



Integrated Coverage User Guide

Product Version 16.11

November 2016

© 2016 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522.

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

All other trademarks are the property of their respective holders.

Restricted Permission: This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

Patents: Cadence Product ICC, described in this document, is protected by U.S. Patents 5,095,454, 5,418,931, 5,606,698, 6,487,704, 7,039,887, 7,055,116, 5,838,949, 6,263,301, 6,163,763, 6,301,578, and 7,424,703.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as

to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Contents

1	10
Preface	10
Audience	10
Related Documentation	11
Typographical Conventions	11
Syntax Conventions	11
About Online Help	12
Launching Cadence Help	12
Getting Help for Cadence Help	13
Getting Help on Commands to Run Tools	13
Getting Help on Coverage Commands	13
Getting Help on Tool Messages	13
Customer Support	14
2	15
Integrated Coverage Overview	15
Coverage Types	15
Code Coverage	16
Functional Coverage	16
Integrated Coverage in the Design Flow	18
3	19
Block, Branch, and Statement Coverage	19
Block Coverage	19
Blocks in Verilog/VHDL	19
Branch Coverage	21
Branches in Verilog/VHDL	21
Verilog Example	23
VHDL Example	23

Statement Coverage	25
Scoring Block, Branch, and Statement Coverage	27
4	29
Expression Coverage	29
Sum-of-Products (SOP) Scoring	30
Full Combinational Coverage (FCC) Scoring	35
Control Scoring	37
Vector Scoring	43
Scoring Expression Coverage	50
Scoring of Operators	51
Scoring Events	53
5	57
Toggle Coverage	57
Types of Transitions	57
Signal Behavior Recognized in Integrated Coverage	59
Scoring Toggle Coverage	60
Scoring Toggle Coverage for SV Enum Signals	60
6	62
Automatic Marking of Coverage	62
COM Analysis	62
COM and Toggle Coverage	63
COM and Expression Coverage	63
COM and Block Coverage	64
COM and Coverage Calculation	65
7	70
Coverage Pragmas	70
Marking Coverage Items Using Pragmas	70
Pragmas and Coverage Calculation	72
Coverage Pragmas and translate_off/translate_on	73

Disabling/Enabling Implicit Block Scoring	74
Disabling Explicit Default Scoring	74
Ignoring Coverage Pragmas	75
8	76
Functional Coverage	76
Control-Oriented Using PSL/SVA Statements	79
Functional Coverage Points in PSL	79
Functional Coverage Points in SVA	83
Data-Oriented Using SystemVerilog Covergroup	85
Defining a Covergroup	86
Defining Coverpoints	91
Defining a Cross	130
Declaring a Cross of a Cross	144
Predefined Coverage Methods	148
Predefined Coverage System Tasks and System Functions	160
Specifying Coverage Options	161
Covergroups in Compilation Units	201
Covergroups in Interfaces	201
Covergroups in Program Blocks	202
Covergroups in Generate Blocks	203
Scope of Covergroup and Covergroup Instances	208
Scoring Functional Coverage	211
Functional Coverage using Incisive Assertion Library	212
9	213
FSM Coverage	213
Basic FSM Model	214
FSM Coverage Types	215
State Coverage	215
Arc Coverage	216
Modeling Styles Supported	218

Two Process Modeling Style	218
Single Process Modeling Style	228
One-Hot Encoding Style	229
FSM Extraction in ICC	231
Unsupported Scenarios for FSM Extraction	234
Unexpected FSM Results Scenarios	237
Scoring FSM Coverage	239
10	241
Generating Coverage Data	241
Generating Coverage Data using Multi-Step Simulation	241
Compiling the Design	242
Elaborating the Design	242
Simulating the Design	246
Generating Coverage Data using Single-Step Simulation	257
Support for Mixed HDL-SystemC (SC) / Analog Mixed Signals (AMS) Designs	259
Support for SystemVerilog Constructs	260
11	261
Analyzing Coverage Data	261
Launching IMC	261
Loading Runs	262
Listing Currently Loaded Run	262
Merging Runs	263
Ranking Runs	263
Generating Reports	263
Refining Metrics Data	263
Exiting IMC	264
12	265
Supported and Unsupported Functionality	265
Block Coverage	265
Expression Coverage	266

Toggle Coverage	267
FSM Coverage	268
Functional Coverage	269
13	270
Coverage Configuration File (CCF) Commands	270
include_ccf	273
set_libcell_scoring	274
set_merge_with_libname	275
set_parameterized_module_coverage	277
select_coverage	280
(de)select_coverage	285
set_inheritance_aware_class_coverage	291
set_com	294
set_com_interface	296
set_glitch_strobe	298
set_hit_count_limit	299
set_subprogram_scoring	299
set_code_fine_grained_merging	301
set_refinement_resilience	303
set_explicit_block_scoring	304
set_assign_scoring	304
set_branch_scoring	304
set_statement_scoring	304
deselect_macro	305
set_expr_scoring	306
set_expr_coverable_statements	311
set_toggle_strobe	312
set_toggle_limit	313
set_toggle_includex	313
set_toggle_includez	313
set_toggle_noports	315

set_toggle_portsonly	315
set_toggle_scoring	316
deselect_fsm	331
set_fsm_reset_scoring	331
set_fsm_arc_scoring	332
set_fsm_scoring	333
Deprecated CCF Commands/Options	353
Using CCF commands across simulations	354
CCF Commands Supported in the MSIE Flow	356
CCF Commands Unsupported in the MSIE Flow	357

Preface

Cadence's Integrated Coverage has well-defined coverage metrics to perform a thorough analysis of verification completeness. Coverage metrics can be classified into code coverage and functional coverage. This manual describes Integrated Coverage and focuses on coverage data generation and introduces Integrated Metrics Center (IMC) as a data analysis tool.

For more details on IMC, see the *Integrated Metrics Center User Guide* in the Metric-Driven Verification (MDV) release.

This preface discusses the following topics:

- [Audience](#)
- [Related Documentation](#)
- [Typographical Conventions](#)
- [Syntax Conventions](#)
- [About Online Help](#)
- [Customer Support](#)

Audience

This manual is written for design and verification engineers who want to identify the areas of design that:

- Have not been fully tested.
- Did not meet the desired coverage criteria.

The prerequisites for using this manual are:

- Working knowledge of HDL and design experience using Verilog or VHDL.
- Knowledge of the Xcelium™ Single Core simulator.

Related Documentation

- The *Integrated Coverage Quick Reference Guide* discusses the available commands for coverage data generation and analysis.
- The *Integrated Metrics Center User Guide* provides in-depth information on merging metrics data, displaying reports, marking metrics items, and analyzing metrics data using IMC.
- The *Integrated Coverage UCIS User Guide* describes a programming procedural interface to UCIS gives an overview of its information model, explains how to create VHPI applications using the C or C++ programming language, provides examples of VHPI applications.

Typographical Conventions

The table lists the typographical conventions used in this manual.

Table 3-1 Typographical Conventions

Font	Meaning	Example
<i>italic</i>	Titles of books	For more information, refer <i>Integrated Coverage Methodology Guide</i> .
literal	Text that you must enter literally in source files or on the command line	% xmvlog *.v
< <i>italic literal</i> >	User-defined arguments for which you must substitute a name or a value.	% xmvlog -f <filename>

Syntax Conventions

The table lists the syntax conventions used in this manual.

Table 3-2 Syntax Conventions

Symbol	Meaning	Example
--------	---------	---------

	Vertical bars (OR-bars) separate possible choices for a single argument. They take precedence over any other operator.	command <i>argument1 argument2</i>
[]	Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices. You can choose one argument from the list.	command [<i>argument1 argument2</i>]
{ }	Braces are used with OR-bars and enclose a list of choices. You must choose one argument from the list.	command { <i>argument1 argument2</i> }
...	Three dots (...) indicate that you can repeat the previous argument. If they are used within brackets, you can specify zero or more arguments. If they are used without brackets, you must specify at least one argument, but you can specify more.	<i>argument...</i> specify at least one [<i>argument</i>] ... you can specify zero or more

About Online Help

The online documentation system is called Cadence Help.

Launching Cadence Help

- Set the path variable so that it includes the path to the executables, which are in `install_dir/tools/bin` and `install_dir/bin`, and then enter the following command at the prompt:
`cdnshelp &`
- You can also launch Cadence Help by selecting an item on the GUI *Help* menu or by clicking the *Help* button on forms and dialog boxes.

Getting Help for Cadence Help

After launching Cadence Help, press `F1` or choose *Help - Contents* to display the help page for Cadence Help.

Getting Help on Commands to Run Tools

You can display a list of options for any of the tools and utilities by typing the tool or utility name followed by the `-help` option as follows:

```
% tool_name -help
```

Example:

```
% xmvllog -help  
% xmvhdl -help  
% xmelab -help  
% xmsim -help
```

Getting Help on Coverage Commands

You can display a list of options for the command by:

```
% help <command_name>
```

Getting Help on Tool Messages

Use the *xmhelp* utility to display extended help on the brief messages generated by the *compiler*, *elaborator*, and *simulator*.

Syntax:

```
% xmhelp tool_name message_code
```

The `message_code` argument is not case-sensitive and can be in uppercase or in lowercase.

Examples:

```
% xmhelp xmvllog BADCLP  
% xmhelp xmvllog badclp  
% xmhelp xmelab CUVWSP
```

Customer Support

There are several ways that you can get help with your Cadence product:

- Customer support

Cadence is committed to keeping your design teams productive by providing answers to technical questions, the latest software updates, and education services to keep your skills updated. For information on Cadence support, go to the following web site:

<http://www.cadence.com/support>

- Cadence Online Support

Customers with a maintenance contract with Cadence can obtain current information on the tools at the following web site:

<http://support.cadence.com>

- Feedback about documentation

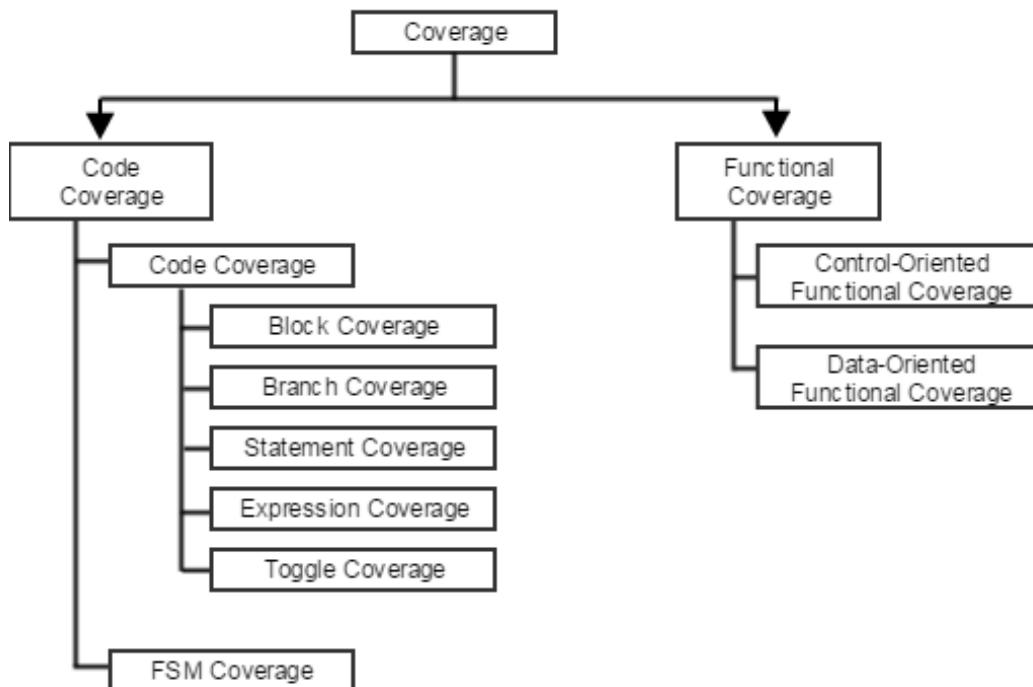
Contact Cadence Customer Support to file a PCR if you find:

- An error in a manual
- An omission of information in a manual
- A problem using the Cadence Help documentation system

Integrated Coverage Overview

Cadence's **Integrated Coverage** solution has well-defined coverage metrics to perform a thorough analysis of verification completeness. Coverage metrics can be classified as code coverage and functional coverage. Code coverage is a method of assessing how well the test cases test the intended behavior and to what extent they execute the design. Functional coverage focuses on functional aspects of a design and provides a very good insight on how the verification goals set by a test plan are being met. This chapter provides an overview of coverage types and describes Integrated Coverage in the design flow.

Coverage Types



Code Coverage

Code coverage is classified as:

- Code Coverage, which includes:
 - Block Coverage - Identifies the lines of code that get executed during a simulation run. It helps you determine if the various testbenches exercise the statements in a block. See [Chapter 2, "Block, Branch, and Statement Coverage"](#), for more information.
 - Branch coverage - Yields more precise coverage details than block coverage by obtaining coverage results for various branches individually. With branch coverage, a piece of code is considered 100% covered when each branch of a conditional statement has been executed at least once. See [Chapter 2, "Block, Branch, and Statement Coverage"](#), for more information.
 - Statement Coverage - Provides information on number of statements within a block. See [Chapter 2, "Block, Branch, and Statement Coverage"](#), for more information.
 - Expression Coverage - Provides information on why a conditional piece of code was executed. It provides statistics for all expressions in the HDL code. See [Chapter 3, "Expression Coverage,"](#) for more information.
 - Toggle Coverage - Provides information about the change of signals and ports, during a simulation run. It measures activity in the design, such as unused signals, signals that remain constant, or signals that have too few value changes. See [Chapter 4, "Toggle Coverage,"](#) for more information.
- FSM Coverage - Interprets the synthesis semantics of the HDL design and monitors the coverage of the FSM representation of control logic blocks in the design. With FSM coverage, you identify what states were visited and which transitions were taken. See [Chapter 8, "FSM Coverage,"](#) for more information.

Functional Coverage

Functional coverage is performed on user-defined functional coverage points, specified using PSL, SystemVerilog assertions, or covergroup statements. These coverage points specify scenarios, error cases, corner cases, and protocols to be covered and also specifies analysis to be done on different values of a variable.

Functional coverage is of following types:

- Control-oriented functional coverage - Is an extension of assertion-based verification and

identifies interesting functions directly. In Integrated Coverage, control-oriented functional coverage points are specified using PSL or SVA `assert`, `assume`, and `cover` directives. The coverage to be measured is directly specified using the PSL/SVA statements or is interpreted from them.

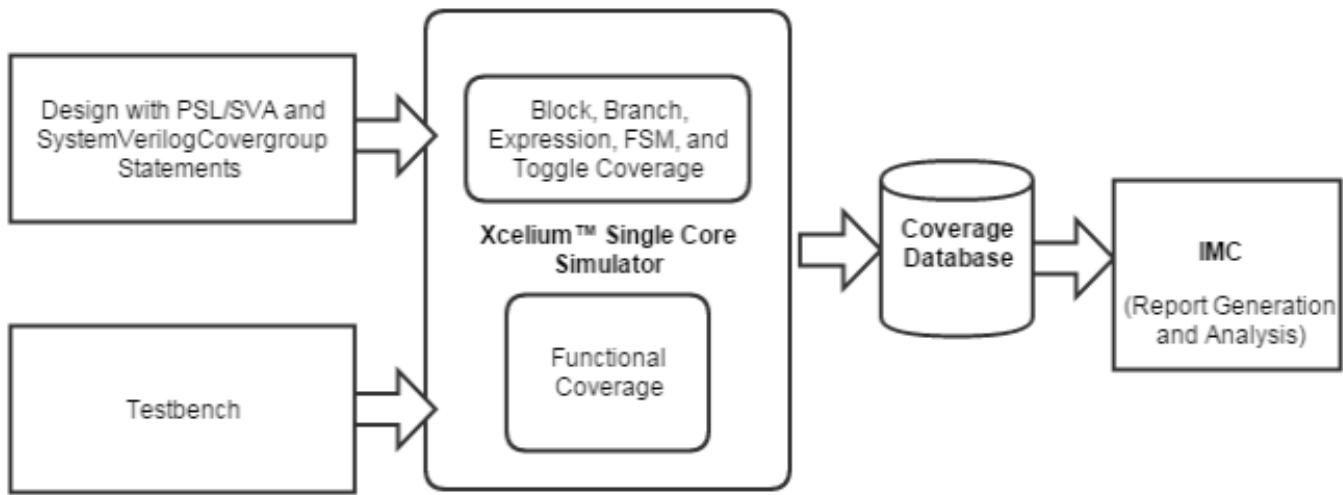
- Data-oriented functional coverage - Focuses on tracking data values. It includes coverage of variable values, binning, specification of sampling, and cross products. It helps design engineers to identify untested data values or subranges. In Integrated Coverage, data-oriented functional coverage is specified using SystemVerilog constructs.

See [Chapter 7, "Functional Coverage,"](#) for more information.

 To access the functional coverage capabilities, you need an Xcelium license.

Integrated Coverage in the Design Flow

You can perform an analysis of code coverage and functional coverage in a single run, as shown below.



The design and testbench are passed to Xcelium™ Single Core Simulator for coverage data generation. The coverage data is stored to a database, which later is analyzed using the coverage reporting tool, IMC. For details on coverage data analysis with IMC, see the *Integrated Metrics Center User Guide* in the Metric-Driven Verification (MDV) release.

See [Chapter 9, "Generating Coverage Data,"](#) for detailed information on coverage data generation.

Block, Branch, and Statement Coverage

This chapter provides an overview of block coverage, branch coverage, and statement coverage. Together these coverages tell you what code did not execute. The chapter also describes what is considered as a block and a branch in Verilog and VHDL code.

This chapter covers the following topics:

- [Block Coverage](#)
 - [Blocks in Verilog/VHDL](#)
- [Branch Coverage](#)
 - [Branches in Verilog/VHDL](#)
 - [Verilog Example](#)
 - [VHDL Example](#)
- [Statement Coverage](#)
- [Scoring Block, Branch, and Statement Coverage](#)

Block Coverage

Block coverage is a basic code coverage mechanism that identifies which blocks in the code have been executed and which have not. Block coverage identifies whether test scenarios exercise the statements in a block. In general, block coverage is an essential first step in the overall verification process.

Blocks in Verilog/VHDL

A block is a statement or a sequence of statements in Verilog/VHDL that executes with no branches or delays. Either none or all of the statements in a block are executed. The following procedural statements are considered as blocks:

- All statements between a matching *begin...end* pair and do not contain a flowbreak statement, or a single statement that could have *begin* and *end* statements added around it. A flowbreak

statement is one that can alter the normal execution sequence of procedural statements at a given time.

Flowbreak Statements in Verilog

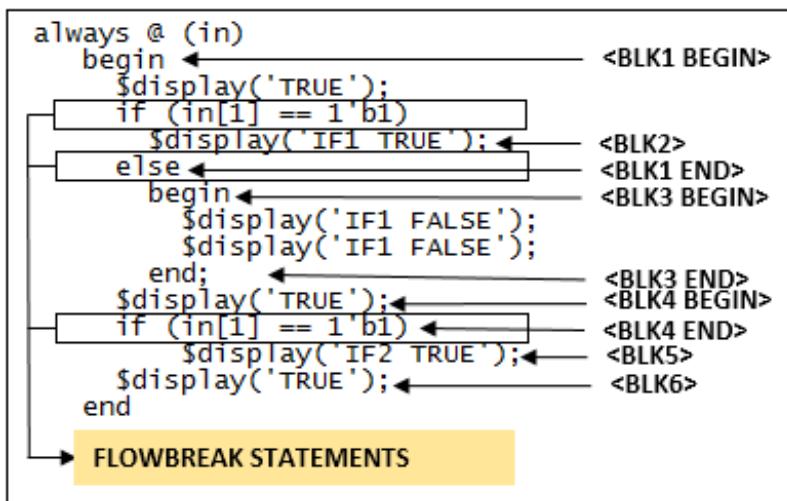
<code>if</code> statement	<code>case</code> item	<code>forever</code> statement
<code>repeat</code> statement	<code>while</code> statement	<code>for</code> statement
<code>wait</code> statement	delay control (#)	event control (@)
<code>disable</code> statement	tasks and function calls	

Flowbreak Statements in VHDL

<code>if</code> statement	<code>case</code> item	<code>loop</code> statement
<code>next</code> statement	<code>exit</code> statement	<code>return</code> statement
<code>wait</code> statement		

- All statements from a *begin* up to and including the completion of the next flowbreak statement.
- All statements from after the completion of a flowbreak statement up to and including the next flowbreak statement or until the final *end* statement.
- All statements between matching *fork...join* statements (Verilog only).

Blocks in behavioral Verilog code are within *initial* and *always* procedural blocks, tasks, and functions. Blocks in behavioral VHDL code are within *process* blocks and subprograms. The following figure displays blocks in a given Verilog code.



- Blocks in Verilog are not scored for continuous assignments (unless explicitly scored using the `set_assign_scoring` command).
- Blocks are not scored for concurrent assignment statements.
- Blocks are not defined for primitives.
- Blocks in VHDL are not scored for a function defined inside a package when the function is called from inside VHDL generate block.

Branch Coverage

Branch coverage complements block coverage by providing more precise coverage results for reporting coverage numbers for various branches individually. With branch coverage, a piece of code is considered 100% covered when each branch of a conditional statement has been executed.

Branches in Verilog/VHDL

A branch is a statement that is executed based on evaluation of a condition used in a conditional statement. Consider the following Verilog continuous assignment statement.

```
assign out2 = (cond1) ? 1'b1 : 1'b0;
```

The above statement has the following branches:

- If `cond1` evaluates to true, `out2` is assigned `1'b1`.
- If `cond1` evaluates to false, `out2` is assigned `1'b0`.

Ideally, coverage results should indicate coverage numbers for both the conditions individually. With block coverage, this statement is considered as a single block, which is considered covered if `cond1` is either true or false. With branch coverage, this statement is considered 100% covered when each branch of the conditional statement has executed.

Branches in Verilog are generated due to:

- `if`, `else if`, `else`, and implicit `else` (also scored as blocks)
- `case` blocks with explicit/implicit defaults (also scored as blocks)
- Single block corresponding to multiple `case` items (scored only if branch scoring is enabled)
- Ternary assignment conditions (scored only if branch scoring is enabled)

Branches in VHDL are generated due to:

- `if`, `elsif`, `else` blocks, and implicit `else` (also scored as blocks)
- `case` blocks (also scored as blocks)
- Conditional signal assignments (scored only if branch scoring is enabled)
- Selected signal assignments (scored only if branch scoring is enabled)

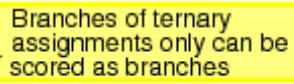
Verilog Example

The following figure displays a block annotated report to demonstrate what gets reported as a block and branch. This report displays source annotated with block coverage data. See the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release for details.

```

27      always @ (in )
28      1           begin                                // <BLK>
29      1           if ( in[2] == 1'b1 )                  // <BR> <BLK>
30      1           $display("IF2 TRUE");             // <BR> <BLK>
31      1           else
32      1           $display("IF2 FALSE");             // <BR> <BLK>
33      end
34
35      always @ (in )
36      1           begin                                // <BLK>
37      1           case(in)
38      1           3'b100:
39      1           $display("NODEF state 100");        // <BR> <BLK>
40      1           3'b010:
41      1           $display("NODEF state 010");        // <BR> <BLK>
42      endcase
43      end
44
45      always @ (in )
46      1           begin                                // <BLK>
47      1           case(in)
48      1           3'b100,
49      1           3'b010:
50      0           $display("DEF state 100, 010");    // <two branches> <one BLK>
51      default:
52      1           $display("DEF state DEF");        // <BR> <BLK>
53      endcase
54      end
55
56      1           assign out1 = in[2];                // <BLK>
57      1           ? 2'b11 // <BR>
58      1           : 2'b00; // <BR>

```



In the above figure, red indicates pure blocks, green indicates branches not scored as block, and blue indicates the code scored as block and branch.

VHDL Example

The following figure displays what gets reported as a block and branch in a given VHDL code.

```

55      -- IF ELSIF ELSE END IF blocks and IMPLICIT ELSE
56      PROCESS
57      BEGIN
58      1      IF phi3 = '1' then    -- <BLK>
59      0          d <= a;    -- <BR>, <BLK>
60      1      ELSIF phi2 = '1' then -- <BLK>
61      1          d <= b;    -- <BR>, <BLK>
62      ELSE
63      1          d <= c;    -- <BR>, <BLK>
64      END IF;
65      1      WAIT FOR 50 NS;  -- <BLK>
66      END PROCESS;
67
68      -- CASE blocks
69
70      PROCESS
71      BEGIN
72      1      WAIT FOR 5 NS;  -- <BLK>
73      1      case phi3 is   -- <BLK>
74          WHEN '0' =>
75          1              a <= b;    -- <BR>, <BLK>
76          WHEN '1' =>
77          0              a <= c;    -- <BR>, <BLK>
78          WHEN OTHERS =>
79          0              a <= a;    -- <BR>, <BLK>
80          END case;
81      1      WAIT FOR 1000 NS; -- <BLK>
82      END PROCESS;
83
84      -- Conditional NEXT and EXIT statements
85
86      PROCESS
87      VARIABLE i : INTEGER;
88      BEGIN
89      1      FOR i IN 0 TO 100 LOOP -- <BLK>
90      1          WAIT FOR 100 NS; -- <BLK>
91      1          IF (i = 11) THEN -- <BLK>
92          1              NEXT;  -- <BLK> <BR>
93          END IF;
94      1          IF (i = 21) THEN -- <BLK>
95          1              EXIT;  -- <BLK> <BR>
96          END IF;
97      END LOOP;
98      END PROCESS;

```

Not scored as blocks due to the use
of conditional signal assignments

Not scored as blocks
due to the use of
selected signal
assignments

```

d <= b WHEN phi1 = '0'  -- <BR>
ELSE c;                -- <BR>

```

-- Selected signal assignments

```

WITH phi3 SELECT
a <= b AFTER 1 NS WHEN '0';    -- <BR>
c AFTER 1 NS WHEN '1';        -- <BR>
a AFTER 1 NS WHEN OTHERS;    -- <BR>

```

In the above figure, red indicates pure blocks, green indicates branches not scored as block, and blue indicates the code scored as block and branch.

Statement Coverage

In a design, an unverified block with more statements is likely to have more errors than the one with a single statement. To ensure verification completeness, design and verification engineers might want to prioritize blocks based on the number of statements in a block. A block coverage report, by default, does not include information on the number of statements within a block. To include this information in a block coverage report, use the `set_statement_scoring` command in the coverage configuration file during elaboration.

With this command, scoring of statements is enabled, and the statement coverage information is also included in the block coverage report.

Consider the following code.

```
module top(r1, r6);
input r1, r6;
reg r2, r3, r4, r5;
always@(r1 or r6)
begin
    if(r1 == 1'b1)           //<BLK1>
        begin
            r2 <= 1'b0;       //implicit else <BLK3>
            r3 <= 1'b1;       //<BLK2>
        end
    case (r1)                //<BLK4>, implicit default <BLK7>
        1'b0, 1'b1:
            r4 <= 1'b1;       //<BLK5>, <BLK6>
        endcase
        r5 <= (r1 == 1'b1)?   //<BLK8>
            1'b1 :
            1'b0;
        if(r6 == 1'b1)         //implicit else <BLK12>
            r4 <= ~r5;        //<BLK11>
    end
endmodule
```

The above code has 12 blocks. A simple block coverage report would help you identify uncovered blocks. However, including the statement coverage information in the report, would help you prioritize the blocks that can be targeted first.

The following report is generated from the above code after enabling branch and statement coverage.

Number of statements: 9
 Covered statements: 7
 Uncovered statements: 2

Module/Entity name: top File name: /home/ruchikas/stmtcov/example.v Number of covered blocks: 7 of 12 Number of covered branches: 4 of 9 Number of covered statements: 7 of 9																																																																																																
Number of blocks marked COV: 0 Number of blocks marked IGN: 0 Number of statements marked COV: 0 Number of statements marked IGN: 0																																																																																																
<hr/>																																																																																																
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;">Cnt</th> <th style="text-align: left;">LineNum</th> <th style="text-align: left;">#Stmts</th> <th style="text-align: left;">Block/Branch type</th> <th style="text-align: left;">Line</th> <th style="text-align: left;">origin</th> <th style="text-align: left;">Description</th> </tr> </thead> <tbody> <tr><td>1</td><td>5</td><td>1</td><td></td><td>5</td><td></td><td>begin // <BLK1></td></tr> <tr><td>1</td><td>7</td><td>2</td><td>true part of</td><td>6</td><td></td><td>if(r1 == 1'b1)</td></tr> <tr><td>0</td><td>6</td><td>0</td><td>* implicit else</td><td>6</td><td></td><td>if(r1 == 1'b1)</td></tr> <tr><td>1</td><td>11</td><td>1</td><td></td><td>11</td><td></td><td>case (r1)</td></tr> <tr><td>0</td><td>13</td><td>1</td><td>a case switch 1 of</td><td>11</td><td></td><td>case (r1)</td></tr> <tr><td>1</td><td>13</td><td>1</td><td>a case switch 2 of</td><td>11</td><td></td><td>case (r1)</td></tr> <tr><td>0</td><td>11</td><td>0</td><td>* implicit default:</td><td>11</td><td></td><td>case (r1)</td></tr> <tr><td>1</td><td>16</td><td>2</td><td></td><td>16</td><td></td><td>r5 <= (r1 == 1'b1)?</td></tr> <tr><td>1</td><td>17</td><td>0</td><td>* ternary 1 true</td><td>16</td><td></td><td>r5 <= (r1 == 1'b1)?</td></tr> <tr><td>0</td><td>18</td><td>0</td><td>* ternary 1 false</td><td>16</td><td></td><td>r5 <= (r1 == 1'b1)?</td></tr> <tr><td>0</td><td>21</td><td>1</td><td>true part of</td><td>20</td><td></td><td>if(r6 == 1'b1)</td></tr> <tr><td>1</td><td>20</td><td>0</td><td>* implicit else</td><td>20</td><td></td><td>if(r6 == 1'b1)</td></tr> </tbody> </table>						Cnt	LineNum	#Stmts	Block/Branch type	Line	origin	Description	1	5	1		5		begin // <BLK1>	1	7	2	true part of	6		if(r1 == 1'b1)	0	6	0	* implicit else	6		if(r1 == 1'b1)	1	11	1		11		case (r1)	0	13	1	a case switch 1 of	11		case (r1)	1	13	1	a case switch 2 of	11		case (r1)	0	11	0	* implicit default:	11		case (r1)	1	16	2		16		r5 <= (r1 == 1'b1)?	1	17	0	* ternary 1 true	16		r5 <= (r1 == 1'b1)?	0	18	0	* ternary 1 false	16		r5 <= (r1 == 1'b1)?	0	21	1	true part of	20		if(r6 == 1'b1)	1	20	0	* implicit else	20		if(r6 == 1'b1)
Cnt	LineNum	#Stmts	Block/Branch type	Line	origin	Description																																																																																										
1	5	1		5		begin // <BLK1>																																																																																										
1	7	2	true part of	6		if(r1 == 1'b1)																																																																																										
0	6	0	* implicit else	6		if(r1 == 1'b1)																																																																																										
1	11	1		11		case (r1)																																																																																										
0	13	1	a case switch 1 of	11		case (r1)																																																																																										
1	13	1	a case switch 2 of	11		case (r1)																																																																																										
0	11	0	* implicit default:	11		case (r1)																																																																																										
1	16	2		16		r5 <= (r1 == 1'b1)?																																																																																										
1	17	0	* ternary 1 true	16		r5 <= (r1 == 1'b1)?																																																																																										
0	18	0	* ternary 1 false	16		r5 <= (r1 == 1'b1)?																																																																																										
0	21	1	true part of	20		if(r6 == 1'b1)																																																																																										
1	20	0	* implicit else	20		if(r6 == 1'b1)																																																																																										

Statements in each block

In the above report, column #Stmts shows the number of statements in the corresponding block. For example, blocks on Line 7 and Line 16 have two statements. The number of statements in module top is 9, which is the total of all statements in column #Stmts.

