗
step1 -> step2	✓
step2 -> start	✓
step2 -> step1	✗
step2 -> step2	✗

In the “As List” view, each line shows a pair of states. The color of the line shows the reachability between these states:

- A white line indicates that you did not load sequence data that contains the sequences of states for the FSM.
- By default, a red line indicates that there is low coverage for reaching between the states in the line. Low coverage can mean that there are multiple ways, or sequences, for the FSM to reach from one of the states to the other and the FSM went through few of these ways or sequences.
- By default, a yellow line indicates moderate coverage.
- By default, a green line indicates high coverage.

The Sequence Filter menu, Param field, and Apply Filter buttons are used for filtering the display in this tab and the Reachability tab. If you use them to *exclude* a state, the color, for all the lines for the two states where the specified state is an intervening state between the two states of the line, changes to red. If you use them to *require* a state, the color, for all the lines for the two states where the specified state is *not* an intervening state between the two states of the line, changes to red.

For example, if an FSM can reach from state `start` to state `step2` using the following sequence:

```
start -> step1 -> step2
```

The line in this tab for `start -> step2` is green and the sequence is shown in green in the Sequences tab. Excluding `step1` changes this line to red in this tab and removes the sequence from the Sequences tab.

Figure 5-6 The Reachability Tab As Matrix View



In the “As Matrix” view, the “from” states are listed down the left side and the “to” states are listed across the top of the matrix.

The default color codes are also used for low, moderate, or high coverage. You can also see the amount of coverage either as a percentage or as a ratio.

The Summary Tab

The Summary tab displays a table of state and transition coverage information showing how many were covered and the percentage covered of the total number of states and possible transitions.

Figure 5-7 The Summary Tab

The screenshot shows a software interface with a title bar 'Summary' and a table below it. The table has four columns: 'fsm3_1', 'TOTAL', 'COVERED', and 'PERCENT'. There are two rows: one for 'States' (6 total, 6 covered, 100.00% percent) and one for 'Transitions' (8 total, 6 covered, 75.00% percent). The background of the main area is light gray.

fsm3_1	TOTAL	COVERED	PERCENT
States	6	6	100.00
Transitions	8	6	75.00

The Source Tab

The Source tab displays a version of the Verilog source file that contains the module definition containing the FSM. This version of the source file contains line numbers. The Source tab highlights the assignment statements that assign the states of the FSM to the reg that holds the current state and the reg or wire that holds the next state.

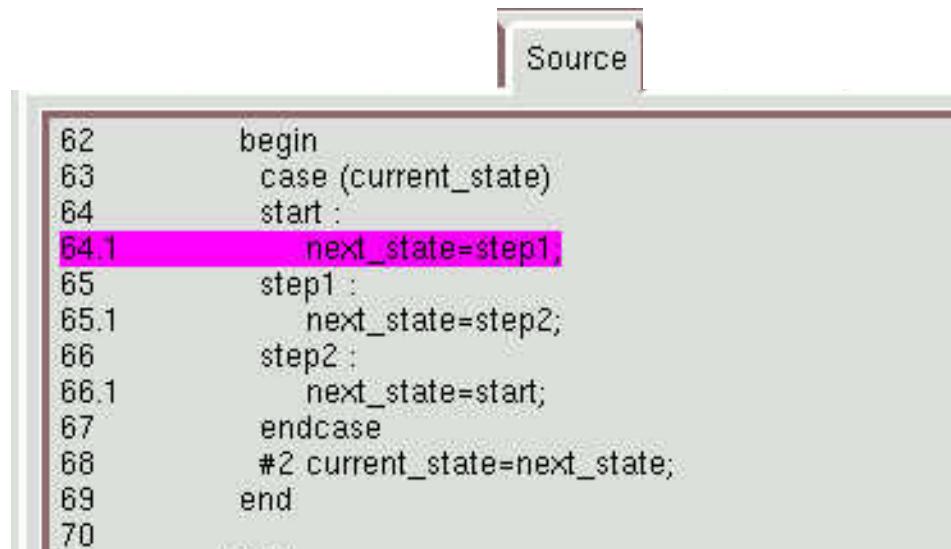
Double-click in the Transitions or States tab to scroll the Source tab to a corresponding assignment statement. Double-click on a state in the States tab to scroll the Source tab to the first assignment of that state.

Double-click on a line or a box in one of the views of the Transitions tab to bring the Source tab to the top in the lower pane, and to scroll to the statement that assigns the “from” state in the transition to the reg or wire that holds the next state of the FSM. If there is no such

statement, the Source tab scrolls to the procedural statement that assigns the “from” state to the reg that holds the current state of the FSM.

Double-click on the line or box again and the Source tab scrolls to the statement that assigns the “to” state to the reg or wire that holds the next state of the FSM. If there is no such statement, it scrolls to the procedural statement that assigns the “to” state to the reg that holds the current state of the FSM.

Figure 5-8 The Source Tab



The screenshot shows a software interface with a tab labeled "Source" at the top. Below the tab, a code editor displays the following Verilog-like pseudocode:

```
62 begin
63 case (current_state)
64 start :
64.1     next_state=step1;
65 step1 :
65.1     next_state=step2;
66 step2 :
66.1     next_state=start;
67 endcase
68 #2 current_state=next_state;
69 end
70
```

The line "64.1 next_state=step1;" is highlighted with a pink background, indicating it is the selected or currently active line of code.

The Sequences Tab

The Sequences tab shows the different sequences of states, including intervening states, that an FSM can follow to reach from one state to another state that you specify in the Reachability tab. The Reachability tab controls the content of the Sequences tab.

Figure 5-9 The Sequences Tab

List of sequences start -> start		
Is Covered	Length	Sequences
<input checked="" type="checkbox"/>	4	start -> step1 -> step2 -> start
<input type="checkbox"/>	3	start -> step1 -> start
Unfiltered Sequences:		Total = 2
Filtered Sequences:		Total = 2
		Covered = 1
		Ratio = 50.00
		Covered = 1
		Ratio = 50.00

The Sequences tab displays a number of lines and columns.

Each line contains a sequence of states. The states in the line could be just the “from” and “to” states that you selected in the Reachability tab, or they could be a longer sequence showing intervening states between the “from” and “to” states. These “from” and “to” states appear at the top of the tab.

The “Is Covered” column indicates whether the sequence in a line was actually followed by the FSM during simulation.

A line that contains a check mark () in the “Is Covered” column shows a sequence of states that the FSM followed during simulation. This is dynamic verification that the sequence is covered. By default these lines are green.

A line that contains an X () in the “Is Covered” column shows a sequence of states that the FSM did not follow, or did not completely follow, during simulation, but Vex static analysis found that the FSM could follow. There is no dynamic verification, so this sequence is not covered. By default these lines are red.

The “Length” column displays the number of states in the sequence. The “Sequence” column displays the sequence of states.

Some sequences might not be displayed because they were filtered out by the filter in the Reachability tab.

At the bottom of the tab are two lines of summary information. The first line illustrates the total number of sequences between the “from” and “to” state that VCS found through static analysis, the total number of sequences that were covered, and the percentage of the covered sequences. The second line is used when you filter sequences in the tab so that it does not show all the sequences between the “from” and “to” state. This second line is also used for coverage information on the sequences that you did not filter out.

Pinched Off Intervening Loops

The Sequences tab omits or “pinches off” intervening loops in the sequence of states that it reports. For example, in the following sequence of states:

A → B → D → B → C → D → A

In this example, there is an intervening loop of B -> D -> B in the sequence. This loop is pinched off and the line in the Sequences is:

A -> B* -> C -> D -> A

The asterisk indicates where the loop was pinched off.

Reordering The Sequences

You can change the order in which cmView displays the sequences in the Sequences tab. You can reorder the display:

- According to whether or not the sequence is covered
- According to the length of the sequence
- Alphanumerically based on the “from” state in the sequence

To reorder the display according to whether or not the sequence is covered, click on the “Is Covered” column title. A blue up arrow  appears in the column title and the sequences are reordered showing the covered sequences first. To reorder the display showing the uncovered sequences first, click on the blue up arrow.

The blue up arrow is replaced by a blue down arrow  and the sequences are reordered showing the uncovered sequences first.

To reorder the display according to the length of the sequence, click on the “Length” column title. A blue up arrow  appears in the column title and the sequences are reordered showing the shortest sequences first. To reorder the display showing the longest sequences first, click on the blue up arrow. The blue up arrow is replaced by a blue down arrow  and the sequences are reordered showing the longest sequences first.

To reorder the display alphanumerically according to the name of the “from” state, click on the “Sequences” column title. A blue up arrow  appears in the column title and the sequences are reordered alphanumerically. To reorder the display in reverse alphanumeric order, click on the blue up arrow. The blue up arrow is replaced by a blue down arrow  and the sequences are reordered in reverse alphanumeric order.

6

Path Coverage

Path coverage determines whether the conditional expressions in `if` and `case` statements were true, and therefore allowed the execution of procedural statements in the initial or always blocks that they control.

You instruct VCS and VCS MX to compile for path coverage with the `-cm path` compile-time option and argument. You specify monitoring for path coverage with the `-cm path` runtime option and argument.

This chapter describes the following:

- “Path Coverage Example”
- “Generating Path Coverage”
- “Path Coverage Reports”

Path Coverage Example

Consider the annotated source code in [Example 6-1](#):

Example 6-1 Annotated Path Coverage Example

```
30 module dev (out,clk,c1,c2,c3,c4,i1,i2);
31 input clk,c1,c2,c3,c4,i1,i2;
32 output out;
33 reg out,c,d,b;
34
35 always @(posedge clk)
36 begin
37     out = 1'b0;
38     if (c1)
39         begin
40             out = i1 && i1;
41             if (c2)
42                 b = i1 || i2;
43         end
44     if (c3)
45         c = ~i1;
46     else
47         c = ~i2;
48     case (c4)
49         1'b0 : d = 1'b0;
50         1'b1 : d = 1'b1;
51     endcase
52 end
53 endmodule
```

In [Example 6-1](#), one of the possible paths through this always block is as follows:

1. `c1` is true in the `if` statement on line 38, enabling the execution of the begin-end block that follows on line 39.
2. `c2` is true in the `if` statement on line 41, enabling the execution of the assignment statement on line 42.

3. c3 is true in the `if` statement on line 44, enabling the execution of the assignment statement on line 45 and preventing the execution of the assignment statement on line 47.
4. c4 is true in the `case` statement beginning on line 48, preventing the execution of the assignment statement on line 49 and enabling the execution of the assignment statement on line 50.

In this path `c1`, `c2`, `c3`, and `c4` were all true. In other possible paths, some or all of these signals are false.

Note:

The graphical user interface (GUI) for cmView does not display path coverage information. You must have cmView write path coverage reports.

VCS and VCS MX do not monitor for path coverage `if` and `case` statements in user-defined tasks and functions and inside loop statements. The following code would have no results in path coverage:

```
module test;

reg r1,r2,r3,r4,r5;
wire w1;
integer int1;

always @ (r2 or r4)
begin
  mtask (r4,r5);
  for(int1=1;int1<10;int1=int1+1)
    if (r2)
      r5=1'bz;
end

task mtask;
  input in1;
  output out1;
```

```
begin
  if (in1)
    out1=1;
  end
  endtask

endmodule
```

VCS and VCS MX do not monitor the `if` statement in the `for` loop statement and the `if` statement in the user-defined task.

Generating Path Coverage

To generate path coverage, perform the following:

1. Compile the design for path coverage. For example:

```
vcs source.v -cm path
```

2. Instruct VCS/ VCS MX to monitor for path coverage during simulation. For example:

```
simv -cm path
```

3. Instruct cmView to write path coverage reports. For example:

```
vcs -cm_pp -cm path
```

Path Coverage Reports

cmView writes its report files in the `./simv.cm/reports` directory. The report files which cmView writes about path coverage are as follows:

`cmView.long_p`

A long report file containing comprehensive information about the path coverage of your design, organized by module instance.

`cmView.long_pd`

Another long report file containing comprehensive information about the path coverage of your design, organized by module definition instead of module instance.

`cmView.hier_p`

Path coverage report where coverage data is presented in subhierarchies in the design. There is a section for each module instance and the path coverage information in these sections is used for the instance and all module instances hierarchically under that instance.

`cmView.mod_p`

Path coverage report where coverage data is used for module instances in the design, not subhierarchies. There is a section for each module instance and the information in these instance sections do not include data for instances hierarchically under the specified instance.

`cmView.mod_pd`

Path coverage report, similar to the `cmView.mod_p` file, where coverage data is used for module definitions instead of module instances.

`cmView.short_p`

A short report file containing only sections for instances in which all paths were not covered. In these sections, cmView only lists the uncovered paths. The report ends with summary information.

`cmView.short_pd`

Another short report file, similar to the `cmView.short_p` file, where coverage data is used for module definitions instead of module instances.

The cmView.long_p File

The following is an example cmView.long_p file. This example file is interrupted in a number of places to explain its contents.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year
```

LONG PATH COVERAGE REPORT

```
////////////////////////////////////////////////////////////////////////  
// MODULE INSTANCE COVERAGE
```

```
// This section contains coverage for each instance of a module
```

```
Test Coverage Result: Total Coverage
```

```
MODULE test
```

```
-----  
//-----
```

```
// Module Coverage Summary
```

```
// No Paths For Summary
```

```
-----  
//-----
```

The report begins with a section on the top-level module. There are no if or case statements in this module so there is no path coverage data to report.

```
MODULE test.d1
```

```
FILE source file path name
```

```
-----
```

EXPRESSION	IF c1	IF c2	IF c3	CASE (c4)	
	-	-	-	none	Covered
	0	-	0	1'b0	Covered

0	-	0	1'b1	Not Covered
0	-	1	none	Not Covered
0	-	1	1'b0	Not Covered
0	-	1	1'b1	Not Covered
1	1	0	none	Not Covered
1	1	0	1'b0	Covered
1	1	0	1'b1	Not Covered
1	1	1	none	Not Covered
1	1	1	1'b0	Covered
1	1	1	1'b1	Covered
1	0	0	none	Not Covered
1	0	0	1'b0	Covered
1	0	0	1'b1	Not Covered
1	0	1	none	Not Covered
1	0	1	1'b0	Not Covered
1	0	1	1'b1	Not Covered

The report then has a section on module instance test.d1. The report has separate sections of coverage data for each module instance.

This section reports on path coverage for an instance of the module definition in [Example 6-1](#). There is one always block in this instance.

We see that the first possible path through the always block is where c1 had a value of 0, it didn't matter what the value of c2 was, c3 had a value of 0, and c4 was neither 1 nor 0. VCS/ VCS MX did execute this instance using this path, so the path is reported as "Covered."

The last possible path is where c1 is 1, c2 is 0, c3 is 1, and c4 is 1. VCS/ VCS MX did not execute the instance using this path so the path is reported as "Not Covered."

```
//-----
//          Module Coverage Summary
//          paths           TOTAL    COVERED    PERCENT
//                         18        6          33.33
//-----
```

There is a summary of the path coverage information for the instance.

```

//*****
//          Total Module Instance Coverage Summary
//          paths      TOTAL    COVERED    PERCENT
//                      18        6        33.33

```

At the end is a summary of path coverage information for the entire design.

Different Format For Long Paths

If there are more than 50 possible paths through an initial or always block, cmView uses a different format to report path coverage. For example:

```

27      begin
28          if (in1 && in2)
29              -1-
30                  t1=in1;
31          if ( t1 <= 10)
32              -2-
33                  case (t1)
34                      -3-
35                          1 : t3 = t2 && t1;
36                          2 : t2 = t1 / 2;
37                          3 : t6 = 1'bz;
38                          4 : t6 = t3 || t4;
39                          5 : t5 = 3 * t2;
40                          6 : t2 = t1 * 2;
41                          7 : t4 = t3;
42                          8 : t3 = t5 / 2;
43                          9 : t2 = 9;
44                          10 : t7 = t4 + t5;
45          endcase
46          else
47              t2 = t1;
48          if (t3 && t4)
49              -4-
50                  if (t4 || t5)
51                      -5-
52                          if (t5 && t6)
53                              -6-
54                                  if (t6 || t7)
55                                      -7-
56                                          t3 = t6;
57          case (t3)
58              -8-
59          1 : begin
60              case (t4)

```

```

      -9-
54      1 : case (t5)
      -10-
55          1 : t6 = 7;
56          2 : t6 = 6;
57          3 : t6 = 5;
58          4 : t6 = 5;
59          5 : t6 = 4;
60          6 : t6 = 3;
61          7 : t6 = 2;
62          8 : t6 = 1;
63          9 : t6 = 0;
64      endcase
65      2 : case (t4)
      -11-
66          1 : t5 = 0;
67          default : t5 = t4;
68      endcase
69      default : t3 = t2;
70      endcase
71      t5=t4;
72  end
73  2 : if (t2 || t3)
      -12-
74      if (t3 && t4)
      -13-
75          if (t4 || t5)
      -14-
76              if (t5 && t6)
      -15-
77                  t1 = 0;
78  default : t1 = 0;

```

First, cmView annotates the always block giving each line its line number from the source file. Then it labels the conditional expressions. For example:

```

28      if (in1 && in2)
      -1-
29          t1=in1;

```

The conditional expression in this `if` statement is labeled 1; it is the first conditional expression in the always block.

Another example:

```

31      case (t1)
      -3-

```

The third conditional expression is in this case statement.

After the annotation and expression labeling, cmView reports only the covered paths. For example:

```
COVERED PATH (1) ->! (2) ->! (4) ->(8, default)
COVERED PATH ! (1) ->(2) ->! (4) ->(8, default)
COVERED PATH ! (1) ->! (2) ->! (4) ->(8, default)
```

In the first covered path:

1. (1) Indicates that the first conditional expression, the one in the first `if` statement (on line 28), was true, so VCS/ VCS MX executed the assignment statement it controls.
2. ! (2) Indicates that the second conditional expression, the one in the second if statement (on line 30), was false, so VCS/ VCS MX did not execute the statements it controls, including the `case` statement that has the third conditional expression. There is no path to this conditional expression in this simulation.
3. ! (4) Indicates that the fourth conditional expression, in the third `if` statement (on line 45), was false.
4. (8, default) Indicates that the conditional expression in the second `case` statement (on line 51), was true, and VCS/ VCS MX executed the default case item.

The cmView.hier_p File

The following is an example of a cmView.hier_p file. This file contains summary information in terms of subhierarchies. Because module instance test.d1 is hierarchically under top-level module test, the data from test.d1 is included in the information for top-level module test.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE HIERARCHICAL INSTANCE COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// instance of a module. The statistics take into account all sub-hierarchies  
//*****  
// MODULE HIERARCHICAL INSTANCE COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// instance of a module. The statistics take into account all sub-hierarchies  
// under the instance.  
  
Test Coverage Result: Total Coverage  


| Module Name | Path<br>(%) |
|-------------|-------------|
| test        | 33.33       |
| test.d1     | 33.33       |

  
//*****  
// Total Module Instance Coverage Summary  


|       | TOTAL | COVERED | PERCENT |
|-------|-------|---------|---------|
| paths | 18    | 6       | 33.33   |


```

The cmView.mod_p File

The following is an example of a cmView.mod_p file. This file contains summary information for module instances. Coverage data for an instance is for just the instance and not for the instances hierarchically under the instance.

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number  
// User: Intelligent User  
// Date: Day Month date hour:minute:second year  
  
//*****  
// MODULE INSTANCE COVERAGE SUMMARY  
  
// This section summarizes coverage by providing statistics for each  
// instance of a module. The statistics do not take into account any  
// sub-hierarchy under the instance.  
  
Test Coverage Result: Total Coverage  


| Module Name | Path (%) |
|-------------|----------|
| test        | --       |
| test.d1     | 33.33    |

  
//*****  
// Total Module Instance Coverage Summary  


| paths | TOTAL | COVERED | PERCENT |
|-------|-------|---------|---------|
|       | 18    | 6       | 33.33   |


```

The cmView.mod_pd File

The cmView.mod_pd file is similar to the cmView.mod_p file except that the coverage information is organized by module definition instead of by module instance. The following is the corresponding cmView.mod_pd file to the cmView.mod_p file above:

```

// Synopsys, Inc.
//
// Generated by: cmView version_number
// User: Intelligent User
// Date: Day Month date hour:minute:second year

//*****MODULE DEFINITION COVERAGE SUMMARY

// This section summarizes coverage by providing statistics for each
// module definition. The coverage is cumulative over all the instances
// of the module

Test Coverage Result: Total Coverage

Module Name          Path
                   (%)  

test                --
dev                 33.33

//*****Total Module Definition Coverage Summary

//          TOTAL    COVERED    PERCENT
// paths      18        6         33.33

```

The cmView.short_p File

The cmView.short_p file contains information for all module instances which do not have 100% coverage, and summary information about the entire design.

```

// Synopsys, Inc.
//
// Generated by: cmView version_number
// User: Intelligent User
// Date: Day Month date hour:minute:second year

```

SHORT PATH COVERAGE REPORT

```

//*****
// MODULE INSTANCE COVERAGE

// This section contains coverage for each instance of a module

Test Coverage Result: Total Coverage

MODULE test.d1

EXPRESSION    IF c1    IF c2    IF c3    CASE (c4)
-----  -----  -----  -----
      0      -      0      1'b1  Not Covered
      0      -      1      none  Not Covered
      0      -      1      1'b0  Not Covered
      0      -      1      1'b1  Not Covered
      1      1      0      none  Not Covered
      1      1      0      1'b1  Not Covered
      1      1      1      none  Not Covered
      1      0      0      none  Not Covered
      1      0      0      1'b1  Not Covered
      1      0      1      none  Not Covered
      1      0      1      1'b0  Not Covered
      1      0      1      1'b1  Not Covered

```

```

//-----
//*****
//          Total Module Instance Coverage Summary
//          paths           TOTAL     COVERED      PERCENT
//                         18         6        33.33

```

The cmView.short_pd File

The cmView.short_pd file is similar to the cmView.short_p file except that the information in it is organized by module definition instead of by module instance.

```

// Synopsys, Inc.
//
// Generated by: cmView version_number
// User: Intelligent User
// Date: Day Month date hour:minute:second year

```

SHORT PATH COVERAGE REPORT

```

//*****
//          MODULE DEFINITION COVERAGE

// This section contains coverage for module definitions.
// The coverage is cumulative over all the instances of the module

```

Test Coverage Result: Total Coverage

MODULE dev

EXPRESSION	IF c1	IF c2	IF c3	CASE (c4)
	-	-	-	-
0	-	0	1'b1	Not Covered
0	-	1	none	Not Covered
0	-	1	1'b0	Not Covered
0	-	1	1'b1	Not Covered
1	1	0	none	Not Covered
1	1	0	1'b1	Not Covered
1	1	1	none	Not Covered
1	0	0	none	Not Covered
1	0	0	1'b1	Not Covered
1	0	1	none	Not Covered
1	0	1	1'b0	Not Covered
1	0	1	1'b1	Not Covered

EXPRESSION	IF c1	IF c2	IF c3	CASE (c4)	
0	-	0	1'b1	Not Covered	
0	-	1	none	Not Covered	
0	-	1	1'b0	Not Covered	
0	-	1	1'b1	Not Covered	
1	1	0	none	Not Covered	
1	1	0	1'b1	Not Covered	
1	1	1	none	Not Covered	
1	0	0	none	Not Covered	
1	0	0	1'b1	Not Covered	
1	0	1	none	Not Covered	
1	0	1	1'b0	Not Covered	
1	0	1	1'b1	Not Covered	

```
//-----
```

```

//*****
//          Total Module Definition Coverage Summary

//          TOTAL      COVERED      PERCENT
//          paths      18          6          33.33

```

Path Coverage

6-378

7

Branch Coverage

Branch coverage analyzes how `if` and `case` statements and the ternary operator (`? :`) establish branches of execution in your Verilog design. It shows you vectors of signal or expression values that enable or prevent simulation events.

You instruct VCS and VCS MX to compile for branch coverage with the `-cm` branch compile-time option and argument. You specify monitoring for branch coverage with the `-cm` branch runtime option and argument.

Note:

Branch coverage is implemented for Verilog simulation only.

This chapter describes the following:

- “[Branch Coverage Example](#)”
- “[Enabling Branch Coverage](#)”

- “Branch Coverage Reports”
 - “Using Pragmas to Limit Branch Coverage”
 - “Viewing Branch Coverage with the Coverage Metrics GUI”
 - “Branch Coverage Limitations”
-

Branch Coverage Example

Branch coverage analyzes how `if` and `case` statements and the ternary operator (`? :`) establish branches of execution in your Verilog design. It shows you vectors of signal or expression values that enable or prevent simulation events.

Consider the code in [Example 7-1](#):

Example 7-1 Branch Coverage Code Example

```
case (r1)
    1'b1      : if (r2 && r3)
                  r4 = (r5 && r6) ? 1'b0 : 1'b1;
    1'b0      : if (r7 && r8)
                  r9 = (r10 && r11) ? 1'b0 : 1'b1;
    default   : $display("no op");
endcase
```

In this block of code there are procedural assignment statements where the value assigned is controlled by the ternary operator (`? :`). These in turn are controlled by `if` statements, and the `if` statements are controlled by a `case` statement.

In this block of code, the possible vectors of signal or expression values that result in simulation events or prevent simulation events are as follows:

r1	(r2 && r3)	(r5 && r6)	(r7 && r8)	(r10 && r11)
1	1	1	-	-
1	1	0	-	-
1	0	-	-	-
0	-	-	1	1
0	-	-	1	0
0	-	-	0	-
default	-	-	-	-

For example, in the first vector, r1 is 1'b1 and the expression (r2 && r3) is true, therefore, the value of r4 depends on the value of (r5 && r6). The values of (r7 && r8) and (r10 && r11) do not matter.

Another example, r1 is 1'b1, but the expression (r2 && r3) is false, and the values of the other expressions do not matter. Therefore, nothing happens.

Branch coverage shows you these vectors and then tells you whether these vectors ever occurred during simulation, in other words, whether they were covered.

By default, VCS and VCS MX do not monitor for branch coverage if and case statements and uses of the ternary operator (? :) if they are in user-defined tasks or functions or in code that executes as a result of a for loop. You can, however, enable branch coverage in this code (see “[For Loops and User-defined Tasks and Functions](#)”).

Branch Coverage With Unknown and High Impedance Values

If the conditional expression in an if statement evaluates to X or Z, branch coverage treats this as a false value and reports that the 0 value for the expression is covered.

If the case expression in a `case` statement evaluates to X or Z, unless there is a case item with the case item expression that evaluates to X or Z, VCS and VCS MX execute the default case item. When this occurs, branch coverage reports that the vector for the default case item is covered. If there is no default case item, branch coverage reports a vector for the missing default case item and reports it as covered.

When the conditional expression for a ternary operator evaluates to X or Z, the vector for the expression is not covered.

Enabling Branch Coverage

To enable branch coverage, perform the following:

1. Compile the design for branch coverage. For example:

```
vcs design.v -cm branch
```

2. Instruct VCS/VCS MX to monitor for branch coverage during simulation. For example:

```
simv -cm branch
```

3. Instruct cmView to write branch coverage reports. For example:

```
vcs -cm_pp -cm branch
```

For Loops and User-defined Tasks and Functions

If the `if` and `case` statements and the ternary operator (`? :`) are in user-defined tasks or functions, or in code that executes as a result of a `for` loop, by default VCS and VCS MX do not monitor them for branch coverage. This is also true for condition coverage.

If you compile your Verilog design for condition coverage and enter the `-cm_cond` compile-time option with the `for` and `tf` keyword arguments, which tell VCS and VCS MX to compile the design for condition coverage in `for` loops and user-defined tasks and functions, VCS or VCS MX will also compile the design for branch coverage in `for` loops and user-defined tasks and functions. For example:

```
vcs design.v -cm cond+branch -cm_cond tf -cm_cond for  
simv -cm cond+branch  
vcs -cm_pp -cm cond+branch
```

Branch Coverage Reports

`cmView` writes its report files in the following `./simv.cm/reports` directory:

`cmView.long_b`

A long detailed report file, organized by module instance, containing comprehensive information about the branch coverage of your design.

`cmView.long_bd`

A long detailed report file, similar to the `cmView.long_b` file, but organized by module definition instead of module instance.

`cmView.hier_b`

A branch coverage report where coverage data is organized by subhierarchies in the design. There is a section for each module instance and the information in these sections is used for the instance and for all module instances hierarchically under that instance.

`cmView.mod_b`

A branch coverage report where coverage data is used for module instances in the design, not subhierarchies. There is a section for each module instance and the information in these instance sections do not include data for instances hierarchically under the specified instance.

`cmView.mod_bd`

A branch coverage report, similar to the `cmView.mod_b` file, but organized by module definition instead of module instance.

`cmView.short_b`

A short report file containing sections for instances in which there is not 100% branch coverage. In these sections, cmView lists only signals that were not covered. The report ends with summary information.

`cmView.short_bd`

A short report file, similar to the `cmView.short_b` file, but organized by module definition instead of by module instance.

Example Branch Coverage Report

The following is an example of the `cmView.long_b` (b for branch) that cmView writes for branch coverage. It contains example branch coverage results for the code shown in [Example 7-1](#).

The example only has a top-level module to provide stimulus and an instantiated module that contains `if` and `case` statements and assignments with expressions containing the ternary operator.

The following is an example `cmView.long_b` file. This example file is interrupted in a number of places to explain its contents.

The report begins with introductory information:

```
// Synopsys, Inc.  
//  
// Generated by: cmView version_number_of_VCS  
// User: your_user_name  
// Date: date_cmView_writes_the_report
```

LONG BRANCH COVERAGE REPORT

The report includes the version number, the user who generated the report, and the date. The report labels itself as the long branch coverage report.

The next part illustrates the information for the top-level module.

```
////////////////////////////////////////////////////////////////////////  
// MODULE INSTANCE COVERAGE  
  
// This section contains coverage for each instance of a module  
  
Test Coverage Result: Total Coverage  
  
MODULE top-level_module_name  
-----  
//  
// Module Coverage Summary  
// No Branches For Summary
```

There are no `if` or `case` statements in the top-level module or uses of the ternary operator, therefore, there are no branch coverage results for the top-level module.

In this example, there is only one module instance. In the long branch coverage report there is information for each module instance in the design.

```
MODULE module_instance_hierarchical_name  
FILE Verilog_source_file_path_name.v  
-----  
45 case (r1)
```

```

46      -1-
        1'b1    : if (r2 && r3)
47          -2-
                  r4 = (r5 && r6) ? 1'b0 : 1'b1;
          -3-
48      1'b0    : if (r7 && r8)
49          -4-
                  r9 = (r10 && r11) ? 1'b0 : 1'b1;
          -5-
50      default : $display("no op");

```

This section on the instance begins with the hierarchical name of the instance and the path name for the source file.

Coverage metrics found in the instance are:

1. A `case` statement with a `case` expression (`r1`) that it labels -1-.
2. An `if` statement nested inside the `case` statement. This `if` statement has a conditional expression (`r2 && r3`) that it labels -2-.
3. An assignment statement nested inside the `if` statement. The assignment is an expression that contains the ternary operator with a conditional expression (`r5 && r6`) that it labels -3-.
4. Another `if` statement nested in the `case` statement. This `if` statement has a conditional expression (`r7 && r8`) that it labels -4-.
5. An assignment statement nested inside the second `if` statement. The assignment is an expression that contains the ternary operator with a conditional expression (`r10 && r11`) that it labels -5-.

The following illustrates the vectors of signal values that enable or prevent the assignment statements in this code and whether these vectors occurred, or were covered, during simulation.

BRANCH	-1-	-2-	-3-	-4-	-5-	
1'b1	1	1	-	-	-	Covered
1'b1	1	0	-	-	-	Covered
1'b1	0	-	-	-	-	Covered
1'b0	-	-	1	1	-	Covered
1'b0	-	-	1	0	-	Not Covered
1'b0	-	-	0	-	-	Covered
default	-	-	-	-	-	Not Covered

//*****

The first vector is:

BRANCH	-1-	-2-	-3-	-4-	-5-	
1'b1		1	1	-	-	Covered

In this vector, when all of the following things occur, signal `r4` is assigned `1'b0`:

1. The case expression (`r1`) (labeled `-1-`) is the `1'b1` choice case item.
2. The conditional expression (`r2 && r3`) (labeled `-2-`) in the first `if` statement is true.
3. The conditional expression (`r5 && r6`) (labeled `-3-`) for the ternary operator in the assignment statement is true.

All these things occurred, therefore, this vector is covered. In the first vector, the other conditional expressions do not matter.

The second vector is:

BRANCH	-1-	-2-	-3-	-4-	-5-	
1'b1	1	0	-	-	-	Covered

In this vector, when all of the following things occur, signal `r4` is assigned `1'b1`:

1. The case expression (`r1`) (labeled `-1-`) is the `1'b1` choice case item.

2. The conditional expression ($r2 \&& r3$) (labeled -2-) in the first if statement is true.
3. The conditional expression ($r5 \&& r6$) (labeled -3-) for the ternary operator in the assignment statement is false.

All of these things occurred, therefore, this vector was covered. In the second vector, the other conditional expressions do not matter.

The third vector is:

BRANCH	-1-	-2-	-3-	-4-	-5-	
1'b1	0	-	-	-	-	Covered

This vector is covered when all of the following things occur:

1. The case expression ($r1$) (labeled -1-) is the 1'b1 choice case item.
2. The conditional expression ($r2 \&& r3$) (labeled -2-) in the first if statement is false.

Both of these things occurred, therefore, this vector was covered.

Because the conditional expression is false, VCS and VCS MX do not execute the assignment statement, therefore, it does not matter what the value is for the conditional expression ($r5 \&& r6$) for the ternary operator.

The fourth vector is:

BRANCH	-1-	-2-	-3-	-4-	-5-	
1'b0	-	-	1	1	-	Covered

In this vector, when all of the following things occur, signal `r9` is assigned `1'b0`:

1. The case expression (`r1`) (labeled `-1-`) is the `1'b0` choice case item.
2. The conditional expression (`r7 && r8`) (labeled `-4-`) in the first `if` statement is true.
3. The conditional expression (`r10 && r11`) (labeled `-5-`) for the ternary operator in the assignment statement is true.

All these things occurred, therefore, this vector was covered. In this vector, the other conditional expressions do not matter.

The fifth vector is:

BRANCH	-1-	-2-	-3-	-4-	-5-	
<code>1'b0</code>	-	-	1	0		Not Covered

In this vector, when all of the following things occur, signal `r9` is assigned `1'b1`:

1. The case expression (`r1`) (labeled `-1-`) is the `1'b0` choice case item.
2. The conditional expression (`r7 && r8`) (labeled `-4-`) in the first `if` statement is true.
3. The conditional expression (`r10 && r11`) (labeled `-5-`) for the ternary operator in the assignment statement is false.

All of these things did not occur, therefore, this vector was not covered. In this vector, the other conditional expressions do not matter.

The sixth vector is:

BRANCH	-1-	-2-	-3-	-4-	-5-	
1'b0	-	-	0	-		Covered

In this vector, when all of the following things occur, the vector is covered:

1. The case expression (`r1`) (labeled -1-) is the `1'b0` choice case item.
2. The conditional expression (`r7 && r8`) (labeled -4-) in the first `if` statement is false.

Both of these things occurred, therefore, this vector was covered.

Because the conditional expression is false, VCS/VCS MX does not execute the assignment statement, therefore, it does not matter what the value is for the conditional expression (`r10 && r11`) for the ternary operator.

The last vector is used for execution of the default case item for the `case` statement.

BRANCH	-1-	-2-	-3-	-4-	-5-	
default	-	-	-	-		Not Covered

This did not occur, therefore, it was not covered.

The section on the instance concludes with summary information.

```
//-----
//          Module Coverage Summary
//      branches      TOTAL    COVERED     PERCENT
//                  7        5         71.43
//-----
```

The reports concludes with summary information about the entire design.

```
//*****  
//      Total Module Instance Coverage Summary  
//  
//      branches      TOTAL    COVERED    PERCENT  
//      7            5          71.43  
//-----
```

Using Pragmas to Limit Branch Coverage

The `//VCS coverage exclude_file` and `//VCS coverage exclude_module` pragmas exclude a source file or module definition from branch coverage.

Additionally, you can use the `//VCS coverage off` and `//VCS coverage on` pragmas to exclude certain code from branch coverage.

When you use `if` statements (as in [Example 7-2](#)), cmView reports a column for each conditional expression (as in [Example 7-3](#)).

Example 7-2 If Statements in a Design

```
always @ (r1 or r2 or r3)
begin
  if (r1)
    begin
      $display("r1 is true");
      r4=r1;
    end
  else
    begin
      $display("r1 not true");
      if (r2)
        begin
```

```

        $display("r2 is true");
        r5=r2;
    end
else
begin
    $display("r2 not true");
    if (r3)
begin
    $display("r3 is true");
    r6=r3;
end
else
    $display("r3 not true");
end
end
$display("no op");
end

```

Example 7-3 Branch Coverage Report

```

23      if (r1)
24          -1-
25          begin
26              $display("r1 is true");
27              r4=r1;
28          end
29      else
30          begin
31              $display("r1 not true");
32              if (r2)
33                  -2-
34                  begin
35                      $display("r2 is true");
36                      r5=r2;
37                  end
38              else
39                  begin
40                      $display("r2 not true");
41                      if (r3)
42                          -3-
43                      begin

```

```

41                      $display("r3 is true");
42                      r6=r3;
43                      end
44      else
45                      $display("r3 not true");
46      end
47  end

```

BRANCH	-1-	-2-	-3-	
1	-	-		Covered
0	1	-		Covered
0	0	1		Covered
0	0	0		Covered

This examples includes branches starting at `r1` and ending at `r1`, `r2`, or `r3`.

You can use these pragmas to exclude one of the conditional expressions for a ternary operator as shown in [Example 7-4](#):

Example 7-4 Excluding a Conditional Expression for a Ternary Operator

```

always @ (r1 or r2 or r3)
begin
if (r1)
begin
    $display("r1 is true");
    r4=r1;
end
//VCS coverage off
else
begin
    $display("r1 not true");
    if (r2)
begin
    $display("r2 is true");
    r5=r2;
end
else
begin

```

```

    $display("r2 not true");
//VCS coverage on
    if (r3)
        begin
            $display("r3 is true");
            r6=r3;
        end
    else
        $display("r3 not true");
end
$display("no op");
end

```

In this case, cmView reports on only the branch starting and ending at `r1`. There is no new branch starting at `r3` because the `if` statement for `r2` is excluded from coverage by the `//vcs coverage off` pragma.

You can also use pragmas (see [Example 7-5](#)) to exclude code that does not affect branch coverage, that is, the branches that VCS or VCS MX identify as unchanged. For example:

Example 7-5 Using Pragmas Without Affecting Branch Coverage

```

always @ (r1 or r2 or r3)
begin
if (r1)
begin
    $display("r1 is true");
    r4=r1;
end
else
begin
//VCS coverage off
    $display("r1 not true");
//VCS coverage on
    if (r2)
begin
    $display("r2 is true");

```

```

        r5=r2;
    end
else
begin
    $display("r2 not true");
    if (r3)
begin
    $display("r3 is true");
    r6=r3;
end
else
    $display("r3 not true");
end
end
$display("no op");
end

```

When you use case statements as shown in [Example 7-6](#), cmView includes a column for each case expression (see [Example 7-7](#)):

Example 7-6 Using case Statements