The number of statements will vary depending on whether branch scoring is enabled or disabled. A case statement with multiple switches in one single statement will be counted as 1 statement if branch scoring is not enabled, but will be counted as separate statements if branch scoring is enabled. For example, line 13 has two switches, as branch scoring is enabled, each of the switch shows up as a separate statement in the block report.

Note: If a block is hit during a simulation run, all the statements within the block are considered hit. As a result, the hit count of a statement is the hit count of its block.

Determining the number of statements within a block

- The number of statements are determined as 0 for:
 - An implicit else block (For example, blocks on Line 6 and Line 20)
 - An implicit default block (For example, the block on Line 11)
 - Branches corresponding to the true and false branch of a Verilog ternary assignment condition (For example, branches on Line 17 and Line 18)

- Branches corresponding to the true and false branch of a VHDL conditional signal assignment and a VHDL selected signal assignment
- Note:** Ternary assignment conditions in Verilog and conditional signal assignments and selected signal assignments in VHDL are scored only if branch scoring is enabled using the [set_branch_scoring](#) command during elaboration.

- The statements enclosed within pragmas are considered. For example, in the following code, ICC determines the number of statements for <BLK2> as 2 even if one of the statements is enclosed within pragmas.

```
begin // <BLK2>
    r2 <= 1'b0;
    //pragma coverage block=off
    r3 <= 1'b1;
    //pragma coverage block=on
end
```

- The statements inside protected code are ignored.

Scoring Block, Branch, and Statement Coverage

The following table provides details on how to score block, branch, and statement coverage.

Coverage Type	How to score
Block	<ul style="list-style-type: none"> ● Pass <code>-coverage block</code> to <code>xmelab</code>, or ● Use the select_coverage command in the coverage configuration file and then pass this file using the <code>-covfile</code> option to <code>xmelab</code>.
Branch	Use the set_branch_scoring command in the coverage configuration file and then pass this file using the <code>-covfile</code> option to <code>xmelab</code> .

Statement	Use the <code>set_statement_scoring</code> command in the coverage configuration file and then pass this file using the <code>-covfile</code> option to xmelab.
-----------	---

-  Branch coverage and statement coverage cannot be scored without enabling block coverage. See [Chapter 9, "Generating Coverage Data,"](#) for more details.

After simulation has dumped block, branch, and statement coverage data, you analyze it using the reporting tool IMC. For details on coverage data analysis with IMC, see the *Integrated Metrics User Guide*.

Expression Coverage

Expression coverage is a mechanism that factorizes logical expressions and monitors them during simulation run. It provides metrics to quantify the degree of verification completeness.

It measures how thoroughly the testbench exercises expressions in assignments and procedural control constructs (if/case conditions). It identifies each input condition that makes the expression true or false and whether that condition happened in simulation. Expression coverage provides finer granularity of coverage metrics than other code coverages, such as block and branch coverage.

Expressions can be scored in many ways, where one trades the amount of data to analyze against the accuracy of the results. Expressions can be scored using following modes:

- Sum of Products (SOP) scoring generates the most concise, easy-to-analyze data but it also is less strict in when it considers a term covered. This is the default mode of expression scoring.
- Full Combinational Coverage (FCC) scoring generates data to analyze whether all combinations of conditions in an expression are covered.
- Control scoring is more strict in its scoring and will require more work to reach 100% coverage. It scores expressions hierarchically as expressions and sub expressions, which makes analysis more complex.
Both SOP and control scoring score vectors as logical, where a vector is either zero or non-zero. This reduces accuracy and allows reaching 100% coverage quicker.
- Vector scoring takes control scoring one step further, and it analyzes vectors in expressions one bit at a time. But that results in more coverage points to analyze and to cover.

Ultimately, as a user, you decide where to study expression coverage and to what level of detail. But before scoring expression coverage, remember to get a high block coverage in your regression.

Expression coverage by default is scored only for Verilog logical operators (`||` and `&&`) and VHDL logical operators (`OR`, `AND`, `NOR`, and `NAND`). The default constructs covered are listed:

Logical Operators	Constructs Covered

Verilog Logical Operators (and &&)	<ul style="list-style-type: none"> • Condition of Ternary operator (?:) in procedural and continuous assignments • Condition of <code>if</code> statement in if procedural statements • Expression of <code>case/casex/casez(expression)</code> in procedural statements • Condition of <code>repeat/while/do-while/for</code> loop in procedural statements
VHDL Logical Operators (OR, AND, NOR, and NAND)	<ul style="list-style-type: none"> • Condition of <code>wait/assert</code> statement in <code>wait/assert</code> sequential statements • Expression of <code>if/elsif</code> condition in if sequential statements • Expression of "case expression" of case sequential statements • Condition of <code>while</code> loop sequential statements • Conditional signal assignments and Selected signal assignments are equivalent to sequential <code>if</code> and <code>case</code> assignments, respectively, as defined by the VHDL language. The expressions in these equivalent statements are scored in the same way as mentioned in previous points.

For more information on scoring other operators, refer to [Scoring of Operators](#).

This chapter explores the different scoring modes and how to interpret the detailed reports.

Typically, you score your entire design using the same mode, but you also can set different scoring modes for selected modules in your design. See [set_expr_scoring](#) for details on specifying scoring mode.

Sum-of-Products (SOP) Scoring

The SOP scoring mode reduces expressions to a minimum set of expression inputs that make the expression both true and false. SOP scoring checks that each input has attained both 1 and 0 state at some time during the simulation.

SOP scoring is inherently first-level. It uses subexpressions only in the following situations:

- Expressions with an arithmetic or relational operator not at the lowest level
- Overly complex expressions (too many rows, especially due to XOR operator)

 With SOP scoring, vector inputs are scored as logical (single bit).

An expression marked for SOP coverage can contain a maximum of 1024 terms. If the number of terms exceeds this limit, coverage skips the expression and its sub-terms, and the data is not scored or dumped in the coverage database. In addition, a warning is displayed with the line and position of the expression during elaboration, and the number of terms in the expression. However, you can change the maximum number of terms in an expression using the `set_expr_scoring` command with the `-max_terms_sop <num>` option, where `num` is a positive integer.

Example 1

Consider the following Verilog expression.

```
b & (c | d)
```

With SOP scoring, the resultant scoring table is:

b & (c d)					
<1>	<2>	<3>			
hit		rval		<1>	<2>

1		1		1	-
1		1		1	-
0		0		-	0
1		0		0	-

The `rval` column displays the resulting value of the expression (either zero or non-zero). The `rval` column is displayed only if the expression has **different** operators. For example, the `rval` column will not be displayed for expression `b && (c && d)`.

Example 2

If an expression has redundant inputs, SOP scoring removes them from the scoring table. For example, in the following expression, the results table displays values for `b` only once.

a = b && (b c)					
<1>	<1>	<2>			
hit		rval		<1>	<2>

1		1		1	-
0		0		0	-

Example 3

By default, if a primary expression includes a subexpression that uses logical equality (`==0`) or

logical inequality (`!=0`) operators, then the terms for the primary expressions are determined after splitting the subexpression. Consider the following expression:

```
b && ((a != 0) || c)
```

This primary expression includes a subexpression (`a != 0`) that uses a logical inequality operator. As a result, the terms of this primary expression are determined as:

```
b && ((a != 0) || c)  
<1> <2-> <3-> <4->
```

To stop splitting such subexpressions when determining the terms of primary expression, use the [set_expr_scoring](#) command with the `-vlog_remove_redundancy` option during elaboration.

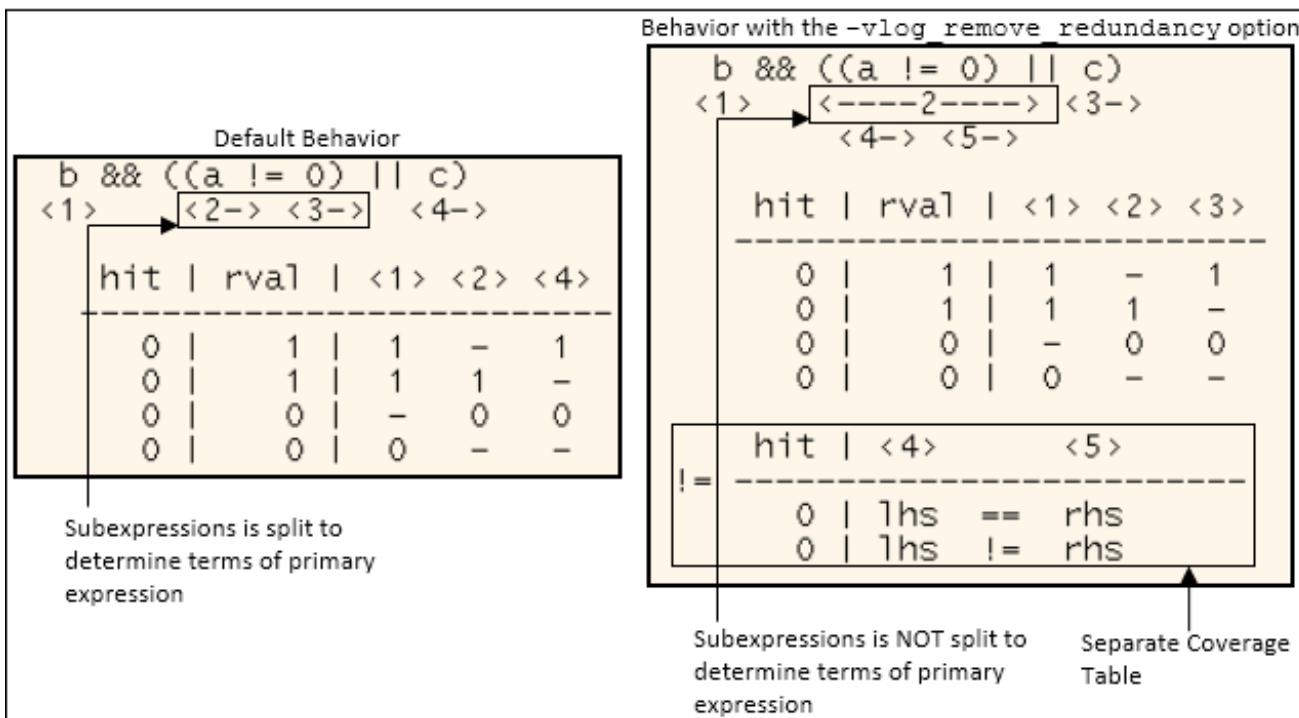
 The `-vlog_remove_redundancy` option has been deprecated and will not be supported in the subsequent release.

With this command, the terms of the primary expression are determined as:

```
b && ((a != 0) || c)  
<1> <----2----> <3->  
      <4-> <5->
```

In addition, the subexpression is scored in a separate coverage table.

The following figure displays the expression coverage results with and without the `-vlog_remove_redundancy` option.

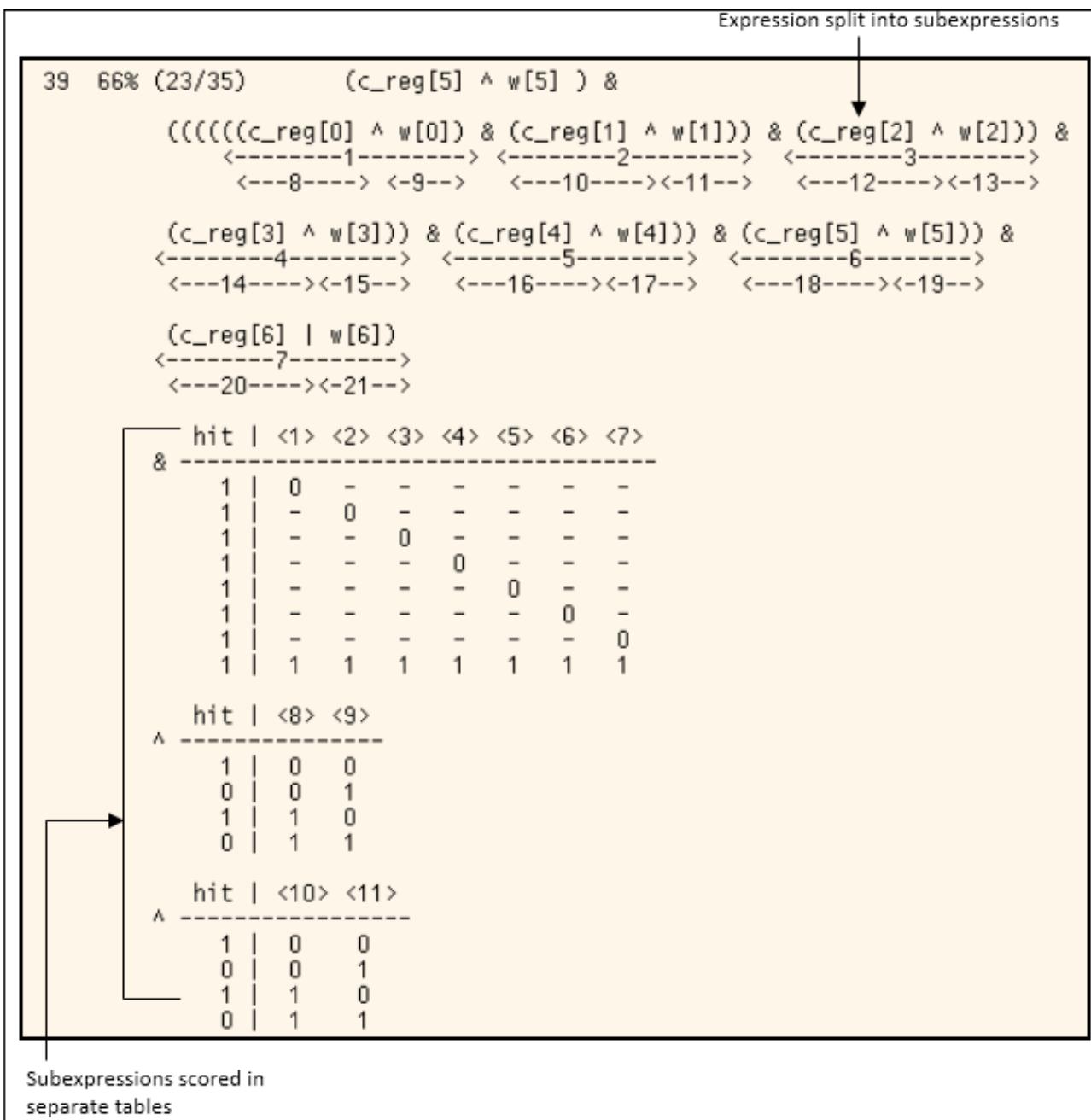


Example 4

In the given scenarios, an expression is split to form more than one coverage tables in following scenarios:

- If the number of terms is greater than 64 for expressions that use more than one operator.
- If the number of rows in a table is expected to exceed the predefined limit of 256.

Consider an example of an overly complex expression where the number of rows is expected to exceed the predefined limit.



In this example, as the number of rows is expected to exceed the predefined limit, the expression is split into subexpressions, and scored in separate tables.

Evaluation of VHDL AND/NAND and OR/NOR operators of BIT and BOOLEAN Types

By default, operands in an expression are evaluated before associating them with operators. However, in the case of VHDL short-circuit operations, you might want to evaluate the right-hand operand only if the left-hand operand has a certain value.

By default, short-circuit evaluation for VHDL AND/NAND and OR/NOR operators of BIT and BOOLEAN type

operands is enabled. You can disable the short-circuit evaluation using the `-no_vhdl_shortcircuit` switch with the `set_expr_scoring` command in the coverage configuration file as:

```
set_expr_scoring -no_vhdl_shortcircuit
```

With the above command, the second operand of VHDL AND/NAND and OR/NOR operators for BIT / BOOLEAN type operands are evaluated if the first operand evaluates to 0/FALSE and 1/TRUE, respectively.

Consider the following SOP table for a VHDL AND operator.

hit		<1>	<2>
0		0	-
0		-	0
0		1	1

By default, for an input combination of term $<1> = 0$ and term $<2> = 0$, only Row 1 will be scored because term $<1>$ will be evaluated but term $<2>$ will not be evaluated.

If short-circuit evaluation is disabled, then for the same input combination, both term $<1>$ and term $<2>$ will be evaluated, and both Row 1 and Row 2 will be scored.

-  ● Operations for which the right-hand operand is evaluated if and only if the left-hand operand has a certain value are called short-circuit operations. VHDL logical operations such as and, or, nand, and nor defined for operands of types BIT and BOOLEAN are short-circuit operations.
- If the `-no_vhdl_shortcircuit` switch is specified with any `set_expr_scoring` command, it disables the short-circuit behavior for all VHDL entity-architecture pairs for which expression coverage is enabled.

Full Combinational Coverage (FCC) Scoring

The FCC scoring mode is an extension of SOP scoring. This mode is used to analyze whether all the combination of terms in an expression are covered. You can enable FCC scoring either for the whole design or for selected Verilog or VHDL units.

To enable FCC scoring for the whole design, use the following command in the coverage configuration file:

```
set_expr_scoring -fcc
```

When you use this command, FCC scoring is enabled for all the modules that are not specifically marked for any other scoring mode.

To enable FCC scoring for specific Verilog and VHDL modules, use the following command:

```
set_expr_scoring -fcc <list of modules instantiated in the design>
```

⚠ In FCC scoring, if an expression contains any parameterized term, bitwise scoring for that expression is not done.

Example 1

Consider the given Verilog expression:

```
z = x && y && p;
```

With FCC scoring, the resultant scoring table is:

Line	Coverage	Expression description
47	75% (6/8)	$z = x \&\& y \&\& p;$
		$(x \&\& y) \&\& p$ $<1-> <2-> <3->$
		hit <1> <2> <3>
&&		
2		0 0 0
2		0 0 1
0		0 1 0
0		0 1 1
14		1 0 0
3		1 0 1
1		1 1 0
3		1 1 1

Example 2

Consider the given VHDL example:

```
if ((a and b and c) = '1')
```

In this example when you use the `set_expr_scoring -fcc` command, the terms of the expression are scored as follows:

Line	Coverage	Expression description
35	80% (8/10)	<pre>if ((a and b and c) = '1') ((A and B) and C) = '1' <->1-----> <->2-> <->3-> <->4-> <->5-> hit <1> <2> = ----- 1 lhs = rhs 6 lhs /= rhs hit <3> <4> <5> and ----- 2 0 0 0 1 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 0 1 0 1 1 1 1 0 1 1 1 1</pre>

Control Scoring

Control scoring mode checks if each input has controlled the output value of the expression at some time during the simulation. If the value of input changes, then the value of output also changes. Control scoring improves verification accuracy by applying stronger requirements in order to call an expression input covered.

The control scoring mode follows a basic rule: *Coverage is scored if and only if an expression term has at least one input (or operand) that controls the result of the expression.* Two or more inputs of an expression may both control the result of an expression.

Note: Control scoring is sometimes referred to as "sensitized condition coverage" or "focused condition coverage".

The control scoring mode breaks an expression into a hierarchy of subexpressions. For example:

```
b & (c | d)
<1> <- - - 2 - - ->
<3> <4>
```

where

- Level 1 is $b \& (c | d)$ with inputs $<1>$ and $<2>$
- Level 2 is $(c | d)$ with inputs $<3>$ and $<4>$

An expression is always analyzed from the top-down level, determining first if the current level's inputs (<1> and <2>) qualify it as a controlling expression. (Level refers to the depth of a nested expression.)

- If no inputs control the expression output, then no scoring of coverage is done for that expression or any of its subexpressions.
- If at least one input is controlling, then the term is scored. Additionally, any expression coverage table corresponding to a subexpression(s) on a controlling input is evaluated for scoring.

The coverage tool will continue to the next lower level of the expression, and check for coverage there. Again, it will check if the current level's terms are controlling. If so, scoring will be done.

With control scoring, the above expression will create the following results tables:

```
b & (c | d)
<1> <----2---->
    <3> <4>

hit | <1> <2>
& -----
0 | 0   1
0 | 1   0
1 | 1   1

hit | <3> <4>
| -----
1 | 1   0
0 | 0   1
0 | 0   0
```

In case of control scoring, a separate results table is created for each operator.

Note: With control scoring, vector inputs are scored as logical (single bit).

The following Verilog code example demonstrates how a truth table is created in case of control scoring.

```
module test;
reg b,c,d,e;
wire sig1 = (b && c || d) && (e != 0);
initial
```

```
$monitor($time,, "b =",b," c =",c," d =",d," e =",e," sig1 =",sig1);
initial
begin
#1 b = 0; c = 0; d = 0;
#1 e = 1;
#1 b = 1;
#1 c = 1;
#1 $finish;
end
endmodule
```

The simulation log for the above code is:

```
0 b =x c =x d =x e =x sig1 =x
1 b =0 c =0 d =0 e =x sig1 =0
2 b =0 c =0 d =0 e =1 sig1 =0
3 b =1 c =0 d =0 e =1 sig1 =0
4 b =1 c =1 d =0 e =1 sig1 =1
```

The expression coverage report for the module `test` is as follows.

Note: The following expression coverage report contains Level and Time comments to aid in the control score analysis that follows the report presentation. These comments are not part of the actual report.

```

Line  Coverage   Expression description
-----
3 63% (7/11)  wire sig1 = (b && c || d) && (e != 0);
              (b && c || d) && (e != 0)
              <----1-----> <----2--->
              <---3--> <4>  <7> <8>
              <5> <6>

Level 1    hit  <1>  <2>
&& -----
      2      0      1          Time: 2, 3
      0      1      0
      1      1      1          Time: 4

Level 2    hit  <3>  <4>
|| -----
      1      1      0          Time: 4
      0      0      1
      2      0      0          Time: 2, 3

Level 2    hit  <7>  <8>
!= -----
      0      lhs == rhs
      1      lhs != rhs      Time: 4

Level 3    hit  <5>  <6>
&& -----
      0      0      1          Time: 3
      1      1      0
      1      1      1          Time: 4

```

Report Analysis

Time	Analysis
Time 0	Inputs of expression: b =x c =x d =x e =x sig1 =x No scoring is recorded since at least one input of the top-level expression is undefined (=x).
Time 1	Inputs of expression: b =0 c =0 d =0 e =x sig1 =x No scoring is recorded since at least one input of the top-level expression is undefined (=x).

Time 2	<p>Inputs of expression: <code>b =0 c =0 d =0 e =1 sig1 =0</code></p> <p>Level 1 (<code>&&</code> Truth Table)</p> <p>Since <code>b && c = 0</code>, and <code>d = 0</code>, input <code><1> = 0</code>. Since <code>e = 1</code>, input <code><2> = 1</code>, and the state of <code><1> && <2></code> becomes <code>0 && 1</code>. This is a controlling term: input <code><1></code> is controlling the outcome of the <code>&&</code> operation. Therefore, the <code>hit</code> increments in the <code>0 1</code> expression term of the truth table.</p> <p>Level 2 (<code> </code> Truth Table)</p> <p>Since input <code><1></code> controls the <code><1> && <2></code> expression of Level 1, this sub-level is considered for scoring. Since input <code><3></code> and input <code><4></code> are both <code>0</code>, <code><3> <4></code> has the state of <code>0 0</code>. This is a term where both inputs (<code><3></code> and <code><4></code>) are controlling the outcome of the <code> </code> operation. Therefore, the <code>hit</code> increments in the <code>0 0</code> expression term of the truth table.</p> <p>Level 2 (<code>!=</code> Truth Table)</p> <p>Since input <code><2></code> is not the controlling input in the <code><1> && <2></code> expression of Level 1, nothing is considered for scoring in sub-levels pertaining to input <code><2></code>.</p> <p>Level 3 (<code>&&</code> Truth Table)</p> <p>Since input <code><3></code> (and in this case, input <code><4></code>) is a controlling input in the <code><3> <4></code> expression of Level 2, this sub-level is considered for scoring. Since <code>b = 0</code>, and <code>c = 0</code>, <code><5> && <6></code> have a state of <code>0 && 0</code>. This is not a controlling input; therefore, no scoring is done.</p>
--------	---

Time 3	<p>Inputs of expression: <code>b =1 c =0 d =0 e =1 sig1 =0</code></p> <p>Level 1 (<code>&&</code> Truth Table) Same as Time 2.</p> <p>Level 2 (<code> </code> Truth Table) Same as Time 2.</p> <p>Level 2 (<code>!=</code> Truth Table) Same as Time 2.</p> <p>Level 3 (<code>&&</code> Truth Table) Since input <code><3></code> (and in this case, input <code><4></code>) is a controlling input in the <code><3> <4></code> expression of Level 2, this sub-level is considered for scoring. Since <code>b = 1</code>, and <code>c = 0</code>, <code><5> && <6></code> have a state of <code>1 && 0</code>, this is a controlling term: input <code><5></code> controls the outcome of the <code>&&</code> operation. Therefore, the <code>hit</code> increments in the <code>1 0</code> expression term of the truth table.</p>
--------	--

Time 4	<p>Inputs of expression: <code>b =1 c =1 d =0 e =1 sig1 =0</code></p> <p>Level 1 (&& Truth Table)</p> <p>Since <code>b && c = 1</code>, and <code>d = 0</code>, input $<1> = 1$. Since input $<1> = 1$, input $<2> = 1$, and the state of $<1> \&\& <2>$ becomes <code>1 && 1</code>. This is a controlling term where both inputs ($<1>$ and $<2>$) control the outcome of the <code>&&</code> operation. Therefore, the <code>hit</code> increments in the <code>1 1</code> expression term of the truth table.</p> <p>Level 2 (Truth Table)</p> <p>Since input $<1>$ (and in this case, input $<2>$) is a controlling input in the $<1> \&\& <2>$ expression of Level 1, this sub-level is considered for scoring. Since input $<3>$ and input $<4>$ are <code>1</code> and <code>0</code> respectively, $<3> \mid\mid <4>$ has the state of <code>1 \mid\mid 0</code>. This is a controlling term. Input $<3>$ controls the outcome of the <code>\mid\mid</code> operation. Therefore, the <code>hit</code> increments in the <code>1 0</code> expression term of the truth table.</p> <p>Level 2 (!= Truth Table)</p> <p>Since input $<2>$ (and in this case, input $<1>$) is a controlling input in the $<1> \&\& <2>$ expression of Level 1, this sub-level is considered for scoring. Since <code>e = 1</code>, and <code>0 = 0</code>, the <code>lhs != rhs</code> expression term of the truth table is met and its <code>hit</code> increments.</p> <p>Level 3 (&& Truth Table)</p> <p>Since input $<3>$ is a controlling input in the $<3> \mid\mid <4>$ expression of Level 2, this sub-level is considered for scoring. Since <code>b = 1</code>, and <code>c = 1</code>, $<5> \&\& <6>$ have a state of <code>1 && 1</code>. This is a controlling term where both inputs ($<5>$ and $<6>$) control the outcome of the <code>&&</code> operation. Therefore, the <code>hit</code> increments in the <code>1 1</code> expression term of the truth table.</p>
--------	---

Note: The expression coverage table (for both SOP and control scoring) is broken into two or more tables if there are more than 10 inputs to report.

Vector Scoring

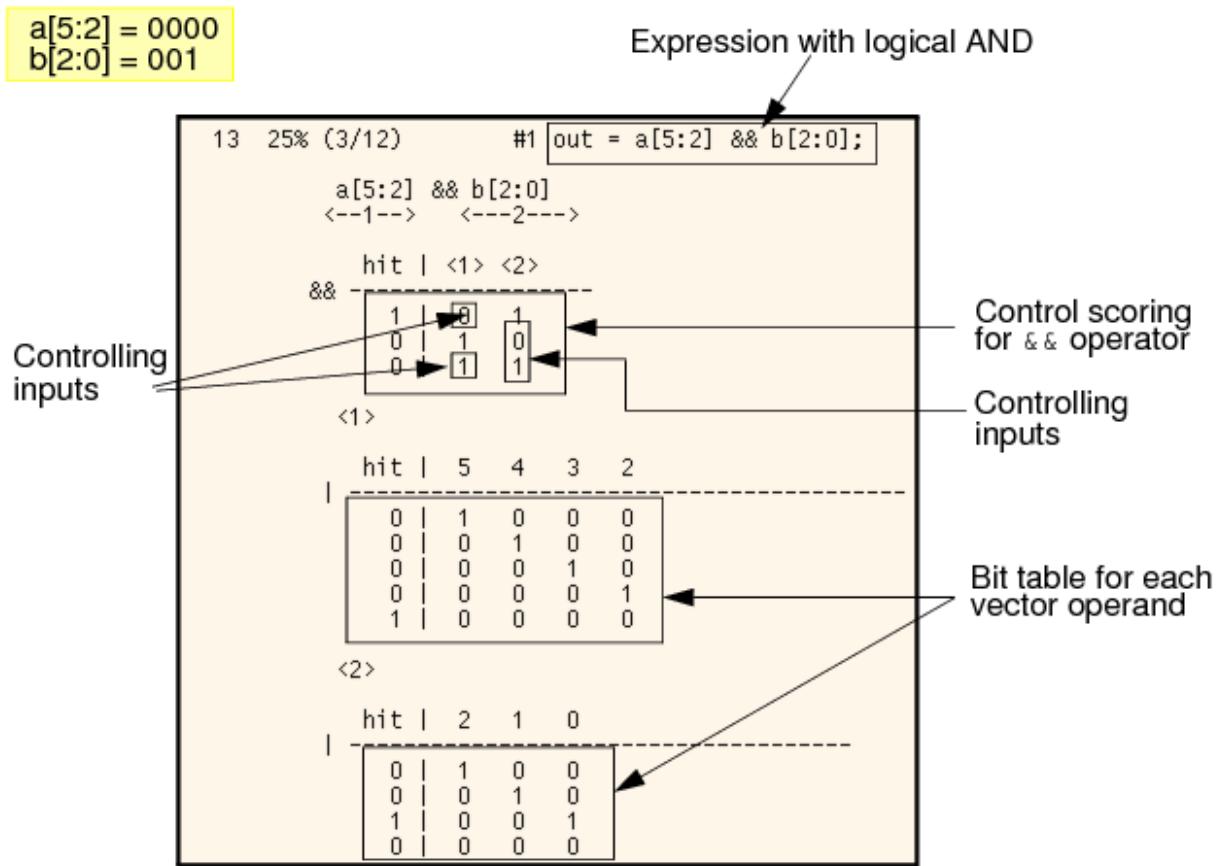
Vector scoring mode is an extension of control scoring mode. With vector scoring, each bit of a multi-bit signal is scored and reported separately and you have lots of data to analyze. The following Verilog code demonstrates how truth tables are created for vector scoring.

```
1 module top;
```

```
2 reg [5:0] a, b, c, out;
3 reg [7:0] d;
4 initial
5 $monitor($time, " a = %b", a, " b = %b", b, " c = %b", c, " d = %b", d,
   "out = %b", out);
6 initial
7 begin
8 #1
9 a = 6'b000000;
10 b = 6'b000001;
11 c = 6'b111111;
12 d[2:0] = 3'b011;
13 #1 out = a[5:2] && b[2:0];
14 #1 out = a | b;
15 #1 out = a ^ b;
16 #1 out = c ? a : b;
17 #1 out = b === a;
18 #1 out = ~b[5:3];
19 #1 out = a[1:0] | b[2:0];
20 #1 c = 6'b100000;
21 #1 out = a && b || c;
22 #1 $finish;
23 end
24 endmodule
```

Scoring of Logical Operators

The truth table for expression containing logical `&&` operator on Line 13 is as follows.



Note: With control scoring, only the first truth table is reported.