```

always @ (r1 or r2 or r3)
case (r1)
    1      : case (r2)
        1      : r4=1;
        0      : r4=0;
        default : $display("r4 not assigned");
    endcase
    0      : case (r3)
        1      : r5=0;
        0      : r5=1;
        default : $display("r5 not assigned");
    endcase
    default : $display("no op");
endcase

```

Example 7-7 cmView Output for case Statement

```

24      case (r1)
-1-

```

```

25      1      : case (r2)
26          -2-
27          1      : r4=1;
28          0      : r4=0;
29          default : $display("r4 not assigned");
30      endcase
31      0      : case (r3)
32          -3-
33          1      : r5=0;
34          0      : r5=1;
35          default : $display("r5 not assigned");
36      endcase
37      default : $display("no op");

```

BRANCH	-1-	-2-	-3-	
	1	1	-	Covered
	1	0	-	Covered
	1	default	-	Not Covered
	0	-	1	Covered
	0	-	0	Covered
	0	-	default	Not Covered
	default	-	-	Not Covered

However, when you use pragmas to exclude a case statement, as shown in [Example 7-8](#), VCS or VCS MX identify no branches for that case expression (see [Example 7-9](#)):

Example 7-8 Using Pragmas to Exclude case Statements

```

case (r1)
    1      : case (r2)
        1      : r4=1;
        0      : r4=0;
        default : $display("r4 not assigned");
    endcase
//VCS coverage off
    0      : case (r3)
        1      : r5=0;
        0      : r5=1;
        default : $display("r5 not assigned");
    endcase
//VCS coverage on
    default : $display("no op");

```

```
endcase
```

Example 7-9 cmView Output When case Statements are Excluded With Pragmas

```
24      case (r1)
           -1-
25          1       : case (r2)
           -2-
26            1       : r4=1;
27            0       : r4=0;
28        default : $display("r4 not assigned");
29      endcase
30 //VCS coverage off
31          0       : case (r3)
32            1       : r5=0;
33            0       : r5=1;
34        default : $display("r5 not assigned");
35      endcase
36 //VCS coverage on
37    default  : $display("no op");
```

BRANCH	-1-	-2-	
	1	1	Covered
	1	0	Covered
	1	default	Not Covered
default		-	Not Covered

When you use the ternary operator as shown in [Example 7-10](#), cmView reports a column for each conditional expression (see [Example 7-11](#)):

Example 7-10 Using the Ternary Operator

```
assign w1 = (r1==1) ?
           ((r2==1) ?
             ((r4==1) ? r6 : r7)
             : r5)
           : r3;
```

Example 7-11 cmView Output for a Ternary Operator

```
4      assign w1 = (r1==1) ?
           -1-
5                  ((r2==1) ?
           -2-
6                  ((r4==1) ? r6 : r7)
           -3-
7                  : r5)
8                  : r3;
9
10     initial
11     begin
```

BRANCH	-1-	-2-	-3-	
1	1	1	1	Covered
1	1	1	0	Covered
1	0	1	-	Covered
0	-	-	-	Covered

If you use these pragmas to exclude one of the conditional expressions for a ternary operator, as in [Example 7-12](#), cmView still reports the column for the conditional expression, as in [Example 7-13](#):

Example 7-12 Excluding a Conditional Operator for the Ternary Operator

```
assign w1 = (r1==1) ?
           //VCS coverage off
           ((r2==1) ?
           //VCS coverage on
           ((r4==1) ? r6 : r7)
           : r5)
           : r3;
```

Example 7-13 cmView Output When a Ternary Operator is Excluded with Pragmas

```
4      assign w1 = (r1==1) ?
```

Branch Coverage

-1-

```
5          //VCS coverage off
6          ((r2==1) ?
7              -2-
8          //VCS coverage on
9          ((r4==1) ? r6 : r7)
10         -3-
11         : r5)
12         : r3;
13     begin
```

BRANCH	-1-	-2-	-3-	
	1	1	1	Covered
	1	1	0	Covered
	1	0	-	Covered
	0	-	-	Covered

However, if you use these pragmas to exclude all the ternary operators and conditional expressions, as shown in [Example 7-14](#), VCS/VCS MX does not compile for or monitor, and cmView does not report, these ternary operators and their conditional expression for branch coverage:

Example 7-14 Excluding All Conditional Operators for the Ternary Operator

```
//VCS coverage off
assign w1 = (r1==1) ?
    ((r2==1) ?
        ((r4==1) ? r6 : r7)
        : r5)
    : r3;
//VCS coverage on
```

If you enter any of these pragmas in your source code, and at later time, want VCS or VCS MX to ignore these pragmas, enter the `-cm_ignorepragmas` compile-time option.

Viewing Branch Coverage with the Coverage Metrics GUI

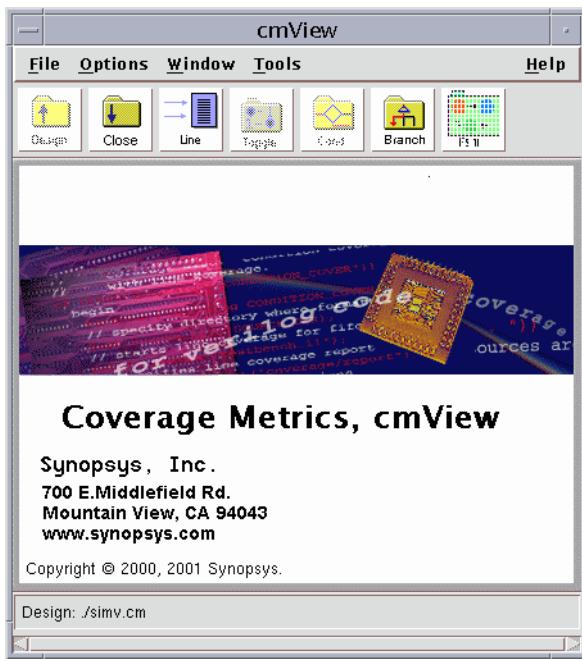
cmView can start the graphical user interface (GUI) for VCS or VCS MX coverage metrics. You use it to see graphical representations of coverage information recorded by VCS or VCS MX. To start the GUI, perform the following:

1. Enter the following command line:

```
vcs -cm_pp -cm branch
```

The main window for cmView appears (see [Figure 7-1](#)):

Figure 7-1 The cmView Main Window



The command line also instructs cmView to perform the following:

- Read the design file:
`./simv.cm/db/verilog/cm.decl_info`
This file provides cmView with information about the design and its hierarchy.
- Read the intermediate data files for branch coverage in the
`./simv.cm/coverage/verilog` directory:

Note:

If at compile-time you specified a different coverage metrics database with the `-cm_dir` compile-time option, also include the option on the cmView (or vcs `-cm_pp gui`) command line.

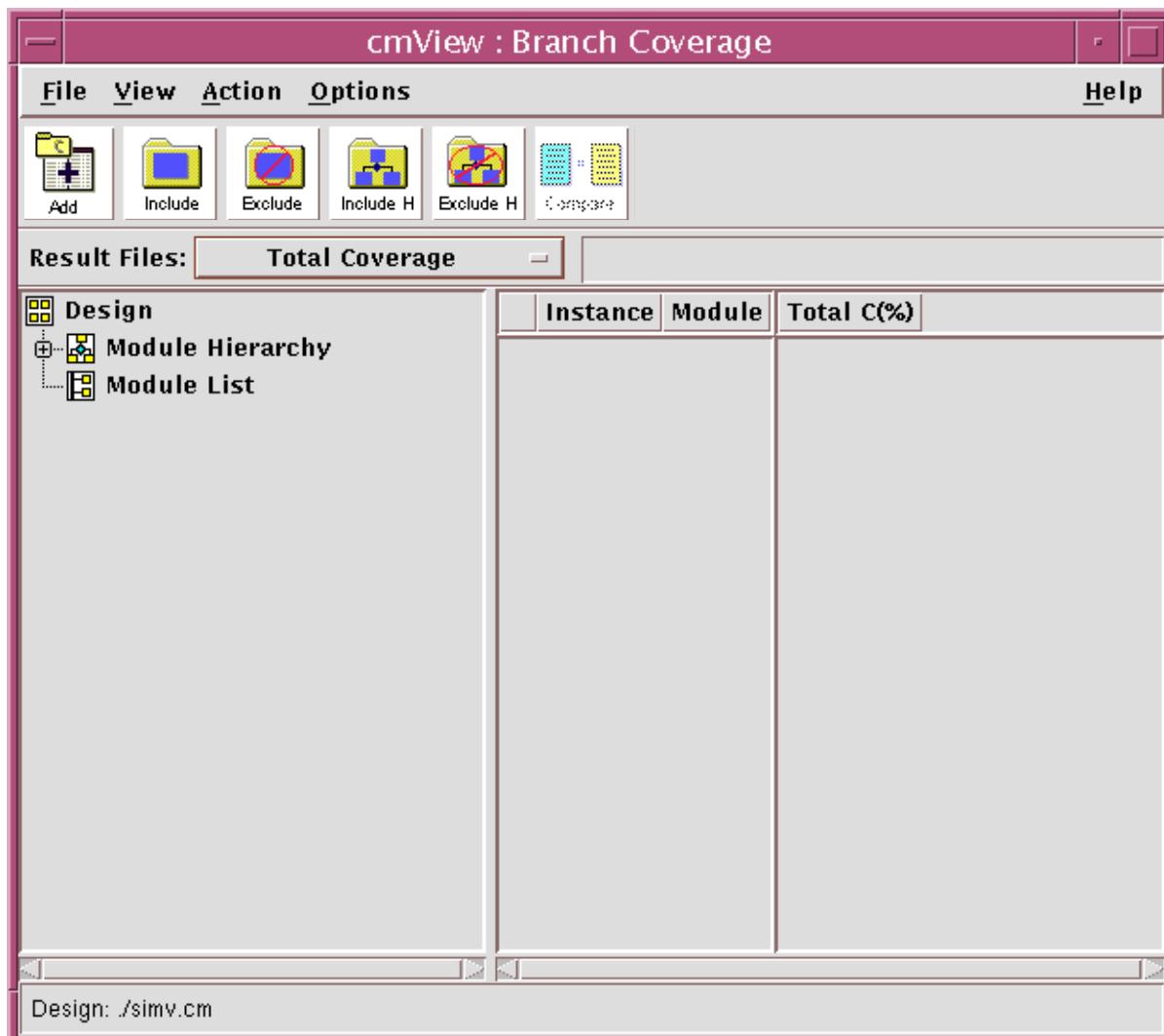
2. Click the branch Coverage toolbar button (see [Figure 7-2](#)):

Figure 7-2 Branch Coverage Toolbar Button



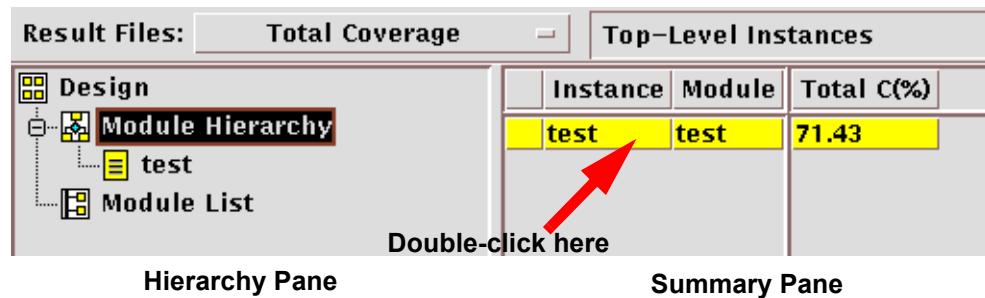
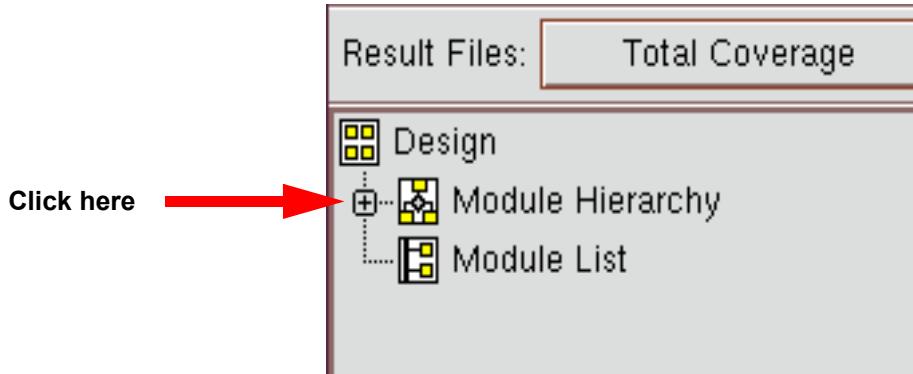
The Branch Coverage window appears (see [Figure 7-3](#)):

Figure 7-3 Branch Coverage Window



3. By default, cmView displays coverage summary information on a per module instance basis, therefore, the FSM Coverage window needs a way to display the design hierarchy so that you can select module instances to examine their FSM coverage. Click the plus symbol next to the icon for the Module Hierarchy to display the top-level modules (in this case one) in the design (see [Figure 7-4](#)):

Figure 7-4 Module Hierarchy Plus Symbol and Top-level Modules



You see that there is 71.43% coverage for the instance.

4. Double-click on the instance in the Summary Pane. This opens the Branch Summary window for the instance, as shown in [Figure 7-5](#).

Figure 7-5 Branch Coverage Instance Summary Window

The screenshot shows the cmView software interface for branch coverage analysis. The title bar reads "cmView – Branch: Instance test". The menu bar includes "File", "View", "Options", "Exclude", and "Help". The toolbar contains icons for "Top of File", "Prev Line", "Next Line", "End of File", "Go Parent", and "Go Child". The "Module Files" dropdown is set to "exp2.v". The main window displays the following Verilog source code:

```

1 module test;
2 reg r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11;
3 initial
4 begin
5   r1=0;
6   #5 r2=0;
7   #5 r3=0;
8   #5 r4=0;
9   #5 r5=0;
10  #5 r6=0;
11  #5 r7=0;
12  #5 r8=0;
13  #5 r9=0;
14  #5 r10=0;
15  #5 r11=0;
16  #5 r1=1;
17  #5 r2=1;
18  #5 r3=1;
19  #5 r4=1;

```

Below the source code are tabs for "Statistics", "Information", "Tests", "Go To", and "Branch". The "Branch" tab is selected, showing the following table:

test	TOTAL	COVERED	PERCENT
Branch	7	5	71.43

The status bar at the bottom shows the path: "/remote/misc2/pmcgee/projects/CovMet/BRANCH/exp2.v".

5. Scroll down to a red line that indicates a start of one or more branches. After the line number, it has the symbol ==>B (indicating a place where branches start). Click on the red line.

Instead of scrolling down, you can use the Go To tab to list all the lines that start branches and highlight them in red. Click in a line in the Go To tab, to instruct cmView to scroll down to that line in the Source window. Then click on the red line in the Source window to put information about the branches starting from the line, into the Branch tab. When you click on the red line, you will notice that the line changes its color to magenta.

6. Look in the Branch tab.

The Branch tab now contains columns and vectors that show branch coverage similar to the branch coverage reports.

Branches that are not covered are highlighted in red and the ==> symbol is positioned to the left of the vector.

[Figure 7-6](#) illustrates the source window with a magenta line and coverage information in the Branch tab.

Figure 7-6 Branch Tab

cmView – Branch: Instance test

File View Options Exclude Help

Module Files: exp2.v

Top of File Prev Line Next Line End of File Ok Parent

```

36      #5 r10=0;
37      #5 r11=0;
38      #5 $finish;
39      end
40
41      always @ (r1 or r2 or r3 or r4 or r5 or r6 or r7 or r8 or r9 or r10 or r11)
42      ==>B case (r1)
43          1'b1 : if (r2 && r3)
44              r4 = (r5 && r6) ? 1'b0 : 1'b1;
45          else
46              r4 = 1'bz;
47          1'b0 : if (r7 && r8)
48              r9 = (r10 && r11) ? 1'b0 : 1'b1;
49          else
50              r9 = 1'bz;
51      default : $display("\n\nNO OP\n\n");
52      endcase
53      endmodule

```

Statistics Information Tests Go To Branch

BRANCH	-1-	-2-	-3-	-4-	-5-	
1'b1	1	1	-	-	-	Covered
1'b1	1	0	-	-	-	Covered
1'b1	0	-	-	-	-	Covered
1'b0	-	-	1	1	1	Covered
==>	1'b0	-	-	1	0	Not Covered
==>	1'b0	-	-	0	-	Covered
==>	default	-	-	-	-	Not Covered

Line	Expression
-1- 42	r1
-2- 43	(r2 && r3)
-3- 44	(r5 && r6)
-4- 47	(r7 && r8)
-5- 48	(r10 && r11)

/remote/misc2/pmcgee/projects/CovMet/BRANCH/exp2.v

Figure 7-7 is a closer look at the Branch tab.

Figure 7-7 Enhanced View of the Branch Tab

BRANCH	-1-	-2-	-3-	-4-	-5-	
	1'b1	1	1	-	-	Covered
	1'b1	1	0	-	-	Covered
	1'b1	0	-	-	-	Covered
	1'b0	-	-	1	1	Covered
=>	1'b0	-	-	1	0	Not Covered
	1'b0	-	-	0	-	Covered
=>	default	-	-	-	-	Not Covered

//*****

Line	Expression
-1- 42	r1
-2- 43	(r2 && r3)
-3- 44	(r5 && r6)
-4- 47	(r7 && r8)
-5- 48	(r10 && r11)

This Branch tab shows the branches for the following block of code:

```
case (r1)
1'b1 : if (r2 && r3)
            r4 = (r5 && r6) ? 1'b0 : 1'b1;
        else
            r4 = 1'bz;
1'b0 : if (r7 && r8)
            r9 = (r10 && r11) ? 1'b0 : 1'b1;
        else
            r9 = 1'bz;
default : $display("\n\nNO OP\n\n");
endcase
```

The branches start with the `r1` expression in line 42 (labeled expression -1-), which controls what occurs in the case statement, so the `r1` expression is column 1.

- If `r1` is true, cmView looks at `(r2 && r3)` on line 43, labeled expression -2-.
- If `(r2 && r3)` is true, cmView looks at `(r5 && r6)` on line 44, labeled expression -3-.
- If `r1` is false, cmView looks at `(r7 && nr8)` on line 47, labeled expression -4-.
- If `(r7 && r8)` is true, cmView looks at `(r10 && r11)` on line 48, labeled expression -5-.

The Branch tab shows a column for each expression. In each column, 1 means covered, 0 not covered, and - indicates that the value does not matter.

Looking horizontally there are rows or vectors of values for the possible branches. Two branches are in red, which indicates that they are not covered.

- The first branch is used when `r1` is 0. When this occurs, only the values of `(r7 && nr8)` and `(r10 && r11)` matter.
 - In the case statement, the second branch is the default case.
7. Click on the first branch that was not covered to display the explanation for this lack of coverage in the Source window. [Figure 7-8](#) shows the changed Source window.

Figure 7-8 Highlighted Lines in the Source Window