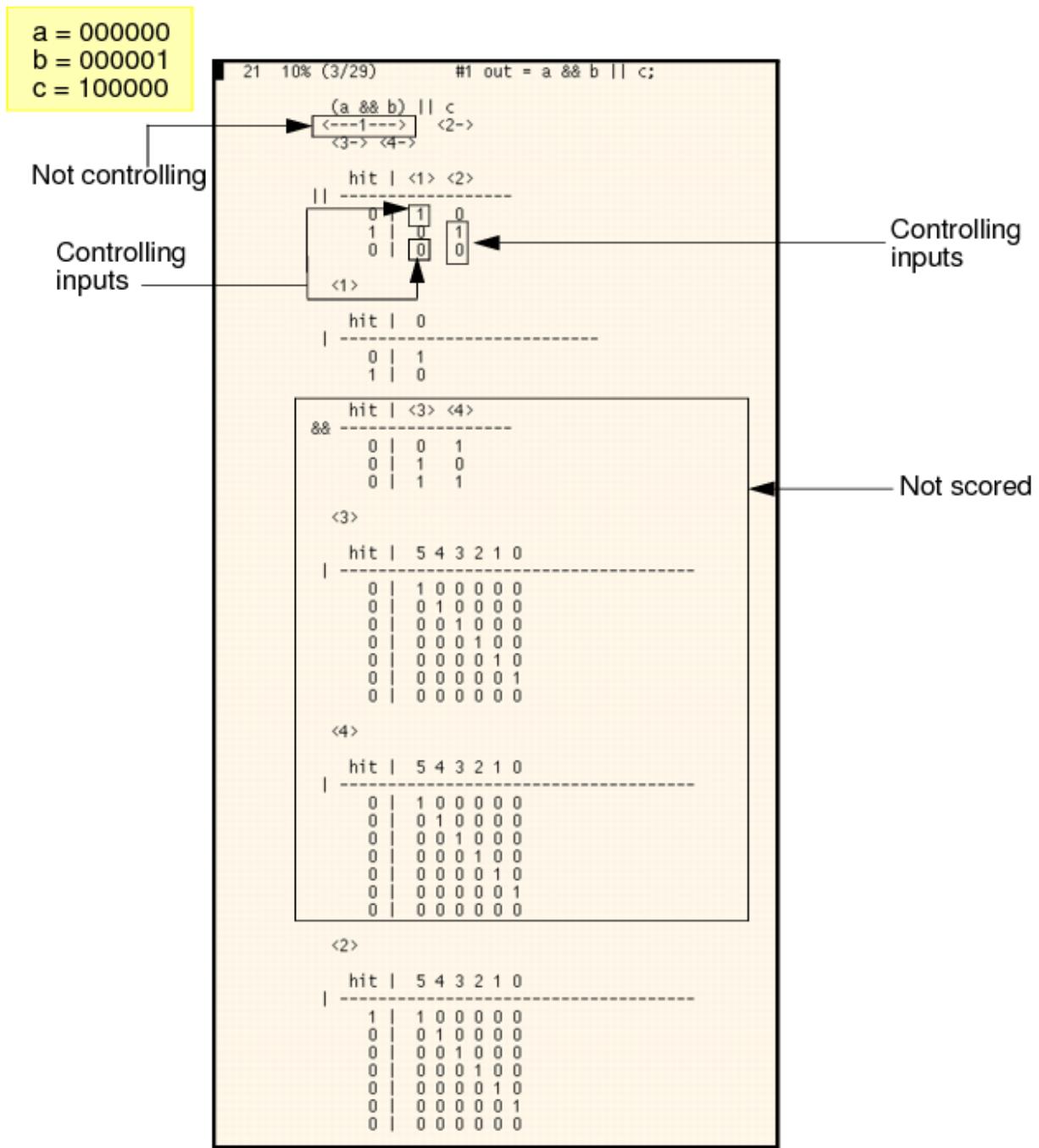
In a logical operator truth table, each operand is scored using an `|` reduction table, and the overall expression is scored using the same table as a scalar `||` or `&&` operator.

The operands for operators `||` and `&&` are first compressed into a single bit, such that if any bit is 1, then the operand is 1. This is effectively an `|` reduction (for both operators) and is implemented in logic as an `OR` gate whose inputs are each bit of the vector. These single bit results are then used to determine the result of the expression, which is either an `OR` or `AND` gate, respectively.

If multiple bits of a vector are 1, then nothing is scored in the lower tables because no single bit is controlling. However, scoring may occur in the higher table since the result of the operand is 1.

Each operand is scored according to its type (vector, scalar, or real). Operands of unequal size are scored according to the size of each operand. If an operand is a constant, no scoring is done for that expression.

Consider another truth table for expression containing logical operators on Line 21.

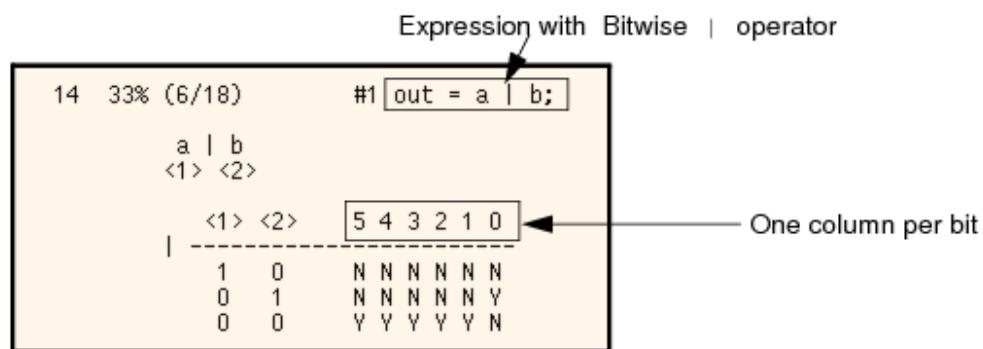


In the above report, expression term <1> is not controlling (because it evaluates to 0). As a result, expression terms <3> and <4> are not scored.

Scoring of bitwise operators

The truth table for expression containing bitwise | operator on Line 14 is as follows.

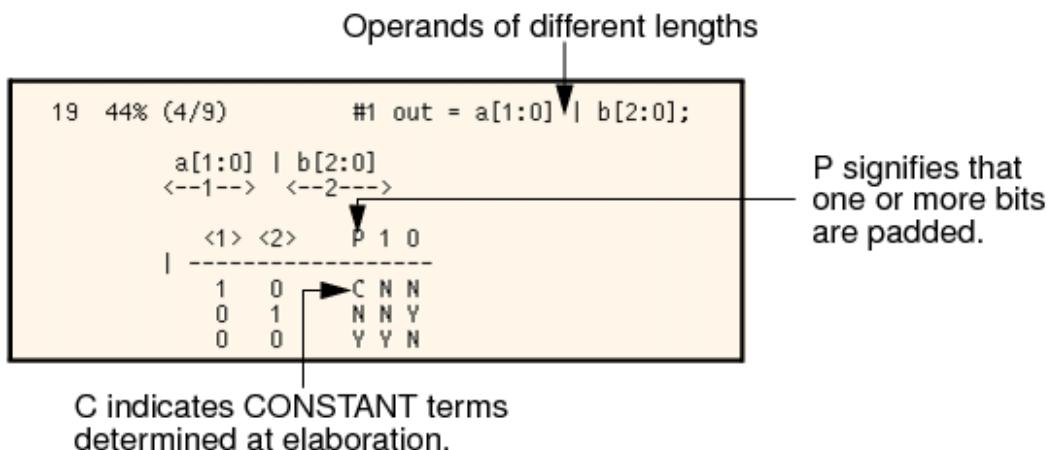
a = 000000
b = 000001



In a bitwise operator truth table, Y indicates that this bit combination has occurred. N indicates that this bit combination has not occurred. The scoring count is number of hits (Y) divided by number of scorable entries ($\text{Y}+\text{N}$), which in this case is $6/18$.

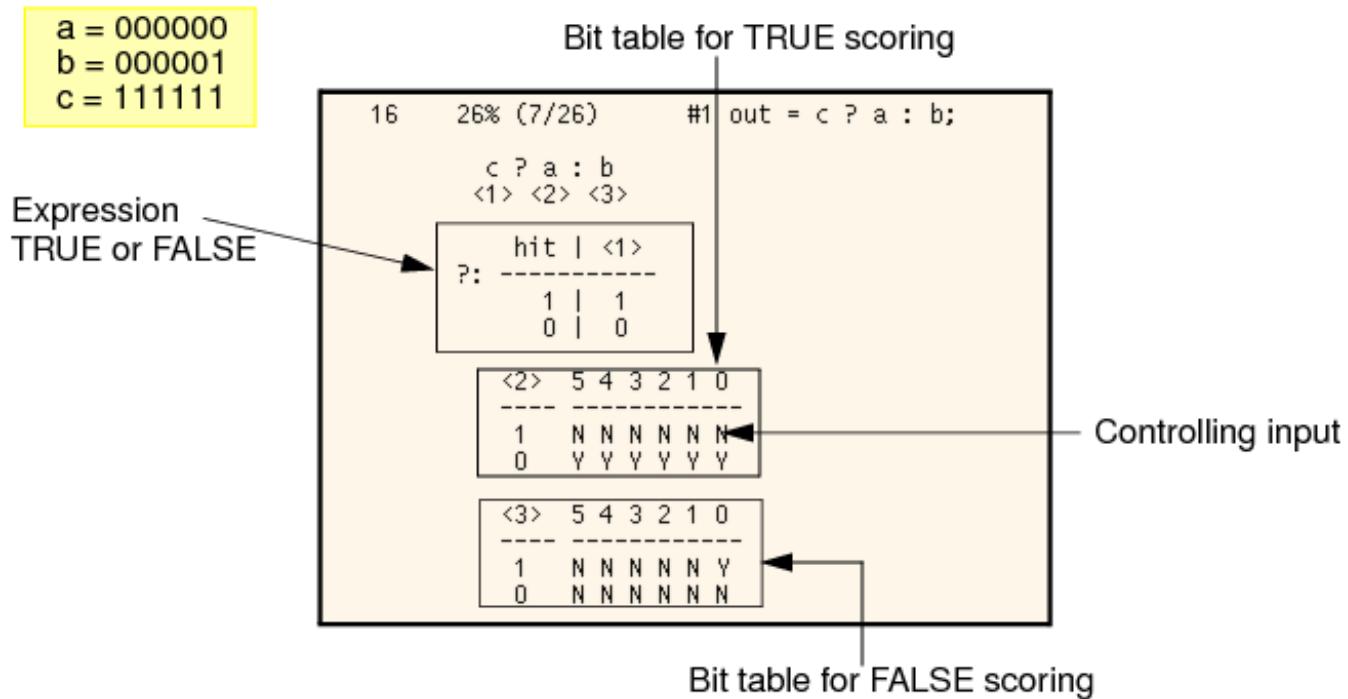
Consider another example where operands of a bitwise expression are of different lengths. The truth table for expression with operands of different lengths on Line 19 is as follows.

a[1:0] = 00
b[2:0] = 001



Scoring of conditional operator

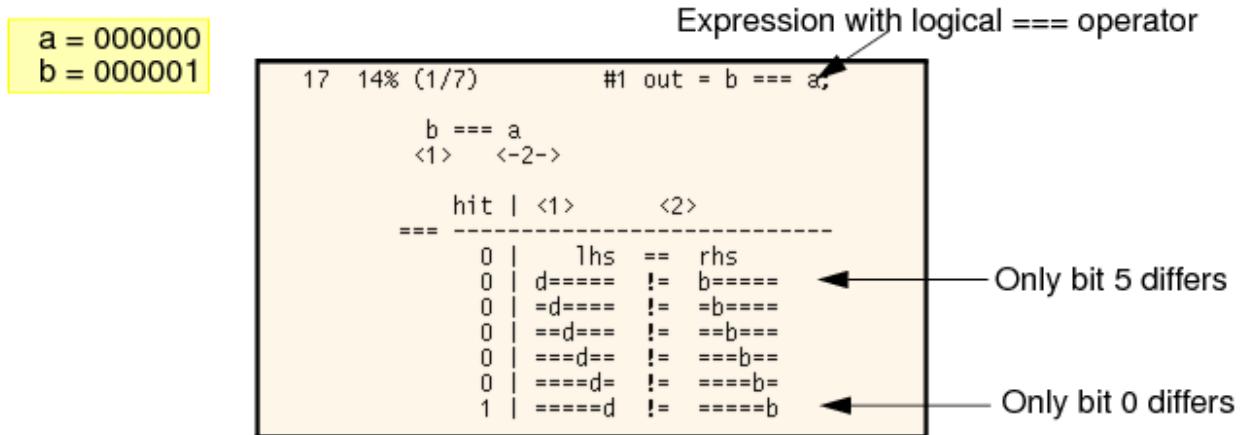
The truth table for expression containing a conditional operator on Line 16 is as follows.



In the case of a conditional operator, the first truth table determines if the expression is true or false. The subsequent tables display bit values for true and false scoring, respectively. Here, for term $\langle 2 \rangle$ operand 0, since all the bits are 0, Y is shown for each bit with value 0.

Scoring of Equality Operators

The truth table for expression containing logical equality operator on Line 17 is as follows.



The truth table for an expression with an equality operator includes `lhs` and `rhs`. The first row in the truth table indicates if the two operands are equal or not. If the `lhs` and `rhs` differ in only a single bit, then that bit is the controlling term. In the above report, `=` indicates that the bits are of the same value. The symbols `b` and `d` are mirror images of each other and signify that the two values are

opposite in value. The scoring count is $N + 1$, where N is the maximum number of the bits in an operand.

Note: Vectors of differing size use the size of the largest vector and the smallest vector is appended with trailing zeros in MSB up to the size of the largest vector.

For details on coverage data analysis with IMC, see the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

Limitations with Vector Scoring

- Expressions involving parameters (directly or indirectly) are not scored.
- Individual bits of constants involved in an expression are not considered for scoring. Consider the following code:

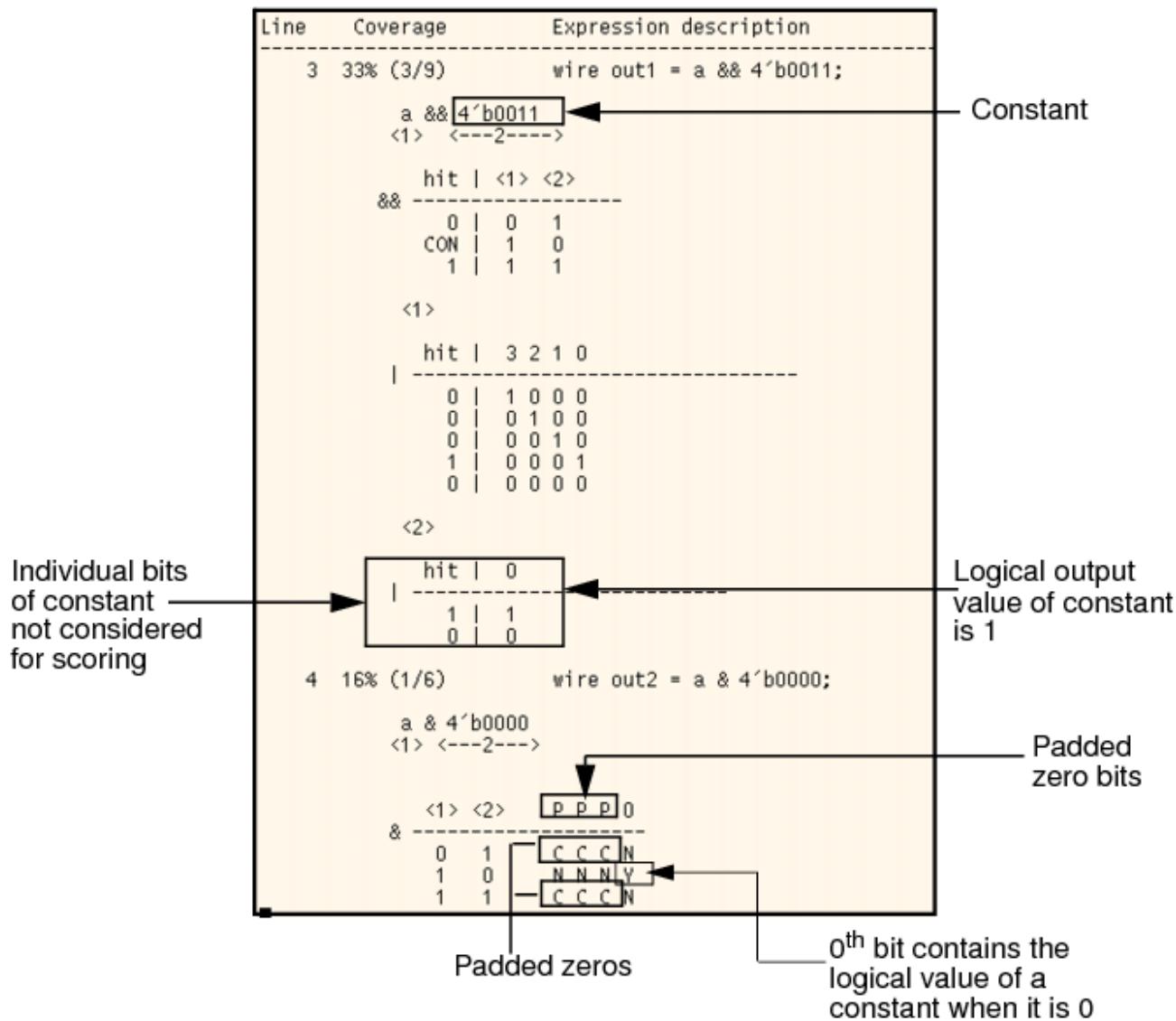
```
module top;

reg [3:0] a;

wire out1 = a && 4'b0011;
wire out2 = a & 4'b0000;

initial
begin
#1 a = 4'b0001;
#1 $finish;
end
endmodule
```

The expression coverage report generated from the above code is:



Scoring Expression Coverage

To score expression coverage, you can either:

- Pass `-coverage expr` to `xmelab`.
- Use the `select_coverage` command in the coverage configuration file and then pass this file using the `-covfile` option to `xmelab`.

See [Chapter 9, "Generating Coverage Data,"](#) for more details.

After simulation has dumped expression coverage data, you analyze it using the reporting tool IMC.

For details on coverage data analysis with IMC, see the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

Scoring of Operators

The following table lists the operators that are scored in different scoring modes when the `set_expr_coverable_operators -all` and `set_expr_coverable_statements -all` commands are used to score all Verilog and VHDL operators in conditions and assignments.

Verilog Operator	SOP Table Created	Control Table Created	Vector Table Created	FCC Table Created
Relational: <code>></code> , <code>>=</code> , <code><</code> , <code><=</code>	Yes	Yes	Yes	Yes
Logical: <code>!</code> , <code>&&</code> , <code> </code> , <code>==</code> , <code>!=</code>	Yes	Yes (except for <code>!</code>)	Yes	Yes
Case: <code>==</code> , <code>!==</code>	Yes	Yes	Yes	Yes
Bit-wise: <code>~</code> , <code>&</code> , <code> </code> , <code>^</code> , <code>^~</code> , <code>^^</code>	Yes	Yes (except for <code>~</code>)	Yes (except for <code>~</code>)	Yes (except for <code>~</code>)
Reduction: <code>&</code> , <code>~&</code> , <code> </code> , <code>~ </code> , <code>^</code> , <code>~^</code> , <code>^^</code>	Yes	Yes	Yes	Yes
Conditional: <code>? :</code>	Yes	Yes	Yes	Yes
Concatenation: <code>{}</code>	No	No	No	No
Replication: <code>{ {} }</code>	No	No	No	No
Arithmetic: <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code>	No	No	No	No
Modulus: <code>%</code>	No	No	No	No

Logical: <<, >>	No	No	No	No
Arithmetic shift: <<<, >>>	No	No	No	No

VHDL Operator	SOP Table Created	Control Table Created	Vector Table Created	FCC Table Created
Logical: and, or, nand, nor, xor, xnor	Yes	Yes	NA	Yes
Relational: =, /=, <, <=, >, >=	Yes	Yes	NA	Yes
Shift: sll, srl, sla, sra, rol, ror	No	No	NA	No
Adding: +, -, &	No	No	NA	No
Sign: +, -	No	No	NA	No
Multiplying: *, /, mod, rem	No	No	NA	No
Miscellaneous: **, abs, not You can enable scoring of VHDL <code>not</code> operator using the <u>set_expr_scoring</u> - <code>vhdl_not_as_operator</code> command in the coverage configuration file at elaboration.	No	No	NA	No
not operator	Yes	Yes	NA	Yes

Note: Vector scoring is not supported for VHDL design units.

Note: Using `xor` or `xnor` operator, an expression with maximum limit of 16 terms can be scored, and any expression with more than 16 terms will not be scored.

Note: You can use the [set_expr_scoring -no vhdl_control](#) command to disable control scoring in all VHDL units.

Expression Coverage Report -- Operators scored by default

The following expression coverage report displays a sample expression which includes multiple operators. In the default mode, coverage is scored only for Verilog logical operator `&&`, as shown below:

Line	Coverage	Expression description
55	67% (2/3)	assign #1 sat_prod = (ovm && ovf) ? (prod[`HMSB]) ? PSAT : NSAT : prod ; (ovm && ovf) ? (prod[31] ? PSAT : NSAT) : prod <-1--> <-2--> && hit <1> <2> 1 0 - 230 - 0 0 1 1

Only logical `&&` scored

Scoring Rules for Vector inputs

The following rules apply for scoring vector inputs. These are listed in the order of precedence.

Input Contains ...	Input Scores as ...
any x or z	not scored
any 1s (ones)	1 (one)
all 0s (zeros)	0 (zero)

Scoring Events

Scoring events helps you examine the possible value changes that trigger the execution of an `always` block. By default, Integrated Coverage does not score events. To enable scoring of Verilog event control (which is introduced by the symbol `@`), use the `-event` option of the [set_expr_scoring](#) command.

Scoring events is independent of the scoring mode (SOP, Control, or Vector) selected.

When you enable scoring of events, a separate truth table (referred as the event table) is created in addition to other truth tables (based on scoring mode selected), as shown in the report below.

```
event ev1, ev2;
reg a, b, c, r;
initial begin
#100 a = 0; b= 0; c = 0;
#100 a = 1; b= 0; c = 0;
#100 a = 1; b= 0; c = 1;
#100 -> ev1
```

Event expression (scored from positive edges)
SOP scoring mode

Event table →

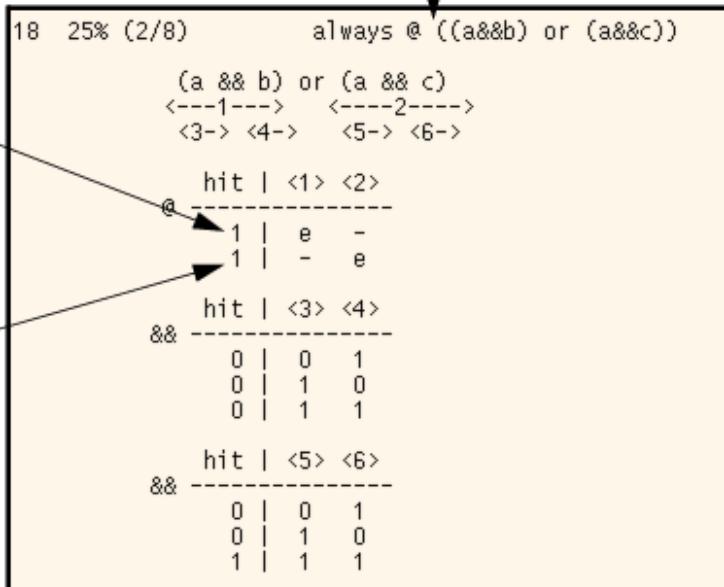
21 25% (2/8) always @(posedge(a&&b) or posedge(a&&c))
posedge (a && b) or posedge (a && c)
<-----1-----> <-----2----->
<3-> <4-> <5-> <6->
hit <1> <2>
@ -----
0 e -
1 - e
hit <3> <4>
&& -----
1 0 -
1 - 0
0 1 1
hit <5> <6>
&& -----
1 0 -
1 - 0
1 1 1

Each event term is represented by symbol `e`, and is independent of any events on the other terms. In the above report, an event occurred on term `<2>`, and therefore it is shown as hit.

Consider another example.

```
event ev1, ev2;
reg a, b, c, r;
initial begin
  #100 a = 0; b= 0; c = 0;
  #100 a = 1; b= 0; c = 0;
  #100 a = 1; b= 0; c = 1;
  #100 -> ev1
```

Event expression scored
Control scoring mode



Term <1> scored from
ab = 00 edge

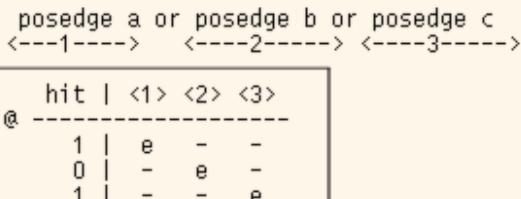
Term <2> scored from
ac = 11 edge

Consider another example that uses a sensitivity list.

```
#100 a = 0; b= 0; c = 0;
#100 a = 1; b= 0; c = 0;
#100 a = 1; b= 0; c = 1;
```

Event expression scored

27 67% (2/3) always @ (posedge a, posedge b, posedge c)



Each event tracks
a single edge

Consider another example that uses an implicit event expression list.

```
#100 a = 0; b= 0; c = 0;  
#100 a = 1; b= 0; c = 0;  
#100 a = 1; b= 0; c = 1;
```

Event expression scored

Event list obtained all nets
on the right-hand side

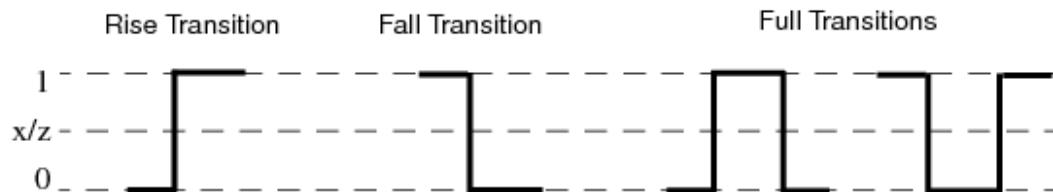
30 100% (3/3) always @ (*)
▶ a or b or c <1> <2-> <3->
hit <1> <2> <3>
@ -----
1 e - -
1 - e -
1 - - e

Toggle Coverage

Toggle coverage measures activity of various signals in a design and provides information on untoggled signals or signals that remain constant during simulation run. It is very useful in gate-level testing. This chapter provides an overview of toggle coverage in Xcelium.

Types of Transitions

Change in signal value during simulation run is a transition, and rise, fall, and full transitions are recorded as shown below:



- Rise transition is a change in value from `0 -> 1` *or* from `x/z -> 1` (*if set_toggle_includex/z is used*)
- Fall transition is a change in value from `1 -> 0` *or* from `x/z -> 0` (*if set_toggle_includex/z is used*)
- Full transition is recorded for signals that have been through both rise and fall transitions.

Toggle coverage scores these rise/fall transitions on the following declarative objects:

Verilog	<ul style="list-style-type: none"> ● Scalar/vector nets of tri, trand, tri0, tri1, wire, want, wor types ● Scalar/vector variables of reg types
SystemVerilog	<ul style="list-style-type: none"> ● Scalar nets of tri, trand, trior, tireg, tri0, tri1, wire, wand, wor types ● Single dimensional packed vector nets of tri, trand, trior, tireg, tri0, tri1, wire, want, wor types ● Scalar variables of bit, logic, reg types ● Single dimensional vector variables of bit, logic, reg types ● Members of a packed/unpacked struct with no dimensions, the members being of any of the above a-d types <p><u>Exception :</u></p> <ul style="list-style-type: none"> ● Type-parameterized nets/variables are not supported
VHDL	<ul style="list-style-type: none"> ● Scalar signals of boolean, bit, std_ulogic, std_logic types ● Single dimensional signals - bit_vector, std_ulogic_vector, std_logic_vector types ● Members of a signal of non-array record type that are of any of the above a-b types

The toggle coverage items are covered in the following scopes:

Verilog/SystemVerilog	<ul style="list-style-type: none"> ● Nets/variables declared inside a Module/Interface are supported. nets/variables declared inside a generate are not supported. ● Ports of a Module/Interface are supported. ● Type-parameterized module/interface is not supported by default. You can enable this support using the set_parameterized_module_coverage.
VHDL	<ul style="list-style-type: none"> ● Signals declared inside architecture are supported. signals declared inside other scopes such as a generate are not supported. ● Ports of an entity are supported.

Signal Behavior Recognized in Integrated Coverage

The default behavior of toggle coverage in Integrated Coverage is described below.

- Toggle coverage by default does not recognize transitions that originate from an unknown state (X or Z). To enable toggle counting for signals that originate from X or Z values, use [set_toggle_includex](#) and [set_toggle_includez](#) commands in the coverage configuration file. With these commands, transitions `x -> 0`, `x -> 1`, `z -> 0`, and `z -> 1` are also recorded.
- Toggle coverage by default records all of the signals within a DUT. It does not allow recording of individual signals. However, you can exclude specific signals from toggle recording using the [set_toggle_excludedfile](#) command in the coverage configuration file at elaboration.

Note: If an uncovered toggle object is excluded for one instance and included for another instance, toggle report for the module treats the toggle object as excluded because present implementation gives higher precedence to exclude vs uncovered resolution.

- By default, glitches are considered for toggle scoring. This can lead to artificially high coverage counts. To prevent glitches from being considered for toggle recording, use the [set_toggle_strobe](#) command in the coverage configuration file. With glitch filtering enabled, a transition is recorded when a net transitions from one stable value to another stable value.

- By default, only toggle signal names of vector signals are reported for unexpanded nets and regs, and individual bits are reported for expanded nets in the toggle exclude file.

See [Chapter 9, "Generating Coverage Data,"](#) for more details on commands that can be provided in a coverage configuration file.

Note: Toggle coverage is supported for module instances under `generate` hierarchy.

Scoring Toggle Coverage

To score toggle coverage, you can either:

- Pass `-coverage toggle` to xmelab.
- Use the `select_coverage` command in a file and then pass this file using `-covfile` option to xmelab.

See [Chapter 9, "Generating Coverage Data,"](#) for more details.

After simulation has dumped toggle coverage data, you analyze it using reporting tool IMC. For more information on analyzing coverage data, refer to the *Integrated Metrics Center User Guide* in the Metric-Driven Verification (MDV) release.

Scoring Toggle Coverage for SV Enum Signals

SystemVerilog enum signals are now supported for toggle coverage. Each state or a name constant in enum declaration is considered toggled when it reaches a different state or in other words traverses to a different named constant in enum declaration. Consider the given example:

```
typedef enum logic [1:0]
{
    STATE1 = 2'b00,
    STATE2 = 2'b01,
    STATE3 = 2'b11,
    STATE4 = 2'b10,
} enum_state;
```

The scoring of SV enum signals is different from the signal bit signals. In the given example, a state is considered as toggled for SV enum when it changes from STATE2->STATE3. And an enum is considered fully toggled when the state has moved or traversed to all states, such as in this case the enum is considered fully toggled only when STATE3,STATE2, STATE4, and STATE1 are hit. When all the states are reached, the enum has a grade of 4/4 for toggle coverage.

Further, the enum is considered as partially toggled if only a few states out of all the declared states are hit. For example, if an enum hits only STATE1 and STATE3, then it is considered as partially toggled and has a grade of 2/4. Overall, these partial or fully toggled enum grades are added to toggle grades for bits and bussed signals to find cumulative toggle grades for the full design.