```
37      #5 r11=0;
38      #5 $finish;
39      end
40
41      always @ (r1 or r2 or r3 or r4 or r5 or r6 or r7 or r8 or r9 or r10 or r11)
42 ==>B case (r1)
43      1'b1 : if (r2 && r3)
44          r4 = (r5 && r6) ? 1'b0 : 1'b1;
45      else
46          r4 = 1'bz;
47      1'b0 : if (r7 && r8)
48          r9 = (r10 && r11) ? 1'b0 : 1'b1;
49      else
50          r9 = 1'bz;
51      default : $display("\n\nNO OP\n\n");
52      endcase
53      endmodule
```

Lines with true conditional expressions are highlighted in green.
Lines with false conditional expressions are highlighted in yellow
as are MISSING_DEFAULT lines in the source window.

In [Example 7-8](#), the uncovered branch is controlled by line 42,
which was already highlighted when you clicked on it. It is also
controlled by line 47 and 48.

$(r7 \&& r8)$ is true. It is the conditional expression in line 47,
therefore, line 47 is highlighted in green.

$(r9 \&& r10)$ is false. It is the conditional expression in line 48,
therefore, line 48 is highlighted in yellow.

Excluding Branches from Coverage Calculation

To exclude a branch from coverage:

1. Click the ==> symbol to the left of the branch in the Branch tab. This changes the symbol to M (for marked).
2. Choose the **Exclude > Recalculate** menu command to:
 - Remove the marked vector from the Branch tab.
 - Change the results in the Statistics tab.

Using a File to Exclude Branches

You can use cmView to write a file containing the branches that are marked for exclusion with the X or M symbol. When you start cmView again and specify this file, the branches in this file will be excluded from coverage.

To use cmView to write a file containing the excluded branches:

1. Choose the **Exclude > Write To File** menu command.

This displays the Specify File For Excluded Branches dialog box.
2. Select the directory where you want to write the file and enter the name of this file.
 - If this is a new file, the process of creating this file is complete.
 - If this is an existing file, the Specify Write Mode dialog box appears and displays a message similar to the following:

```
File exclude_file_path_name is Not Empty  
Overwrite or Append?
```

You have a choice of overwriting the file (the file will only include entries for the newly excluded files), or you can choose to append new entries for the newly excluded files.

3. Select Append or OverWrite.

Append adds new entries. OverWrite deletes previous entries and only includes entries for the newly excluded files.

Appended entries for the same module instance are accumulated or merged together.

Note:

This feature works in similar ways in line and toggle coverage and you can use the same file to exclude lines, signals, and branches.

4. The next time you start cmView, either with the cmView GUI, or in batch mode to write reports, specify this file with the `-cm_elfile` command-line option. For example:

```
vcs -cm_pp gui -cm branch -cm_elfile exclude_file
```

or

```
vcs -cm_pp -cm branch -cm_elfile exclude_file
```

The excluded branches have the X symbol in the Branch tab or reports and they are not counted in the statistics.

Comments in the File

If you want to include documentation, you can add comments to the output file containing excluded branches. To do this, first recalculate the exclusion of these branches, then perform the following:

1. Choose the **Exclude > Comment** menu command.

The Provide The Comment For Last Set Of Excluded Branches dialog box displays.

2. Enter a comment.
3. Click Save.
4. Choose the **Exclude > Write To File** menu command and specify the file using the Specify File For Excluded Branches dialog box.

Clearing Marked Branches

After marking a branch for exclusion, you can clear it by clicking again on the branch. The symbol will then change from M back to $=>$.

If a branch is already excluded, as indicated by the X symbol, clicking the symbol changes it to the XM symbol. When you choose the **Exclude > Recalculate** menu command, the statistics change to show that it is not covered and the branch's symbol changes to $=>$.

To clear M and MX symbols from multiple branches, use the following menu commands:

- **Exclude > Unmark This Module**
- **Exclude > Unmark All Modules**

This changes M symbols to $=>$ and MX symbols to X .

Inconsistency in the Exclude File

If the exclude file you specify with the `-cm_elfile` option has an entry to exclude a branch from coverage, but after revising the design, the current coverage results show that the branch is covered, cmView performs the following depending on if you are using the batch report writing mode or GUI mode.

In batch mode cmView displays the following:

```
//Error: Module version has changed since el-file was created  
//The excluded information is ignored  
//-----  
//module_name elfile: exclude_file
```

cmView ignores the entry in the exclude file and the report shows the branch as covered.

In GUI mode cmView opens the Inconsistencies Detected In Elf file dialog box to display the same message that it would display in batch report writing mode.

The dialog box has two buttons:

Ignore

Enables you to continue the cmView GUI session. For the instance, the Branch Coverage summary window does not show the covered branch as excluded.

Abort

Ends the cmView GUI session.

Excluding Covered Branches

You can exclude covered branches in the exclude file. If you choose to do this and then start cmView again, entering the `-cm_elfile` option with the exclude file containing entries for covered branches, cmView performs the following depending on if you are using the batch report writing mode or GUI mode.

In batch report writing mode, cmView reports the branch as covered and puts an A symbol to the left of the row or vector for the branch, indicating that there was an attempt to exclude the covered branch that was ignored. For example:

BRANCH	-1-	-2-	-3-	-4-	-5-	
A	1'b1	1	1	-	-	Covered
A	1'b1	1	0	-	-	Covered
A	1'b1	0	-	-	-	Covered
	1'b0	-	-	1	1	Not Covered
	1'b0	-	-	1	0	Covered
	1'b0	-	-	0	-	Covered
	default	-	-	-	-	Not Covered

In GUI mode, cmView opens the Inconsistencies Detected In Elf file dialog box to display a message similar to the following:

```
//Warning: Attempt to exclude the covered branch, attempt  
is ignored  
//-----  
-----  
// module line line_number vector vector_number elfile:  
exclude_file
```

The dialog box has two buttons:

Ignore

Enables you to continue the cmView GUI session. The Branch Coverage summary window for the instance shows the A symbol to the left of the vector that was covered, but there was an attempt to exclude it (see [Figure 7-9](#)).

Abort

Ends the cmView GUI session.

Figure 7-9 Branch Tab with Covered Branches Attempted to Exclude

BRANCH	-1-	-2-	-3-	-4-	-5-	
A	1'b1	1	1	-	-	Covered
	1'b1	1	0	-	-	Covered
	1'b1	0	-	-	-	Covered
==>	1'b0	-	-	1	1	Not Covered
	1'b0	-	-	1	0	Covered
	1'b0	-	-	0	-	Covered
==>	default	-	-	-	-	Not Covered

Branch Coverage Limitations

Branch coverage only works on Verilog code; it does not work on VHDL code.

The following compile-time options for coverage metrics do not work with branch coverage:

`-cm_count`

Adds an execution count to reports.

`-cm_cond_sop`

Modifies condition coverage compilation, replacing sensitized vectors with condition SOP coverage.

The following cmView command-line options (or vcs -cm_pp) do not work with branch coverage:

-cm_report testslist

Instructs cmView to indicate which test files caused the line or condition to be covered in its line and condition coverage reports.

-cm_report_testlists_max *int*

An alternative to the -cm_report testslist option and argument. This option specifies the number of test files to use when indicating which test files caused a line or condition to be covered. With the -cm_report testslist option and argument, cmView uses the first three test files in alphanumeric order of the file names.

The following do not work with branch coverage:

- The cmView Tcl interface
- API tasks and API functions for real time coverage

Branch Coverage

7-418

8

Using the Graphical User Interface

cmView can start a Graphical User Interface (GUI) for coverage metrics enabling you to interactively view coverage results from multiple simulations.

Note:

For more information about the Unified Coverage Reporting and Viewing Coverage in DVE, see the URG User Guide under the Coverage category and DVECoverageGUI under the LCA category respectively in the VCS Online Documentation.

The GUI includes the following features:

- A hierarchy browser that enables you to traverse the design hierarchy with ease
- Tabulated coverage results which you can view with the aid of a variety of controls

- Detailed annotated listing windows which enable you to look at your original source files with uncovered lines highlighted
- Test comparison functionalities that enable you to obtain incremental and differential coverages from two tests
- Test grading windows and tabulation grids that enable you to compare tests to help you determine which tests cover which portions of the design for statement, condition, and toggle coverage

This chapter describes the following:

- “[cmView Command-line Options](#)”
- “[Common Operations in cmView Windows](#)”
- “[Test Grading](#)”
- “[User Preferences](#)”

cmView Command-line Options

cmView accepts command-line options by using one of the following methods:

- From a command line beginning with the `cmView` command followed by the command-line options.
- From a `vcs` command with the `-cm_pp` option which instructs VCS to start cmView instead of starting compilation. We recommend this method to ensure that you start the correct version of cmView.

The `-cm_pp` option includes the following optional keyword arguments:

`gui`

VCS starts the cmView graphical user interface to display coverage data.

`batch`

Specifies the default operation. In other words, it instructs cmView to write reports in batch mode.

Note:

For mixed language design, use the

`-cm_dir directory_path_name` option with `vcs -cm_pp` to invoke the cmView GUI (the following includes additional information about the `-cm_dir` option).

After you enter `cmView` or `vcs -cm_pp`, you can enter the following command-line options.

`-cm`

Specifies reading the design file and the test files for the specified type or types of coverage. The arguments specify the types of coverage:

`line`

Line or statement coverage.

`cond`

Condition coverage.

`fsm`

FSM coverage.

`tgl`

Toggle coverage.

`path`

Path coverage.

branch

Branch coverage.

If you want cmView to read the intermediate data files for more than one type of coverage, use the plus (+) character as a delimiter between arguments. For example:

```
vcs -cm_pp -cm line+cond+fsm+tg1
```

-cm_annotation *type_of_coverage*

Specifies adding FSM or condition coverage to the annotated files in the .simv.cm/reports/annotated directory which contains line coverage information. You can specify the following types of coverage:

cond

Specifies adding condition coverage information.

fsm

Specifies adding FSM coverage information.

You can specify adding both types of information with the plus (+) character delimiter. For example:

```
vcs -cm_pp -cm line+cond+fsm+tg1 -cm_annotation fsm+cond
```

-cm_autograde *percentage*

Instructs cmView to perform test autograding in batch mode and write a test autograding report.

`-cm_cond_branch`

When you use the Verilog conditional operator (`? :`) in a continuous assignment, by default, VCS and VCS MX monitor the conditional expression operand for the conditional operator. Enter this option if you would rather have cmView report what values were assigned in the continuous assignment instead of whether the conditional expressions were covered (see “[Specifying Condition SOP Coverage](#)” on page 4-219).

`-cm_dir directory_path_name`

Specifies an alternative name and location for the `simv.cm` directory.

`-cm_elfile`

Allows you to exclude the following from coverage calculations:

- Source code lines that you marked for exclusion from line coverage in a previous cmView GUI session (see “[Using a File to Exclude Lines](#)” on page 2-140).
- Signals that you marked for exclusion from toggle coverage in a previous cmView GUI session (see “[Using a File to Exclude Signals](#)” on page 3-191).
- Branches that you marked for exclusion from branch coverage in a previous cmView GUI session (see “[Using a File to Exclude Branches](#)” on page 7-411).

`-cm_exclude filename`

Excludes the conditions listed in the specified exclude file (see “[Excluding Conditions and Vectors From Reports](#)” on page 4-231).

`-cm_hier config_filename`

Specifies a configuration file that includes the parts of the design in which you do, or do not, want to see coverage information. This file has the same syntax as the file specified with the `-cm_hier` compile-time option.

`-cm_log filename`
Specifies a log file.

`-cm_map module_name | entity_name | library_name.entity_name`
Maps the coverage for a subhierarchy from one design to another.

`-cm_name filename`
Specifies the name of the report files.

`-cm_report keyword_argument`
The keyword arguments you can specify are as follows:

- `annotate [+module | +instance]`
Instructs cmView to write annotated files in the `simv.cm/reports/annotated` directory. Annotated files show where you are missing line coverage.
- `module`
Only writes annotated files for each module (or entity) definition.
- `instance`
Only writes annotated files for each instance.
- `cond_ids`
Instructs cmView to add condition ID numbers in its reports and write the `cm_excludeCond_generated` file (see “[Excluding Conditions and Vectors From Reports](#)” on page 4-231).
- `disable_sequence`
Does not list any sequences in the report.
- `disable_sequence_loop`
Does not list any sequence that starts and ends with the same state.

`portdir`

Instructs cmView to annotate the toggle coverage report with the direction of each port signal (see “[Displaying Port Signal Direction in the Toggle Coverage Reports](#)” on page 3-169).

`summary`

Instructs cmView to write a summary file of its reports. In VCS MX, do not use with a VHDL top design.

`summary_on_top`

Report files begin with a summary instead of at the end.

`testlists`

Instructs cmView to indicate which test files caused the line or condition to be covered in its line and condition coverage report files.

`worst_first`

Sections for instances begin with those containing the lowest coverage.

`-cm_report_testlists_max int`

An alternative to the `-cm_report testlists` option and argument. This option specifies the number of test files to use when indicating which test files caused a line or condition to be covered. With the `-cm_report testlists` option and argument, cmView uses the first three test files in alphanumeric order of the file names.

`-cm_tests filename`

Specifies a file containing a list of test files. cmView only reads these test files.

`-cm_nocasedef`

Instructs cmView not to display or report line or condition coverage information about default case statements that are not executed.

- cm_verbose
 - Instructs cmView to display summary information after writing reports.
- cm_summary instance | noemail
 - The keyword arguments specify the following:
 - instance
 - Instructs cmView to include summary information on module instances in the summary report file.
 - noemail
 - Instructs cmView not to e-mail the summary report file to you.
- help
 - Displays a help message.

Figure 8-1 The cmView Main Window



Common Operations in cmView Windows

This section describes the operations you can perform with cmView regardless of the type of coverage.

Hiding or Showing the Tool and Status Bars

By default, both the toolbar and status bar are displayed in the main window. However, you can hide and/or restore the display of the toolbar and the status bar as follows:

- To hide the toolbar, choose the **Options > Toolbar** menu command.

- To display the toolbar, choose the **Options > Toolbar** menu command again.

Toolbar is a toggle option: clicking Toolbar alternately hides and displays the toolbar.

- To hide the status bar, choose the **Options > Status Bar** menu command.
- To display the status bar, choose the **Options > Status Bar** menu command again.

Status Bar is a toggle option: clicking Status Bar alternately hides or displays the status bar.

Controlling the Display of Multiple Windows in cmView

All open windows can be reduced to icons at the same time by choosing the **Window > Minimize all** menu command:

This creates icons for all windows in the cmView session (see [Figure 8-2](#)):

Figure 8-2 cmView Icon



Double-click the cmView icon to restore the main window, then choose the **Window > Restore All** menu command in the main window to restore all windows to their original state, that is, before they were made into icons.

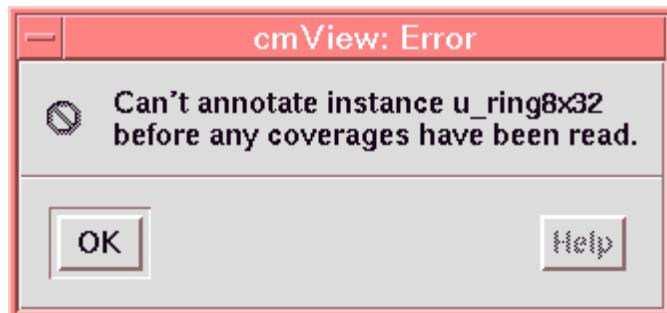
You can also display a list of all open windows in the current session and make a particular window active as follows:

1. Click Window in the menu bar.
2. Click a window in the drop-down list to make it the active window and bring it to the front.

Displaying the Log of Error and Warning Messages

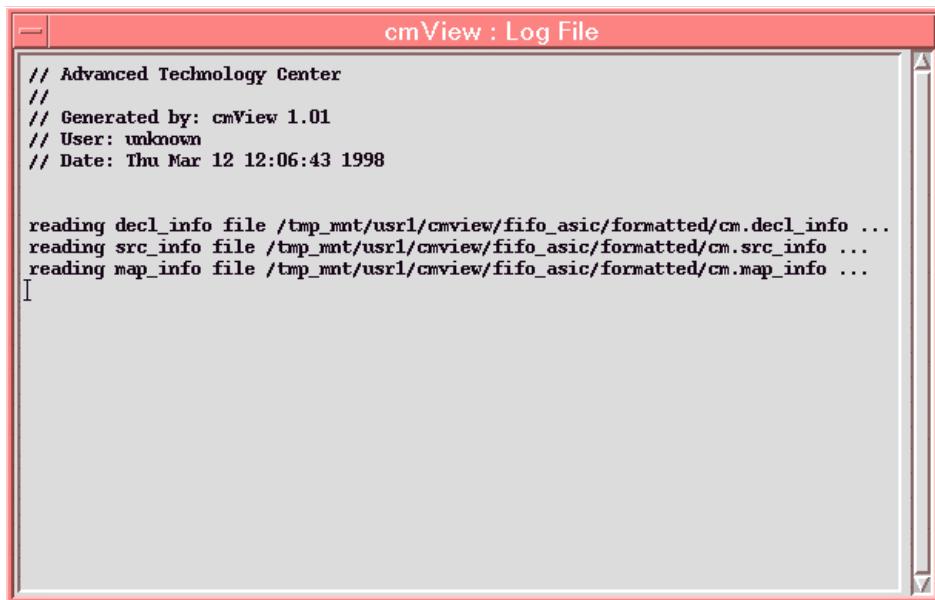
When an error occurs in a cmView session, an error dialog box is displayed with a brief description of the problem (see [Figure 8-3](#)).

Figure 8-3 cmView Error Dialog Box



You can display additional information about the error by choosing the **Window > Error Log** menu command. This command provides you with detailed information about the error and displays a list of all messages logged during the current session (see [Figure 8-4](#)).

Figure 8-4 Detailed Error Information and Message Log



When cmView issues a warning message during a session, it writes the warning to the error log and displays the warning icon (see below), in the appropriate window. Clicking on the warning icon displays the Log File window (see [Figure 8-4](#)).



The Hierarchy Pane

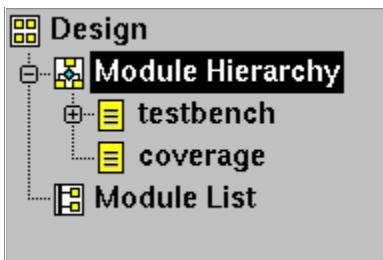
You can use the hierarchy window to traverse the module hierarchy in the design. To do this, perform one of the following:

1. Click Module Hierarchy  in the window
or

2. Click the plus sign  located to the left of the folder icon to expand the hierarchy.

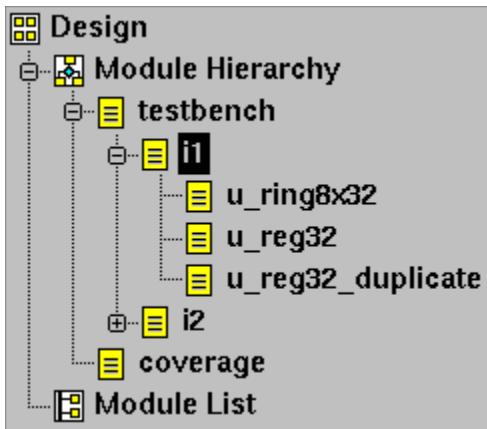
Expanding the module hierarchy displays the top-level modules in the design (see [Figure 8-5](#)).

Figure 8-5 Module Hierarchy, Top Level



Module instances that contain module instantiations are also displayed with a plus sign (+) to the left, which allows you to expand the hierarchy at that level. Clicking the plus sign at each subsequent level allows you to see the entire hierarchy of modules contained in the design.

Figure 8-6 Module Hierarchy, All Current Levels



Similarly, once a hierarchy has been expanded, you can collapse it by clicking the minus sign (-). For example, clicking the minus sign to the left of the test bench collapses the hierarchy (see [Figure 8-5](#)).

The Summary Pane

When you click a module instance in the hierarchy pane, the summary pane displays coverage statistics for that instance and all its submodules. The statistics are displayed in a tabulated grid. You

have control over modifying the presentation of the data in a variety of ways. For example, all columns in this window are resizable. You can resize any column by dragging the right hand border of the heading cell at the top of the column.

The rows in this summary pane are color coded according to the percentage of coverage displayed in the fourth column.



Red All instances with a low coverage of less than 20% are displayed in red.

Yellow All instances with medium coverage of between 20% and 80% are displayed in yellow.

Green All instances with a high coverage greater than 80% are displayed in green.

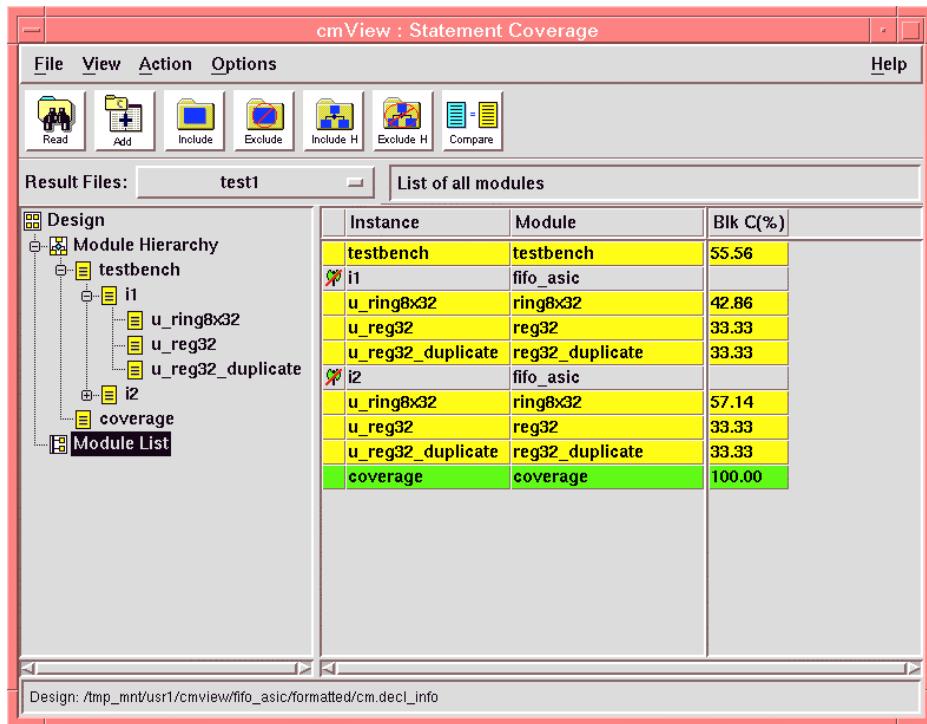
These coverage levels and their corresponding colors are the defaults. You can change them by using the Preferences dialog (see “[User Preferences](#)”).

If you want to see the summary for all modules in the design without regard to their hierarchical relationships, select Module List in the hierarchy pane (see below).



This displays a flat list of all modules in the design along with their corresponding coverage statistics in the summary pane. An example of the statistics displayed for statement coverage is shown in [Figure 8-7](#).

Figure 8-7 Module List and Corresponding Coverage Results



Click Design in the hierarchy pane to display the total coverage analysis for the entire design in the summary pane.

Sorting the Coverage Results Data

Clicking the heading cell at the top of any column sorts the coverage data using the data in that column as the sort key. For example:

- Click any column-heading cell to sort the coverage data in ascending order by the data contained in that column.
- Click the same column heading to sort the data in descending order.

The Result Files Menu

The coverage data displayed in the summary pane is the result of the test displayed in the Result Files menu. Once the results of a test are read into cmView, the test is added to this list. You can select any test in the list to display its corresponding coverage results in the summary pane.

In addition to the individual tests read, there are three special categories of results that can be viewed. These are referred to as total, incremental, differential (or Diff) coverage. The following table briefly describes the different categories:

Results Category	Explanation
Total Coverage	Total coverage is the cumulative result of all the tests added in the current session. Total coverage is the default.
Incremental Coverage	Incremental coverage is the amount of coverage provided by one selected test over another selected test.
Differential Coverage	Differential (<i>Diff</i>) Coverage is the difference in the coverage provided by any two selected tests.

The following sections further describe each of these categories:

- “[Adding the Results of a Test to the Total Results](#)”
- “[Viewing the Incremental Coverage of an Added Test](#)”
- “[Viewing the Differential Coverage Between Two Tests](#)”

Click the same column again to return the data to its original state.

Opening a Design File

The `cm.decl_info` file is the design file that VCS writes about the design and that cmView reads in order to report or display coverage results.

VCS creates the `./simv.cm/db` directory and writes the `cm.decl_info` file in the `db` directory. When you start cmView, it looks for the `/simv.cm/db` directory and the `cm.decl_info` file. If cmView locates it, the file is read. Therefore, there is no need to go through the process of opening a design file unless you want to look at coverage data stored elsewhere, perhaps for another design.

To open a design file, perform one of the following:

- Click the Design icon (see below):

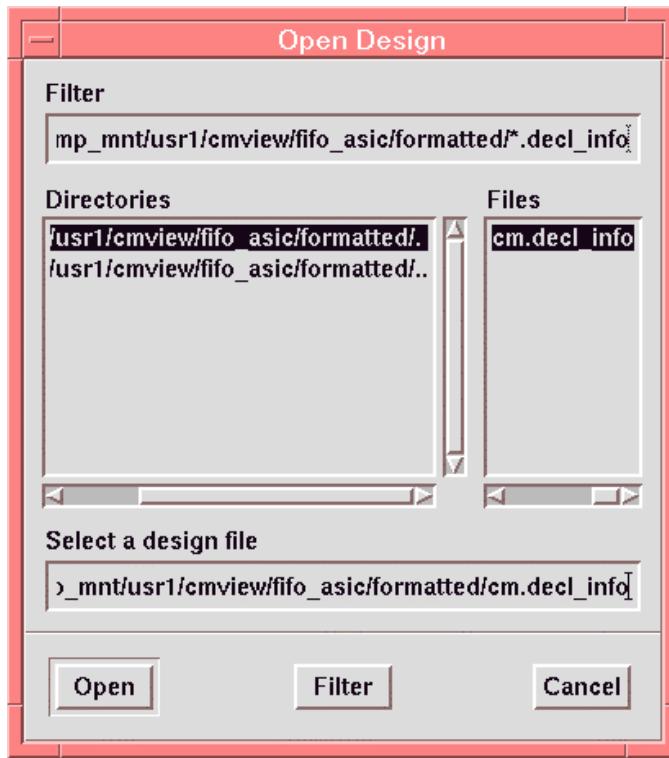


Design

- Choose the **File > Open Design** menu command.

This displays the Open Design dialog box (see [Figure 8-8](#)).

Figure 8-8 The Open Design Dialog Box



This dialog box has two fields and two panes:

- The Filter field defines the file selection criteria for the current directory. By default, the filter field ends with the following string:
`*.decl_info`
This indicates that the Files field displays only those files with the `decl_info` extension.
- The currently selected directory is highlighted in the Directories pane.
- The Files pane contains a list of all files that match the selection criteria specified in the Filter field.

- The Select a design file field shows the name of the chosen design file.

Double-click on any directory in the Directories pane to select that directory. Once you have located the information file directory, double-click on the `cm.decl_info` file in that information file directory to load the design. If you compiled your design with the `-cm` compile-time option, by default the information file directory is `./cm.dir`.

If an error occurs during the loading process, an error dialog box is displayed, briefly describing why the load failed. For a detailed description of the error, check the error log (see “[Displaying the Log of Error and Warning Messages](#)”). When a design is successfully loaded, the name of the design is displayed in the status bar.

Adding the Results of a Test to the Total Results

By default, the coverage data displayed in the summary window is the cumulative total of all tests added so far. To add a test to this cumulative total, perform the following:

- Click the Add icon (see below), or



- Choose the **File > Add Coverage** menu command.

This command displays the Add Coverage dialog box (see “[The Add Statement Coverage Dialog Box](#)”).

The default extension is displayed in the file selection field. This extension will be either `.line` for statement coverage or `.tg1` for toggle coverage.

All files in the selected directory, whose extensions match the default extension, are displayed in the Files field.

As in the previous case, traversing directories displays all the tests in the Files field.

1. Click a test to add the coverage results data for that test to the cumulative total coverage.
2. Click the Result Files menu as follows (in this example, Total Coverage is the current selection in that menu and clicking Total Coverage reveals the menu):

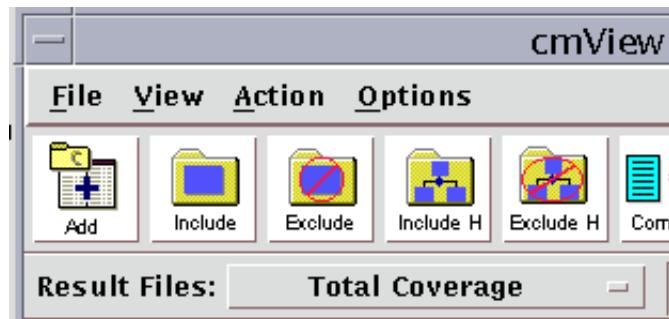
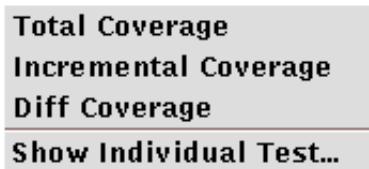


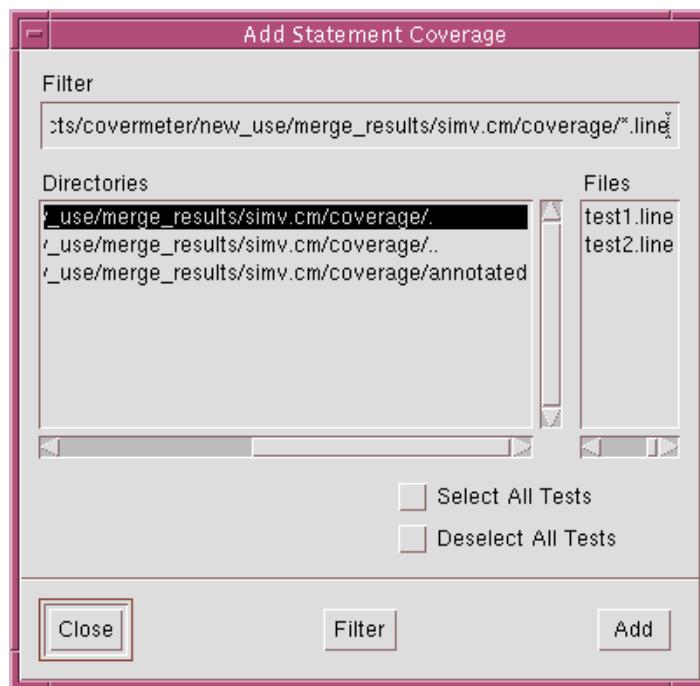
Figure 8-9 Example Results Files Option Menu



3. Select Total Coverage to display the cumulative coverage in the summary pane.

See “[How to Display the Current Test List](#)” for instructions on how to see which tests have been added so far and what currently comprises total coverage.

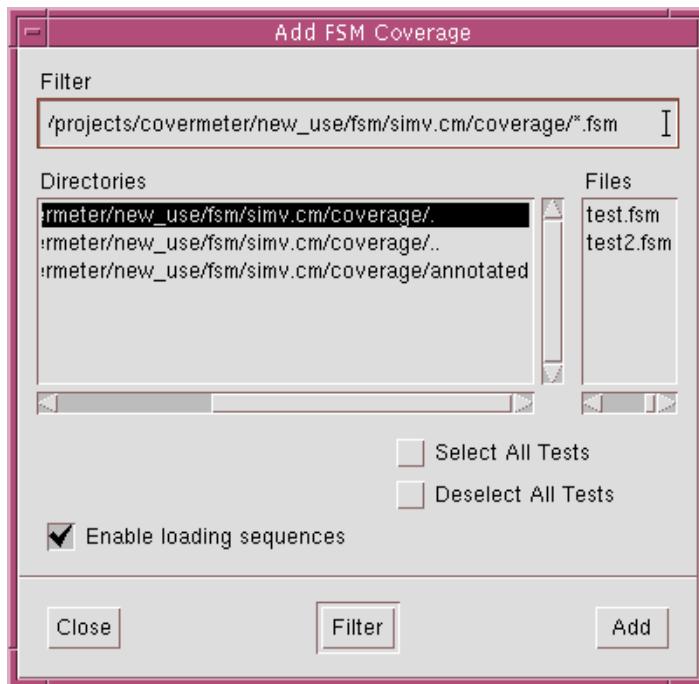
Figure 8-10 The Add Statement Coverage Dialog Box



4. Double-click on an intermediate report file in the Files list box to add the test recorded in that intermediate file and close the dialog box.
5. Single-click on an intermediate report file in the Files list box, then click the Add button to add the test without closing the dialog box.

If you are adding FSM coverage data, the Add Coverage dialog box has an additional feature (see [Figure 8-11](#)).

Figure 8-11 The Read FSM Coverage Dialog Box



The Enable loading sequences check box enables you to see dynamic as well as static information in the Reachability and Sequences tabs of the FSM Summary window. Static information is whether an FSM could reach from one state to another as determined by an analysis of the Verilog code before simulation starts. Dynamic information is whether the FSM did reach from one state to another during simulation. For more information see [“The Reachability Tab” on page 5-352](#).

Note:

Loading sequence information can be time consuming.

Viewing the Incremental Coverage of an Added Test

To view incremental coverage of an added test, click Incremental Coverage in the Result Files menu. This shows you the incremental coverage of the last test added with the Add Coverage option over the previously accumulated total coverage.

The summary pane reflects this incremental coverage.

To view the incremental coverage of one test over another, perform the following:

1. Select two tests to compare as described in the section, "[How to Select Two Tests for Comparison](#)".
2. Select Incremental Coverage in the Result Files menu.

The summary pane now displays the incremental coverage of the first test selected over the second.

Viewing the Differential Coverage Between Two Tests

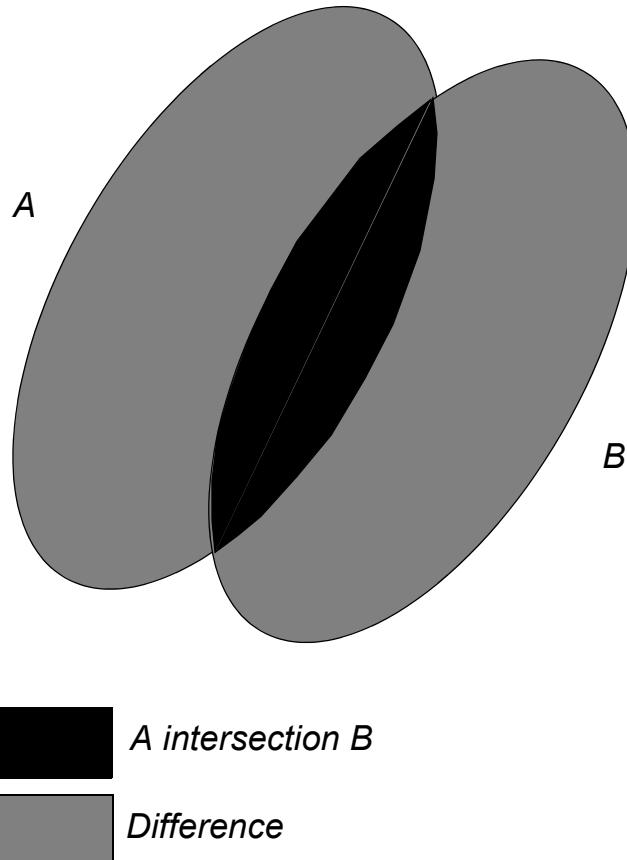
To view the difference in coverage between two tests, click Diff Coverage in the Result Files menu. This shows you the difference in coverage between the last test added with the Add Coverage option and the previously accumulated total coverage. The definition of difference comes from set theory and is calculated as follows:

$$\text{Difference} = (A \cup B) - (A \cap B)$$

In this instance, B represents the covered code in the last test that you added, and A represents the covered code for all tests that you added before you added the last one. ($A \cup B$) is the covered

code from all tests including the last one and (A Intersection B) is the covered code in the last test that was also covered by previous tests (see the Venn diagram in [Figure 8-12](#)).

Figure 8-12 Venn Diagram of Difference, Union, and Intersection in Set Theory



For example, a series of tests cover lines 100-200 and lines 300-400 of a source file, and an additional test covers lines 250-350. (A Union B) would be lines 100-200 and 250-400, (A Intersection B) would be lines 300-350, so the difference is lines 100-200, 250-300, and 350-400.

The summary pane reflects this difference in coverage.

To view the differential coverage between two individual tests, perform the following:

1. Select two tests to compare as described in the section “[How to Select Two Tests for Comparison](#)”.
2. Select the Diff Coverage item in the Result Files menu.

The summary pane now displays the difference in coverage between the two tests.

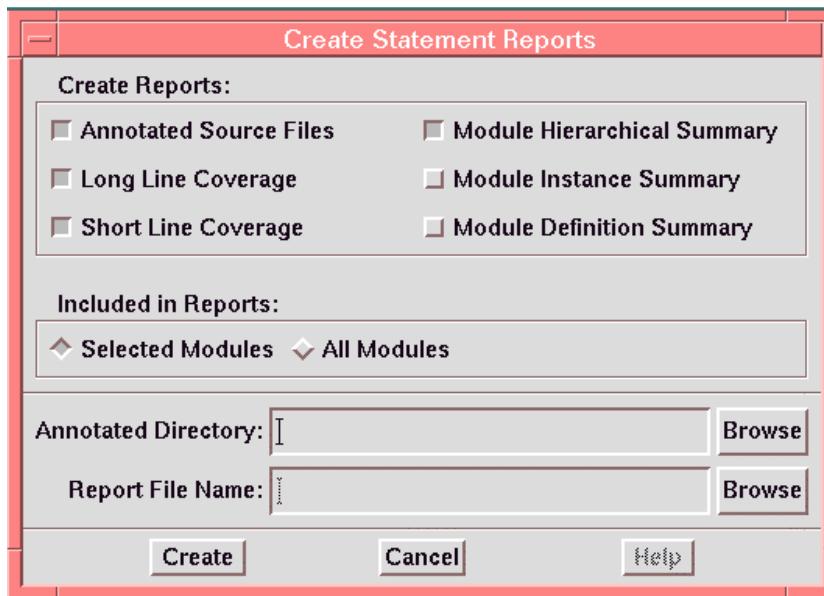
Creating Reports

cmView can create reports either in batch mode or interactively using a series of dialog boxes for the different type of reports. See the section, “[cmView Command-line Options](#)”, for a listing of the command-line options for batch mode.

To create reports interactively, perform the following:

1. Choose the **File > Reports** menu command in the Statement Coverage window to display the Create Statement Reports dialog box. The same menu command in the Toggle Coverage window displays the Create Toggle Reports dialog box, and the corresponding menu commands for condition and FSM coverage have their corresponding dialog boxes for reports. The layout of these dialog boxes is the same.

Figure 8-13 Create Statement Reports Dialog Box



2. Choose the options you need in the report that you want to create:

The fields and selections in this dialog box are as follows:

Annotate Source Files

Specifies that you want cmView to write annotated source files.

These files are copies of the module definitions from the module instances that you selected or specified, with the lines numbered and with --> text symbols to indicate line coverage. This check box is only active in the Create Statement Reports dialog box.

Long Line|Toggle|Condition|FSM Coverage

Writes comprehensive reports — the `cmView.long_l`, `cmView.long_t`, `cmView.long_c`, and `cmView.long_f` files.

Short Line|Toggle|Condition|FSM Coverage

Writes short reports with summary information and only instances that do not have full coverage — the `cmView.short_l`, `cmView.short_t`, `cmView.short_c`, and `cmView.short_f` files.

Module Hierarchy Summary

Generates reports on hierarchies, where module instances coverage results includes the coverage data for module instances under them — the `cmView.hier_l`, `cmView.hier_t`, `cmView.hier_c`, and `cmView.hier_f` files.

Module Instance Summary

Reports on module instances where coverage data does not include coverage data from module instances under them — the `cmView.mod_l`, `cmView.mod_t`, `cmView.mod_c`, and `cmView.mod_f` files.

This check box is active when the view in the coverage window is set to instances, which is the default setting.

Module Definition Summary

Specifies reporting coverage for all instances of module definitions. Instead of reporting on instances, `cmView` reports on module definitions. This check box is active when the view for the coverage window is set to module definitions instead of module instances (see “[The View Menu](#)”).

Selected Module

Writes reports only about the module instance or module definition selected in the summary pane.

All Modules

Writes reports about all module instances or definitions in the summary pane.

Annotated Directory

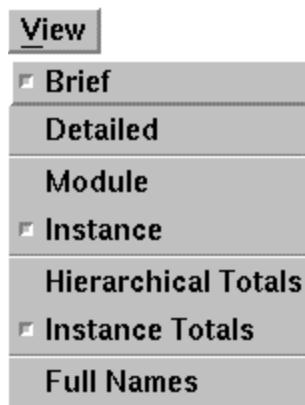
Specifies the directory where cmView writes the annotated source files. This field is only active in the Create Statement Reports dialog box and only when you click Annotated Source Files.

Report File Name

Specifies the name of all the report files, but not their extensions.

3. Click Create.

The View Menu



By default, the summary pane displays a brief summary of the specific coverage results data for the selected modules.

To obtain a detailed report instead of the summary, choose the **View > Detailed** menu command. The details displayed depend on the type of coverage that is being viewed. The appropriate coverage window section describes this information.

To see a brief summary, select the **View > Brief** menu command.

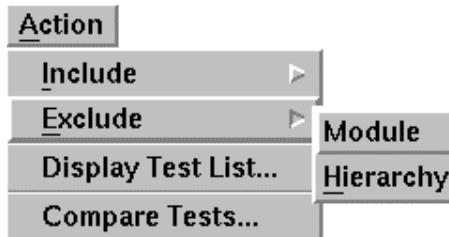
By default, the summary window displays coverage statistics for the module instance selected in the hierarchy window. You have the following options:

- To see the coverage analysis for a module definition, choose the **View > Module** menu command. This instance column is not displayed in the summary window. The totals are displayed for all modules.
- Choose the **View > Instance** menu command to return to the instance view.

By default, totals displayed in the instance view are instance totals only. In other words, the coverage results data apply only to that instance. You have the following options:

- To see hierarchical totals, choose the **View > Hierarchical Totals** menu command. The totals displayed in this case are displayed for each instance and its hierarchy.
- Choose the **View > Instance Totals** menu command to see the instance totals again.
- To see all instance names displayed in their fully qualified form, choose the **View > Full Names** menu command. In this case, all instances are displayed in their long form.
- To revert back to short names, choose the **View > Full Names** menu command again.

The Action Menu



You can use the Action menu to exclude or include an instance or module from coverage analysis. You can also use it to exclude or include a complete hierarchy from coverage.

Excluding an Instance or Module From Coverage

To exclude an instance or module, perform the following:

1. Click the instance or module in the summary pane.
The row is highlighted.
2. Click the Exclude icon (see below), or choose the **Action > Exclude > Module** menu command.



Exclude