You can enable the support for SV enum signals by using the [`set_toggle_scoring -sv_enum`](#) command. You can also extend the behavior of this command to enable the scoring of enumerated members declared inside SystemVerilog structures by using the [`- sv_enum_with_struct`](#) option.

Automatic Marking of Coverage

Design code often includes coverage items that cannot be exercised by a testbench. Some of these un-exercised items (or coverage points) could be false coverage holes present due to design errors or due to constant drivers in the design. These un-exercised coverage items reduce coverage figures due to which designers are unable to achieve coverage targets.

It is imperative for the designer to detect these un-exercisable coverage items and analyze them so as to either fix the design errors or omit the un-exercisable items from coverage figures.

The Constant Object Marking (COM) feature enables you to automatically detect many un-exercisable coverage items in the design, mark them, and ignore them from coverage figures. The tool performs an extensive analysis and identifies many constant items. However, it is not as complete as for instance a complex formal analysis, and remaining items may need to be identified and manually marked. This chapter discusses automatic marking of coverage items using COM.

Note: In Integrated Coverage, you can also use manual marking of coverage items through pragmas. See [Chapter 6, "Coverage Pragmas,"](#) for details on manual marking.

COM Analysis

COM analysis is a two-step process.

- Step 1: Identification of constant objects/expressions and inactive blocks
Identification of constant objects is done during the elaboration phase. To enable COM, you use the `set_com` command in the coverage configuration file and pass this file to xmelab.
- Step 2: Marking of constant objects/expressions and inactive blocks
During the simulation phase, the identified objects are automatically marked and saved to the coverage database and `icc.com` file.

During COM analysis, first constant nets are identified, then constant expressions, and finally constant blocks. COM analysis saves valuable verification effort because these constant objects can never be covered by the testbench.

COM and Toggle Coverage

Objects in a design might not toggle because either the designer has intended to keep the circuit in a particular mode or such objects could have been left inadvertently unconnected or are driven constant. During COM analysis, the following toggle objects are marked.

Verilog	<ul style="list-style-type: none"> ● <code>wire</code> ● <code>reg</code>
VHDL	<ul style="list-style-type: none"> ● <code>signal</code> (<code>std_ulogic</code>, <code>std_logic</code>, <code>std_ulogic_vector</code>, <code>std_logic_vector</code>, <code>bit</code>, <code>bit_vector</code>, <code>boolean</code>) ● <code>variable</code> (<code>std_ulogic</code>, <code>std_logic</code>, <code>std_ulogic_vector</code>, <code>std_logic_vector</code>, <code>bit</code>, <code>bit_vector</code>, <code>boolean</code>)

COM analysis for toggle coverage involves identifying and marking objects `IGN` that never toggle during simulation run. If `set_com` is specified for a particular module-instance and a net is marked as `IGN`, then the nets directly connected to this net are also identified as `IGN` even if `set_com` is not specified for the module-instance in which they are declared.

COM and Expression Coverage

COM analysis for expression coverage involves analyzing all of the expressions in the design to determine either if individual expression terms cannot happen or if the complete expression is constant. Expressions containing the following operators can be marked.

Verilog	VHDL (from standard and std_logic_1164 packages)
Concatenation and multi-concatenation	Logical operators (<code>and</code> , <code>nand</code> , <code>or</code> , <code>nor</code> , <code>xnor</code> , <code>xor</code> , <code>not</code>)
Logical operators (<code>&&</code> , <code> </code> , <code>!</code>)	Relational operators (<code>></code> , <code><</code> , <code><=</code> , <code>=</code> , <code>/=</code> , <code>>=</code>)
Bitwise operators (<code>&</code> , <code> </code> , <code>~</code> , <code>^</code> , <code>~^</code>)	
Reduction operators (<code>&</code> , <code> </code> , <code>~&</code> , <code>~ </code> , <code>~^</code>)	

Unary operators (+, -)	
Relational operators (==, !=, ==, !=, >, >=, <, <=)	
Conditional operator (?:)	

Note: Verilog expressions containing an event operator (@) or event-or operator are not marked. Support for COM is limited to scalar and one-dimensional arrays.

Expression terms that the tool determines cannot happen are marked IGN and excluded from coverage. If the complete expression is constant, then all terms are marked IGN and the expression is excluded from coverage.

COM and Block Coverage

Verilog and VHDL blocks that are controlled by `if`, `if-else`, and `case` expressions are marked if the corresponding controlling condition can be marked. Blocks that are immediate children of a control statement (`if` or `case`) are marked as `IGN` if the controlling expression determines that this block cannot execute.

When branch coverage is scored, expressions controlling each branch are analyzed to mark corresponding blocks. For example, if branch coverage is enabled and COM is turned ON, the block consisting of the false part of the following ternary statement will be marked as `IGN` in the coverage report.

```
assign w = (1'b1) ? a:b;
```

COM and Coverage Calculation

Consider the following Verilog code.

```

module tb();
    wire [dout;] → Constant objects
    top U1(2'b10, dout);
endmodule

module top(a, z);
    input [1:0] a;
    output [z;]
    reg r;

    assign z = 1'b1;
    always@{a
        begin
            if(a == 2'b00) → Expression evaluates to constant (logic FALSE)
                $display("Inside if block"); → Inactive block
        end
    endmodule

```

If COM is turned ON (using `set_com` in coverage configuration file) and block, expression, and toggle coverage is enabled, then the following `icc.com` file is generated.

Line Number	<pre> ;; --Compilation-- ;; In Emacs: Click on warnings to get to their sources ;; This file has been generated because constant object marking ;; feature was enabled through the coverage configuration file. tmp/test.v:2: Toggle object 'dout' evaluates to constant '1' and will be excluded from coverage (tb) tmp/test.v:15: Block is inactive and will be excluded from coverage (tb.U1) tmp/test.v:14: Relational expression evaluates to constant (tb.U1) tmp/test.v:7: Toggle object 'a[1]' evaluates to constant '1' and will be excluded from coverage (tb.U1) tmp/test.v:7: Toggle object 'a[0]' evaluates to constant '0' and will be excluded from coverage (tb.U1) tmp/test.v:8: Toggle object 'z' evaluates to constant '1' and will be excluded from coverage (tb.U1) tmp/test.v:9: Toggle object 'r' evaluates to constant 'X' and will be excluded from coverage (tb.U1) </pre>
Location of design file	Item marked

This file details the coverage items marked due to COM. If you are using an editor like emacs, then the items in this file link directly to the related source files.

For toggle objects, the constant value of the object is printed in the `icc.com` file. The constant value can be `0`, `1`, or `X`. A value `X` is printed for objects that evaluate to anything other than constant `0` or `1`.

Un-exercised toggle objects are reported using the `-marked` option in the detailed report, as:

Number of signal bits fully toggled: 0 of 0 Number of signal bits partially toggled(rise): 0 of 0 Number of signal bits partially toggled(fall): 0 of 0 Number of signal bits marked COV: 0 Number of signal bits marked IGN: 1	→ Marked IGN
Hit(Full) Hit(Rise) Hit(Fall) Signal	
IGN IGN IGN dout	
Instance name: tb.U1 Module/Entity name: top File name: /tmp/test.v Number of signal bits fully toggled: 0 of 0 Number of signal bits partially toggled(rise): 0 of 0 Number of signal bits partially toggled(fall): 0 of 0 Number of signal bits marked COV: 0 Number of signal bits marked IGN: 4	
Hit(Full) Hit(Rise) Hit(Fall) Signal	
IGN IGN IGN a[1] IGN IGN IGN a[0] IGN IGN IGN z IGN IGN IGN r	

Inactive blocks are marked IGN and are not considered in coverage calculations. Inactive blocks are reported using the `-marked` option in the detailed report as:

Number of blocks marked COV: 0 Number of blocks marked IGN: 1	→ Marked IGN
mark marked block line no. line origin description	
IGN 15 true part of 14 if(a == 2'b00)	

Expressions marked IGN are not considered in coverage calculations and are reported using the `-marked` option in the detailed report, as:

Instance name: tb.U1 Module/Entity name: top File name: /tmp/test.v Number of expr items marked COV: 0 Number of expr items marked IGN: 2	→ Marked IGN
Line Coverage Expression description	
14 --% (0/0) if(a == 2'b00) a == 2'b00 <1> <--2---> hit <1> <2> == IGN lhs == rhs IGN lhs != rhs	

There will be no effect on blocks, expressions, or objects that are otherwise (in the absence of COM) not reported, such as objects inside protected code or expressions determined constant and reported in COVSEC warnings.

Evaluation of Objects by Integrated Coverage

An object is evaluated as a constant/variable based on the following principles:

- Ports on top-level modules are variable.
- Everything outside or driven from outside the [DUT Interface](#) is considered variable.
- A net bit that has exactly one continuous driver is marked with the same constant marking as its driver.
- A net bit that has a number of continuous drivers is marked according to resolution of the driving objects as shown below.

Driver value	0	x	1	0x	1x	V	UNC
0	0	x	x	0x	x	0x	0
x	x	x	x	x	x	x	x
1	x	x	1	x	1x	1x	1
0x	0x	x	x	0x	x	0x	0x
1x	x	x	1x	x	1x	1x	1x
V	0x	x	1x	0x	1x	V	V
UNC	0	x	1	0x	1x	V	UNC

where,

- x - Unknown bit value
- 1 - One bit value
- 0 - Zero bit value
- v - Variable bit value
- 0x - Bit that cannot be one
- 1x - Bit that cannot be zero
- UNC - Unconnected bit value

- For a reg/signal to be evaluated, there should be a single assignment (bit/part/full) statement for it.
 - A reg/signal that has exactly one sequential assignment (including an initial value) is evaluated to RHS of the assignment.
 - A reg/signal that has more than one sequential assignments is evaluated to a variable even if a bit is not assigned multiply in those assignments, and may have evaluated to a constant.
- In VHDL, an index/slice object with globally static index/slice expression value is not analyzed and the object is evaluated to a variable.
- Net, reg, or parameter references are evaluated to values of corresponding nets, regs, or parameters.
- Unconnected signals are treated as `x` when determining values.
- A function call expression is evaluated to a variable.
- A task/function input, output, inout is evaluated to a variable.
- Verilog force/release/assign/deassign assignments are evaluated to a variable.
- A bit select expression is evaluated to the corresponding constant value of the selected signal if the bit index is globally static or it can be evaluated to a constant string that represents an integer. If the index expression is evaluated to a constant string that has `x` or `u`, this bit select is evaluated to `x`. If the index expression is evaluated to a variable, the signal constant string should be checked and if it has the same values in all possible positions, the corresponding constant value will be a result, or a variable otherwise.
- A part select is evaluated similar to a bit select expression.
- Verilog memories/multi-dimensional arrays are evaluated to a variable.
- VHDL signals of type other than `std_ulogic`, `std_logic`, `std_ulogic_vector`, `std_logic_vector`, `bit`, `bit_vector`, and `boolean` are evaluated to a variable.
- Verilog nets driven by UDPs/interconnect path delays/MOS/bidirectional switches are evaluated to a variable.
- VHDL implicit/guard signal objects are evaluated to a variable.
- When an expression condition of an `if/case` statement evaluates to constant `UNC/x`, it is treated as a constant 0 (FALSE).

 The following constraints are ignored for evaluation:

- Delays in assignments
- Simulation time assignments using TCL commands such as `force`
- Simulation time assignments via VHPI/VPI
- Simulation time initialization using `-ncinitialize` switch

DUT Interface

Different testbenches might exercise different parts of the design code. DUT inputs may have constant values that vary between simulations. To ensure that we do not falsely ignore coverage of code that may be disabled by a constant input in one simulation, all DUT ports are considered variable. The DUT interface is determined as follows:

- By default, the top-level module is assumed as the DUT interface.
- If the `-covdut` or `+nccovdut` option is used during elaboration, then only the modules defined with these options are considered as the DUT interface.
- If the `set_com_interface` command is used in the coverage configuration file, then the modules defined with this command are considered as the DUT interface. The DUT interface defined with `set_com_interface` overrides the above two DUT definitions.

Note: It is recommended that when the complete DUT is scored for coverage, you use `- covdut` to define the DUT interface. When only a sub-set of the DUT is scored for coverage, you use `set_com_interface` to define the true DUT interface.

See [Chapter 9, "Generating Coverage Data,"](#) for more details.

Coverage Pragmas

This chapter discusses use of coverage pragmas to mark uncovered items that should not be considered in the overall analysis of coverage. You embed coverage pragmas in the source code to disable scoring on those portions of the design. The portions disabled using pragmas are marked IGN and are ignored from coverage counts. Coverage pragmas can only reduce scoring already enabled through a coverage configuration file or the `-coverage` option. Coverage pragmas added to

a module apply to all instances of that module.

Coverage pragmas are embedded in the source code (pre-simulation marking). Coverage pragmas save run time, but if later you want to check coverage on the disabled areas, you must remove the pragmas in the HDL code and re-simulate.

Marking Coverage Items Using Pragmas

By embedding pragmas, you mark portions in the design for which you want to disable scoring (block/expression/fsm/toggle/all).

To disable/enable coverage scoring, use:

```
<comment_indicator> pragma coverage [{coverage_type} = ] {on | off}
```

where,

```
<comment_indicator> ::= // | -- | /*  
<coverage_type> ::= block | expr | toggle | fsm
```

Note: You cannot enable/disable scoring of few branches/statements within a block. Pragmas are applied to the block. If the complete block is embedded within pragmas, then all the statements/branches within the block are ignored. If few statements within a block are embedded within pragmas, then pragmas are ignored, and coverage for that block (including branch and statement, if enabled) is scored.

Coverage pragmas take effect until a subsequent coverage pragma reverses their effect, or until the end of the translation unit (source file). For example, the pragmas defined in the code below disable scoring of block coverage from Line 91 to 95.

```
88 wire bit_cnt_reset = reset | spi_fs ;
89
90 // pragma coverage block = off
91 always @(posedge m_spi_clk or posedge bit_cnt_reset)
92     if (bit_cnt_reset)
93         bit_cnt <= 0 ;
94     else if (not_full)
95         bit_cnt <= bit_cnt + 1 ;
96 // pragma coverage block = on
```

To disable scoring of all coverage types simultaneously, use:

```
// pragma coverage off
```

This pragma is also equivalent to:

```
// pragma coverage block = off, expr = off, fsm = off, toggle = off
```

 To disable FSM coverage scoring using pragmas, specify coverage pragmas around the state register declaration, as shown below:

```
//pragma coverage fsm = off
reg state_variable;
//pragma coverage fsm = on

always @()
begin
    ...
    case state_variable
        ...
    end
```

If different coverage pragmas provide conflicting instructions, the last pragma instruction always supersedes the preceding instructions.

 Pragmas within an inactive `ifdef are ignored.

Pragmas and Coverage Calculation

Coverage items disabled using pragmas (pragmatized items) are marked IGN and are ignored from coverage calculations. For the above code, if we generate block coverage data without embedding pragmas, coverage results are:

Number of covered blocks: 30 of 33				
Number of blocks turned off by user: 0				
Number of blocks marked COV: 0				
Number of blocks marked IGN: 0				
cnt	covered block	line no.	line	origin description
50	92		92	if (bit_cnt_reset)
18	93	true part of	92	if (bit_cnt_reset)
50	94	false part of	92	if (bit_cnt_reset)
50	95	true part of	94	else if (not_full)
0	94	* implicit else	94	else if (not_full)
50	101		101	if (not_full)
50	102	true part of	101	if (not_full)
0	101	* implicit else	101	if (not_full)

Coverage results after adding pragmas are:

Blocks 91 - 95 not reported	Number of covered blocks: 26 of 28 → 28 instead of 33				
	Number of blocks turned off by user: 0				
	Number of blocks marked COV: 0				
	Number of blocks marked IGN: 5 → Ignored due to pragmas				
	cnt	covered block	line no.	line	origin description
	50	101	101	if (not_full)	
	50	102	101	if (not_full)	true part of
	0	101	101	if (not_full)	* implicit else

You can view items marked, using the `-marked` option of the `report_detail` command.

Note: Pragmatized items still are tracked in the coverage model file and contribute to the design checksum.

Coverage Pragmas and translate_off/translate_on

Code within `translate_off` and `translate_on` of synopsys and pragma types is not instrumented until either a matching `translate_on` pragma is encountered or a coverage pragma explicitly enables coverage instrumentation. By default, coverage pragmas are always enabled unlike simulation pragmas, which are enabled using the `-lexpragma` option. Consider the given example, in which simulation pragmas have been enabled using the `-lexpragma` option. In this case, simulation pragmas are turned off in line 38 and then turned back on in line 46. Further, the toggle coverage pragma on line 43 will have no effect. This is because any code that has been disabled for compilation, and consequently simulation, through, for example, simulation pragmas cannot be analyzed for coverage including coverage pragmas.

```
33 -- pragma coverage block = on, expr = on
34 process(rst, clk) begin
35 if (rst = '1') then Ai <= 0; Ao <= 0; diff <= 0;
36 elsif (rising_edge(clk)) then
37 if (push = '1') and (f = '0') and (pop = '0') then
38 -- pragma translate off
39     Ai <= bump(Ai); diff <= diff + 1;
40     elsif (pop = '1') and (e = '0') and (push = '0') then
41         Ao <= bump(Ao); diff <= diff - 1;
42     end if;
43 -- pragma coverage toggle = on
44 end if;
45 end process;
46 -- pragma translate on
```

Instrumentation suspended
for block and expression

Instrumentation explicitly
enabled for toggle will have no effect

Note: For information on how pragmas are processed in a run, see [NC-Verilog Simulator Help](#) guide.

Disabling/Enabling Implicit Block Scoring

To disable/enable scoring of implicit `else` and `default` blocks using pragmas, use:

```
<comment_indicator> pragma coverage
    [{implicit | implicit_else | implicit_default} = ] {on | off}
```

where,

- `<comment_indicator>` can be `//` or `--` or `/*`
- `implicit` indicates disabling/enabling both `implicit else` and `default` blocks.
- `implicit_else` indicates disabling/enabling `implicit else` blocks.
- `implicit_default` indicates disabling/enabling `implicit default` blocks.

To disable `implicit else` scoring, use:

```
// pragma coverage implicit_else = off
```

To disable scoring of both `implicit else` and `default` blocks simultaneously, use:

```
// pragma coverage implicit = off
```

This pragma is equivalent to:

```
// pragma coverage implicit_else=off, implicit_default=off
```

Disabling Explicit Default Scoring

You can disable scoring of a `default` statement in a fully described `case` using pragmas, as shown below.

```
case (ps)
    ST1: ...
    ST2: ...
    ST3: ...
    // pragma coverage block = off
    default: $display("Error - This should never happen");
    // pragma coverage block = on
endcase
```

Ignoring Coverage Pragmas

To ignore coverage pragmas, use:

```
// pragma coverage pragmas = off
```

With this statement, subsequent coverage pragmas have no effect on coverage instrumentation until a matching `pragmas = on` statement is encountered as shown below:

```
// pragma coverage pragmas = on
```

You can also ignore coverage pragmas by commenting them out, as shown below:

```
////pragma coverage off
-----pragma coverage off
```

Functional Coverage

Functional coverage can help you identify untested functionality in the design and provide a view of verification completeness from a functional point of view. It offers an insight on how the verification goals set by a test plan are being met, and can be performed on user-defined coverage points, specified using PSL, SystemVerilog assertions, or covergroup statements. These coverage points specify scenarios, error cases, corner cases, and protocols to be covered and also specify analysis to be done on different values of a variable.

By exercising the important areas of the design with the help of user-defined functional coverage points, you can target 100% coverage.

This chapter covers the following topics:

- Control-Oriented Using PSL/SVA Statements
 - Functional Coverage Points in PSL
 - PSL cover Directive
 - PSL assert Directive
 - Functional Coverage Points in SVA
 - SVA assert directive
 - SVA assume directive
 - SVA cover directive
- Data-Oriented Using SystemVerilog Covergroup
 - Defining a Covergroup
 - Example: Defining a Covergroup
 - Example: Defining Covergroups with Arguments
 - Defining Coverpoints
 - Coverpoint Bins
 - Example: Creating a Scalar Bin

- Example: Bins for an enum type coverpoint
 - Example: Coverpoint Bins with Type Real Based Variables
 - Example: Covergroup input Argument to Specify Bin Ranges
 - Example: Covergroup ref Argument to Specify Coverpoint Expressions
 - Example: Coverpoint Specified on a Type Real Expression
 - Example: Bins with Clause for Coverpoint Bins
 - Example: Using Dynamic Arrays in set_covergroup_expression for Coverpoint Bins
 - Transition Bins
 - Automatic Bin Creation for Coverpoints
 - Specifying Guard Expressions
 - Defining Wildcard Bins
 - Excluding Coverpoint Values or Transitions or Cross Bins from Coverage Results
 - Example: Empty bins
 - Example: Automatic bins
 - Example: The auto_bin_max option
 - Example: Guard expressions
 - Example: Fixed Size Vector Bins
 - Example: Transition Bins
 - Example: Cross bin
 - Example: Real Coverpoint - Singleton Value in Ignore Bin
 - Example: Real Coverpoint - Fully Overlapping Range in Ignore Bin
 - Example: Real Coverpoint - Partial Overlapping Range in Ignore Bin
 - Example: Illegal/Ignore Bins Containing a with Clause
 - Specifying Illegal Coverage Point Values or Transitions
- Defining a Cross
 - Hierarchical Cross Coverage

- Automatic Cross Bins
- User-Defined Cross Bins
- Specifying Guard Expressions with Crosses
- Specifying with Clause at the Cross Bin Level
- Declaring a Cross of a Cross
- Predefined Coverage Methods
- Predefined Coverage System Tasks and System Functions
 - `$set_coverage_db_name()`
 - `$get_coverage`
- Specifying Coverage Options
 - Instance-Specific Covergroup Options
 - Example: Using weight Option
 - Example: Using per_instance Option
 - Example: Using at_least, goal, name, and comment Options
 - Example: Using the cross_auto_bin_max Option
 - Example: Using the detect_overlap Option
 - Example: Using the cross_num_print_missing Option
 - Example: Using Covergroup Input Arguments to Set Coverage Options
 - Type-Specific Covergroup Options
 - Example: Using comment Type Option
 - Example: Using weight Type Option
 - Example: Using goal Type Option
 - Example: Using strobe Type Option
 - Example: Using real_interval Type Option
 - Example: Using merge_instances Type Option
 - Procedural Assignment of Covergroup Options
 - Covergroups in Parameterized Classes/Modules

- Covergroups in classes - Performance considerations
 - Covergroups in Compilation Units
 - Covergroups in Interfaces
 - Covergroups in Program Blocks
 - Covergroups in Generate Blocks
 - Declare and instantiate the covergroup in same generate block
 - Declare the covergroup in parent scope and instantiate it in the generate block
 - Scope of Covergroup and Covergroup Instances
- Scoring Functional Coverage
- Functional Coverage using Incisive Assertion Library

 To access the functional coverage capabilities, you need an Xcelium license.

Control-Oriented Using PSL/SVA Statements

Control-oriented functional coverage is an extension of assertion-based verification and identifies interesting functions directly. It helps you verify the temporal aspect of the design behavior, or how it behaves as simulation time progresses. In Integrated Coverage, control-oriented functional coverage points are specified using PSL or SVA assert, assume, and cover directives. The coverage to be measured is directly specified using the PSL/SVA statements or is interpreted from them.

For more on writing assertions in PSL, see the [Assertion Writing Guide](#).

Functional Coverage Points in PSL

Integrated Coverage scores PSL assert, assume, and cover directives.

The following verification directives are not considered functional coverage points, even if they are present in the description:

- assume_guarantee

- restrict
- restrict_guarantee
- fairness
- strong

PSL cover Directive

The PSL `cover` directive causes the verification tool to check if a certain path is covered during verification. The PSL `cover` directive can be applied only to PSL sequences.

The PSL `cover` statement can be specified in the following ways.

- ```
// psl sequence ThreeSucc = {a==3;a==4;a==5};
// psl cover ThreeSucc @ (posedge clk);
```

The above statements first define a PSL sequence and then apply the `cover` directive to this sequence. As the `cover` directive is not labeled, reports related to this `cover` directive show an internally generated name `__cover<n>_ThreeSucc`, where `<n>` is an internally generated positive integer.

- ```
// psl int_cover_once: cover S2;
```

This `cover` directive applies to a predefined sequence `S2`. As the `cover` directive is labeled, the reports related to this `cover` directive will show the name `int_cover_once`.

A few more examples of `cover` directives are as follows.

```
// psl sequence COV_FIFO = {!rst && rose(count==FIFO_size - 1)};  
// psl sequence COV_FIFO_EMPTY = {!reset && rose (count == 0)};  
// psl cover COV_FIFO @ (posedge clk);  
// psl cover COV_FIFO_EMPTY @ (posedge clk);
```

These `cover` directives specify FIFO full and empty situations.

PSL assert Directive

The PSL `assert` directive expresses required design behavior. The simulator checks that the design conforms to the specified behavior. The `assert` directive is applied to PSL property declarations. The primary objective of adding the `assert` directive is to specify interesting scenarios and corner cases and to ensure that the scenarios specified by the properties are not violated during verification. The main purpose of the PSL `assert` directive is to identify bugs during verification.

The PSL `assert` directive can be specified in the following ways:

- `// psl assert always {P; Q; R} |=> {V; W};`

This `assert` directive specifies the condition in which the signals V and W are high in successive clock cycles, after signals P, Q, and R have been high in the preceding three successive cycles. Reports related to this `assert` directive show an internally generated name `__assert<n>`, where `<n>` is an internally generated positive integer.

- `// psl property P1 = always ((mux_out==0) || (mux_out==1));`
`// psl AssertP1: assert P1;`

The above specification defines a property P1, which states that the value of the variable `mux_out` must always be either 0 or 1, and asserts this property. As the `assert` directive is labeled, the reports show the name `AssertP1`.

Although the primary utility of PSL `assert` directives is to identify bugs in the design, they are also useful in providing coverage information. Consider the following assertion:

```
// psl property P1 always (pre -> post) @ (posedge clk);  
// psl assert P1;
```

This assertion states that whenever `pre` evaluates to true, `post` must also evaluate to true in the same clock cycle. However, if `pre` never occurs, the assertion will never fail. If the user only monitors the failures of assertions, there can be a false sense of security, as far as this assertion is concerned because the failure count for this assertion is shown as 0. From a coverage perspective, it is important to know that the condition was tested. This is equivalent to having the following statement:

```
// psl cover {pre} @ (posedge clk);
```

Another advantage of the `assert` directive for functional coverage points is to have assertions that are specifically added to identify interesting scenarios, and would never fail.

Consider the following assertion:

```
// psl property DATA32_AFTER_REQ_ACK = always ({req; ack} |=> {data[=32]}) @
(posedge clk);
// psl assert DATA32_AFTER_REQ_ACK;
```

This assertion identifies the case when the data bit has been high for 32 consecutive cycles, after request and acknowledge have occurred in two successive cycles.

At any time during simulation, an assertion can have one of the following states:

inactive	There are currently no partial matches of the sequence of conditions described by the assertion.
active	The first sequence of the enabling condition is satisfied, and the assertion has not finished or failed.
finished	The fulfilling condition for the assertion has evaluated to true or has discharged without failing.
failed	The fulfilling condition for the assertion has evaluated to false.
disabled	The property has been disabled by an <code>assertion -disable</code> command or other external control.

Note: Specifying functional coverage points using the `assume` directive is similar to using the `assert` directive from a coverage point of view.

The finished count of a PSL `cover`, `assert`, or `assume` statement is provided as an indicator of the functional coverage. These counts can be observed during simulation run, and can be stored in the coverage database (based on user-specified commands) for post-process analysis through the reporting tool IMC.

Note: Supported PSL directives written inside a SystemC module are considered as functional coverage points. For more information on usage of assertion inside the SystemC module, refer to the *NC-SC Simulator Reference* guide.

Note: You can use the `sysvbind` flag to enable/disable selected coverage types on all instances that are binded.

Functional Coverage Points in SVA

Integrated Coverage scores SVA `assert`, `assume`, and `cover` directives excluding negative assertions specified with the `not` construct.

SVA directives specify how an assertion is used during the verification process -- as a behavior to be checked, or for collecting coverage information. Only properties with a verification directive are evaluated. To specify functional coverage points in SVA, you first define a property and then apply the `assert`, `assume`, or `cover` directives to it.

Consider the following property definition:

```
property P1;  
  @(negedge clk)  
  (a[*2] ##3 b[*2]) | => (d);  
endproperty
```

After defining a property, you apply the SVA `assert`, `assume`, or `cover` directives to it.

SVA assert directive

The `assert` directive specifies the property to be checked during the simulation. The `assert` directive is used to enforce a property as a checker and is specified as:

```
assert property (P1);
```

SVA assume directive

The `assume` directive specifies a property that must hold true throughout verification. When used in simulation, it checks that only valid stimuli is tested.

Note: Specifying the `assume` directive is similar to the `assert` directive.

SVA cover directive

The `cover` directive causes the property to be checked during simulation, and the results are used to generate coverage information. The `cover` statement can specify an action to take when the property passes.

The following example defines a property, `c1`, and uses the property to generate coverage information:

```
property C1;
  @(posedge clk)
  a || b ##2 c;
endproperty
cover property (C1) $display("Covering C1");
```

Creating Sequential Regular Expressions

Properties are often constructed out of sequential behavior. A sequence is a list of SystemVerilog boolean expressions in a linear order of time. You can declare a sequence with optional formal arguments. When the sequence is instantiated, actual arguments can be passed to the sequence. Consider the following code:

```
sequence S1 (term2, term3);
  (a ##1 term2 ##1 term3);
endsequence
sequence S2 (CC, DD);
  (a ##1 CC) or (b ##1 DD);
endsequence
assert property
  (@(negedge clk) (S1(b,c)) | => S2((e||f), g));
```

In the above code, the `S1` sequence is defined with two formal arguments, `term2` and `term3`. When the sequence is used in an assertion, actual arguments `b` and `c` are substituted for the formal arguments. The `S2` sequence is used similarly.

For more on writing SystemVerilog assertions, refer to [Assertion Writing Guide](#).

i Functional coverage scores:

- Concurrent PSL and SVA assertions of types assume, assert, and cover.
- Type-based coverage of System Verilog immediate assertions inside a class declared in a Package or a Compilation Unit.
- Functional coverage does not score:
 - System Verilog immediate assertions inside a class declared in a module.
 - Instance-based coverage of immediate assertions inside a class.
 - VHDL asserts
 - Plain properties
- Concurrent assertions describe patterns of behavior that span clock cycles. Concurrent assertions are distinguished by the property keyword, and are triggered by a clock, which must be explicit. Immediate assertions are executed like procedural code during simulation. An immediate assertion tests an expression when the statement is executed in the procedural code. For more on immediate and concurrent assertions, refer to the [*Assertion Writing Guide*](#).

Data-Oriented Using SystemVerilog Covergroup

Data-oriented functional coverage focuses on tracking the number of times a variable receives specific values, transitions, or combination of these during a simulation run. As data-oriented functional coverage focuses on the change in data values received by variables at different simulation events, it is easy to interpret. However, the complexity increases with the increase in value ranges, and inclusion of transitions, cross-products, combinations, and invalid values.

Data-oriented functional coverage helps you identify untested data values or subranges. For this, one needs a mechanism to track values of specific variables in a design. Verification engineers can track the values of those variables by grouping them in one or more covergroups. Variables grouped in a covergroup are known as coverpoints.

Note: This guide discusses SystemVerilog functional coverage. For details on SystemC functional coverage, see [*NC-SC CVE Library Reference*](#).

This section discusses:

- [**Defining a Covergroup**](#)

- [Defining Coverpoints](#)
- [Defining a Cross](#)
- [Predefined Coverage Methods](#)
- [Specifying Coverage Options](#)
- [Covergroups in Classes](#)
- [Covergroups in Compilation Units](#)
- [Covergroups in Interfaces](#)
- [Covergroups in Program Blocks](#)
- [Covergroups in Generate Blocks](#)
- [Covergroups in Packages](#)
- [Scope of Covergroup and Covergroup Instances](#)
- [Adding Covergroups in Verification Units](#)

Defining a Covergroup

A covergroup is a collection of coverpoints (a group of variables/expressions whose values are simultaneously sampled and scored). You define a covergroup using the `covergroup` construct. A `covergroup` construct is similar to a user-defined type, which is defined once and many variables of the type are created subsequently. Similarly, you declare a `covergroup` construct once and subsequently create instances of that construct to track coverage.

Following is the supported Backus-Naur Form (BNF) for defining a covergroup. BNF is widely used for describing the syntax of a programming language.

```
covergroup_declaration ::=  
covergroup covergroup_identifier [([ tf_port_list])] [coverage_event] ;  
{ coverage_spec_or_option }  
endgroup [ : covergroup_identifier ]  
  
tf_port_list ::= tf_port_item { , tf_port_item }  
  
tf_port_item ::= {attribute_instance}  
[tf_port_direction] data_type_or_implicit  
[port_identifier [ = expression ]]
```

```
tf_port_direction ::= port_direction  
  
port_direction ::= input | ref  
  
port_identifier ::= identifier  
  
coverage_spec_or_option ::=  
{attribute_instance} coverage_spec  
| {attribute_instance} coverage_option ;  
  
coverage_spec ::=  
cover_point  
| cover_cross  
  
coverage_event ::=  
clocking_event  
| with function sample ( [ tf_port_list ] )  
  
coverage_option ::=  
option. member_name = expression  
type_option.member_name = constant_expression  
  
variable_decl_assignment ::=  
...  
| covergroup_variable_identifier = new
```

- i** The BNF specified for covergroup and coverpoint is a subset of one defined by the IEEE 1800 standard. For information on complete coverage specification, refer to the SystemVerilog LRM.

Example: Defining a Covergroup

Consider the following covergroup type definition:

```
covergroup cg @(posedge clk);  
...  
endgroup  
cg cg_inst = new(); //instance of cg
```

The above statements defines a covergroup type `cg` and creates an instance `cg_inst` of covergroup `cg`. Use of parentheses `()` after the `new` keyword is optional. "..."represents the contents of a covergroup, which are coverpoints and cross definitions. See [Defining Coverpoints](#) and [Defining a Cross](#) for more details.

Considerations when defining covergroups

When defining covergroups, remember that direct assignments between covergroup instances, as shown below, are not supported.

```
covergroup cg @(posedge clk);  
  ...  
endgroup  
cg cg_inst1;  
cg cg_inst2;  
cg_inst1 = cg_inst2; //direct assignment between covergroup instances
```

Example: Defining Covergroups with Arguments

Consider the following covergroup type definition:

```
covergroup cg (input int size) @ (clk);  
  ...  
endgroup  
cg cg_inst1; //instance of cg declared  
cg cg_inst2; //instance of cg declared  
cg_inst1 = new(2); // value passed to formal argument size during instantiation  
cg_inst2 = new(5); // value passed to formal argument size during instantiation
```

In the above code, covergroup `cg` has a formal argument `size` of type `input`. Values `2` and `5` are passed to the formal arguments during covergroup instantiation for `cg_inst1` and `cg_inst2`, respectively.

For the covergroup arguments of the type `input`, you can also define their default values in the covergroup declaration. For instance, in the above example the default value of `size` can be specified as `2` by modifying the covergroup type definition as:

```
covergroup cg ( input int size = 2 ) @ (clk); // default value of the input argument specified
...
endgroup

cg cg_inst1; // instance of cg declared
cg cg_inst2; // instance of cg declared
cg_inst1 = new; // value not specified for formal argument size during instantiation
cg_inst2 = new(5); // value passed to formal argument size during instantiation
```

In the above code, in `cg_inst1` the value for `size` is not specified and therefore, `size` is set to the default value 2. However, in `cg_inst2`, the value for `size` is specified as 5, which overrides the default value.



Covergroup arguments can be of type `input` or `ref`. Default values can only be specified for `input` arguments, and only integral and string values are currently supported as the default values.

Covergroup arguments of type `input` can be used to:

- Specify bin values/ranges/size
- Set covergroup options
- Specify coverpoint expressions, cross expressions, and guard expressions

Covergroup arguments of type `ref` can be used to specify coverpoint expressions and guard expressions.

Note: When using covergroup arguments of type `input` in coverpoint/cross/guard expressions, only a single value (the value available during covergroup instantiation) is sampled throughout the simulation. However, in the case of covergroup arguments of type `ref`, all values are sampled during simulation run.

Consider the following code snippet.

```

covergroup cg (input logic[1:0] size, ref logic[1:0] f1) @(clk);
    A: coverpoint size {
        bins b1[] = {[0:3]};
    }
    B: coverpoint f1 {
        bins b2[] = {[0:3]};
    }
endgroup
cg cg_inst1;
...
v1 = 3;
cg_inst1 = new(v1,v1); cg_inst1.set_inst_name("cg_inst1");
v1 = 2; ... v1 = 0; ... v1 = 1;
...

```

In the above code, covergroup `cg` has two coverpoints `A` and `B`. Coverpoint `A` is declared on covergroup input argument `size`, while coverpoint `B` is declared on covergroup span style="font-size: 10.0pt;line-height: 13.0pt;font-family: Arial , Helvetica , FreeSans , sans-serif;background-color: transparent;"> argument `f1`. For coverpoint `A`, only a single value 3 will be sampled throughout the simulation. However, for coverpoint `B` all values for variable `v1` will be sampled.

The type of the covergroup argument can be explicitly specified, or inherited from the previous argument. If the type is not explicitly specified, then the default type is `input`, if it is the first argument in the covergroup definition. Otherwise, the type is inherited from the previous argument. Consider the following covergroup definition:

```
covergroup cg (ref int size1, int size2) @(clk);
```

In the above covergroup definition, the type of the argument `size2` is inherited from the previous argument and, therefore, is considered `ref`.

For detailed examples on defining covergroups with arguments, see:

- [Example: Covergroup input Argument to Specify Bin Ranges.](#)
- [Example: Covergroup ref Argument to Specify Coverpoint Expressions.](#)

Considerations when defining covergroups with arguments

When defining covergroups with arguments, remember that:

- Covergroup arguments can only be of integer data types (`shortint`, `longint`, `int`, `byte`, `bit`, `logic`, `reg`, `integer`). Use of non-integral types, such as arrays and their part selects is not supported.
- Specifying default values for covergroup arguments of the type `ref` is currently not supported.

```
covergroup cg (ref reg[2:0] r_in = 5 ) @(clk); //Parsing error as not supported
```

```
A:coverpoint r_in;  
endgroup
```

- Covergroup arguments cannot be specified in sampling conditions.
- Covergroup `input` arguments of type `string` can be used only to set covergroup options `name` and `comment`.
- Covergroup `ref` arguments cannot be specified in any of the following:
 - Bin declaration
 - Setting covergroup options
 - Clocking events (not supported)

 You can create and instantiate a covergroup within a module, interface, program block, or a class. For information on scope of covergroup and its instances, refer to [Scope of Covergroup and Covergroup Instances](#).

Defining Coverpoints

The variables whose values should be tracked are defined as coverpoints within one or more covergroups. During a simulation run, the values for variables defined as coverpoints is tracked and stored in the coverage database.

Following is the supported BNF for defining a coverpoint.

```
cover_point ::=  
[cover_point_identifier : ] coverpoint expression [ iff( expression ) ] bins_or_empty  
  
bins_or_empty ::=  
{ {attribute_instance} { bins_or_options ; } }  
| ;  
  
bins_or_options ::=  
coverage_option  
  
[ wildcard ] bins_keyword bin_identifier [[[ expression ]]] = {open_range_list}  
[with (expression)] [iff ( expression )]  
| [ wildcard ] bins_keyword bin_identifier [[[ expression ]]]  
= cover_point_identifier with [( expression )] [iff ( expression )]  
| bins_keyword bin_identifier [ [ ] ] = trans_list [ iff ( expression ) ]
```

```
|bins_keyword bin_identifier [ [ expression ] ] = default [ iff ( expression
) ]
|bins_keyword bin_identifier = default sequence [ iff ( expression ) ]

bins_keyword ::= bins | illegal_bins | ignore_bins

open_range_list ::= open_value_range { , open_value_range }

open_value_range ::= value_range

value_range ::= expression | [expression:expression]

trans_list ::= ( trans_set ){,( trans_set )}

trans_set ::= trans_range_list { => trans_range_list}

trans_range_list ::=

trans_item

| trans_item [ *repeat_range ]
| trans_item [ -> repeat_range ]
| trans_item [ = repeat_range ]

trans_item ::= range_list

range_list ::= value_range { , value_range }

repeat_range ::= expression | expression : expression

covergroup_expression ::= expression
```

A covergroup expression can be a:

- Constant expression
- Global and instance constants, which are members of the enclosing class
- Non-reference arguments

Function calls may participate in a `covergroup_expression`, but the following semantic restrictions are imposed:

- Functions shall not contain output or non-const ref arguments (const ref is allowed).
- Functions shall be automatic (or preserve no state information) and have no side effects.
- Functions shall not reference non-constant variables outside the local scope of the function.
- System function calls are restricted to constant system function calls

To define a coverpoint named B1 for tracking the values of variable var1, use the following

statement in the covergroup definition:

```
B1: coverpoint var1;
```

Here, B1 is the `cover_point_identifier` that is used while reporting. With the above definition, all values for the variable var1 are tracked. You can also track type real variables and expressions in coverpoints.

For detailed examples on specifying coverpoints on type real variables and expressions, see:

- [Example: Coverpoint Specified on a Type Real Expression](#)

While defining a coverpoint, you can associate a label with it as:

```
A: coverpoint expr;
```

Here, 'A' is the label for the coverpoint.

If a label is not specified and the coverpoint expression is a simple variable, then the variable name is used as the label for the coverpoint. It is a good practice, as well as highly recommended to label the coverpoints within a covergroup with a user-specified label. If these coverpoints are left unlabelled then the tool generates coverpoint names internally, which in some cases may cause downstream issues for merging, resilience, etc and may limit the effective usage. Labeling coverpoints also helps with the analysis of these coverpoints within detailed reports and in IMC GUI.

```
covergroup cov_trans_beat @cov_transaction_beat;  
  
option.per_instance = 1;  
beat_addr : coverpoint addr {  
    option.auto_bin_max = 16; }  
beat_dir : coverpoint trans_collected.read_write;  
beat_data : coverpoint data {  
    option.auto_bin_max = 8; }  
beat_wait : coverpoint wait_state {  
    bins waits[] = { [0:9] };  
    bins others = { [10:$] }; }  
beat_addrXdir : cross beat_addr, beat_dir;  
beat_addrXdata : cross beat_addr, beat_data;  
endgroup : cov_trans_beat
```

In the given example the labels `beat_addr`, `beat_dir`, `beat_data`, and `beat_wait` are added for the coverpoints.

A new parser time switch, `-cp_expr_as_name`, has been added to `xmvlog` and `xrun`. When this parser time switch is used, the names of the unlabeled coverpoints are generated based on the following rules:

- For coverpoints with simple identifier, the name is generated based on the identifier. For example:

```
covergroup cg;  
    coverpoint addr; //Coverpoint will be created with name "addr"  
endgroup
```

- For coverpoints containing hierarchical identifier, such as bit or part or member select expression, decompiled string of expression is used for naming. For example:

```
covergroup cg;  
    coverpoint arr[2]; // Coverpoint will be created with name "arr[2]"  
    coverpoint obj.trans; // Coverpoint will be created with name "obj.trans"  
    coverpoint ubus.addr[2].sig; // Coverpoint will be created with name  
    "ibus.addr[2].sig "  
endgroup
```

- For the coverpoints that are not in the above two categories, escaped name decompiled string of expression will be used for naming. For example:

```
covergroup cg;  
    coverpoint asig[0]+bsig[0]; //Coverpoint will be created with name  
    "\asig[0]+bsig[0] "  
    coverpoint addr*3; //Coverpoint will be created with name "\addr*3"  
endgroup
```

 As the internal names will change for unlabeled crosses, it will impact the usage of these names in cross definition, merging, and refinement.

Limitations of Coverpoints on Type Real Variable/Expression

The given scenarios are not supported:

- Coverpoint expressions with both real and integral variables
- Coverpoints with shortreal and realtime data types.
- Transition/wildcard bins with real based coverpoints. In these cases, relevant error messages are specified

- Use of greater than 32 bits integral values as bin values of real coverpoints
- Use of real expression, such as 4.5+3.99, as bin values
- Use of bin values with x/z/? characters
- Use of arguments, both input and reference, in coverpoint expressions
- Use of auto bins
- Creating vector bins out of default bins
- Bin-level merging for vector bins

 To use type real variables in coverpoints, the licence string Virtuoso_MMSIM_Incubation 1.0 is required.

Coverpoint Bins

A coverpoint bin is used to define the values that should be tracked and stored during a simulation run. A bin is defined using the `bins` keyword, as shown below:

```
[ wildcard ] bins_keyword bin_identifier [[[ covergroup_expression ]]]  
= {covergroup_range_list} [with (with_covergroup_expression)][iff ( expression )]  
| [ wildcard ] bins_keyword bin_identifier [[[ covergroup_expression ]]]  
= cover_point_identifier with [( with_covergroup_expression )] [iff ( expression )]  
| [ wildcard ] bins_keyword bin_identifier [[[ covergroup_expression ]]] =  
set_covergroup_expression [iff ( expression )]
```

In the above definition,

- `bin_identifier` is the name of the bin.
- `[[[expression]]]` indicates if the bin is scalar or vector. In the absence of `[]`, a scalar bin is created. A scalar bin creates a single bin for all values in the `open_range_list`. Use of square brackets `[]` indicates creation of a vector bin. A vector bin creates a separate bin for each value in the `open_range_list`. To create a fixed number of vector bins, specify a single positive integral expression inside the square brackets.
- `open_range_list` specifies the set of values to be tracked for the variable. It can be a single value, a range of values, or an open range. An open range is defined using a `$`, where `$` at the right bound indicates maximum value of the expression on which the coverpoint is declared, and `$` at the left bound indicates minimum value of the expression on which the coverpoint is

declared. An open range can be:

Range	Description
[\$: value]	Defines the range as all values less than or equal to "value"
[value : \$]	Defines the range as all values greater than or equal to "value"
[\$: \$]	Defines range as all values for the associated coverpoint variable

\$ is legal only as part of a range specification. It cannot be used to specify a single value for a bin.

 The range can be of any base, such as binary, hexadecimal, decimal, or octal. In addition, range can be constant expressions, instance constants (for classes only), or non-ref arguments to the covergroup. The range can also be a variable expression and are evaluated at covergroup instantiation.

Examples of range definition of a bin:

- To define the range as values between 0 to 5 and 10, use:

```
bins b1[] = {[0:5], 10};
```
- To define the range as values between 0 to 5 and 9 to 14, use:

```
bins b1[] = {[0:5], [9:14]};
```
- To define the range as values 0, 2, and 7, use:

```
bins b1[] = {0,2,7};
```
- To define the range as hexadecimal values between 0 and F, use:

```
bins b1[] = {[`h0:`hF]};
```
- To define the range as values less than or equal to 5 and 10, use:

```
bins b1[] = {[:$:5], 10};
```
- To define the range as values greater than or equal to 7, use:

```
bins b1[] = {[7:$]};
```

 - To define the range of a cross AXB using a variable v2, use:

```
bins b1 = binsof(A) intersect{v2};
```

- `with` clause specifies that only the values in the `open_range_list` for which the expression evaluates to true are included in the bin. In the expression, the term "item" is used to represent the candidate value, which is of the same type as the coverpoint. You can use the name of the coverpoint instead of the `open_range_list` to denote all the values of the coverpoint. However, note that only the name of the coverpoint containing the bin being defined is allowed and any other name is not allowed.

When using the `with` clause, by default the expression is applied to the set of values in the `open_range_list` prior to distribution of values to the bins. If you want to allow the distribution of values before expression application, use the `distribute_first` covergroup option.

Note that only constant expressions, global and instance constants, or non-ref arguments to the covergroup can be used as variables in the expression.

The following scenarios are not supported when using `with` clause and an error is reported:

- `with` clause is specified in transition or default bins
- Bin definition specifies a coverpoint name other than its enclosing coverpoint
- A coverpoint identifier is used in bin definition without including a `with` clause
- Explicit name of a coverpoint, in case the coverpoint expression is not a simple variable, is not used in bin definition having `with` clause
- Any variable other than constant expressions, global and instance constants, or non-ref arguments to the covergroup are used as variables in expression in the `with` clause.

Limitations:

In the current release, the `distribure_first` covergroup option, wildcard bins containing a `with` expression, and `with` expressions for bins of real coverpoints are not supported. The support for these features is planned for future releases.

For detailed examples on coverpoint bins containing bins `with` clause, see:

- [Example: Bins with Clause for Coverpoint Bins](#)
- [Example: Illegal/Ignore Bins Containing a with Clause](#)

- `set_covergroup_expression` syntax enables you to specify an expression yielding an array of values that define the bin. In this case, any expression that yields an array of values, packed or unpacked, is supported. When you use `set_covergroup_expression` syntax, each element of the resultant array should be of integral type and may differ from the type of the coverpoint expression. If the elements of the array are of a type different than the coverpoint expression, assignment compatibility rules are applied as per section 6.22.3 of the IEEE 1800-1200 SystemVerilog LRM.

When using `set_covergroup_expression` syntax, an error is reported if:

- a syntax does not yield an array of values.
- any element of an array is non-integral.

Limitations:

- `set_covergroup_expression` syntax is not supported for the bins of real coverpoint.
- A coverpoint having bin with “`set_covergroup_expression`” syntax will be merged from two or more databases in IMC standard merge only if:
 - Bin identifier matches
 - Same set of valid bin values are present.
- A coverpoint having bin with “`set_covergroup_expression`” syntax will be merged from two or more databases in IMC bin-level merge, only if:
 - Bin identifier matches
 - If Vector bin/Auto bin has single value, set of valid bin values need not be same. If valid bin values across databases are different, then values from secondary database which are present in primary are only merged.
 - If Scalar bin/Sized vector bin/Auto bin has multiple values per bin, same set of valid values are present in bin.

For an example on using dynamic arrays in `set_covergroup_expression` syntax for coverpoint bins, see:

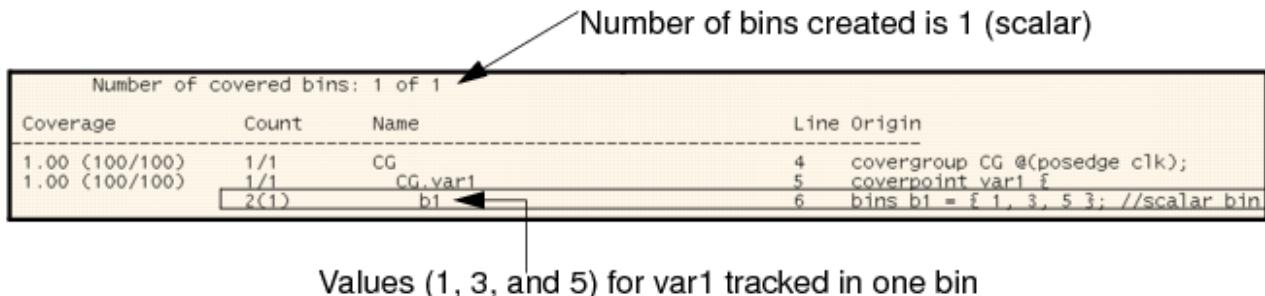
- [Example: Using Dynamic Arrays in `set_covergroup_expression` for Coverpoint Bins](#)

Example: Creating a Scalar Bin

Consider the following code:

```
coverpoint var1 {
    bins b1 = { 1, 3, 5 }; //scalar bin
}
```

The above code creates a scalar bin `b1` to track values of variable `var1`.



A screenshot of a SystemVerilog coverage report. At the top, it says "Number of covered bins: 1 of 1". Below is a table:

Coverage	Count	Name	Line Origin
1.00 (100/100)	1/1	CG	4 covergroup CG @(posedge clk);
1.00 (100/100)	1/1	CG.var1	5 coverpoint var1 f
	2(1)	b1	6 bins b1 = { 1, 3, 5 }; //scalar bin

Annotations: An arrow points from the text "Number of bins created is 1 (scalar)" to the "Number of covered bins" header. Another arrow points from the text "Values (1, 3, and 5) for var1 tracked in one bin" to the "b1" entry in the table.

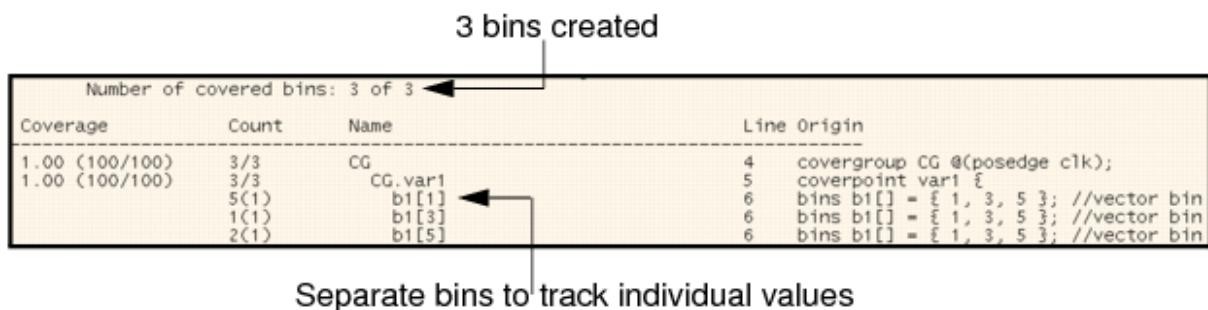
With a scalar bin, a single bin is created and you cannot identify how many times variable `var1` was assigned a particular value from the given range list.

Example: Creating a Vector Bin

Consider the following code:

```
coverpoint var1{
    bins b1[] = {1, 3, 5}; // vector bin
}
```

In the above code, use of square brackets [] indicates that `b1` is a vector bin and the set of values specified in the range list are tracked individually. The above code creates three bins: `b1[1]`, `b1[3]`, and `b1[5]` to store the value of variable `var1`.



A screenshot of a SystemVerilog coverage report. At the top, it says "Number of covered bins: 3 of 3". Below is a table:

Coverage	Count	Name	Line Origin
1.00 (100/100)	3/3	CG	4 covergroup CG @(posedge clk);
1.00 (100/100)	3/3	CG.var1	5 coverpoint var1 {
	5(1)	b1[1]	6 bins b1[] = {1, 3, 5}; //vector bin
	1(1)	b1[3]	6 bins b1[] = {1, 3, 5}; //vector bin
	2(1)	b1[5]	6 bins b1[] = {1, 3, 5}; //vector bin

Annotations: An arrow points from the text "3 bins created" to the "Number of covered bins" header. Another arrow points from the text "Separate bins to track individual values" to the "b1[1]", "b1[3]", and "b1[5]" entries in the table.

In this example, count is incremented individually for each of these bins depending on the number of times the value of `var1` was 1, 3, or 5 during simulation run.

⚠ If the range list for a vector bin includes duplicate values, a single bin is created to score duplicate values. For example, the following bin definition creates seven bins.

```
bins b1[] = {[1:7], 1};
```

Here, 1 is repeated twice in the range list. However, a single bin is created for it, as shown:

Number of covered bins: 7 of 7			Line	Origin
Coverage	Count	Name		
1.00 (100/100)	7/7	CG		
1.00 (100/100)	7/7	CG.var1		
5(1)		b1[1]	4	covergroup CG @ (posedge clk);
1(1)		b1[2]	5	coverpoint var1 {
1(1)		b1[3]	7	bins b1[] = {[1:7], 1};
3(1)		b1[4]	7	bins b1[] = {[1:7], 1};
2(1)		b1[5]	7	bins b1[] = {[1:7], 1};
4(1)		b1[6]	7	bins b1[] = {[1:7], 1};
3(1)		b1[7]	7	bins b1[] = {[1:7], 1};

Example: Creating a Fixed Size Vector Bin

Consider the following code:

```
coverpoint var1{
    bins b1[3] = {[1:10]}; // sized vector bin
}
```

In the above code, value 3 within the square brackets indicates that b1 is a fixed size vector bin of size 3, and the set of values specified in the range list is distributed across 3 bins.

You can also specify input argument for the size expression as shown below:

```
covergroup cg (int size);
    bins b1[size] = {[1:10]};
endgroup
cg cg_inst = new(3);
```

Distribution of bin values in the case of a fixed vector bin is explained below.

Assume,

N = Number of possible values

S = Size of vector bin

Q = N/S

R = N mod S (remainder)

First (S-1) bins will contain Q values, and last bin will contain (Q + R) values.

In this case,

N = 10

S = 3

Q = 3

R = 1

Therefore, the first ($3-1 = 2$) bins will include three values each, and the last bin will include four values, as:

```
b1[0] => will score values 1, 2, and 3
b1[1] => will score values 4, 5, and 6
b1[2] => will score values 7, 8, 9, and 10
```

The following instance-based report displays these bins.

Three bins created				
Coverage	Count	Name	Line	Origin
0.75 (75/100)	4/5	cg_mod	5	covergroup CG @(posedge clk);
1.00 (100/100)	3/3	cg_mod.var1	7	coverpoint var1 {
	1(1)	b1[0]	8	bins b1[3] = {[1:10]};
	2(1)	b1[1]	8	bins b1[3] = {[1:10]};
	1(1)	b1[2]	8	bins b1[3] = {[1:10]};
0.50 (50/100)	1/2	cg_mod.var2	12	coverpoint var2 {
	10(1)	b2[0]	13	bins b2[2] = {[64'h00000000000000000000000000000000 :64'h00000000000000000000000000000003]};
	0(1)	b2[1]	13	bins b2[2] = {[64'h00000000000000000000000000000000 :64'h00000000000000000000000000000003]};

Note: Duplicate values are retained in a fixed size vector bin. In the following code, value 1 is repeated in the range list, and is scored in both bins:

```
bins b1[2] = {[1:7], 1};
```

Here, value 1 is repeated in the range list. The above code creates two bins and the value 1 is scored in both bins, as shown below:

Number of covered bins: 2 of 2				
Coverage	Count	Name	Line	Origin
1.00 (100/100)	2/2	cg_mod	4	covergroup CG @(posedge clk);
1.00 (100/100)	2/2	cg_mod.var1	5	coverpoint var1 {
5(1)		b1[0]	7	bins b1[2] = {[1:7], 1};
5(1)		b1[1]	7	bins b1[2] = {[1:7], 1};

Duplicate value retained and scored in both bins

- ⓘ Do not specify size when creating transition bins. As per SystemVerilog LRM, it is illegal to specify a size with a transition bin. For details on transition bins, see [Transition Bins](#).

Example: Bins for an enum type coverpoint

An enum type variable is declared when you have a limited set of possible values as shown in the following example.

```
enum {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY} day1;
covergroup CG @(posedge clk);
    A: coverpoint day1{
        bins a1[] = {TUESDAY, THURSDAY, SATURDAY};
        bins a2[] = {$:WEDNESDAY};
    }
endgroup
```

In the above code:

Vectored bin `a1` creates three bins as: `a1[TUESDAY]`, `a1[THURSDAY]`, and `a1[SATURDAY]`, which stores the values `TUESDAY`, `THURSDAY`, and `SATURDAY` respectively.

Vectored bin `a2` creates four bins as: `a2[SUNDAY]`, `a2[MONDAY]`, `a2[TUESDAY]`, and `a2[WEDNESDAY]`, which stores the values `SUNDAY`, `MONDAY`, `TUESDAY`, and `WEDNESDAY` respectively.

Consider the following instance-based report generated from the above code.

Number of covered bins: 6 of 7					
Coverage	Count	Name	Line Origin		
0.86 (86/100)	6/7	CG	4	covergroup CG @(posedge clk);	
0.86 (86/100)	6/7	CG.A	5	A: coverpoint day1{	
	1(1)	a1[TUESDAY]	6	bins a1[] = {TUESDAY, THURSDAY, SATURDAY};	
	0(1)	a1[THURSDAY]	6	bins a1[] = {TUESDAY, THURSDAY, SATURDAY};	
	1(1)	a1[SATURDAY]	6	bins a1[] = {TUESDAY, THURSDAY, SATURDAY};	
	2(1)	a2[SUNDAY]	7	bins a2[] = {\$:WEDNESDAY};	
	1(1)	a2[MONDAY]	7	bins a2[] = {\$:WEDNESDAY};	
	1(1)	a2[TUESDAY]	7	bins a2[] = {\$:WEDNESDAY};	
	1(1)	a2[WEDNESDAY]	7	bins a2[] = {\$:WEDNESDAY};	

Enum constants as bin indices

In the above report, notice that enum constants are used for showing bin indices and uncovered bins are shown individually. For more on reporting, the *Integrated Metrics Center User Guide* in the Metric-Driven Verification (MDV) release.

When using an `enum` type coverpoint, it is recommended that:

- Explicit values to enum constants in the list (though allowed) should be avoided.

```
enum {ONE=1, THREE, SEVEN=7, EIGHT} e;
```
- In such declarations, the unassigned enum constants take increasing numbers from the previous numbered ones. In this case, enum constant `THREE` will be assigned a value 2, which might generate confusing output.

- Enum constants from the enum constant list are used to specify the bin range.

```
enum {ONE=1, TWO, SEVEN=7, EIGHT, THREE=3, FOUR} e;  
  
covergroup CG @(posedge clk);  
  
A: coverpoint e {  
  
    bins a1[] = {[1:7]};  
  
}  
  
endgroup
```

The above code generates a warning because bin range is specified using integer values for an enum type coverpoint variable. During simulation, if the values sampled for coverpoint `e` are 2, 3, 7, and 8, then only 2 and 7 will be considered as covered. The number of bins in this case will be three because only `ONE`, `TWO`, and `SEVEN` fall in this range.

For more on SystemVerilog enumerated data types, see the [SystemVerilog Reference Guide](#).

Example: Coverpoint Bins with Type Real Based Variables

Consider the following code:

```
real r1,r2;  
logic[3:0] i1;  
logic[1:0] rg1;  
  
...  
  
covergroup CG@(posedge clk);  
option.per_instance = 1;  
option.weight = 10;  
option.goal = 88;  
type_option.real_interval=0.1;  
  
C1:coverpoint r1{  
    bins b1[] = {0.001,[0.001:0.3]};  
    bins b1_1[] = {0.001,[3.001:3.3)};  
    bins b2[] = {[1.5:1.3)};  
    bins b3 = {1.15};  
    bins b4 = {1.005};  
    ignore_bins b5 = {1.005};  
    illegal_bins b6 = {1.15};  
}  
  
C2:coverpoint r2{
```

```
type_option.goal = 30;
bins b1 = {-1.001,3.0};
bins b2 = {[ -1:0]};

}

C3:coverpoint i1{
    bins b1 ={1,2,3,4};
}

C4:coverpoint rg1{
    bins b1 ={2'b0,2'b1};
}

CR1: cross C1,C2{
    bins my_b1 = binsof (C1.b1);
}

CR2: cross C1,C3;
endgroup
```

 For more information, refer to the [real interval](#) type option.