The instance or module is excluded from coverage analysis. A blue circle with a red line through it is displayed in the first column to indicate that the instance or module has been excluded.

Excluding a Hierarchy from Coverage

To exclude a hierarchy from coverage, perform the following:

1. Click the parent instance of the hierarchy in the summary pane.
The row is highlighted.
2. Click the Exclude Hierarchy icon (see below), or choose the **Action > Exclude > Hierarchy** menu command.



The instance and its hierarchy are excluded from coverage analysis. A blue circle with a red line through it is displayed in the first column of all instances in the hierarchy to indicate that the instance has been excluded.

Including an Instance or Module in Coverage Analysis

To include an instance or a module in the coverage, perform the following:

1. Click the instance or module in the summary pane.
The row is highlighted.
2. Click the Include icon (see below), or choose the **Action > Include > Module** menu command.



Including a Hierarchy in Coverage Analysis

To include an instance and its hierarchy in the coverage, perform the following:

1. Click the parent instance of the hierarchy in the summary pane.

2. Click the Include Hierarchy icon (see below), or choose the **Action > Include > Hierarchy** menu command.



How to Display the Current Test List

To display a list of tests that have been read or added for the specified coverage type in the current session of cmView, choose the **Action > Display Test List** menu command.

This list includes the full path names of all tests. Also listed are the tests that have been set up to be compared in both Incremental and Diff coverages.

[Figure 8-14](#) shows an example of the tests read for statement coverage during a cmView session.

Figure 8-14 Statement Coverage: Total Tests Window

Statement Coverage: Total Tests	
Incremental & Diff	Total Coverage /tmp_mnt/usr1/cmview/fifo_asic/coverage/test0
Tests Added	/tmp_mnt/usr1/cmview/fifo_asic/coverage/test0
Tests Read	/tmp_mnt/usr1/cmview/fifo_asic/coverage/test0 /tmp_mnt/usr1/cmview/fifo_asic/coverage/test1

The list contains all tests that have been read into the database from the current session for the chosen coverage type. These include tests that have been both read using the Read Coverage option and added using the Add Coverage option in the File dropdown menu.

How to Select Two Tests for Comparison

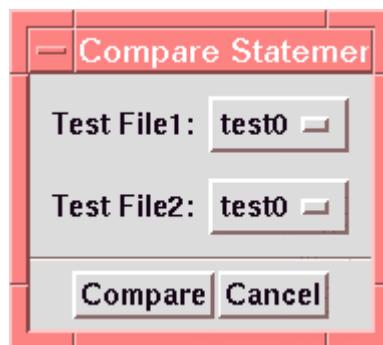
To select two tests for comparison, do the following:

1. Click the Compare icon (see below), or choose the **Action > Compare Tests** menu command.



This displays the Compare Tests dialog box (see [Figure 8-15](#)).

Figure 8-15 Compare Tests Dialog Box



This dialog box presents two option menus for selecting the two tests:

- If **Incremental Coverage** is to be viewed, the incremental coverage of the first test over the second is displayed.
- Similarly, if **Diff Coverage** is selected, the difference in coverage between the two tests is displayed.

The Options Menu

The Options menu allows you to hide or show the tool and status bars and set up or modify the user preferences (for additional information, see "[Hiding or Showing the Tool and Status Bars](#)" and "[User Preferences](#)").

Test Grading

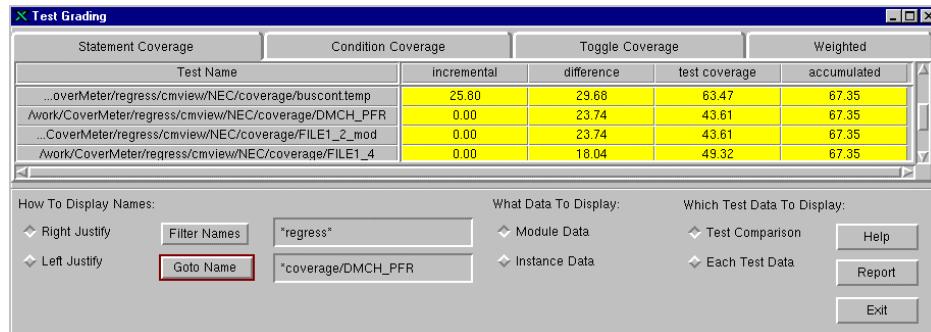
Test grading is determining whether a test or set of tests provides a specific amount of coverage. A test is an intermediate data file from a simulation run for a type of coverage. In this sense, the files `test.line` and `test.tgl` are tested for line and toggle coverage.

To open the Test Grading window and examining the coverage obtained for each test added to the current session, perform the following:

1. Choose the **Tools > Grading** menu command in the cmView main window:

This displays the Test Grading window (see [Figure 8-16](#)).

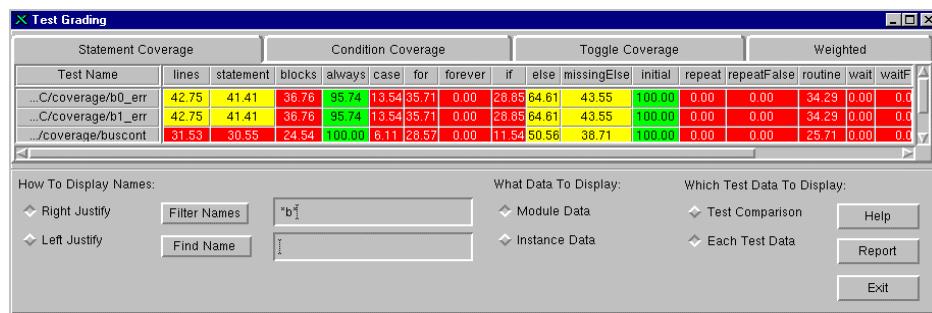
Figure 8-16 The Test Grading Window, “Test Comparison” Selected



2. Click on any one of the three tabs for Statement, Condition, or Toggle coverage to view the data for each test listed in the Test Name column.
3. Click the Weighted tab to view the relative coverage levels obtained for each test listed in the Test Name column.

- Manipulate how the data for each tab is displayed by utilizing the features provided in the control panel in the bottom portion of the Test Grading window. For example, click the radio button for Each Test Data to obtain a detailed view of the coverage for each test. The coverage is broken down by features in the code (see [Figure 8-17](#)).

Figure 8-17 The Test Grading Window, “Each Test Data” Selected



- To sort the data when “Each Test Data” is selected, click the heading cell at the top of any column in the display to sort the data using the data in that column as the sort key. For example:
 - Click any column heading cell to sort the data in ascending order by the data contained in that column.
 - Click the same column heading to sort the data in descending order.
 - Click the same column heading again to return the data to its original state.
- All columns in the Test Grading window are resizable. To resize columns, click the border located to the right of the heading cell at the top of a column and drag the border left or right to the desired width.

Displaying the List of Test Names

There are controls available that enable you to display the list of test names in different ways.

Justification

The justification controls enable you to scroll the path names in the Test Names column to the left and right when the path name is too long to fit in the viewable portion of the column:

- Click the Right Justify radio button to see the name of the file at the end of the path name.
- Click the Left Justify radio button to see the start of the path name.

Selecting a Specific Group or Subset of Tests for Display

The Filter Names control allows you to use pattern matching with a wildcard character (an asterisk) to group tests together by virtue of a common string of characters:

1. Click in the File Names field and type a string of one or more characters you want to use to display a list of tests whose names contain the same string. Use an asterisk as a wildcard character to match any character in a test name.
2. Click the File Names button to display the tests.

Focusing the Display on One Specific Test

The Go To Name control allows you to focus your attention on one specific test, assuming you know either the exact name of the test or the last few characters of the name of the test.

1. Click in the Go To Name field and type the name of the test you want to display, or type a string of characters preceded by an asterisk, as shown in [Figure 8-16](#).
2. Click the Go To Name button.

The test is displayed in the Test Name column, centered vertically in the column.

Displaying Different Kinds of Data

Two sets of radio buttons are provided that enable you to choose the data you want to display:

- Click the Module Data radio button to enable the display of test grading based on module definitions, then click either the Test Comparison or the Each Test Data radio button, as desired.
- Click the Instance Data radio button to enable the display of test grading based on module instances, then click either the Test Comparison or the Each Test Data radio button, as desired.

Either test comparison data or data for each test can be displayed for both module and instance data.

Test Comparison Data

Test comparison data is displayed in a 4-column tabulated grid as shown in [Figure 8-18](#) for statement coverage, [Figure 8-19](#) for condition coverage, and [Figure 8-20](#) for toggle coverage.

Figure 8-18 Test Comparison Data for Statement Coverage

Statement Coverage		Condition Coverage		Toggle Coverage		Weighted	
Test Name	incremental	difference	test coverage	accumulated			
...fifo_asic/cm.coverage/test	41.09	41.09	41.09	41.09			
...fifo_asic/cm.coverage/test0	8.53	10.85	47.29	49.61			
...fifo_asic/cm.coverage/test1	0.00	8.53	41.09	49.61			
...fifo_asic/cm.coverage/test2	2.33	13.95	40.31	51.94			

Figure 8-19 Test Comparison Data for Condition Coverage

Statement Coverage		Condition Coverage		Toggle Coverage		Weighted	
Test Name	incremental	difference	test coverage	accumulated			
/work/CoverMeter/regress/cmview/NEC/coverage/b0_err	28.57	28.57	28.57	28.57			
/work/CoverMeter/regress/cmview/NEC/coverage/b1_err	2.34	4.75	29.19	30.61			
...CoverMeter/regress/cmview/NEC/coverage/busconttemp	0.00	0.00	0.00	0.00			
/work/CoverMeter/regress/cmview/NEC/coverage/buscont	3.38	19.93	15.42	31.96			

Figure 8-20 Test Comparison Data for Toggle Coverage

Statement Coverage		Condition Coverage		Toggle Coverage		Weighted	
Test Name	incremental	difference	test coverage	accumulated			
/work/CoverMeter/regress/cmview/NEC/coverage/b0_err	39.02	39.02	39.02	39.02			
/work/CoverMeter/regress/cmview/NEC/coverage/b1_err	10.98	21.95	39.02	50.00			
...CoverMeter/regress/cmview/NEC/coverage/busconttemp	0.00	0.00	0.00	0.00			
/work/CoverMeter/regress/cmview/NEC/coverage/buscont	2.13	37.80	16.46	52.13			

Four kinds of coverage data are used to compare coverage from test to test in the order in which the coverage results for each test were added to the cmView session. Tests are listed in the Test Name column in the order in which they were added to the session. The four kinds of coverage data include:

- **Incremental** data is the amount of additional coverage provided by a given test over the previous test.
- **Difference** data is the difference in coverage between a given test and the accumulated total up to the previous total.
- **Test coverage** data is the coverage for a given test. That is, the data reflects the total coverage for that test only.
- **Accumulated** data is the cumulative result of all tests added in the current session up to that point in the order of tests.

Data for Each Test: Statement Coverage

Detailed information for each test is displayed in an 18-column tabulated grid (see [Figure 8-21](#) for an example using statement coverage).

This information allows you to compare tests according to the specific features of Verilog. For statement coverage, the data is shown for each block type.

Figure 8-21 Detailed Information for Each Test for Statement Coverage

Test Name		lines	statement	blocks	always	case	for	forever	if	else	missingElse	initial	repeat	repeatFalse	routine	wait	waitF
...C/coverage/b0_err		42.75	41.41	36.76	95.74	13.54	35.71	0.00	28.65	64.61	43.55	100.00	0.00	0.00	34.29	0.00	0.0
...C/coverage/b1_err		42.75	41.41	36.76	95.74	13.54	35.71	0.00	28.65	64.61	43.55	100.00	0.00	0.00	34.29	0.00	0.0
...coverage/buscont		31.53	30.55	24.54	100.00	6.11	28.57	0.00	11.54	50.56	38.71	100.00	0.00	0.00	25.71	0.00	0.0

Data for Each Test: Condition Coverage

Detailed information for each test is displayed in a tabulated grid (see [Figure 8-22](#) for an example using condition coverage). For condition coverage, the data includes logical, non-logical, and events.

Figure 8-22 Detailed Information for Each Test for Condition Coverage

Test Name		logical	non-logical	event
/work/CoverMeter/regress/cmview/NEC/coverage/b0_err		26.57	100.00	94.85
/work/CoverMeter/regress/cmview/NEC/coverage/b1_err		26.19	73.20	94.85
...CoverMeter/regress/cmview/NEC/coverage/buscont.temp		0.00	0.00	0.00
/work/CoverMeter/regress/cmview/NEC/coverage/buscont		15.42	29.45	94.23

Data for Each Test: Toggle Coverage

Detailed information for each test is displayed in a tabulated grid (see [Figure 8-23](#) for an example using toggle coverage). For toggle coverage, the data includes registers, register bits, nets, and net bits.

Figure 8-23 Detailed Information for Each Test for Toggle Coverage

Test Name	Statement Coverage		Condition Coverage		Toggle Coverage		Weighted		
	total	regs	regBits	regBits 0->1	regBits 1->0	nets	netBits	netBits 0->1	netBits 1->0
..._asic_big/cm.coverage/test1	40.32	29.69	38.87	39.17	38.87	55.32	44.24	45.04	44.50
..._asic_big/cm.coverage/test2	29.99	20.31	26.98	29.28	26.98	36.17	32.71	33.51	32.98
..._asic_big/cm.coverage/test3	29.91	20.31	25.82	26.11	25.82	42.55	41.02	41.82	41.29

How To Display Names: What Data To Display: Which Test Data To Display:

Right Justify Filter Names Module Data Test Comparison Help
 Left Justify Goto Name Instance Data Each Test Data Report
 Report Exit

Weighted Coverage

Weighted coverage is a measure of the importance you place on the different types of coverage. Weights are assigned using the Weights tab in the User Preferences folder (see “[User Preferences](#)”). Test comparison data is displayed in a 4-column tabulated grid as shown in [Figure 8-24](#). Detailed information for each test is displayed as shown in [Figure 8-25](#).

Figure 8-24 Weighted Coverage for Test Comparison Data

Test Name	Statement Coverage		Condition Coverage		Toggle Coverage		Weighted	
	incremental	difference	test coverage	accumulated				
/work/CoverMeter/regress/cmview/NEC/coverage/b0_err	33.80	33.80	33.80	33.80				
/work/CoverMeter/regress/cmview/NEC/coverage/b1_err	6.66	13.35	33.61	40.30				
.../coverMeter/regress/cmview/NEC/coverage/buscont.temp	0.00	0.00	0.00	0.00				
/work/CoverMeter/regress/cmview/NEC/coverage/buscont	2.76	28.87	15.94	42.05				

Figure 8-25 Weighted Coverage for Detailed Information for Each Test

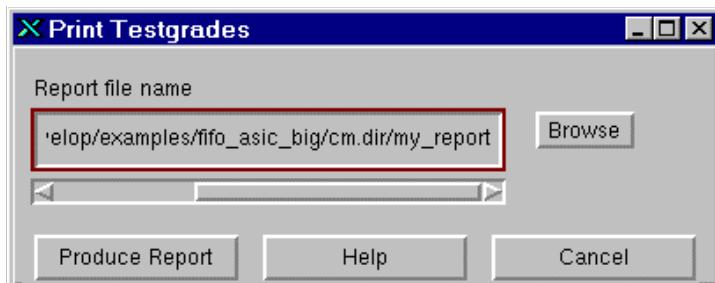
Test Name	Statement Coverage		Condition Coverage		Toggle Coverage		Weighted	
	statement	condition	toggle	weighted				
/work/CoverMeter/regress/cmview/NEC/coverage/b0_err	36.76	28.57	39.02	33.80				
/work/CoverMeter/regress/cmview/NEC/coverage/b1_err	36.76	28.19	39.02	33.81				
.../coverMeter/regress/cmview/NEC/coverage/buscont.temp	63.47	0.00	0.00	0.00				
/work/CoverMeter/regress/cmview/NEC/coverage/buscont	0.00	15.42	16.46	15.94				

How to Print the Test Grading Information

1. Click the Report button.

The Print Testgrades window is displayed as shown in [Figure 8-26](#).

Figure 8-26 Print Testgrades Dialog Box



2. Enter a name for the report in the Report file name field and click the Produce Report button.

Closing the Test Grading Window

Click the Exit button to close the Test Grading window.

Getting Help

Click the Help button to access the online help facility.

Using the Automatic Grading Window

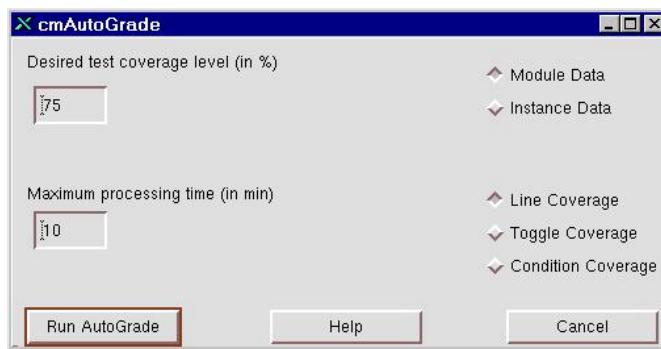
You can use the Automatic Grading window to obtain a minimum set of tests for a desired coverage goal.

The general procedure for opening the Automatic Grading Window is as follows:

1. Choose the **Tools > Autograding** menu command in the cmView main window.

The Automatic Grading dialog box is displayed:

Figure 8-27 Automatic Grading Dialog Box



2. Select the desired settings and click Run AutoGrade.
3. Upon completion of the automatic grading process, a test grading window is displayed. The test grading window is similar to the test grading windows described in "[Test Grading](#)".

The information in the test grading window reflects the selections you made for automatic grading.

The message, Tests optimally ordered for n % coverage, where n is the value you entered for the desired test coverage level in the cmAutograde window, is displayed at the bottom of the test grading window to indicate that the data was produced by the autograding process.

4. Click the Report button in the test grading window to produce a report of the grading information (see "[How to Print the Test Grading Information](#)").

User Preferences

User preferences enable you to modify the coverage levels, change the color representations of coverage levels and highlighting, control how you save your work in the current cmView session, how the log file is handled, and assign different weights to the different coverage types, if desired.

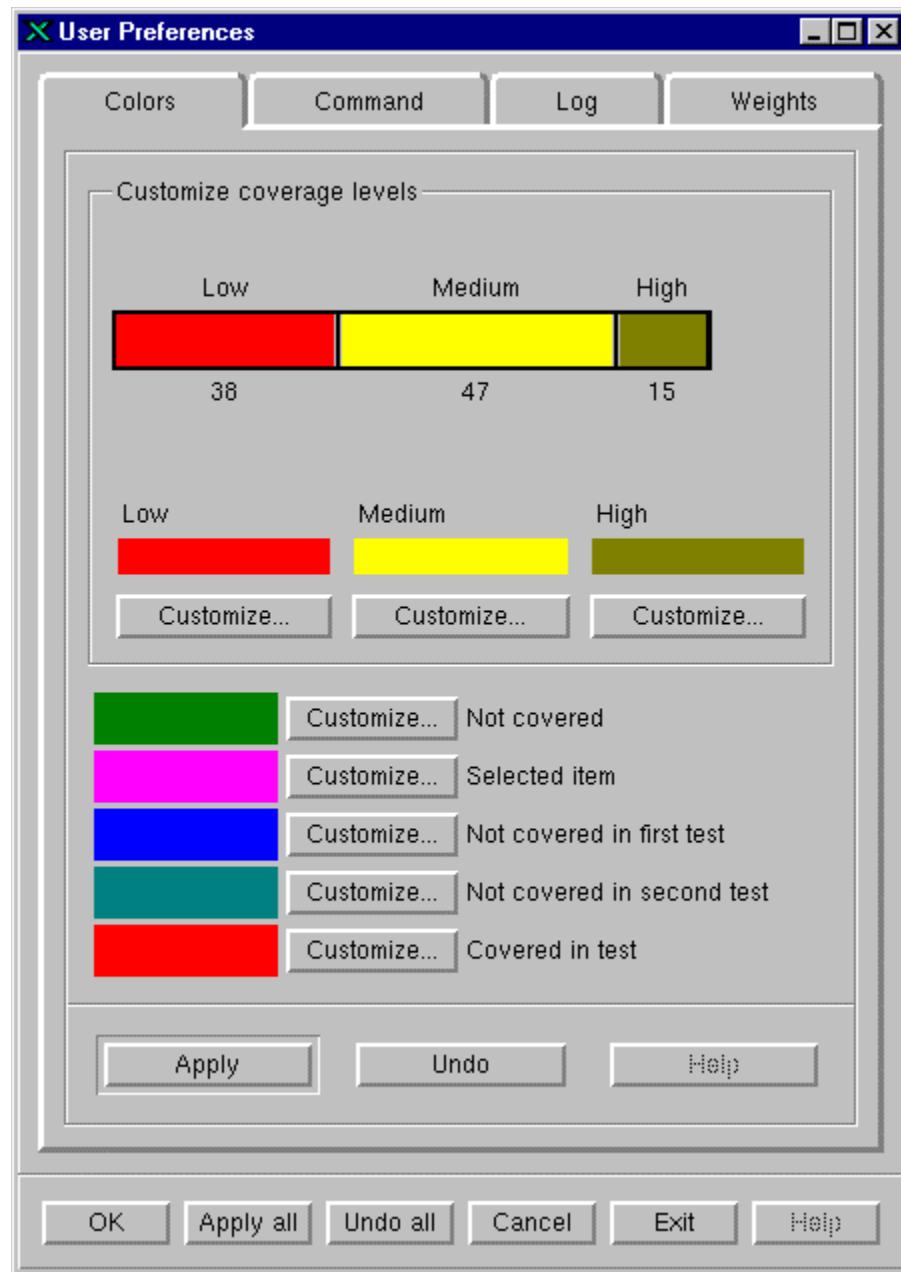
The User Preferences window displays a folder with four tabs. To initiate the dialog and access this folder, select the **Options > Preferences** menu command.

[Figure 8-28](#) illustrates the User Preferences window.

The Colors Tab

The Colors tab is the default tab in the User Preferences dialog (see [Figure 8-28](#)). When the Colors tab is active, the upper portion of the display contains a set of controls you can use to customize coverage levels. These include a bar to change the coverage levels followed by buttons to control the colors in which the coverage levels are displayed. The bottom portion of the display contains controls you can use to customize the colors in which various lines are displayed in different windows.

Figure 8-28 The Colors Tab in the User Preferences Folder



The section entitled, “[The Summary Pane](#)”, describes rows in the summary pane of the Coverage Window that are color coded according to the level of coverage for the instance or module. You can change these levels and their corresponding colors using the

Colors tab. In the coverage level bar, you can change the percentage for low, medium and high coverages by selecting and dragging either of the two dividing sliders. As you do this, you will notice the percentages change along the bottom to reflect the new coverage levels. By default, all instances with a low coverage of less than 20% are displayed in red; instances with medium coverage of between 20% and 80% are displayed in yellow; and instances with a high coverage greater than 80% are displayed in green.

To change the colors associated with the different coverage levels, click Customize below the desired coverage level. This displays the Color Selector dialog as shown in [Figure 8-29](#). You can change the color by selecting another color from the existing palette.

Figure 8-29 Color Selector Dialog Box



The controls in the bottom half of this window are used to customize the colors in which different items are displayed in the Annotated and Toggle Summary windows. To change the color in which uncovered items are displayed, click the Customize button for this color. This displays the same color selector dialog shown in [Figure 8-29](#). Controls to customize colors for the following items are provided:

- Item not covered
- Currently selected item

- For Diff Coverage, item not covered by first test
 - For Diff Coverage, item not covered by second test
 - For Incremental and Condition Coverage, item covered by the current test
-

The Command Tab

The Command tab is shown in [Figure 8-30](#):

Figure 8-30 The Command Tab in the User Preferences Folder



The Command feature allows you to capture your work in the current cmView session for use in a later cmView session, if desired. By default, the work in the current session is not saved. Consequently, you must enable command file generation to capture your work.

When command file generation is enabled, cmView writes your work to a script file. The script file, or command file, can then be used in a later cmView session. The following options are provided:

- Disable command file generation

- Discard command file when exiting (the default)
- Keep only current version of command file
- Back up the most recent version of the command file

If you choose either of the last two options, you can select the script file name either by typing it in the box provided or by clicking the Browse button. If you click Browse, the standard File Selection Box appears. In this case, specify the name of the script file and click OK.

The Log Tab

The Log tab is shown in [Figure 8-31](#):

Figure 8-31 The Log Tab in the User Preferences Folder



By default, cmView writes detailed information for errors and warnings that occur during the current session to a log file and deletes the log file when the session ends. If you have no need for the log file and you want to save some processing time, you can disable log file generation and avoid some unnecessary overhead.

If, however, you want to retain the log file, you can use one of the available Log options to keep, backup, or append the log file to an existing file. The following options are provided:

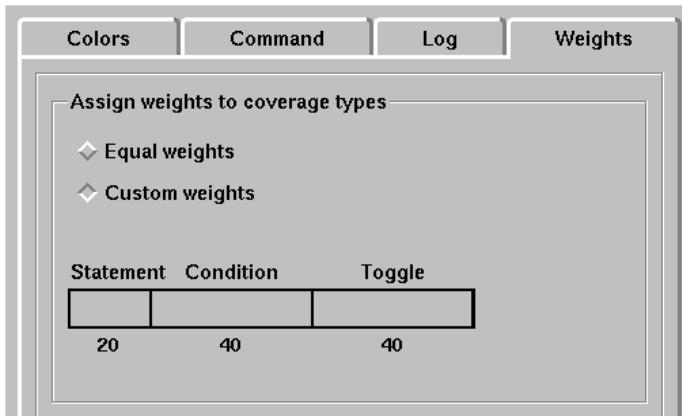
- Disable log file generation
- Discard log file when exiting (the default)
- Keep only current version of log file
- Back up the most recent version of the log file
- Append log file to previous log file

If you choose any of the last three options, you can select the log file name either by typing it in the box provided or clicking the Browse button. If you click Browse, the standard File Selection Box appears. In this case, specify the name of the log file and click OK.

The Weights Tab

The Weights tab is shown in [Figure 8-32](#). Weights assigned to the different types of coverages apply only to test grading. By default, all weights are equal.

Figure 8-32 The Weights Tab in the User Preferences Folder



You can use this tab to assign different weights to the different coverage types. In the coverage weights bar, you can change the percentage for statement, condition and toggle coverages by selecting and dragging either of the two dividing sliders. As you do so, you will notice the percentages change along the bottom to reflect the new weights.

9

Real Time Coverage

VCS has system functions for a real time API that enable you to write test fixture modules that can do the following:

- Find out what type of coverage VCS is running for part of a design
- Start or stop a type of coverage for part of a design
- Determine how close you are to recording all the coverage data that you can in a simulation run for a part of the design.

The parts of the design for which you can determine this information or start or stop collecting coverage information can be any of the following:

Note:

You can specify multiple parts of the design.

- All instances of a specific module definition

- All instances of a specific module definition and all module instances hierarchically under these module instances
- A specific module instance
- A specific module instance and all module instances hierarchically under this module instance

The user-defined system functions that enable you to do these things are as follows:

`$cm_coverage`

Starts or stops monitoring for a particular type of coverage or returns a value telling you what type of coverage data VCS is gathering for a particular part of the design.

See “[The \\$cm_coverage System Function](#)” on page 472.

`$cm_get_coverage`

Used with `$cm_get_limit`, its return value represents how much of a specific type of coverage data VCS has gathered so far.

See “[The \\$cm_get_coverage System Function](#)” on page 476.

`$cm_get_limit`

Used with `$cm_get_coverage`, its return value represents the maximum amount of a specific type of coverage data VCS can gather for part of a design. See “[The \\$cm_get_limit System Function](#)” on page 478.

You use the `$cm_coverage` system function to both determine what type of coverage information VCS is gathering and to enable or disable gathering that information.

You can use the `$cm_get_coverage` and `$cm_get_limit` system functions together, comparing their return values, to see how close you are to gathering all the coverage information you can. There are arguments for specifying the type of coverage and for looking at how close you are for only part of the design.

The arguments to these system functions are for the most part integer values. Synopsys provides a source file that defines text macros for these integer values so that you can more easily see what you are specifying in these system functions. For example:

```
$cm_coverage(3,4,11,"top.dev1");
```

and

```
$cm_coverage(`CM_CHECK,`CM_FSM,`CM_HIER,"top.dev1");
```

Both specify checking to see if VCS is gathering FSM coverage for the part of the design that is hierarchically under module instance top.dev1. The second example is a lot more obvious because it uses the macros defined in the \$VCS_HOME/include/CoverMeter.vh file.

To use the text macros in the CoverMeter.vh file, do the following:

- Specify including the file with the `include compiler directive:

```
'include "CoverMeter.vh"
```
- Tell VCS where to look for this file with the +includer compile-time option:

```
+includer+$VCS_HOME/include
```

Note:

These system functions do not work if you don't enable monitoring for coverage at runtime with the -cm runtime option.

The \$cm_coverage System Function

The \$cm_coverage system function starts or stops VCS from monitoring for a particular type of coverage, or returns a value telling you what type of coverage data VCS is gathering, for a particular part of the design. Its syntax is as follows:

```
$cm_coverage(mode, type, include_hierarchy,  
"module_or_instance", ...)
```

Here:

mode

Specifies starting, stopping, or checking for a certain type of coverage. These modes, their text macros, and their integer equivalents are as follows:

- | | | |
|-----------|---|---|
| 'CM_START | 1 | Specifies starting the type of coverage specified in the <i>type</i> argument for the part or the design specified in the <i>include_hierarchy</i> and <i>module_or_instance</i> argument. |
| 'CM_STOP | 2 | Specifies stopping the type of coverage specified in the <i>type</i> argument for the part or the design specified in the <i>include_hierarchy</i> and <i>module_or_instance</i> argument. |
| 'CM_CHECK | 3 | Specifies looking to see if VCS is gathering coverage information for the type of coverage specified in the <i>type</i> argument for the part or the design specified in the <i>include_hierarchy</i> and <i>module_or_instance</i> argument.
The return value from this system function, when you specify this mode, tells you whether VCS is gathering this information. |

type

Specifies the type of coverage you want to start, stop, or check for. These types, their text macros, and their integer equivalents are as follows:

'CM_SOURCE	1	Line coverage
'CM_CONDITION	2	Condition coverage
'CM_TOGGLE	3	Toggle coverage
'CM_FSM	4	FSM coverage

include_hierarchy

Specifies whether or not you want to also start, stop, or check for the type of coverage in the design hierarchy under all instances of the module or modules you specify in the *module_or_instance* argument.

The text macros and integer equivalents that you can enter for this argument are as follows:

'CM_MODULE	10	Specifies <i>not</i> also starting, stopping, or checking for coverage in the design hierarchy under the instances you specify in the <i>module_or_instance</i> argument.
'CM_HIER	11	Specifies also starting, stopping, or checking for coverage in the design hierarchy under the instances you specify in the <i>module_or_instance</i> argument.

module_or_instance

The module identifier (or name) of a module or the hierarchical name of a module instance. When you specify a module identifier, you specify all instances of this module.

Always enclose this argument in quotation marks.

You can specify more than one module identifier or module instance. If you do, separate these arguments with commas and enclose each in quotation marks, for example:

"*module_identifier*", "*module_instance_hierarchical_name*"

Consider the following matrix for the *include_hierarchy* and *module_or_instance* arguments:

	Specify a module identifier in the <i>module_or_instance</i> argument	Specify a hierarchical name for a module identifier in the <i>module_or_instance</i> argument
Specify 'CM_MODULE or its equivalent integer for the <i>include_hierarchy</i> argument	Applies to all instances of the specified module but none of the module instances hierarchically under these instances.	Applies only to the specified instance
Specify 'CM_HIER or its equivalent integer for the <i>include_hierarchy</i> argument	Applies to all instances of the specified module and all the module instances hierarchically under these instances.	Applies to the specified instance and all module instances hierarchically under the specified instance.

Return Values

The \$cm_coverage system function returns positive or negative integer values. The meaning of these values is determined by the *mode* argument. The \$VCS_HOME/include/CoverMeter.vh file also contains text macros for these return values.

When the *mode* argument is 'CM_CHECK the text macros for, integer values of, and meaning of the return values are as follows:

'CM_NOERROR	0	VCS is gathering the type of coverage information specified with the <i>type</i> argument for the module instances specified with the <i>include_hierarchy</i> and <i>module_or_instance</i> arguments.
'CM_ERROR	-1	There is an invalid argument.
'CM_NOCOV	-2	There is no coverage data of the type specified for the instances specified.

'CM_PARTIAL	-3	There is coverage data of the type specified for some, but not all, of the instances specified
-------------	----	--

When the *mode* argument is 'CM_START the text macros for, integer values of, and meaning of the return values are as follows:

'CM_NOERROR	0	VCS has started gathering the type of coverage information specified with the <i>type</i> argument for the module instances specified with the <i>include_hierarchy</i> and <i>module_or_instance</i> arguments.
'CM_ERROR	-1	There is an invalid argument.
'CM_NOCOV	-2	VCS cannot gather the type of coverage information specified with the <i>type</i> argument. Usually this is because the source code is not compiled for the type of coverage specified.
'CM_PARTIAL	-3	VCS cannot gather the type of coverage information specified with the <i>type</i> argument for all instances specified, but it can for some. Usually this means only some of the specified instances are not compiled for this type of coverage, while other specified instances are. The coverage data collected will be no different than if you specified only the instances compiled for the specified coverage type.

When the *mode* argument is 'CM_STOP the text macros for, integer values of, and meaning of the return values are as follows:

'CM_NOERROR	0	Coverage successfully stopped
'CM_ERROR	-1	There is an invalid argument.

The \$cm_get_coverage System Function

You can use the `$cm_get_coverage` system function to return a value that you compare with the return value from the `$cm_get_limit` system function to determine how close you are to gathering all the coverage data you can gather in the current simulation, of a specified coverage type, and from a specified part of the design.

The return value from this system task is an integer representing how much of the specified type of coverage information VCS has gathered so far for the specified part of the design.

The syntax of the `$cm_get_coverage` system function is as follows:

```
$cm_get_coverage(type, include_hierarchy,  
"module_or_instance",...)
```

Here:

type

Specifies the type of coverage. These types, their text macros, and their integer equivalents are as follows:

'CM_SOURCE	1	Line coverage
'CM_CONDITION	2	Condition coverage
'CM_TOGGLE	3	Toggle coverage
'CM_FSM	4	FSM transition coverage
'CM_FSM_TRANS	4	FSM transition coverage (same as 'CM_FSM)
'CM_FSM_STATE	5	FSM state coverage

include_hierarchy

Specifies whether you want to see how close you are to gathering all the coverage information you can in the design hierarchy under the module instances you specify in the *module_or_instance* argument.

The text macros and their integer equivalents for specifying whether to include the subhierarchies under these instances are as follows:

'CM_MODULE	10	Specifies <i>not</i> including in the design hierarchy under the instances or instance of the module or instance you specify in the <i>module_or_instance</i> argument.
'CM_HIER	11	Specifies including the design hierarchy under the instances or instance of the module or instance you specify in the <i>module_or_instance</i> argument.

module_or_instance

The module identifier (or name) of a module or the hierarchical name of a module instance.

Note:

Less detail is provided here for the *include_hierarchy* and *module_or_instance* arguments than is provided for the \$cm_coverage system function. This is to avoid repeating a lot of details because these arguments work the same way for this system function as for the \$cm_coverage system function. You use these arguments to specify, for which part of the design, the operation (in this case determining how much coverage data VCS has gathered so far) applies.

Return Values

If there are no error conditions, the `$cm_get_coverage` system function returns either a zero or a positive integer value. You compare this return value with the return value of the `$cm_get_limit` system function. When they return matching positive integer values for a specific type of coverage and for a matching part of the design, then the current simulation run yields no more additional coverage data. Without this comparison, which positive integer is returned from this system function has no particular significance.

When there is an error this system function returns negative integer values. The text macros for, integer values of, and meaning of the return values are as follows:

'CM_ERROR	-1	There is an invalid argument.
'CM_NOCOV	-2	The specified type of coverage data is not available in this simulation run.

The `$cm_get_limit` System Function

You use the `$cm_get_limit` system function to return a value that you compare with the return value from the `$cm_get_coverage` system function to determine how close you are to gathering all the coverage data you can gather in the current simulation of a specified coverage type and from a specified part of the design.

The return value from this system task is a integer representing how much of the specified type of coverage information, for the specified part of the design, VCS could gather during the simulation. In

contrast, the return value of the `$cm_get_coverage` system function is an integer representing how much coverage information of the specified type VCS has gathered so far.

The syntax of the `$cm_get_limit` system function is, with the exception of the system function name, identical to the syntax of the `$cm_get_coverage` system function.