In the above code, real, ignore, and illegal values are a part of scalar and vector bin declaration. In addition, real values have been used in the binsof construct. The following are supported in covergroups:

- Scalar and vector bins with real based coverpoints
- Real values in scalar and vector bin declaration
- Real values in ignore/illegal scalar and vector bin declaration
- Real values in binsof/intersect construct
- Use of real input arguments and \$ in bin values

Example: Covergroup input Argument to Specify Bin Ranges

Consider the following code:

```
...
reg [2:0] v1;
covergroup cg (input int size) @(clk);
    coverpoint v1 {
```

```
bins b1[] = {[0:size]};  
}  
endgroup  
  
cg cg_inst1;  
cg cg_inst2;  
  
...  
cg_inst1 = new(2); cg_inst1.set_inst_name("cg_inst1");  
cg_inst2 = new(4); cg_inst2.set_inst_name("cg_inst2");  
...
```

In the above code, covergroup `cg` has a formal argument `size` of type `input`. This formal argument is used to specify the bin range of bin `b1`. In the code, two instances of covergroup `cg` are created. The bin range for both the instances differ based on the value passed to the covergroup argument `size`. The bin range for `cg_inst1` is `[0:2]` and for `cg_inst2` is `[0:4]`.

The above code uses covergroup method `set_inst_name`. For details on this method, see [Using the set_inst_name method](#).

 Covergroup `input` arguments can be used to specify ranges for transition/cross bins as well.

Example: Covergroup ref Argument to Specify Coverpoint Expressions

Consider the following code:

```
...  
covergroup cg ( ref int f1 ) @ (clk);  
option.per_instance = 1;  
A: coverpoint f1 {  
    bins b1[] = {[0:2]};  
}  
endgroup  
  
cg cg_inst1;  
cg cg_inst2;  
int x, y;  
  
...  
cg_inst1 = new(x); cg_inst1.set_inst_name("cg_inst1");  
cg_inst2 = new(y); cg_inst2.set_inst_name("cg_inst2");  
...
```

In the above code, covergroup `cg` has a formal argument `f1` of type `ref`. This argument is used to specify the expression of the coverpoint `A`. The covergroup has two instances, `cg_inst1` and `cg_inst2`. Instance `cg_inst1` samples variable `x`, and instance `cg_inst2` samples variable `y`.

Example: Coverpoint Specified on a Type Real Expression

Consider the following code:

```
module top;
    reg clk;
    real v1;
    real v2;

    covergroup cg @(clk);
        A: coverpoint v1+v2 {
            bins b1 = { 2.5 };
        }
    endgroup

    cg cg_inst1;
    initial
    begin
        cg_inst1 = new; cg_inst1.set_inst_name("cg_inst1");
        ...
    end
endmodule
```

In the above code, `cg` has a coverpoint declared on a real expression `v1+v2`. The coverpoint has a user-defined scalar bin `b1` which contains a single real literal value. In this example, coverpoint expression should be equal to 2.5 in order to result in a match with bin `b1`.

Considerations When Creating Coverpoints on Variables with size > 32 Bits

When creating coverpoints on variables with size greater than 32 bits, remember that:

The report will always show the bin values in hexadecimal format, irrespective of the range format specified in the HDL.

Consider the following code.

```
reg clk;
reg [63:0] v1;
logic [127:0] v2;
```

```
covergroup cg @(clk);
option.per_instance =1;
A: coverpoint v1 {
bins b1[] = {64'h0101010101010101};
}
B: coverpoint v2 {
bins b2[] = {128'd12000000}; //size specified with decimal constant
}
endgroup
```

The following instance-based report is generated from the above code:

Number of covered bins: 2 of 2			
Coverage	Count	Name	Line Origin
1.00 (100/100)	2/2	cg_inst1	5 covergroup cg @(clk);
1.00 (100/100)	1/1	cg_inst1.A	7 A: coverpoint v1 {
	1(1)	b1[64'h0101010101010101]	8 bins b1[] = {64'h0101010101010101};
1.00 (100/100)	1/1	cg_inst1.B	10 B: coverpoint v2 {
	1(1)	b2[128'h00000000000000000000000000000000b71b00]	11 bins b2[] = {128'd12000000 };

Bin shown in hexadecimal format

The above report shows the bin values corresponding to bin `b2` in hexadecimal format, even though the value is specified in the decimal format in the code.

If the size of the coverpoint is more than 32 bits and is not a multiple of 32, then the bin value is reported with size in multiples of 32. Consider the following code.

```
reg clk;
reg [55:0] v1;
covergroup cg @(clk);
A: coverpoint v1 {
bins b1[] = {56'h0101010101010101}; //value more than bin size
}
endgroup
```

The following instance-based report is generated from the above code:

Number of covered bins: 1 of 1			
Coverage	Count	Name	Line Origin
1.00 (100/100)	1/1	cg_inst1	5 covergroup cg @(clk);
1.00 (100/100)	1/1	cg_inst1.A	7 A: coverpoint v1 {
	1(1)	b1[64'h0000010101010101]	8 bins b1[] = {56'h0101010101010101};

Bin shown in multiples of 32

The above report shows bin value as `b1[64'h0001010101010101]` instead of `b1[56'h0101010101010101]`, which is in multiples of 2^{32} because the bin value is more than the size of the coverpoint variable.

Example: Bins with Clause for Coverpoint Bins

Consider the following code:

```
module top;
reg[15:0] buff;
covergroup cg@(posedge clk);
Buff : coverpoint buff{
    bins bin_b1[] = {[0:20]} with (item % 2 == 0);
}
endgroup
...
endmodule
```

In the given example, covergroup `cg` has coverpoint named `Buff`. Bin definition `bin_b1` specifies `with` clause to the bin range `([0:20])`. Here, the expression `(item % 2 == 0)` is applied to the each value of the range. Only those values for which the expression `(item % 2 == 0)` evaluates to true are included in the bin. So, in the given example, bin definition `bin_b1` selects only even values and odd values, including 1,3,5,7,9,11,13,15,17 and 19, are eliminated.

The sample report as it appears in IMC is shown.

Name	Average, Covered Grade	Line	Source Code
cg	0%, 0% (0/11)	7	covergroup cg;
--Buff	0% (0/11)	8	Buff: coverpoint buff{
--bin_b1[0]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[2]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[4]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[6]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[8]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[10]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[12]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[14]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[16]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[18]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);
--bin_b1[20]	0% (0/1)	9	bins bin_b1 = {[0:20]} with (item % 2 == 0);

Example: Using Dynamic Arrays in set_covergroup_expression for Coverpoint Bins

Consider the given example:

```

module top;
    reg [7:0] k ;
    reg [7:0] arr[] ;
    covergroup CG ;
        option.per_instance = 1 ;
        A : coverpoint k
        {
            bins b1[] = arr ;
        }
    endgroup
    CG tcov ;
    initial begin
        arr = new[3] ;
        arr[0] = 3 ;
        arr[1] = 5 ;
        arr[2] = 8 ;
        tcov = new ;
    end
endmodule

```

When you use `set_covergroup_expression`, bin definition `b1` selects values 3, 5, and 8. The sample report of the above example is shown:

<pre> CG ---A ---b1[3] ---b1[5] ---b1[8] </pre>	<pre> 0.00%, 0.00% (0/3) 0.00% (0/3) 0.00% (0/1) 0.00% (0/1) 0.00% (0/1) </pre>	<pre> 4 covergroup CG ; 6 A : coverpoint k 8 bins b1[] = arr ; 8 bins b1[] = arr ; 8 bins b1[] = arr ; </pre>
---	---	--

Transition Bins

A transition bin is used to define the value transitions that should be tracked and stored during a simulation run. BNF for defining a transition bin is:

```

[wildcard] bins_keyword bin_identifier [ [ ] ] = trans_list
bins_keyword ::= bins | illegal_bins | ignore_bins
trans_list ::= ( trans_set ) { ,( trans_set ) }

trans_set ::= trans_range_list { => trans_range_list}

trans_range_list ::=
trans_item
| trans_item [ * repeat_range ]
| trans_item [ -> repeat_range ]
| trans_item [ = repeat_range ]

trans_item ::= range_list

```

```

range_list ::= value_range { , value_range }

value_range ::= expression | [expression:expression]

repeat_range ::= expression | expression : expression
    
```

Using the above BNF, you can define and track:

Simple transitions

Consecutive repetitions

Non-consecutive repetitions

Simple Transitions

A simple transition is a sequence of sampled value changes for a coverpoint.

```

A: coverpoint a {
    bins b1 = (4=>1=>3); //simple transition
}
    
```

The above coverpoint definition creates a scalar transition bin `b1` for transition `4=>1=>3`. This bin is considered hit when a sequence of sampled values for coverpoint `A` is 4, followed by 1, followed by 3.

Sequence of sampled values for coverpoint `A` at different sampling times

sampling time	1	2	3	4	5	6	7	8	9	10	11	12	13	14
value	1	2	4	1	3	5	2	5	1	8	4	1	3	2

With the above sequence, bin `b1` is incremented at the 5th and 13th samples.h

Consider another example where, set of transitions is specified as a range list.

```

A: coverpoint a {
    bins b1[] = (4=>5=>6), ([7:9], 10=>11, 12);
}
    
```

The above bin definition creates two bins:

`b1[4=>5=>6]`, which stores transition `4=>5=>6`.

`b1[[7:9],10=>11,12]`, which stores transition `7=>11, 8=>11, 9=>11, 10=>11, 7=>12, 8=>12, 9=>12, and 10=>12`.

Sequence of sampled values for coverpoint A at different sampling times

sampling time	1	2	3	4	5	6	7	8	9
value	4	5	7	11	8	12	2	2	3

With the above sequence, bin `b1[7:9], 10=>11,12` is incremented at 4th and 6th samples for transitions `7=>11` and `8=>12`, respectively.

Note: Transition vector bins are expanded to report all possible transitions under a given definition. Consider the following code:

```
module top;
reg [7:0] a=0;
reg clk=0;
covergroup cg @(posedge clk);
    A: coverpoint a {
        bins b1[] = (1=>1,2,3) ;
    }
endgroup
cg cg_inst = new();
always
    #10 clk = ~clk;
initial
begin
    clk = 0;
    #20 a =1;
    #20 a =1;
    #20 a =3;
    #20 $finish;
end
endmodule
```

For the given example, the instance-based report is generated with multiple transitions for `b1` as follows:

Number of covered bins: 2 of 3				
Coverage	Count	Name	Line Origin	
0.67 (67/100)	2/3	cg_inst@1_1	4	covergroup cg @(posedge clk);
0.67 (67/100)	2/3	cg_inst@1_1.A	6	A: coverpoint a {
	1(1)	b1[1=>1]	7	bins b1[] = (1=>1,2,3);
	0(1)	b1[1=>2]	7	bins b1[] = (1=>1,2,3);
	1(1)	b1[1=>3]	7	bins b1[] = (1=>1,2,3);

All transitions reported

- i** In a transition vector bin, duplicate transition bins are not identified and removed across multiple transition sets. Consider the following code:

```

covergroup cg ()@(clk);
option.per_instance = 1;
A : coverpoint a {
    bins trans_b[] = (4,5 => 4,5), (5[*2]);
}
B : coverpoint a {
    bins trans_b[] = (4,5 => 4,5), (4[*2]);
}
CR1 : cross A,B;
CR2 : cross A,B {
    bins b1 = binsof (A) intersect {4};
}
CR3 : cross A,B {
    bins b1 = binsof (A) intersect {4};
    ignore_bins ig_b1 = binsof (A) intersect {4};
}
endgroup

```

For the given example, the instance-based report is generated with multiple entries for bin `trans_b[5=>5]` with same hit count as shown. In this report, if `trans_b[5=>5]` is hit all the duplicate entries show the hit counts and if the transition bin is not hit the hit counts for all the entries are displayed as zero.

Coverage	Count	Name	Line	Origin
0.80 (80/100)	4/5	cg1.A	8	A : coverpoint a {
	0(1)	trans_b[4=>4]	9	bins trans_b[] = (4,5[*2]), (5[*2]);
	0(1)	trans_b[4=>5]	9	bins trans_b[] = (4,5[*2]), (5[*2]);
	0(1)	trans_b[5=>4]	9	bins trans_b[] = (4,5[*2]), (5[*2]);
	3(1)	trans_b[5=>5]	9	bins trans_b[] = (4,5[*2]), (5[*2]);
	3(1)	trans_b[5=>5]	9	bins trans_b[] = (4,5[*2]), (5[*2]);

In the given example, the total number of bins in coverpoint `cg1.A` is 5 as against the expected 4 bins. In addition, if `trans_b[5=>5]` gets sampled, all the entries display the same hit count and are counted as covered, resulting in the total number of covered bins in coverpoint `cg1.A` as 2 against the expected 1 covered bin. This change in total number of bins and total covered bins also impacts the actual coverage percentage, which will be different from the expected coverage percentage.

Note that if a coverpoint containing duplicate transition bins is involved in a cross, then it will impact the total number of cross bins and also the total covered cross bins.

Consecutive Repetitions

Consecutive repetitions for transitions are specified as:

```
bins bin_identifier = trans_item [* repeat_range]
```

where `trans_item` is repeated for `repeat_range` times.

For example,

```
bins b1[] = (4 [* 5]); //5 times consecutive 4s
```

is the same as:

```
4=>4=>4=>4=>4
```

Consider another example:

```
bins b1[] = (4 [* 3:5]);
```

is same as:

```
(4=>4=>4), (4=>4=>4=>4), (4=>4=>4=>4=>4)
```

Consider another example:

```
A: coverpoint A {
    bins b1[] = (2=>3[* 2:4]=>1), (3=>4=>5);
    bins b2[] = (3[* 2:4]);
}
```

Sequence of sampled values for coverpoint A at different sampling times

sampling time	1	2	3	4	5	6	7	8	9
value	1	2	3	3	3	3	1	5	3
bin b1[2=>3[*2:4]=>1] hit									
bin b2[3[*2:4]] hits at 4 th , 5 th , and 6 th samples									

With the above sequence, bin `b1[(2=>3[* 2:4]=>1)]` is hit at 7th sample and bin `b2 [3[* 2:4]]` is hit at the 4th, 5th, and 6th samples.

Non-Consecutive Repetitions

Non-consecutive repetitions for transitions are specified as:

```
bins bin_identifier = trans_item [ -> repeat_range]
| trans_item [= repeat_range]
```

In the above BNF, both `->` and `=` indicate non-consecutive occurrence of `trans_item` for `repeat_range` times.

 Non-consecutive means 0 or more values other than `trans_item`.

For example:

```
A: coverpoint a {
    bins b1 = (7 => 12 [-> 2] =>5) ;
    bins b2 = (7 => 12 [= 2] =>5) ;
}
```

The above transition bins are translated as:

```
b1 = ...7=>12=>...=>12=>5
b2 = ...7=>12=>...=>12=>...=>5
```

The difference in the above sequences is:

For `b1` to be hit, the transition must start at sample value 7, followed by two non-consecutive 12s, immediately followed by 5.

For `b2` to be hit, the transition must start at sample value 7, followed by two non-consecutive 12s, followed by 5 at any sample (0 or more), but before any other occurrence of 12.

Sequence of sampled values for coverpoint A at different sampling times

sampling time	1	2	3	4	5	6	7	8	9	10
value	3	12	7	12	5	10	11	12	8	5
	↑ Transition starts for b1 and b2				b2 considered hit					

For the above sampled values, transition starts at 3rd sample. In this case, bin `b1` is not hit because after two non-consecutive 12s, value 5 is not sampled at 9th sample. However, bin `b2` is hit because after two non-consecutive 12s, value 5 is sampled at 10th sample but before any other occurrence of 12.

If the above bin definition is modified as:

```
bins b1 = (12 [-> 2] =>5) ;
```

Then for the above sampled values, bin `b1` is hit at 5th sample with transition details as:

Transition Starts	Transition Ends	Hit (Yes/No)
2 nd sample	5 th sample	Yes, because after two non-consecutive 12s, value 5 is sampled at 5 th sample.
4 th sample	9 th sample	No, because after two non-consecutive 12s, value 5 is not sampled at 9 th sample.
8 th sample	In complete	No, because transition is incomplete.

Note: Do not specify non-consecutive/unbounded transitions as vector bins.

Automatic Bin Creation for Coverpoints

While defining a coverpoint, if you do not specify any bin, SystemVerilog automatically creates a vector bin of size 64. For example, the following code creates a vector bin `cov_bin.auto` of size 64 with elements `cov_bin.auto[0]` to `cov_bin.auto[63]`.

```
covergroup cg @(posedge clk);
    cov_bin : coverpoint a_var;
endgroup
```

By default, the number of automatically created bins is 64. However, if required, you can modify the number of automatically created bins a particular coverpoint using the `auto_bin_max` option, as shown below:

```
option.auto_bin_max = expression;
```

where `expression` is the value to be specified for the option. You can also specify parameters as expressions.

The `auto_bin_max` option can be defined at different levels, as shown in following examples:

To define the number of automatically created bins for coverpoint `a` as 3, use:

```
covergroup cg @(posedge clk);
    coverpoint a{
        option.auto_bin_max = 3;
    }
endgroup
```

To define the number of automatically created bins for all the coverpoints within covergroup `cg` as 3, use:

```
covergroup cg @(posedge clk);
    option.auto_bin_max = 3;
    coverpoint a;
    coverpoint b;
endgroup
```

To define the number of automatically created bins for coverpoint `a` as 3 and for coverpoint `b` as 5, use:

```
covergroup cg @(posedge clk);
    option.auto_bin_max = 5;
    coverpoint a{
        option.auto_bin_max = 3;
    }
    coverpoint b;
endgroup
```

How are bins created when no bins are defined?

If no bins are defined for a coverpoint, SystemVerilog performs the following two steps to determine the required bins for that coverpoint.

Step #	Description	Example
1	<p>SystemVerilog calculates the number of bins (N) as minimum of 2^M and value of <code>auto_bin_max</code> option, where M is the number of bits needed to represent the coverpoint.</p>	<p>Code snippet</p> <pre>reg [0:3] a; covergroup cg @(posedge clk); coverpoint a{ option.auto_bin_max = 3; } endgroup</pre> <p>In the above code, the value of <code>auto_bin_max</code> is 3 and 2^M is 16. As the value of <code>auto_bin_max</code> is the lower of the two, the number of bins created will be 3.</p>

2	<p>SystemVerilog determines how the possible values (V) will be stored across number of bins (N) as:</p> <p>If $N < V$, then the values are evenly distributed among N bins.</p> <p>If V is not divisible by N, then the last bin includes the remaining values.</p>	<p>As the value 3 is less than 16, the values will be evenly distributed among the three bins. In this example, 16 is not divisible by 3. As a result, the last bin will include the remaining values. In this example, the total number of automatic bins created will be three, and the values will be distributed as:</p> <table style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; width: 30%;">Bin</th><th style="text-align: left; width: 70%;">Values</th></tr> </thead> <tbody> <tr> <td>auto[0:4]</td><td>(0, 1, 2, 3, 4)</td></tr> <tr> <td>auto[5:9]</td><td>(5, 6, 7, 8, 9)</td></tr> <tr> <td>auto[10:15]</td><td>(10, 11, 12, 13, 14, 15)</td></tr> </tbody> </table>	Bin	Values	auto[0:4]	(0, 1, 2, 3, 4)	auto[5:9]	(5, 6, 7, 8, 9)	auto[10:15]	(10, 11, 12, 13, 14, 15)
Bin	Values									
auto[0:4]	(0, 1, 2, 3, 4)									
auto[5:9]	(5, 6, 7, 8, 9)									
auto[10:15]	(10, 11, 12, 13, 14, 15)									

The calculation of the number of bins in the case of an `enum` type coverpoint varies. In the case of automatic bins, the total number of bins is computed based on the total possible value set for the coverpoint rather than the `enum` type. In addition, the bin index in the report displays enum constants instead of integer values. The following code illustrates this point.

```
event COV_EVENT;
typedef enum reg[2:0] {ADD,SUM,MUL,DIV} enum_val;
enum_val e1=ADD;
covergroup cg @(COV_EVENT);
    A: coverpoint e1;
endgroup // CG
```

During simulation, if the values sampled for coverpoint `e1` are ADD, SUM, MUL, and DIV, then the instance-based report is:

Number of bins			
Coverage	Count	Name	Line Origin
0.50 (50/100)	2/4	cg	7 covergroup cg @(COV_EVENT);
0.50 (50/100)	2/4	cg.A	8 A: coverpoint e1;
0(1)		auto[ADD]	8 A: coverpoint e1;
1(1)		auto[SUM]	8 A: coverpoint e1;
1(1)		auto[MUL]	8 A: coverpoint e1;
0(1)		auto[DIV]	8 A: coverpoint e1;

Note:

For more on reporting, see the Integrated Metrics Center User Guide in the Metric-Driven Verification (MDV) release.

Specifying Guard Expressions

Guard expressions help you control sampling and binning. Guard expressions are specified with the `iff` construct and are evaluated every time during sampling. If the guard expression evaluates to true at a sampling point, the coverpoint is sampled. If the guard expression evaluates to false, the coverpoint is ignored and is not sampled. Any valid expression can be specified as a guard expression.

In the following code, coverpoint `var1` is sampled only if the expression is true at the sampling event, which is the posedge of `clk`.

```
covergroup cg1 @(posedge clk);  
    coverpoint var1 iff (x || y && z) ;  
endgroup
```

Consider another code snippet.

```
covergroup cg1 @(posedge clk);  
    coverpoint var1 {  
        bins a[] = { 200, 201, 202 } iff (x) ;  
    }  
endgroup
```

During the simulation run, the bin count is incremented only if `x` is true at the sampling event.

 The value (true/false) of guard expression does not impact the total number of bins for a coverpoint. For example, in the above code, the total number of bins is three irrespective of the value of the guard expression `x`.

Defining Wildcard Bins

If a sampled value has `x`, `z`, or `?` in some bit position, then the value is ignored and is not sampled. Using the `wildcard` keyword in the bin definition, you can cause `x`, `z`, and `?` to be treated as wildcards for 0 and 1. For example,

```
wildcard bins b1 = {4`b11xx};
```

causes the count of bin `b1` to be incremented when the sampled value is between 12 and 16 that is: 1100, 1101, 1110, and 1111.

In the absence of `wildcard` keyword, the count for bin `b1` would have incremented if the sampled value was `11xx`.

As the use of `wildcard` keyword cause `x`, `z`, and `?` to be treated as wildcards for 0 and 1, you do not need to define bin values individually. For example,

```
reg [3:0] a;  
bins b1[] = {0010, 0100, 0110, 0000};
```

can be defined as:

```
wildcard bins b1[] = {0xx0};
```

OR

```
wildcard bins b1[] = {0zz0};
```

OR

```
wildcard bins b1[] = {0??0};
```

Transition bins can also be defined as wildcard bins. For example,

```
wildcard bins tb_1 = (2'b0x => 2'b1x);
```

causes the count of transition bin `tb_1` to be incremented for the following transitions:

```
00 => 10  
00 => 11  
01 => 10  
01 => 11
```

Wildcard bins with constant wildcard values or expressions containing wildcard values are supported. This stands true for wildcard vector bins, wildcard scalar bins, wildcard transition bins, wildcard sized vector bins, and wildcard illegal and ignore bins.

Consider the given example in which a wildcard bin with an expression containing wildcard values is supported:

```
module top;  
  
reg [5:0] a=0;  
reg clk;  
  
always begin  
    #10 clk = ~clk;  
    a = (a + 1) % 16;  
end  
  
covergroup cg @ (clk) ;  
    option.per_instance = 1 ;  
    coverpoint a {
```

```
wildcard bins b[] = { {{2'bx0}, {2'b0x}} } ; // concatenation expression
}

endgroup

cg cgi = new();

initial
begin
    clk = 0;
    cgi.set_inst_name("cg_inst");
    a = 0;
    #200 $finish;
end

endmodule
```

The instance-based report for the given example is shown:

```
All Data-oriented Functional Coverage Detail Report, Instance-Based
-----
Coverage Table Legend
-----
Coverage = Grade (Coverage%/Goal%)
Count for covergroup, coverpoint and cross = Covered bins/Total bins
Count for bins = Hit count (At least option value)

Instance name: top
Module/Entity/Interface/Program name: top
File name: /servers/scratch03/irfanc/wildcard_err/testcase/wildcard_expression/test1.v
Number of covered bins: 4 of 4

Coverage      Count     Name          Line Origin
-----
1.00 (100/100) 4/4   cg_inst      12   covergroup cg @ (clk) ;
1.00 (100/100) 4/4   cg_inst.a   14   coverpoint a {
    1(1)      b[0]       15   wildcard bins b[] = { {{2'bx0}, {2'b0x}} } ; // concatenation expression
    2(1)      b[1]       15   wildcard bins b[] = { {{2'bx0}, {2'b0x}} } ; // concatenation expression
    1(1)      b[8]       15   wildcard bins b[] = { {{2'bx0}, {2'b0x}} } ; // concatenation expression
    1(1)      b[9]       15   wildcard bins b[] = { {{2'bx0}, {2'b0x}} } ; // concatenation expression
```

⚠ Wildcard values in bin ranges are not supported in any wildcard or non-wildcard bin definition. The bin range can neither be a wildcard value nor an expression resulting in a wildcard value, and using such values is reported as an error.

Excluding Coverpoint Values or Transitions or Cross Bins from Coverage Results

You can exclude a set of values or transitions associated with a coverpoint from coverage by specifying them as `ignore_bins`. The values/transitions specified with ignore bins are excluded from the sampled values of coverpoints and does not result in any bin count increment. The bins specified with the `ignore_bins` keyword are removed from the total bin count of a coverpoint.

Consider the following code:

```
covergroup cg1 @(posedge clk);
    coverpoint a {
        bins b1[] = {0, 1, 2, 3};
        ignore_bins ignore_vals = {0, 3};
    }
endgroup
```

In the above example, if the sampled value of coverpoint `a` is 3, it is ignored and the count for bin `b1[3]` is not incremented. If during a simulation run, coverpoint `a` takes values 0, 1, 2, and 3, the coverage percentage is reported as 2/2, which is 100%. The values specified with `ignore_bins` have been removed from the total bin count.

- i** The default keyword is not allowed with `ignore_bins`. While defining `ignore_bins`, you must specify the value(s) that should be excluded from coverage results. For example, the following code will generate an error:

```
coverpoint a {
    ignore_bins ignore_vals = default;
}
```

Example: Empty bins

The `ignore_bins` specification in the bin declaration of a coverpoint may result in "empty" bins for a coverpoint. An empty bin is essentially a user-defined/automatic bin of a coverpoint for which all values are ignored using `ignore_bins`. If `ignore_bins` specification results in empty bins for a coverpoint, then empty bins will not be dumped in the coverage database. Consider the following code:

```
covergroup cg1 @(posedge clk);
    coverpoint a {
```

```
bins b1[] = {0, 1, 2, 3};  
bins b2 = {[5:8], 9};  
    ignore_bins ignore_vals = {0, 3, 1, 2};}  
}  
endgroup
```

In the above example, bin `b1` is an empty bin and will not be dumped because all of the values associated with the bin `b1` are specified as `ignore_bins`.

However, this is not true for following cases:

Scalar wildcard bins

In the case of scalar wildcard bins, even if the bin becomes empty due to `ignore_bins` or `illegal_bins` specification, it is still dumped to the coverage database. For example, the following wildcard bin definition results in "empty" bins.

```
covergroup CG @(posedge clk);  
  coverpoint var1 {  
    wildcard bins b1 = {1'bx};  
    ignore_bins ign = {0,1};  
  }  
endgroup // CG
```

In the above description, even though the scalar wildcard bin `b1` becomes empty due to `ignore_bins` specification, it is still dumped to the coverage database.

Transition wildcard bins

In the case of transition wildcard bins, even if the bin becomes empty due to `ignore_bins` or `illegal_bins` specification, it is still dumped to the coverage database. For example, the following wildcard transition bin definition results in "empty" bins.

```
covergroup CG @(posedge clk);  
  coverpoint var1 {  
    wildcard bins b1[] = (3'bz00 => 1'bx => 3);  
    wildcard ignore_bins ib[] = (3'bz00 => 1'bx => 3);  
  }  
endgroup // CG
```

In the above description, even though the wildcard transition bin `b1` becomes empty due to `ignore_bins` specification, it is still dumped to the coverage database. The hit count for the bin is displayed as 0.

⚠ If all the bins in a coverpoint become empty due to the `ignore_bins` specification, then that coverpoint is not dumped to the coverage database and the simulator generates a warning.

Example: Automatic bins

You specify `ignore_bins` for automatic bins, as shown below:

```
reg[0:2] a;
covergroup cg1 @(posedge clk);
    coverpoint a {
        ignore_bins ignore_vals = {0, 3} ;
    }
endgroup
```

In the above example, coverpoint `a` is a 3-bit variable and therefore 8 automatic bins are created. If during simulation run, all possible 8 values (0..7) occur, the percentage coverage is reported as 6/6, which is 100%. Values 0 and 3 are ignored and excluded from coverage results as they are specified with `ignore_bins`.

Example: The auto_bin_max option

The following code illustrates the impact of `ignore_bins` on calculation of coverage percentage when the `auto_bin_max` option is used.

```
reg[0:2] a;
covergroup cg1 @(posedge clk);
    coverpoint a {
        option.auto_bin_max = 4;
        ignore_bins ignore_vals = {2, 3} ;
    }
endgroup
```

For the above code, the total number of automatic bins will be four, as shown below:

Bin	Values
auto[0]	(0, 1)
auto[1]	(2, 3)

auto[2]	(4, 5)
auto[3]	(6, 7)

During sampling, values 2 and 3 are ignored because they are specified with `ignore_bins`. The count of `auto[1]` will not be incremented during sampling. As a result, `auto[1]` becomes an "empty" bin. During coverage reporting, `auto[1]` is ignored and the total number of automatic bins for coverpoint `a` is calculated as three and coverage is reported accordingly.

If the values specified with `ignore_bins` is (1, 3) instead of (2, 3), none of the automatic bins become empty bins and hence the coverage results change.

 The ignore values are applied after laying out the values across the automatic bins.

Example: Guard expressions

If `ignore_bins` is specified with a guard expression, then it is effective only if the guard expression is true at the time of sampling. Consider the following code:

```
covergroup cg1 @(posedge clk);
    coverpoint a {
        bins b1[] = {0, 1, 2, 3};
        ignore_bins ignore_vals = {0, 3} iff c+d ;
    }
endgroup
```

If during a simulation run, coverpoint `a` takes values 0, 1, 2, and 3, values 0 and 3 are ignored if and only if `c+d` is true at the point of sampling. If `c+d` is true throughout the simulation, the coverage percentage is reported as 2/4, which is 50%.

If the value of expression `c+d` is false at the time of sampling and the sampled values are 0 or 3, then these values are not ignored and the count corresponding to bins `b1[0]` and `b1[3]` is incremented. In this situation, coverage percentage is reported as 4/4, which is 100%.

Example: Fixed Size Vector Bins

In the case of a fixed size vector bin, all the values are first distributed across fixed size bins and then, the values specified with the `ignore_bins` are removed from the range list. For example:

```
A: coverpoint a
{
    bins a1[3] = {0,2,3,[4:6],8,9};
    ignore_bins ig = {2,3};
}
```

The above code creates following bins:

Number of covered bins: 2 of 3				
Coverage	Count	Name	Line	Origin
0.67 (67/100)	2/3	cgl_inst1	6	covergroup cg @ (negedge clk);
0.67 (67/100)	2/3	cgl_inst1.A	8	A: coverpoint a
1(1)		a1[0]	10	bins a1[3] = {0,2,3,[4:6],8,9};
0(1)		a1[1]	10	bins a1[3] = {0,2,3,[4:6],8,9};
9(1)		a1[2]	10	bins a1[3] = {0,2,3,[4:6],8,9};

For the above code, the following bins with the given values should be created:

```
a1[0] => will score value 0
a1[1] => will score value 4
a1[2] => will score values 5,6,8, and 9
```

Example: Transition Bins

For transition bins, even if a transition occurs, it cannot be considered hit if it is specified with `ignore_bins`. Consider the following code:

```
A: coverpoint a {
    bins b1[] = (2=>5=>1), (1=>4=>3);
    ignore_bins ignore_trans = (2=>5);
}
```

Sequence of sampled values for coverpoint A at different sampling times

sampling time	1	2	3	4	5	6	7	8	9	10	11	12	13	14
value	1	2	4	1	4	3	7	5	2	5	1	4	3	2

In this case, even though transition $2 \Rightarrow 5 \Rightarrow 1$ occurs at the 11th sample, it is never hit because transition $2 \Rightarrow 5$ is specified as `ignore_bins`. Here, bin `b1[1=>4=>3]` is incremented at the 6th and 13th samples.

 Do not specify unbounded length transitions inside `ignore_bins` or `illegal_bins`.

Example: Cross bin

The `ignore_bins` will not be considered for calculating total number of cross-coverage bins. If the sampled value of any of the coverpoints specified in a cross declaration is ignored at a particular sampling event, the count of the corresponding cross bin will not be incremented as well. Consider the following code:

```
reg [1:0] a;
reg [3:0] b;
covergroup cg @(posedge clk);
    coverpoint a {
        bins b1[] = {0, 1, 2, 3};
        ignore_bins ignore_vals = {0, 3};
    }
    B: coverpoint b;
    cross a, B;
endgroup
```

In the above code, if the sampled value of coverpoint `a` is 3 during simulation run, then the count for the corresponding cross bin will not be incremented. For more on cross bins, see [Defining a Cross](#).

Example: Real Coverpoint - Singleton Value in Ignore Bin

Consider the following example, in which covergroup, `cg1`, has a coverpoint declared on a real variable, `x_r`. The vector bin, `x1`, contains range expressions and ignore_bin, `ix`, contains a value which falls in bin range of `x1`:

```
module top;
    logic clk;
    real x_r, y_r;
    covergroup cg1 @(clk);
        type_option.real_interval = 0.1;
        coverpoint x_r {
            bins x1[] = {[2.4:2.8)};
            ignore_bins ix = {2.5};
        }
    endgroup : cg1
    cg1 cover_inst = new;
```

```
initial begin
#1 x_r = 2.5; clk=~clk;
end
endmodule
```

In the above code, the value 2.5 will be ignored during sampling and the rest of the bin will be sampled. The count of the bin b1[2.5:2.6) will not be incremented when the value is 2.5.

Example: Real Coverpoint - Fully Overlapping Range in Ignore Bin

In the following example, covergroup, `cg1`, has a coverpoint declared on a real variable, `x_r`. The vector bin, `x1`, contains range expressions and ignore_bin, `ix`, contains a range. One bin range completely falls in range specified by ignore bin.

```
module top;
logic clk;
real x_r, y_r;
covergroup cg1 @ (clk);
  type_option.real_interval = 0.1;
  coverpoint x_r {
    bins x1[] = {[2.4:2.8]};
    ignore_bins ix = {[2.5:2.6)};
  }
endgroup : cg1
cg1 cover_inst = new;
initial begin
#1 x_r = 2.5; clk=~clk;
end
endmodule
```

In the above code, the bin [2.5:2.6) will get removed as it falls completely inside the ignore_bins range. The rest of the bins will be sampled.

Example: Real Coverpoint - Partial Overlapping Range in Ignore Bin

In the following example, covergroup, `cg1`, has a coverpoint declared on a real variable, `x_r`. The vector bin, `x1`, contains range expressions and ignore_bin, `ix`, contains a ranges and value and none of the bin ranges completely fall in ignore_bin ranges.

```
module top;
logic clk;
real x_r, y_r;
```

```

covergroup cg1 @(clk);
    type_option.real_interval = 0.1;
    coverpoint x_r {
        bins x1[] = {[2.4:2.8)};
        ignore_bins ix = {[2.59:2.63], 2.79 };
    }
endgroup : cg1
cg1 cover_inst = new;
initial begin
#1 x_r = 2.5; clk=~clk;
end
endmodule

```

In the above code, none of the bin ranges is removed as none of them are completely within the ranges that are being ignored. The values that belong to the ignore_bins ranges will be excluded from sampling.

Example: Illegal/Ignore Bins Containing a with Clause

Consider the following code:

```

module top;
reg[15:0] buff;
covergroup cg@(posedge clk);
Buff : coverpoint buff{
    bins bin_b1[] = {[0:20]} with(item % 2 == 0)
    ignore_bins ign = Buff with(item % 3 == 0)
}
endgroup
-----
endmodule

```

In the given example, covergroup `cg` has a coverpoint `Buff`, which further contains two bins, `bins bin_b1` and `ignore_bin ign`. As with clause is specified, `bins bin_b1` selects only even values after applying the with expression and `ignore_bin ign` selects only the multiples of 3. Therefore, after applying with expression bins include the following values:

```

bins bin_b1[] = {0,2,4,6,8,10,12,14,16,18,20};
ignore_bins ign = {0,3,6,9,12,15,18,21,24,27...};

```

The sample report as it appears in IMC is shown.

Name	Average, Covered Grade	Line	Source Code
cg	0%, 0% (0/8)	7	covergroup cg;
--Buff	0% (0/8)	8	Buff: coverpoint buff{
--bin_b1[0]	0% (0/1)	9	bins bin_b1 = [0:20] with (item % 2 == 0);
--bin_b1[2]	0% (0/1)	9	bins bin_b1 = [0:20] with (item % 2 == 0);
--bin_b1[4]	0% (0/1)	9	bins bin_b1 = [0:20] with (item % 2 == 0);
--bin_b1[8]	0% (0/1)	9	bins bin_b1 = [0:20] with (item % 2 == 0);
--bin_b1[10]	0% (0/1)	9	bins bin_b1 = [0:20] with (item % 2 == 0);
--bin_b1[14]	0% (0/1)	9	bins bin_b1 = [0:20] with (item % 2 == 0);
--bin_b1[16]	0% (0/1)	9	bins bin_b1 = [0:20] with (item % 2 == 0);
--bin_b1[20]	0% (0/1)	9	bins bin_b1 = [0:20] with (item % 2 == 0);

Specifying Illegal Coverage Point Values or Transitions

You can mark a set of values or transitions associated with a coverpoint as illegal by specifying them as `illegal_bins`. If the sampled value or transition of a coverpoint matches with a value specified with `illegal_bins`, simulation generates an error.

The coverage calculation for `illegal_bins` is identical to `ignore_bins`. The only difference between the two is that in the case of `illegal_bins`, if the sampled value or transition of a coverpoint matches with a value specified with `illegal_bins`, simulation generates an error. No error is reported in the case of `ignore_bins`.

Illegal bins take precedence over any other bins, which implies they will result in a run-time error even if they are also included in another bin.

```
covergroup cg1 @(posedge clk);
    coverpoint a {
        bins b1[] = {0, 1, 2, 3};
        illegal_bins ill_vals = {1, 2};
        ignore_bins ign_vals = {2,3}
    }
endgroup
```

In the above code, value 2 is specified with both `illegal_bins` as well as `ignore_bins`. During simulation run, if the sampled value for the coverpoint `a` is 2, an error is reported. This is because `illegal_bins` takes precedence over any other bins.

Defining a Cross

To help designers track values received by more than one variable together as a group, a cross is defined between the coverpoints/variables whose values have to be tracked together. Besides this, a cross can also be specified in another cross. For more information on cross of a cross, refer to [Declaring a Cross of a Cross](#).

Following is the BNF for defining a cross:

```
cover_cross ::=  
[cross_identifier : ] cross list_of_coverpoints [ iff( expression ) ] bins_or_empty  
  
list_of_coverpoints ::= cross_item , cross_item { , cross_item}  
  
cross_item ::=  
cover_point_identifier | variable_identifier | hierarchical_cover_point_identifier  
  
hierarchical_cover_point_identifier ::=  
{ identifier constant_bit_select. } covergroup_instance_identifier.cover_point_identifier  
  
constant_bit_select ::= { [constant_expression] }  
  
  
bins_or_empty ::=  
{ {attribute_instance} { bins_or_options ; } }  
| ;  
  
bins_or_options ::=  
coverage_option  
| bins_keyword bin_identifier = select_expression  
  
bins_keyword ::= bins | illegal_bins | ignore_bins  
  
select_expression ::= select_condition  
| ! select_condition  
| select_expression && select_expression  
| select_expression || select_expression  
| ( select_expression )  
| select_expression with ( expression )  
| cross_identifier  
  
select_condition ::= binsof ( expression ) [ intersect { open_range_list } ]
```

```
expression ::= variable_identifier | cover_point_identifier [.bin_identifier]  
| hierarchcal_cover_point_identifier [.bin_identifier]
```

```
open_range_list ::= open_value_range { , open_value_range }
```

```
open_value_range ::= expression | expression : expression
```

The following code defines a few crosses between coverpoints:

```
reg [1:0] a, c;  
reg [3:0] b, d;  
covergroup cg @(posedge clk);  
    A: coverpoint a;  
    B: coverpoint b;  
    cross A, B; //cross on coverpoints of same covergroup  
    cross c, d; // cross directly on variables c and d  
    cross a, B; // cross on variable a and coverpoint B  
endgroup
```

 When a cross is defined directly on variables, a coverpoint is created implicitly for the variables participating in a cross. However, coverpoints created implicitly through a cross cannot be tracked through system tasks or methods. In addition, these coverpoints do not participate in coverage calculations and are not available in post-processing reports in IMC.

 If the total number of cross products for a cross exceeds the limit of 4294967269, then that cross will not be sampled and saved to the coverage database.

While defining a cross, you can associate a label with it as:

```
crossAB: cross A, B;
```

Here, crossAB is the label for the cross defined between coverpoints A and B. A label creates a hierarchical scope and is used for referring and reporting. It is a good practice, as well as highly recommended to label crosses within a covergroup with a user specified label. If these crosses are left unlabelled then the tool generates names internally, which in some cases may cause downstream issues for merging, resilience etc and limit the effective usage. Labeling the crosses also helps with the analysis of these crosses within detailed reports.

```
covergroup cov_trans_beat @cov_transaction_beat;  
    option.per_instance = 1;  
    beat_addr : coverpoint addr {  
        option.auto_bin_max = 16; }  
    beat_dir : coverpoint trans_collected.read_write;
```

```
beat_data : coverpoint data {
    option.auto_bin_max = 8; }
beat_wait : coverpoint wait_state {
    bins waits[] = { [0:9] };
    bins others = { [10:$] };
}
beat_addrXdir : cross beat_addr, beat_dir;
beat_addrXdata : cross beat_addr, beat_data;
endgroup : cov_trans_beat
```

In the given example, the labels `beat_addrXdir` and `beat_addrXdata` are added for the crosses.

Hierarchical Cross Coverage

Hierarchical references to coverpoints are supported in cross declaration. This enables the use of coverpoints from different covergroup scopes in a cross.

Note that for all hierarchically referenced coverpoints, the last sampled values are reused while sampling the cross. New sampling is not done for hierarchical coverpoints at the time of sampling the cross. Further, in case the hierarchical coverpoints have not been sampled even once, cross sampling is skipped. Cross sampling is also skipped if last sampled values for coverpoints are either ignore or illegal.

Besides hierarchical coverpoint names, hierarchical coverbin names are also supported. The support for hierarchical coverbin names enables writing cross bin expressions (binsof) by referring to the bins that are declared in a hierarchically accessed coverpoint.

Limitations of Hierarchical Cross Coverage

If the covergroup instance containing the coverpoints and the covergroup instance containing the cross are sampled at the same timeslot, the order of sampling of these two covergroup instances will depend on the order of construction ("new") of these instances. Therefore, if you want the covergroup instance with coverpoints to be sampled first, ensure that the corresponding covergroup instance is created before the one containing the cross.

If the hierarchical reference to a coverpoint changes because of object assignments, as shown in the given example, a run time error will be generated.

```
covergroup cg;
    cross o1.cg1.cp;
endgroup

initial
begin
    o2 = o1; // now o2 holds the reference to the coverpoint
    o1 = null; // object used in hierarchical cross reference is null
end
```

Using arrays to access the coverpoints in cross declaration is not supported. For example, in the given code an error will be generated:

```
covergroup cg;
  cross objs[1].cg1.A, B; // objs is array of class objects
endgroup
```

Hierarchical instantiation of a covergroup object having formal arguments in its declaration is not supported. For example, the given code is not supported:

```
module test;
  covergroup cg(reg [3:0] var1);
    A: coverpoint var1 {bins b1[] = {1,2,3,4};}
  endgroup
  cg cgi;
endmodule

module top;
  test i1();
  reg [3:0] a;
  initial begin
    i1.cgi = new(a); //Unsupported Error
  end
endmodule
```

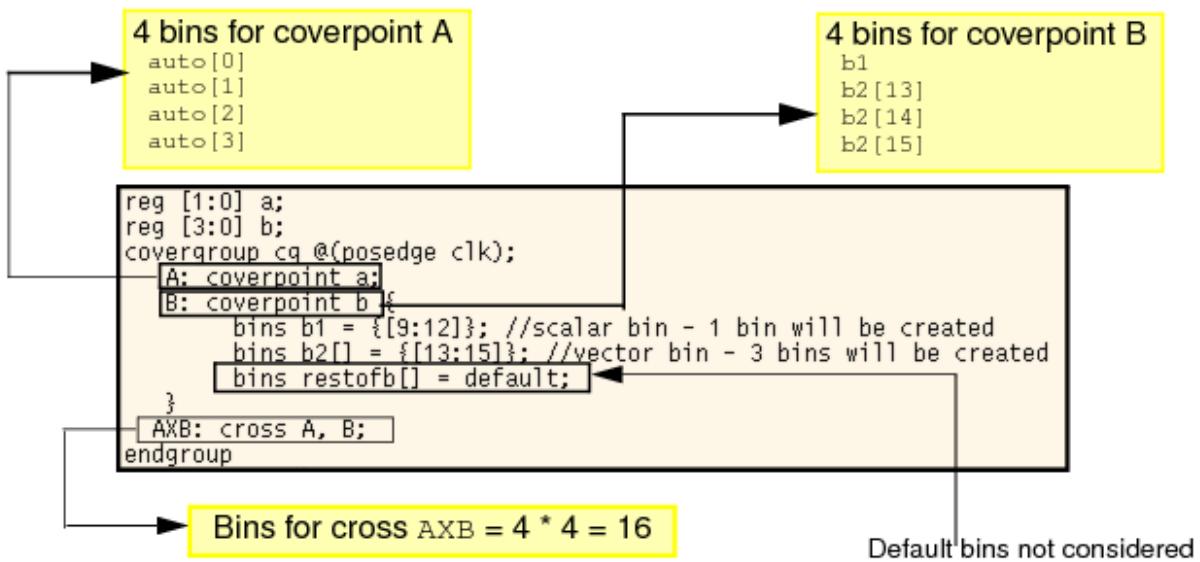
Automatic Cross Bins

When defining a cross, if you do not specify any bin for the cross, SystemVerilog automatically determines the number of bins for the cross as:

```
Bins for cross= Bins of coverpoint_1 * Bins of coverpoint_2 * Bins of coverpoint_N
```

These bins are calculated after removing the illegal and ignore values. In addition, default bins do not participate in a cross.

The following figure illustrates this point.



The cross of coverpoint A and B creates 16 cross products as:

Cross products of cross AXB	
0/16	cg.AXB
0(1)	<auto[0],b1>
0(1)	<auto[0],b2[13]>
0(1)	<auto[0],b2[14]>
0(1)	<auto[0],b2[15]>
0(1)	<auto[1],b1>
0(1)	<auto[1],b2[13]>
0(1)	<auto[1],b2[14]>
0(1)	<auto[1],b2[15]>
0(1)	<auto[2],b1>
0(1)	<auto[2],b2[13]>
0(1)	<auto[2],b2[14]>
0(1)	<auto[2],b2[15]>
0(1)	<auto[3],b1>
0(1)	<auto[3],b2[13]>
0(1)	<auto[3],b2[14]>
0(1)	<auto[3],b2[15]>
14	AXB: cross A, B;

The cross products created for bins of a cross is dependent on the order in which the coverpoints are specified. As a result, `cross a, b;` and `cross b, a;` will result in different cross products.

- ⚠** In the above report, all of the cross tuples are uncovered. By default, all the uncovered automatic cross tuples/bins are reported. To print a specified number of cross tuples/bins (instead of printing all of the cross tuples), use the covergroup option `cross_num_print_missing`.

User-Defined Cross Bins

If no bins are defined for a cross, then SystemVerilog automatically creates cross bins for cross products, as discussed in section [Automatic Cross Bins](#). With user-defined cross bins, you can group cross products that should be reported together. A user-defined cross bin is defined using the `bins` keyword, as shown below:

```
[cross_identifier:] cross list_of_coverpoints {
    bins bin_identifier = binsof (expression) [intersect {open_range_list}];
}
```

In the above definition,

`bin_identifier` is the name of the cross bin.

`expression` defines the coverpoint, or bins of a coverpoint that should be stored in bin `bin_identifier`.

`intersect` construct further refines the bins defined within the `expression` in the `binsof` construct.

Considering the [same example discussed above](#), to create a user-defined bin `my_b2` that includes all the cross products related to bin `b2` of coverpoint `B`, modify the cross definition as:

```
AXB: cross a, B {
    bins my_b2 = binsof(B.b2);
}
```

With the above cross definition, cross products related to bin `b2` of coverpoint `B` are reported together in user-defined bin `my_b2`, and remaining cross products are reported below it, as shown here:

User-defined cross bin (stores cross products related to B.b2)	
0.00 (0/100)	0/5
0(1)	cg_AXB
0(1)	my_b2
0(1)	<auto[0],b1>
0(1)	<auto[1],b1>
0(1)	<auto[2],b1>
0(1)	<auto[3],b1>
Other cross products	
14	AXB: cross a, B
15	bins my_b2 = binsof(B.b2);
14	AXB: cross a, B

With the `intersect` construct, you can further refine the cross products reported with the `expression` stated in the `binsof` construct.

For example, to create cross bins such that odd values are included in one bin and even values are included in another bin, you can modify the cross definition as:

```
AXB: cross a, B{
    bins my_b2_odd = binsof(B.b2) intersect {13,15}; //stores odd values
```

```

        bins my_b2_even = binsof(B.b2) intersect {14}; //stores even values
    }
    
```

With the above cross description, two user-defined cross bins are created as:

		Stores odd values			
0.00 (0/100)	0/6	cg_AXB		14	AXB: cross a, B{
	0(1)	my_b2_odd		15	bins my_b2_odd = binsof(B.b2) intersect {13,15};
	0(1)	my_b2_even		16	bins my_b2_even = binsof(B.b2) intersect {14};
	0(1)	<auto[0],b1>		14	AXB: cross a, B{
	0(1)	<auto[1],b1>		14	AXB: cross a, B{
	0(1)	<auto[2],b1>		14	AXB: cross a, B{
	0(1)	<auto[3],b1>		14	AXB: cross a, B{

Stores even values

Excluding Cross Products from Coverage Results

To exclude a set of cross products from coverage results, use the `ignore_bins` construct, as:

```
[cross_identifier:] cross list_of_coverpoints {
    ignore_bins bin_identifier = binsof (expression) [intersect {open_range_list}];
}
```

The `expression` within the `binsof` construct is used to define the coverpoint or the bin of a coverpoint whose cross products should be ignored. The ignored cross products are not reported. Considering the [same example discussed above](#), to exclude the cross products `<auto[1],b1>`, `<auto[2],b1>`, and `<auto[3],b1>` from coverage results, modify the cross definition as:

```

AXB: cross a, B{
    bins my_b2_odd = binsof(B.b2) intersect {13,15};
    bins my_b2_even = binsof(B.b2) intersect {14};
    ignore_bins ign = binsof(A) intersect {[1:3]};
}
    
```

With the above cross definition, cross products `<auto[1],b1>`, `<auto[2],b1>`, and `<auto[3],b1>` are excluded from coverage results (not reported in cross products and also removed from coverage counts).

Specifying Illegal Cross Products

You can mark a set of cross products as illegal by specifying them as `illegal_bins`. Similar to `ignore_bins`, the cross products marked as `illegal_bins` are not included in coverage results. The only difference between the two is that in the case of `illegal_bins`, if the sampled cross product matches the one specified with `illegal_bins`, simulation generates an error. No error is reported in the case of `ignore_bins`.

Illegal cross bins take precedence over any other bins, which implies that they will result in a run-time error even if they are also included as ignore bins.

 The cross is not dumped to the coverage database if all possible tuples/cross products are mentioned as ignore/illegal cross bins.

Considerations when defining bins for a cross

When defining bins for a cross, remember that:

The expression within the `binsof` construct can only be a coverpoint, or a bin of a coverpoint defined in the same scope as the cross.

```
A: coverpoint a{  
    bins b1[] = {2, [4:5]};  
    bins b2[] = {6, 7};  
    bins b3[] = default;  
}  
  
B: coverpoint b{  
    bins b1[] = {3, [7:9]};  
}  
  
C: coverpoint c{  
    bins b1[] = {1,4};  
}  
  
cross A, B{  
    bins cb1 = binsof(A); // correct use  
    bins cb2 = binsof(A.b1); // correct use  
    bins cb3 = binsof(B.b2); // incorrect use as b2 is not a bin of B  
    bins cb4 = binsof(C); // incorrect use as cross does not include C  
}
```

The expression within the `binsof` construct should not be a default bin.

```
bins cb1 = binsof(A.b3); // incorrect use because b3 is a default bin
```

You cannot declare a vector bin inside a cross.

```
cross A, B{  
    bins cb1[] = binsof(A); // incorrect use as cb1 is declared as vector  
    bins cb2 = binsof(A.b1); // correct use  
}
```

You cannot declare default bins inside a cross.

```
cross A, B{  
    bins cb1 = default; // incorrect use  
    bins cb2 = binsof(A); // correct use  
}
```

You cannot declare bins with the same name inside a cross.

```
cross A, B{
```

```
bins cb1 = binsof(A);  
bins cb1 = binsof(A.b2);  
}
```

Specifying Guard Expressions with Crosses

Guard expressions help you control sampling and binning. Guard expressions are specified with the `iff` construct and are always evaluated during sampling. Any valid expression can be specified as a guard expression. Following is the BNF for specifying guard expressions with a cross.

```
cover_cross ::=  
[cross_identifier:] cross list_of_coverpoints [ iff ( guard_expression ) ]  
select_bins_or_empty
```

If the `guard_expression` evaluates to false at the sampling time, the cross will not be evaluated. Consider the following code:

```
covergroup cg @(posedge clk);  
    A: coverpoint a;  
    B: coverpoint b;  
    cross A, B iff (x || y && z) ;  
endgroup
```

In the above code, bins are incremented only if the expression `(x || y && z)` is true at the sampling point.

 If any of the coverpoints specified in the cross are not sampled because the guard expression is false, then the corresponding cross bin will not be incremented. Consider the following code:

```
covergroup cg @(posedge clk);  
    A: coverpoint a iff (x) ;  
    B: coverpoint b;  
    cross_AB: cross A, B;  
    cross_ab: a, b;  
endgroup
```

In the above code, if the value of `x` is false at the sampling time, then the coverpoint `A` will not be sampled, and related cross bins of `cross_AB` will not be incremented. In addition, cross bins generated for `cross_AB` and `cross_ab` will be same because coverpoints `A` and `B` are reused in `cross_ab`.

Specifying with Clause at the Cross Bin Level

`with` clause can be specified in cross bins as part of the select expression. The BNF for the `select_expression` containing a `with` clause is:

```
select_expression ::=  
select_condition  
| select_expression with (with_covergroup_expression)
```

In the `select_expression`, `with` clause is used to specify that only those bin tuples in the subordinate `select_expression` for which sufficiently many value tuples satisfy the given `with_covergroup_expression` are selected.

In the `with_covergroup_expression`, occurrences of `cross_items`, including the `cover_point_identifiers` or `variable_identifiers` occurring in the `list_of_cross_items` for the cross, represent corresponding values in the value tuples of the candidate bin tuples.

When a `cross_identifier` is used as a `select_expression`, it selects all possible bin tuples. When used with a `with` clause, the cross bin can be completely described using a `with_covergroup_expression`. Only the `cross_identifier` of the enclosing cross may be used; other `cross_identifiers` are not allowed.

Only constant expressions, global and instance constants, or non-ref arguments to the covergroup can be used as variables in a `with_covergroup_expression`

Consider the given example:

```
module top;
    logic [ 0 : 3 ] a, b;
    parameter [ 0 : 3 ] mask;
    covergroup cg@(posedge clk);
        coverpoint a{
            bins low[] = {[ 0 : 7 ]} ;
            bins high = {[ 6 : 16 ]} ;
        }
        coverpoint b{
            bins two[] = b with (item% 2 == 0) ;
            bins three[] = b with (item% 3 == 0) ;
        }
        X: cross a,b
        {
            bins plum = binsof ((b.two) with (b > 5)) || binsof ((a.low) with (a <7 )) ;
        }
    endgroup
    .....
endmodule
```

In the given example, covergroup `cg` has a cross named `x`. A cross bin `plum` is defined using `a with clause` along with the `binsof` expression. Here, the cross bin `plum` demonstrates a `select_expression` composed of `with` covergroup expressions. The first `with` covergroup expression selects bin tuples that contain bins in the `b.two` bin array with values greater than 5.

```
<low[0], two[6]>
<low[0], two[8]>
...
<low[0], two[14]>
<low[1], two[6]>
...
<low[7], two[14]>
<high, two[6]>
...
<high, two[14]>
```

The `||` operator then adds the bin tuples selected by the second `with` covergroup expression,

namely those containing a bin from `a.low` and for which the expression `(a < 7)` satisfies. In this case, the selected tuples will be:

```
<low[0], two[0]>
<low[0], two[2]>
<low[0], two[14]>
...
<low[0], three[0]>
<low[0], three[3]>
...
<low[0], three[15]>
<low[1], two[0]>
<low[1], two[2]>
...
<low[1], three[15]>
<low[2], two[6]>
...
<low[6], three[15]>
```

Final bin tuples contributing to cross bin `plum` will be:

```
<low[0], two[0]>
<low[0], two[2]>
<low[0], two[14]>
...
<low[0], three[0]>
<low[0], three[3]>
...
<low[0], three[15]>
<low[1], two[0]>
<low[1], two[2]>
...
<low[1], three[15]>
<low[2], two[6]>
...
<low[6], three[15]>
<low[7], two[6]>
...
<low[7], two[14]>
<high, two[6]>
...
<high, two[14]>
```

The rest of the tuples will be part of auto bin space. The sample report as it appears in IMC is

shown.

Name	Average, Covered Grade	Line	Source Code
cg	0%, 0% (...)	7	covergroup cg;
--....			
--X	0% (0/7)	8	X : cross a,b
--plum	0% (0/1)	9	bins plum = (binsof(b.two) with (b>5).. ;
--low[7],two[0]	0% (0/1)	8	X : cross a, b
--low[7],two[2]	0% (0/1)	8	X : cross a, b
--low[7],two[4]	0% (0/1)	8	X : cross a, b
--high,two[0]	0% (0/1)	8	X : cross a, b
--high,two[2]	0% (0/1)	8	X : cross a, b
--high,two[4]	0% (0/1)	8	X : cross a, b

Limitations

Following scenarios are not supported when using `with` clause at the cross binlevel:

- Optional “matches” clause is specified in the cross bin definition.
- A cross identifier other than that of enclosing cross is used in bin definition.
- `with` expression contains coverpoint identifier not contributing to cross.
- A cross identifier is used in bin definition without having “`with`” clause.
- “`with_covergroup_expression`” refers to a coverpoint hierarchically.
- Any variable other than constant expressions, global and instance constants, or non-ref arguments to the covergroup are used as variables in a `with_covergroup_expression`.

Specifying cross_set_expression syntax at the Cross Bin Level

`cross_set_expression` syntax enables you to specify an expression to yield a queue of elements to define a cross bin. In this case, the specified expression should yield queue of the `CrossQueueType` with the elements of the type `CrossValType`. The BNF for using `cross_set_expression` syntax is:

```

select_expression ::= select_condition
| ! select_condition
| select_expression && select_expression
| select_expression || select_expression
| ( select_expression )
| select_expression with ( expression ) [ matches integer_covergroup_expression ]
| cross_identifier
| cross_set_expression [ matches integer_covergroup_expression ]

```

Consider the given example:

```

module mod_m;
    int a;
    logic [7:0] b;
    covergroup cg(int cg_lim);
        A: coverpoint a {
            bins x[] = {[0:4]} ;
        }
        B: coverpoint b {
            bins y[] = {[0:4]} ;
        }
        axb : cross A, B
        {
            bins one = '{ {0,0}, {1,1}, {2,2}, {4,4} } ;
        }
    endgroup
    cg cg_inst = new(3);
endmodule

```

The report for the given example is shown:

Name	Average, Covered Grade	Line	Source Code
cg_inst	0.00%, 0.00% (0/32)	4	covergroup cg(int cg_lim);
--A	0.00% (0/5)	6	A: coverpoint a {
--x[0]	0.00% (0/1)	7	bins x[] = {[0:4]} ;
--x[1]	0.00% (0/1)	7	bins x[] = {[0:4]} ;
--x[2]	0.00% (0/1)	7	bins x[] = {[0:4]} ;
--x[3]	0.00% (0/1)	7	bins x[] = {[0:4]} ;
--x[4]	0.00% (0/1)	7	bins x[] = {[0:4]} ;
--B	0.00% (0/5)	9	B: coverpoint b {
--y[0]	0.00% (0/1)	10	bins y[] = {[0:4]} ;
--y[1]	0.00% (0/1)	10	bins y[] = {[0:4]} ;
--y[2]	0.00% (0/1)	10	bins y[] = {[0:4]} ;
--y[3]	0.00% (0/1)	10	bins y[] = {[0:4]} ;
--y[4]	0.00% (0/1)	10	bins y[] = {[0:4]} ;
--axb	0.00% (0/22)	12	axb : cross A, B
--one	0.00% (0/1)	14	bins one = '{ {0,0}, {1,1}, {2,2}, {4,4} } ;
--x[0],y[1]	0.00% (0/1)	12	axb : cross A, B
--x[0],y[2]	0.00% (0/1)	12	axb : cross A, B
--x[0],y[3]	0.00% (0/1)	12	axb : cross A, B
--x[0],y[4]	0.00% (0/1)	12	axb : cross A, B
--x[1],y[0]	0.00% (0/1)	12	axb : cross A, B
--x[1],y[2]	0.00% (0/1)	12	axb : cross A, B
--x[1],y[3]	0.00% (0/1)	12	axb : cross A, B
--x[1],y[4]	0.00% (0/1)	12	axb : cross A, B
--x[2],y[0]	0.00% (0/1)	12	axb : cross A, B
--x[2],y[1]	0.00% (0/1)	12	axb : cross A, B
--x[2],y[3]	0.00% (0/1)	12	axb : cross A, B
--x[2],y[4]	0.00% (0/1)	12	axb : cross A, B
--x[3],y[0]	0.00% (0/1)	12	axb : cross A, B
--x[3],y[1]	0.00% (0/1)	12	axb : cross A, B
--x[3],y[2]	0.00% (0/1)	12	axb : cross A, B
--x[3],y[3]	0.00% (0/1)	12	axb : cross A, B
--x[3],y[4]	0.00% (0/1)	12	axb : cross A, B
--x[4],y[0]	0.00% (0/1)	12	axb : cross A, B
--x[4],y[1]	0.00% (0/1)	12	axb : cross A, B
--x[4],y[2]	0.00% (0/1)	12	axb : cross A, B
--x[4],y[3]	0.00% (0/1)	12	axb : cross A, B

Limitations

- matches clause is not supported with cross_set_expression syntax.