```
$cm_get_limit(type, include_hierarchy,  
"module_or_instance",...)
```

The arguments, their text macros, and integer values are identical to those of the `$cm_get_coverage` system function. When you use these two system functions as intended, you use the same arguments to specify the type of coverage, whether to include the design hierarchy under the specified module definitions and instance hierarchical names and matching module definitions and instance hierarchical names.

Examples

[Example 9-1](#) shows how to use this API to stop simulation when you have sufficient coverage data. [Example 9-2](#) show how to monitor how close you are to full coverage data.

Example 9-1 Stopping Simulation When You Have Sufficient Coverage Data

```
'timescale 1 ns / 1 ns

module top;
reg r1,r2;
wire w1, w2;
integer coverage_on_or_off,check_coverage,get_coverage,get_limit;

design design1 (w1,r1);

initial
begin
#0 get_limit=$cm_get_limit('CM_SOURCE,'CM_MODULE,"top.design1");
#5 coverage_on_or_off= $cm_coverage('CM_STOP,'CM_SOURCE,'CM_HIER,"top");
#5 coverage_on_or_off=
$cm_coverage('CM_START,'CM_SOURCE,'CM_MODULE,"top.design1");
r1=0;
#1000 $finish;
end

always
begin
#25 r1=~r1;
get_coverage=$cm_get_coverage('CM_SOURCE,'CM_MODULE,"top.design1");
if (get_coverage>=get_limit*0.9)
$finish;
end
endmodule
```

In Example 9-1, the initial block does the following:

1. Assigns to integer `get_limit` the return value of `$cm_get_limit`, which represents how much line coverage data for module instance `top.design1` VCS could gather during the simulation.
2. Turns off line coverage for the entire design after 5 time units. The delay is specified so that turning off line coverage happens after it is started by the `CoverMeter.tasks` file.

3. Turns on line coverage 5 time units after that but only for module instance top.design1.
4. Initializes a stimulus reg.
5. Schedules simulation to end at simulation time 1010.

Also in [Example 9-1](#), the always block does the following:

1. Toggles the stimulus reg.
2. Assigns to integer get_coverage the return value of \$cm_get_coverage, which represents how much line coverage data for module instance top.design1 VCS has gathered so far.
3. Looks to see if the integer assigned to get_coverage is greater than or equal to 90% of the integer assigned to get_limit, and if it is, terminates the simulation.

Example 9-2 Displaying How Close You Are To Full Coverage Data

```
initial
begin
    covdata = $cm_coverage('CM_CHECK, 'CM_SOURCE, 'CM_HIER, "i1");
    $display($time,, "Source coverage check: %d", covdata);
#1; // wait until after the coverage stuff is initialized
    covdata = $cm_coverage('CM_CHECK, 'CM_SOURCE, 'CM_HIER, "i1");
    $display($time,, "Source coverage check: %d", covdata);
    covdata = $cm_get_limit('CM_SOURCE, 'CM_HIER, "i1");
    $display($time,, "Source limit : %d", covdata);
#999;
forever
begin
    covdata = $cm_get_coverage('CM_SOURCE, 'CM_HIER, "i1");
```

In [Example 9-2](#), the initial block does the following:

1. At time 0, assigns to integer `covdata` the return value of the `$cm_coverage` system function that is checking to see if VCS is gathering line coverage information about module instance `i1` and the hierarchy under `i1`.
2. Displays the return value.
3. Waits 1 time unit and then once again assigns to integer `covdata` the return value of the `$cm_coverage` system function that is checking for the same thing.
4. Displays the return value again. This value should be different now that the `CoverMeter.tasks` file has started VCS for line coverage.

5. Assigns to integer covdata the return value of the \$cm_get_limit system function for the same type of coverage and the same part of the design.
6. Displays the return value from the \$cm_get_limit system function.
7. After 999 time units begins a forever loop that does the following:
 - Assigns to integer covdata the return value of the \$cm_get_coverage system function for the same type of coverage and the same part of the design.
 - Displays this return value.
 - Waits 100,000 time units before beginning the loop again.

Real Time Coverage

9-484

Glossary

condition A subexpression.

coverage Test coverage. The execution of a line of code for a given function under test. Also, refers to the extent to which test cases being run against the design exercise the code.

coverage file An internal user-defined Verilog file, for example, coverage.v, in which you gather a set of cmMonitor tasks to organize the monitoring activity. Typically, the coverage file contains only cmMonitor tasks.

Coverage Metrics database During compilation, VCS and VCS MX create a directory that contains subdirectories for intermediate data files (test files), reports, and information files about the design that are read by VCS, VCS MX and cmView. By default this file is named simv.cm.

declaration information file cm.decl_info file created by cmMonitor, contains declarations of the hierarchy and the port. If you don't have this file, you can't run cmView.

design information file The cm.decl_info file is written by VCS about the design. cmView reads this file to understand the intermediate data files.

implied else An implied or *missing* else refers to the action taken when the conditional test fails for an `if` statement written without an `else` clause.

information file directory The .simv.cm/db directory that VCS creates and in which it writes information files about the design.

intermediate data file Created by VCS. Contains the coverage results data generated by cmMonitor during the simulation for a particular type of coverage. Also referred to as results data file.

missing else See *implied else*.

results data file See *intermediate data file*.

sensitized Refers to a certain path being taken based on the setting of one of a simultaneous set of values related to each other by a logical AND or a logical OR.

subexpression A condition.

test file See *intermediate data file*.

vector Refers to a set of values used to test an expression.

Index

Symbols

--synopsys coverage_off pragma 1-32
--synopsys coverage_on pragma 1-32
-cm branch 8-422
-cm cond 8-421
-cm fsm 8-421
-cm line 8-421
-cm path 8-421
-cm tgl 8-421
-cm_annotation 1-56
-cm_annotation cond 8-422
-cm_annotation fsm 8-422
-cm_autograde 1-45
-cm_cond 4-203
-cm_cond_branch 8-423
-cm_constfile 4-218
-cm_dir 1-4
-cm_exclude 4-234, 8-423
-cm_glitch 1-34
-cm_hier 1-43
-cm_ignorepragmas 3-158, 7-400
-cm_libs 1-6
-cm_line contassign 2-86
-cm_log 1-20
-cm_map 1-70
-cm_name 1-19, 1-39
-cm_nocasedef 1-44, 8-425
-cm_noconst 1-7
-cm_report 8-424
-cm_report annotate 1-55, 2-95
-cm_report annotate+module 2-99
-cm_report cond_ids 4-231
-cm_report simtime 1-50
-cm_report summary 1-52
-cm_report summary_on_top 1-50
-cm_report testlists 1-55, 2-111, 2-112, 4-248
-cm_report testlists_max 4-249
-cm_report worst_first 1-51
-cm_report_testlist_max 1-55
-cm_report_testlists_max 8-425
-cm_summary instance 1-53, 8-426
-cm_summary noemail 1-54, 8-426
-cm_tests 1-41, 8-425
-cm_tgl 3-150
-cm_timelimit 1-47
-cm_verbose 1-51, 8-426
-o 1-4
-v 1-6
-y 1-6
\$cm_coverage system function 9-472
\$cm_get_coverage system function 9-476

\$cm_get_limit system function 9-478
&&
 in condition coverage 4-199, 4-200
.decl_info 8-436
+incdir 9-471
/* VCS enum enumeration_name */ pragma
5-307
/* VCS state_vector signal_name */ pragma
5-307
//VCS coverage off pragma 1-28
//VCS coverage on pragma 1-29
//VCS exclude_file pragma 1-29
//VCS exclude_module pragma 1-29

A

Action Menu
 in toggle coverage 3-182
Action menu 8-448–8-451
adding a test 8-437–8-440
All Modules
 in the Create Reports dialog box 8-445
allops argument to the -cm_cond option 4-205,
4-206, 4-208, 4-238
Annotate Source Files
 in the Create Reports dialog box 8-444
Annotated Directory
 in the Create Reports dialog box 8-446
annotated directory 1-2, 2-95
annotated source files 2-95–2-99, 4-263, 8-444
Annotated window
 for condition coverage 4-260–4-269
 for line coverage 2-129–2-135
automatic grading
 graphical user interface 8-459
 in batch mode 1-44
Automatic Grading window 8-459–8-460

B

basic argument to -cm_cond 4-215, 4-230
basic argument to the -cm_cond 4-229
basic argument to the -cm_cond option 4-204
blocks
 code blocks in line coverage 2-82–2-86
 in line coverage 2-88
Branch coverage
 viewing results in the Coverage Metrics GUI
 7-400–7-416
Branch Coverage window 7-402
Branch Summary window 7-404

C

case statement
 in line coverage 2-86
celldefine 1-6
cells
 compiling for coverage 1-6
 specifying coverage for 1-6
-cm branch 8-422
-cm cond 8-421
-cm fsm 8-421
-cm line 8-421
-cm path 8-421
-cm tgl 8-421
-cm_annotate 1-56
-cm_annotate cond 8-422
-cm_annotate fsm 8-422
-cm_autograde 1-45, 8-422
-cm_cond 4-203
-cm_cond_branch 8-423
-cm_constfile 4-218
\$cm_coverage system function 9-472
-cm_dir 1-4, 8-423
-cm_elfile 8-423
-cm_exclude 4-234, 8-423
-cm_fsmopt allowTemp 5-315

-cm_fsmopt allowTmp 5-315
-cm_fsmopt optimist 5-311
-cm_fsmopt report2StateFsms 5-316
-cm_fsmopt reportvalues 5-315
-cm_fsmopt reportWait 5-317
-cm_fsmopt reportXassign 5-319
-cm_fsmopt sequence 5-297
-cm_fsmresetfilter 5-321
\$cm_get_coverage system function 9-476
\$cm_get_limit system function 9-478
-cm_glitch 1-34
-cm_hier 1-43, 8-423
-cm_ignore pragmas 3-158, 7-400
-cm_libs 1-6
-cm_line contassign 2-86
-cm_log 1-20, 8-424
-cm_mailsub 1-54
-cm_map 1-70, 8-424
-cm_name 1-19, 1-39, 8-424
-cm_nocasedef 1-44, 8-425
-cm_noconst 1-7
-cm_report 8-424
-cm_report annotate 1-55, 2-95
-cm_report annotate+instance 2-99
-cm_report annotate+module 2-99
-cm_report cond_ids 4-231
-cm_report disable_sequence 5-341
-cm_report disable_sequence_loop 5-341
-cm_report simtime 1-50
-cm_report summary 1-52
-cm_report summary_on_top 1-50
-cm_report testlists 1-55, 2-111, 2-112, 4-248
-cm_report worst_first 1-51
-cm_report_testlists_max 1-55, 2-112, 4-249,
8-425, 2-112
-cm_fsmresetfilter 5-321
-cm_summary instance 1-53, 8-426
-cm_summary noemail 1-54, 8-426
-cm_tests 1-41, 8-425
-cm_tgl 3-148, 3-150
-cm_timelimit 1-47
-cm_verbose 1-51, 8-426
cm.decl_info file 3-173, 4-256, 5-342, 7-401,
8-435
cmView
 command line options 8-420
 described 8-419
 viewing branch coverage results 7-400–
 7-416
 viewing condition coverage results 4-255–
 4-269
 viewing FSM coverage results 5-341–5-361
 viewing line coverage results 2-119–2-135
 viewing toggle coverage results 3-172–3-196
cmView files 1-39
cmView.hier_b file 7-383
cmView.hier_c file 4-241, 4-250
cmView.hier_f file 5-323, 5-331
cmView.hier_l file 2-94, 2-113–2-114
cmView.hier_p file 6-367
cmView.hier_t file 3-159, 3-164–3-165
cmView.long_b file 7-383
cmView.long_bd file 7-383
cmView.long_c file 4-241, 4-244
cmView.long_cd file 4-241
cmView.long_f file 5-323, 5-324–5-329
cmView.long_fd file 5-323
cmView.long_l file 2-93, 2-105–2-108
cmView.long_id file 2-94
cmView.long_p file 6-367
cmView.long_pd file 6-367
cmView.long_t file 3-158, 3-161–3-163
cmView.long_td file 3-158
cmView.mod_b file 7-384
cmView.mod_bd file 7-384
cmView.mod_c file 4-241, 4-251
cmView.mod_cd file 4-241

cmView.mod_f file 5-323, 5-332
cmView.mod_fd file 5-324
cmView.mod_I file 2-94, 2-114–2-115
cmView.mod_Id file 2-94
cmView.mod_p file 6-367
cmView.mod_pd file 6-367
cmView.mod_t file 3-159, 3-165
cmView.mod_td file 3-159
cmView.short_b file 7-384
cmView.short_bd file 7-384
cmView.short_c file 4-242, 4-252
cmView.short_cd file 4-242
cmView.short_f file 5-324, 5-334
cmView.short_fd file 5-324
cmView.short_I file 2-94, 2-116–2-117
cmView.short_Id file 2-94
cmView.short_p file 6-367
cmView.short_pd file 6-367
cmView.short_t file 3-159, 3-167–3-168
cmView.short_td file 3-159
condition coverage
 adding conditions 4-203
 defined 4-198–??
 disabling vector conditions 4-229–4-231
 enabling conditions from more operators
 4-208
 enabling event control conditions 4-206
 for all possible combinations of vectors
 4-226–4-227
 modifying 4-203
 viewing results in the Coverage Metrics GUI
 4-255–4-269
Condition Coverage window 4-257
Condition tab
 in condition coverage 4-267
conditional expression
 in condition coverage 4-198
conditional operator
 in condition coverage 4-198
configuration file
argument to the -cm_fsmcfg option 5-303
for FSM coverage 5-298
continuous assignment FSMs 5-289
continuous assignment statements
 in condition coverage 4-200
 in FSM coverage 5-289
coverage directory 1-2
coverage metrics database 1-2
coverage metrics directory 1-2
 specifying the name and location 1-4
CoverMeter.vh file 9-471
Create Statement Reports dialog box 8-443

D

db directory 1-2, 3-173, 4-256, 5-342, 7-401, 8-435
define compiler directives 5-281, 5-290
design file 2-120, 3-173, 4-256, 5-342, 7-401, 8-435
detailed view
 in condition coverage 4-262
 line coverage 2-127–2-128
Diff Coverage
 in Results Files menu 8-441
Difference
 in Test Grading window 8-456
Differential Coverage
 in Results Files menu 8-434
differential coverage 8-441–8-443

E

Each Test Data
 in Test Grading window 8-453
encoded FSMs 5-278
event argument to the -cm_cond option 4-205, 4-207
event controls
 in condition coverage 4-205, 4-206

F

for argument to the -cm_cond option 4-205, 4-210
forever statement
 in line coverage 2-85
FSM coverage
 coding styles 5-278
 continuous assignment FSMs 5-289
 criteria for an FSM 5-273
 one hot FSMs 5-286
 reachability analysis 5-352
 things to avoid 5-290
 using a configuration file 5-298
 viewing results in the Coverage Metrics GUI 5-341–5-361
FSM Coverage window 5-343
FSM Summary window 5-346
full argument to -cm_cond 4-215
full argument to the -cm_cond option 4-203, 4-208, 4-226, 4-227, 4-238
full vectors 4-203

G

glitch suppression 1-34
Go To Name
 in Test Grading window 8-454
Go To tab
 in condition coverage 4-267
 in line coverage 2-134

H

-help 8-426
hot bit FSMs 5-286

I

if statement
 in line coverage 2-80, 2-85
if-else statement

 in line coverage 2-85
+incdir 9-471
include 9-471
Incremental
 in Test Grading window 8-456
Incremental Coverage 4-263
 in Results Files menu 8-434
Information tab
 in condition coverage 4-266
 in line coverage 2-131
intermediate data files
 naming 1-19

L

libraries
 compiling Verilog libraries for coverage 1-6
 specifying coverage for 1-6
line coverage
 defined 2-80–2-86
 difference from statement coverage 2-83, 2-121
 reports 2-93–2-119
 viewing results in the Coverage Metrics GUI 2-119–2-146
Log File window 8-430
log files
 for coverage metrics during simulation 1-20
 from writing coverage metrics reports 1-20
logical conditions 4-203

M

mapping coverage for subhierarchies also used in another design 1-69–1-75
merged intermediate data files 1-38
merging intermediate data files 1-38
MISSING_ELSE
 missing code in line coverage 2-107
Module Definition Summary
 in the Create Reports dialog box 8-445

M
Module Hierarchy
 in hierarchy pane 8-430
Module Hierarchy Summary
 in the Create Reports dialog box 8-445
Module Instance Summary
 in the Create Reports dialog box 8-445
Module List
 in hierarchy pane 8-432
Multiple conditions 4-203

N
non-logical conditions 4-203

O
-o 1-4
one hot FSMs 5-286
Open Design dialog box 8-435
Options menu 8-451

P
path coverage
 reports 6-366
ports argument to the -cm_cond option 4-205, 4-212
Pragmas 2-125
pragmas for coverage metrics 1-28, 1-32
Print Testgrades window 8-459
procedural assignment statement
 in condition coverage 4-198
procedural assignment statements
 in condition coverage 4-200

R
reachability analysis for FSMs 5-352
Reachability tab
 in FSM coverage 5-352–5-355

R
Report File Name
 in the Create Reports dialog box 8-446
reports directory 1-2, 3-158, 4-241, 5-323, 6-366, 7-383
Result Files menu 8-434

S
Selected Module
 in the Create Reports dialog box 8-445
sensitized condition coverage 4-245–4-247
sensitized conditions 4-204
sensitized multiple condition coverage vectors 4-213–4-216
sensitized vectors 4-204
Sequences tab
 in FSM coverage 5-357–5-361
simv.cm directory 1-2, 1-4
sop argument to the -cm_cond option 4-205, 4-219, 4-223, 4-225
Source tab
 in FSM coverage 5-348, 5-350, 5-356–5-357
statement coverage
 difference from line coverage 2-83, 2-121
Statement Coverage window 2-120
States tab
 in FSM coverage 5-351
Statistics tab
 in condition coverage 4-266
 in line coverage 2-131
status bar
 displaying 8-428
 hiding 8-428
std argument to the -cm_cond option 4-204
std argument to the -cm_cond option 4-215
Summary tab
 in FSM coverage 5-355
--synopsys coverage_off pragma 1-32
--synopsys coverage_on pragma 1-32

T

task definition
 in line coverage 2-84
test autograding 1-45
 report example 1-48
Test Comparison
 in Test Grading window 8-455
Test coverage
 in Test Grading window 8-456
test files
 naming 1-19
test grading 1-44, 8-452–8-460
 printing the report 8-459
 using the test grading window 8-452–8-459
test grading window 8-452
 resizing columns 8-453
 Weighted tab 8-452
Test List
 displaying 8-450
test.branch file 1-19
test.cond file 1-19
test.fsm file 1-19
test.line 8-452
test.line file 1-19
testlists argument to the -cm_report option
2-112
test.path file 1-19
Tests tab
 in condition coverage 4-267
 in line coverage 2-133
test.tgl 8-452
test.tgl file 1-19
tf argument to the -cm_cond option 4-205,
4-211
toggle coverage
 defined 3-147
 excluding a net or register 3-188, 7-411
 reports 3-158–??
 viewing results in the Coverage Metrics GUI
 3-172–3-196

Toggle Coverage window 3-174
toolbar
 displaying 8-428
 hiding 8-427
Total Coverage
 in Result Files menu 8-434
Transitions tab
 in FSM coverage 5-347

U

User Preferences window
 Colors tab 8-461
 Command tab 8-464
 Log tab 8-465
 Weights tab 8-466

V

-v 1-6
//VCS coverage off pragma 1-28
//VCS coverage on pragma 1-29
//VCS exclude_file pragma 1-29
//VCS exclude_module pragma 1-29
Verilog libraries
 compiling for coverage 1-6
View Menu
 in condition coverage 4-268
 in line coverage 2-135
View menu 8-446–8-447

W

wait statement
 procedural delay is interpreted as 2-83
Weighted tab
 in test grading window 8-452
Weights tab
 in User Preferences 8-458
while statement
 in line coverage 2-80, 2-85

WHILE_FALSE

missing code in line coverage 2-107

Y

-y 1-6