- `cross_set_expression` syntax in user-defined bins of cross is not supported if the cross has real coverpoints.
- `cross_set_expression` syntax in a user defined bin of a cross having hierarchical coverpoint is not supported.
- `cross_set_expression` syntax are supported in primary cross but not in secondary cross.
- `with expression` cannot be used with `cross_set_expression` syntax.
- In IMC standard merge, if `cross_set_expression` syntax is specified in cross bin, then crosses across two or more databases will be merged, only if:
 - Coverpoints contributing to the cross gets merged.
 - Text of `cross_set_expression` syntax across database matches.
- In IMC bin-level merge, if `cross_set_expression` syntax is specified in cross bin, then crosses across two or more databases will be merged, only if:
 - Coverpoints contributing to the cross get merged.
 - The cross has same valid space across two or more databases.

Declaring a Cross of a Cross

Cross of a cross is an extension of cross coverage as described in IEEE 1800-1200 SystemVerilog LRM. As per LRM, a cross can be specified using only coverpoints, but this feature enables you to specify a cross in another cross. It is supported for:

Cross of two or more existing crosses or

Pre defined cross with coverpoints

In the current release, this feature is supported only for auto bins. Only illegal/ignore user defined bins can be specified. However, the support for valid user defined bins is planned for future releases.

As the current support for cross of a cross is mainly for auto bins, if the `set_covergroup - cross_auto_bin_max_default_zero` CCF command is specified or `option.cross_auto_bin_max = 0` is specified in the covergroup, it leads to an empty cross in the list for cross of a cross. Therefore, such cross is neither dumped to the database nor is visible in the IMC GUI.

Note that all the features of regular cross apply to the cross of a cross.

- i** The cross of a cross is supported only with the `set_covergroup -optimize_model` command.

When using this feature, a cross which is referenced in another cross must be declared prior to the cross which references it. In addition, only the label specified in a cross declaration can be used to declare the cross in another cross. In case a label is not specified, the unlabelled cross cannot be used in another cross declaration scope.

The current limitations of cross of a cross are highlighted below:

Cross of a different covergroup cannot be accessed in another cross.

Non-illegal/ ignore user-defined bins are not supported in both secondary and primary cross.

The BNF for declaring a cross in another cross declaration is shown. The changes in LRM-based BNF to support cross of a cross are highlighted.

```
cover_cross ::=  
[ cross_identifier : ] cross list_of_cross_items [ iff ( expression ) ]  
select_bins_or_empty  
  
list_of_cross_items ::= cross_item , cross_item { , cross_item }
```

```
cross_item ::=  
cover_point_identifier  
| variable_identifier  
| cross identifier
```

```
cross identifier::=  
identifier
```

—

Cross bin expressions (`binsof`) is also supported for cross of a cross. The BNF excerpt to specify `binsof` expression is shown. The changes in LRM-based BNF to support cross of a cross are highlighted.

```
select_condition ::= binsof ( bins_expression ) [ intersect { covergroup_range_list } ]  
  
bins_expression ::=  
variable_identifier  
| cover_point_identifier [. bin_identifier ]
```

| cross identifier [. bin identifier]

Consider the given example for the cross of a cross in which covergroup `cg` has a cross `cr_of_cr`, which accesses cross `cp_buff0_x_cp_buff1` and cross `cp_buff1_x_cp_buff2`. Notice that there are no user defined bins in all the crosses.

```

module top;
reg[1:0] buff_0 = 0;;
reg[1:0] buff_1 = 3;
reg[2:0] buff_2 = 5;

covergroup cg @(posedge clk);
    cp_buff0 : coverpoint buff_0{
        bins valid_b0[] = {0,1};
    }
    cp_buff1 : coverpoint buff_1{
        bins valid_b1 = {2,3};
    }
    cp_buff2: coverpoint buff_2{
        bins valid_b2[] = {4,5};
    }
    cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1;
    cp_buff1_x_cp_buff2: cross cp_buff1,cp_buff2;
    cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff1_x_cp_buff2;
endgroup
...
endmodule

```

The sample report of the given example is shown:

Name	Average, Covered	Grade	Line	Source Code
<code>cg</code>	91.67%	85.00% (11/13)	8	covergroup cg;
<code>--cp_buff0</code>	100.00%	(2/2)	9	cp_buff0 : coverpoint buff_0{
<code>--valid_b0[0]</code>	100.00%	(1/1)	10	bins valid_b0[] = {0,1};
<code>--valid_b0[1]</code>	100.00%	(1/1)	10	}
<code>--cp_buff1</code>	100.00%	(1/1)	12	cp_buff1 : coverpoint buff_1{
<code>--valid_b1</code>	100.00%	(2/1)	13	bins valid_b1 = {2,3};
<code>--cp_buff2</code>	100.00%	(2/2)	15	cp_buff2: coverpoint buff_2{
<code>--valid_b2[4]</code>	100.00%	(1/1)	16	bins valid_b2[] = {4,5};
<code>--valid_b2[5]</code>	100.00%	(1/1)	16	}
<code>--cp_buff0_x_cp_buff1</code>	100.00%	(2/2)	18	cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1;
<code>--valid_b0[0],valid_b1</code>	100.00%	(1/1)	18	cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1;
<code>--valid_b0[1],valid_b1</code>	100.00%	(1/1)	18	cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1;
<code>--cp_buff1_x_cp_buff2</code>	100.00%	(2/2)	19	cp_buff1_x_cp_buff2: cross cp_buff1,cp_buff2;
<code>--valid_b1,valid_b2[4]</code>	100.00%	(1/1)	19	cp_buff1_x_cp_buff2: cross cp_buff1,cp_buff2;
<code>--valid_b1,valid_b2[5]</code>	100.00%	(1/1)	19	cp_buff1_x_cp_buff2: cross cp_buff1,cp_buff2;
<code>--cr_of_cr</code>	100.00%	(4/4)	20	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff1_x_cp_buff2;
<code>--(valid_b0[0],valid_b2[4]),(valid_b0[1],valid_b2[5])</code>	0.00%	(0/1)	20	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff1_x_cp_buff2;
<code>--(valid_b0[0],valid_b2[4]),(valid_b1,valid_b2[5])</code>	100.00%	(1/1)	20	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff1_x_cp_buff2;
<code>--(valid_b0[1],valid_b2[4]),(valid_b1,valid_b2[5])</code>	100.00%	(1/1)	20	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff1_x_cp_buff2;
<code>--(valid_b0[1],valid_b2[5])</code>	0.00%	(0/1)	20	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff1_x_cp_buff2;

Circular brackets indicate tuples generated for the cross

As shown in the above report, cross `cr_of_cr` contains cross products of auto bins of cross `cp_buff0_x_cp_buff1` and cross `cp_buff1_x_cp_buff2`. Notice that cross `cr_of_cr` contains cross product, `(valid_b0[0],valid_b1)` and `(valid_b1,valid_b2[4])`, enclosed in circular brackets indicating that these are tuples generated for the cross.

Consider the given example of the `binsof` expression used in cross of a cross. In the given example, cross `cr_of_cr` consists of cross `cp_buff0_x_cp_buff1` and coverpoint `cp_buff2`. Ignore bin `ign_1` specified in cross `cr_of_cr`, selects bins `valid_b2` of coverpoint `cp_buff2` by using `binsof` operator.

```

module top;

reg[1:0] buff_0 = 1;
reg[1:0] buff_1 = 3;
reg[2:0] buff_2 = 5;

covergroup cg;
    cp_buff0 : coverpoint buff_0{
        bins valid_b0[] = {0,1};
    }
    cp_buff1 : coverpoint buff_1;
    cp_buff2: coverpoint buff_2{
        bins valid_b2[] = {4,5};
    }
    cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1{
        ignore_bins ign_0 = binsof (cp_buff0.valid_b0) intersect {0};
    }
    cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff2{
        ignore_bins ign_1 = binsof (cp_buff2.valid_b2) intersect{4};
    }
endgroup
...
endmodule

```

The sample report for the given example is shown.

Name	Average, Covered Grade	Line	Source Code
cg	75.00%, 69.00% (11/16)	8	covergroup cg;
--cp_buff0	100.00% (2/2)	9	cp_buff0 : coverpoint buff_0{
--valid_b0[0]	100.00% (1/1)	10	bins valid_b0[] = {0,1};
--valid_b0[1]	100.00% (2/1)	10	bins valid_b0[] = {0,1};
--cp_buff1	75.00% (3/4)	12	cp_buff1 : coverpoint buff_1;
--auto[0]	100.00% (1/1)	12	cp_buff1 : coverpoint buff_1;
--auto[1]	0.00% (0/1)	12	cp_buff1 : coverpoint buff_1;
--auto[2]	100.00% (1/1)	12	cp_buff1 : coverpoint buff_1;
--auto[3]	100.00% (1/1)	12	cp_buff1 : coverpoint buff_1;
--cp_buff2	100.00% (2/2)	13	cp_buff2 : coverpoint buff_2{
--valid_b2[4]	100.00% (1/1)	14	bins valid_b2[] = {4,5};
--valid_b2[5]	100.00% (2/3)	14	bins valid_b2[] = {4,5};
--cp_buff0_x_cp_buff1	50.00% (2/4)	16	cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1{
--valid_b0[1],auto[0]	0.00% (0/1)	16	cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1{
--valid_b0[1],auto[1]	0.00% (0/1)	16	cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1{
--valid_b0[1],auto[2]	100.00% (1/1)	16	cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1{
--valid_b0[1],auto[3]	100.00% (1/1)	16	cp_buff0_x_cp_buff1: cross cp_buff0,cp_buff1{
--cr_of_cr	50.00% (2/4)	19	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff2{
--(valid_b0[1],auto[0]),valid_b2[5]	0.00% (0/1)	19	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff2{
--(valid_b0[1],auto[1]),valid_b2[5]	0.00% (0/1)	19	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff2{
--(valid_b0[1],auto[2]),valid_b2[5]	100.00% (1/1)	19	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff2{
--(valid_b0[1],auto[3]),valid_b2[5]	100.00% (1/1)	19	cr_of_cr: cross cp_buff0_x_cp_buff1,cp_buff2{

Note: Auto bins of primary cross can be accessed from the `binsof` expression in secondary cross using auto but intersect list with this `cross_identifier` is not allowed.

Predefined Coverage Methods

The following coverage methods are available for a covergroup.

Method	Can be called on			Description
	Covergroup	Cross	Coverpoint	
<code>stop()</code>	Yes	Yes	Yes	Stops sampling if default sampling condition is specified
<code>start()</code>	Yes	Yes	Yes	Starts sampling if default sampling condition is specified
<code>sample()</code>	Yes	No	No	Triggers sampling of covergroup irrespective of the <code>start</code> or <code>stop</code> method or the default sampling condition
<code>set_inst_name(string)</code>	Yes	No	No	Assigns name to a covergroup instance
<code>get_coverage()</code>	Yes	Yes	Yes	Returns cumulative coverage number (0...100)
<code>get_inst_coverage()</code>	Yes	Yes	Yes	Returns coverage number (0...100)

<u>get_hitcount()</u>	Yes	Yes	No	Returns hit count for coverpoint bins in covergroup type.
<u>get_inst_hitcount()</u>	Yes	Yes	No	Returns hit count for coverpoint bins in covergroup inst.

Using stop, start, and sample methods

By default, a covergroup is sampled on the default sampling condition specified with the covergroup declaration. Coverage methods `stop`, `start`, and `sample` allow you to control sampling for different instances of a covergroup. The following examples demonstrate the use of these methods.

Example 1 - Default Sampling Condition Specified

```
module dut;
...
covergroup cg @(posedge clk);
A: coverpoint a;
B: coverpoint b;
C: coverpoint c;
endgroup
cg cg_inst = new();
...
initial
begin
#5 clk = 1'b0;
#5 cg_inst.sample(); // Trigger sampling at this time for cg_inst
#20 cg_inst.start(); // No effect as sampling is already ON
#5 cg_inst.sample(); // Trigger sampling at this time for cg_inst
#5 cg_inst.A.stop(); // Stop sampling for coverpoint A
#5 cg_inst.sample(); // Trigger sampling at this time for cg_inst
#20 cg_inst.stop(); // Stop sampling of all the coverpoints and crosses
#200 $finish;
end
endmodule
```

In the above code, the covergroup is instantiated and `start` and `stop` methods are called on its instance, coverpoint, and cross at different time units. The effect of each statement is specified as comments. Note the impact of `start` and `stop` methods.

Example 2 - Default sampling condition not specified

```
module dut;
```

```
...
covergroup cg
A: coverpoint a;
B: coverpoint b;
C: coverpoint c;
endgroup
...
initial
begin
#5 clk = 1'b0;
#5 cg_inst.sample(); // Trigger sampling at this time for cg_inst
#20 cg_inst.start(); // No effect as no sampling condition is specified
#5 cg_inst.A.stop(); // No effect as no sampling condition is specified
#5 cg_inst.sample(); // Trigger sampling at this time for cg_inst
#20 cg_inst.stop(); // No effect as no sampling condition is specified
#200 $finish;
end
endmodule
```

In the above code, default sampling condition is not specified. As a result, `start` and `stop` methods will have no effect. Coverage sampling will happen through the `sample` method.

At times, you might come across situations when covergroup data does not sample. This could be because there is no time elapsing between creating of covergroup instance and triggering of events. Consider the following code:

```
module test();
event cg_event;
byte my_state;
covergroup cg @(cg_event);
coverpoint my_state{bins b1[] = {$:$};}
endgroup
cg cov;
initial begin
cov = new();
my_state = 23;
repeat(2) -> cg_event; //covergroup event trigger without any delay
#10 $finish;
end
endmodule
```

In the above code, there is no time delay between creating of covergroup instance `cov` and triggering of event `cg_event`. As a result, coverage data is not sampled. To avoid this issue, do any

of the following:

Specify a time delay, as shown below:

```
initial begin
  cov = new();
  my_state = 23;
  repeat(2) #10 -> cg_event; //delay added
  #10 $finish;
end
```

Replace `->cg_event` with `cov.sample()`:

Overriding the predefined sample method

Overriding the predefined `sample` method helps you to sample coverage data from contexts other than the enclosing covergroup. For example, an overridden `sample` method can be called with different arguments so that the covergroup data can be sampled from within an automatic task or function, or from within a particular instance of a process, or from within a sequence or property of a concurrent assertion.

To override the predefined `sample` method, use the keyword `with function sample`, as shown below:

```
covergroup covergroup_identifier [([ tf_port_list])] with function sample
  ([tf_port_list])
endgroup [ : covergroup_identifier ]
```

In the above BNF,

`with function sample` is the keyword used to override the `sample` method.

`tf_port_list` specifies the formal arguments of the overridden `sample` method.

⚠ Currently, the port direction of the formal arguments of the overridden `sample` method can be of type `input` only. If the port direction is not explicitly specified, then by default, it is assumed as `input`.

For the complete covergroup BNF, see [Defining a Covergroup](#).

Consider the following example, where the formal argument of the `sample` method is used as a coverpoint expression.

```
module top;
```

```
reg clk, a, b;
always #2 clk = ~clk;
always #5 a = ~a;
always #7 b = ~b;
covergroup cg with function sample(int cpoint1) ;
    c1 : coverpoint cpoint1
    {
        bins b1 = {[0:1]} ;
        bins b2 = {2} ;
    }
endgroup
cg cg_inst1 ;
initial begin
    cg_inst1 = new ;
    clk = 0;
    a = 1;
    b = 1;
    #10 $finish;
end
property P1;
    int x1 ;
    @(posedge clk) (a, x1=2) |=> (b, cg_inst1.sample(x1));
endproperty
cp1: cover property (P1);
initial
begin
    cg_inst1 = new();
end
endmodule
```

In the above code, covergroup `cg` overrides the `sample` method with a formal argument named `cpoint1`. This argument is used to specify the coverpoint expression of coverpoint `c1`. The `sample` method is called from within property `P1` to sample values of argument `x1`, which is local to property `P1`.

Considerations when overriding the sample method

When overriding the `sample` method, remember that:

Formal arguments of the overridden `sample` method can be used only as:

- Coverpoint/cross expressions
- Guard expressions

Formal arguments of the overridden `sample` method can be only of integer data types (`shortint`, `longint`, `int`, `byte`, `bit`, `logic`, `reg`, `integer`).

Formal arguments of the overridden `sample` method cannot be specified in either of the following:

- Bin declaration
- Setting covergroup options

Identifiers used in the formal arguments of the overridden `sample` method cannot be the same as the formal arguments of the covergroup.

```
covergroup cg (ref int x) with function sample (int x); //Incorrect usage
```

Using the `set_inst_name` method

The `set_inst_name(string)` method assigns a name to a covergroup instance. The string supplied as an argument is the name assigned to the covergroup instance.

The following example demonstrates the use of the `set_inst_name` method.

```
module test;
    logic clk=0;
    bit [3:0] by;
    covergroup CG @(posedge clk);
        option.per_instance = 1;
        A: coverpoint by;
        option.auto_bin_max = 4;
    endgroup // CG
    CG g1;
    CG g2 = new();
    CG g3;
    initial begin
        repeat(10) @(negedge clk) by = $urandom;
        $finish;
    end
    initial begin
        g1 = new();
        g1.set_inst_name("obj_g1"); //name assigned to instance g1
        g3 = new();
        g2.set_inst_name("obj_g2"); //name assigned to instance g2
        g3.set_inst_name("obj_g3"); //name assigned to instance g3
    end
    always #2 clk = ~clk;
endmodule // test
```

The instance-based report from the above code is:

Number of covered bins: 12 of 12					
Coverage	Count	Name	Line	Origin	
1.00 (100/100)	4/4	obj_g1	4	covergroup CG @(posedge clk);	
1.00 (100/100)	4/4	obj_g1.A	6	A: coverpoint by;	
	3(1)	auto[0:3]	6	A: coverpoint by;	
	2(1)	auto[4:7]	6	A: coverpoint by;	
	2(1)	auto[8:11]	6	A: coverpoint by;	
	3(1)	auto[12:15]	6	A: coverpoint by;	
1.00 (100/100)	4/4	obj_g2	4	covergroup CG @(posedge clk);	
1.00 (100/100)	4/4	obj_g2.A	6	A: coverpoint by;	
	3(1)	auto[0:3]	6	A: coverpoint by;	
	2(1)	auto[4:7]	6	A: coverpoint by;	
	2(1)	auto[8:11]	6	A: coverpoint by;	
	3(1)	auto[12:15]	6	A: coverpoint by;	
1.00 (100/100)	4/4	obj_g3	4	covergroup CG @(posedge clk);	
1.00 (100/100)	4/4	obj_g3.A	6	A: coverpoint by;	
	3(1)	auto[0:3]	6	A: coverpoint by;	
	2(1)	auto[4:7]	6	A: coverpoint by;	
	2(1)	auto[8:11]	6	A: coverpoint by;	
	3(1)	auto[12:15]	6	A: coverpoint by;	

Coverage for individual named instances

The above report is an instance-based report generated using the `-instance` option of the `report_detail` command.

The above report shows coverage for individual named instances (named through `set_inst_name` method). If a name is not assigned to a covergroup instance, then a unique name is automatically generated for it. This covergroup name is object hierarchy based. Hence, it is easier to relate covergroup instances in the coverage report with the covergroup instances in the HDL. Consider the given example with an embedded covergroup and its instance:

```
module top;
    reg clk;
    class C1;
        reg [1:0] v1;
        covergroup CG @(clk);
            option.per_instance = 1;
            CP1 : coverpoint v1 {
                bins b1 = {0, 1};
            }
        endgroup

        function new();
            CG = new;
        endfunction
    endclass

    C1 cobj;
```

```

initial
begin
    cobj = new;
    ...
end
endmodule

```

The instance-based report from the above code is:

Name	Grade	Line	Source Code
cobj.CG	100%(1/1)	10	covergroup CG @ (clk)
--CP1	100%(1/1)	11	CP1 : coverpoint v1 {
--b1	100%(1/1)	12	bins b1 = {0, 1};

In the given example, the covergroup name is generated as `cobj.CG` as the covergroup is instantiated inside `cobj` and it remains in the same hierarchy till the end of the simulation.

Note: In UVM, the default name for some covergroup objects that are outside the quasi-static object hierarchy, may be very complex with multiple levels of UVM objects. This can make it difficult to relate the covergroup instances in the coverage report with the covergroup instances in the HDL. Therefore, it is recommended to use `set_inst_name()` to name such covergroup instances explicitly.

Note: A module-based report is generated with `-module` option and it does not display individual named instances. It displays cumulative coverage for all instances created for the covergroup. It is also known as *Type-Based*.

Note: Covergroup instance name prefixed with logical path of `uvm_test_top` will be truncated. Additionally, if a covergroup instance name is not prefixed with logical path of `uvm_test_top` but has `uvm_test_top` as a part of the complete name, the covergroup instance name is not truncated.

Prior to this release, type-based information was reported along with the instance-based information in the covergroup report using the `-instance` option of the `report_detail` command. With this release, type-based information is reported if a module-based report is generated with the `-module` option of the `report_detail` command.

For more details on reporting, see the *Integrated Metrics Center User Guide* in the Metric-Driven Verification (MDV) release.

If more than one instance is named identically, then the coverage of instances with the same name is merged and displayed. If the above code is modified as:

```
initial begin
    g1 = new();
    g1.set_inst_name("obj_g1"); //name assigned to instance g1
    g3 = new();
    g2.set_inst_name("obj_g2and3"); //name assigned to instance g2
    g3.set_inst_name("obj_g2and3"); //same name assigned to instance g3
end
```

In the above code, instance g2 and g3 are named identically. The instance-based report is generated as:

Number of covered bins: 8 of 8					
Coverage	Count	Name	Line	Origin	
1.00 (100/100)	4/4	obj_g1	4	covergroup CG @(posedge clk);	
1.00 (100/100)	4/4	obj_g1.A	6	A: coverpoint by;	
	3(1)	auto[0:3]	6	A: coverpoint by;	
	2(1)	auto[4:7]	6	A: coverpoint by;	
	2(1)	auto[8:11]	6	A: coverpoint by;	
	3(1)	auto[12:15]	6	A: coverpoint by;	
1.00 (100/100)	4/4	obj_g2and3	4	covergroup CG @(posedge clk);	
1.00 (100/100)	4/4	obj_g2and3.A	6	A: coverpoint by;	
	6(1)	auto[0:3]	6	A: coverpoint by;	
	4(1)	auto[4:7]	6	A: coverpoint by;	
	4(1)	auto[8:11]	6	A: coverpoint by;	
	6(1)	auto[12:15]	6	A: coverpoint by;	

Coverage merged for identically named instances

Note: Covergroup instances local to tasks or functions are not reported. Coverage data for instances local to tasks or functions is merged with their covergroup and is reported in the cumulative coverage for the covergroup.

Using get_coverage and get_inst_coverage methods

Coverage methods `get_coverage` and `get_inst_coverage` allow you to measure coverage statistics during a simulation run. Using these methods, you can monitor coverage at run time and based on coverage numbers constrain the stimulus, or stop the simulation. Both of these coverage methods can take either zero or two arguments, and these optional set of arguments must be `int` type reference values. If you use any other number of arguments or datatype, an error will be generated.

The `get_coverage` method returns the cumulative coverage for covergroup item (covergroup instance/coverpoint/cross) on which it is invoked. The `get_coverage` method is a static method and hence can be invoked using both `(::)` and `(.)` operators.

The `get_inst_coverage` method returns coverage for covergroup item (covergroup instance/coverpoint/cross) on which it is invoked. It can be invoked using only the `(.)` operator.

Note: Both the methods (`get_coverage` and `get_inst_coverage`) return a real number in the range of 0 to 100.

The following example demonstrates how to use `get_coverage` and `get_inst_coverage` methods.

```
module top;
...
real cov_type, cov_inst, cov_type2;
int covered, total;
covergroup cg @(posedge clk);
A: coverpoint a;
B: coverpoint b;
AXB: cross A, B;
endgroup
cg cg_inst1;
...
initial
begin
cg_inst1 = new();
cov_type = cg::get_coverage(); //returns cumulative coverage for covergroup cg
if (cov_type > 80)
...
cov_inst = cg_inst1.A.get_inst_coverage(); //returns coverage for coverpoint A
if (cov_inst > 80)
...
if (cg_inst1.AXB.get_inst_coverage() > 90) //get_inst_coverage invoked on cross
AXB
...
cov_type2 = cg::AXB::get_coverage(covered, total); //get_coverage invoked with
two integer arguments on cross AXB
...
#200 $finish;
end
endmodule
```

When the `get_coverage` method is invoked with two integer arguments in the cross AXB, the cross returns the number of covered bins and the total number of bins in the covergroup cg. Note the use of `(::)` and `(.)` operators. In the above code, `cg::get_coverage()` can be replaced with `cg.get_coverage()`.

Note: If a testbench includes the `get_coverage()` method or the `get_inst_coverage()` method, then the covergroup coverage is always scored. However, covergroup coverage is dumped to the coverage database only if functional coverage is enabled using the `-coverage functional` option at

elaboration.

Using get_hitcount and get_inst_hitcount methods

Coverage methods `get_hitcount` and `get_inst_hitcount` allow you to measure hit counts for coverpoint bins. Using these methods, you can monitor hit count for coverpoint bins in both type-based and instance-based covergroups.

The `get_hitcount` method returns the hit count for coverpoint bins in covergroup type.

The `get_inst_hitcount` method returns the hit count for coverpoint bins in covergroup inst.

Note: Both the methods (`get_hitcount` and `get_inst_hitcount`) return a real number.

The following example demonstrates how to use `get_hitcount` and `get_inst_hitcount` methods.

```
module top;
reg clk;
reg [4:0] v1;
int coverage = 0, total;
initial
clk = 0;
always #5 clk = !clk;
covergroup CG;
    CP1: coverpoint v1 {
        bins b1[] = {[2:5]};
        bins b2 = {4};
    }
    option.per_instance = 1;
endgroup

CG cg_inst1;
CG cg_inst2;

initial
begin
    cg_inst1 = new;
    cg_inst2 = new;
    cg_inst1.set_inst_name("cg_inst1");
    cg_inst2.set_inst_name("cg_inst2");
end

initial
begin
#1 v1 = 4 ;
#5 cg_inst1.sample() ;
cg_inst2.sample() ;
#5 coverage = CG.CP1.get_hitcount("b1[4]");//returns hit count for coverpoint bins in covergroup
type
#1 coverage = cg_inst1.CP1.get_inst_hitcount("b2");//returns hit count for coverpoint bins for
covergroup inst
#1 coverage = cg_inst1.CP1.get_inst_hitcount("b1[4]");//returns hit count for for coverpoint bins
for covergroup inst
#1 coverage = cg_inst2.CP1.get_inst_hitcount("b2");//returns hit count for coverpoint bins for
covergroup inst
#10 $finish;
end
```

In the above code, the effect of `get_hitcount` and `get_inst_hitcount` methods is specified as comments.

The following limitations apply to `get_hitcount` and `get_inst_hitcount` methods:

As embedded covergroup is not considered a static member, the `get_hitcount` and `get_inst_hitcount` methods cannot be invoked using `(::)` operator.

A warning is issued if methods are called upon bins which do not exist.

Limitations while Using Predefined Methods

Out of module references to predefined methods is not supported.

Predefined Coverage System Tasks and System Functions

The given system task and function are supported to enable managing coverage data collection:

`$set_coverage_db_name()`

The `$set_coverage_db_name(filename)` system task sets the filename of the coverage database in which the coverage information is saved at the end of a simulation run. Here, the filename argument must be a string type and an error is reported if any other datatype is used.

The `$set_coverage_db_name(filename)` system task perform same function as the `coverage - setup - testname < testname >` command and the `-covtest xmsim/xrun` command-line option. If the coverage database name is set using either the `coverage - setup - testname < testname >` command or the `-covtest` option, and the `$set_coverage_db_name(filename)` system task, the name specified by `$set_coverage_db_name(filename)` is overridden.

In addition, in this case, all existing covergroup constructs, such as, procedural assignments and built-in method calls, and all post-processing analysis, such as, merging, will continue to use the name set using `$set_coverage_db_name(filename)`. Also, all the existing covergroup features that are not using `$set_coverage_db_name(filename)` are not impacted.

Note: The argument to the `$set_coverage_db_name(filename)` system task can only be a logical test name, and the use of physical path, both absolute and relative, is not supported.

`$get_coverage`

`$get_coverage` returns a real number in the range 0 to 100. This value indicates the overall coverage of all covergroup types in the design. Consider the given example in which two covergroups are declared in the same module `top`. In this example, each covergroup has `type_option weight = <value>` in the respective declaration, and `$get_coverage` returns the weighted average of coverage of both the covergroups using the `type_option`:

```
module top;  
  
logic [1:0] opcode1, opcode2;  
  
real type_cov1, type_cov2, all_type_cov, d_get_cov;
```

```
covergroup cg1 @(opcode1);
    type_option.weight = 2;
    option.per_instance=1;
    c1: coverpoint opcode1;
endgroup : cg1

cg1 cg1_inst1 = new;

covergroup cg2 @(opcode2);
    option.per_instance=1;
    type_option.weight = 3;
    c1: coverpoint opcode2;
endgroup : cg2

cg2 cg2_inst1 = new;

initial begin
    opcode1 = 'b01; opcode2 = 'b01;
    #1 opcode1 = 'b11; opcode2 = 'b11;
    #1 opcode1 = 'b10; opcode2 = 'b10;

#4 d_get_cov = $get_coverage();
    type_cov1 = cg1::get_coverage();
    type_cov2 = cg2::get_coverage();

    all_type_cov = (type_cov1*2 + type_cov2*3) / (2+3);
    $display("The value of $get_coverage is %g", all_type_cov);

end
endmodule
```

Specifying Coverage Options

Options control the behavior of the covergroup, coverpoint, and cross. There are two types of options:

- **Instance-Specific Covergroup Options** (Apply to an instance of a covergroup)

Type-Specific Covergroup Options (Apply to covergroup type as a whole)

Instance-Specific Covergroup Options

The following table describes the instance-specific covergroup options. These options apply at instance-level. Therefore, different instances of a covergroup can assign different values to these options. The specified value affects only that instance.

Option Name	Default	Description
<u>at least</u>	1	Minimum number of hits for each bin to be considered covered.
<u>goal</u>	100	Specifies target goal for a covergroup instance or for a coverpoint or a cross of an instance.
<u>weight</u>	1	Specifies the weight of the coverpoint when computing coverage of the covergroup and the weight of the covergroup when computing coverage of the instance or module.
<u>name</u>	Unique name	Specifies the name for the covergroup instance. If not specified, the tool automatically generates it.
<u>comment</u>	" "	Specifies a comment with the instance of a covergroup, or a coverpoint, or a cross. The comment is saved to the coverage database and included in coverage reports.
<u>per instance</u>	0	<p>Specifies if each instance contributes to the overall coverage.</p> <p>Value 1 indicates that each instance of the covergroup is saved in the coverage database and included in the coverage report.</p> <p>Default value is 0, which indicates that the coverage of the covergroup instances should not be saved to the coverage database.</p>
<u>auto bin max</u>	64	Maximum number of automatically created bins, when no bins are explicitly defined for the coverpoint.

<u>cross_auto_bin_max</u>	NA	<p>Specifies if the information about automatically-generated cross bins will be included in the coverage computation. The value of this option can only be specified as 0.</p> <p>When this option is specified, automatically-created cross bins are excluded from the coverage computation. They are neither saved in the coverage database nor included in the coverage report. You can use the <code>cross_auto_bin_max</code> option at the covergroup and cross level but not at the coverpoint level.</p> <p>If this option is not specified, all cross coverage bins, both the user-defined cross bins and automatically-generated cross bins, will be reported.</p>
<u>detect_overlap</u>	0	<p>If true, generates a warning when there is an overlap between the range list (or transition list) of bins of a coverpoint.</p> <p>Default value is 0, which indicates no warning will be generated for overlapping range lists.</p>
<u>cross_num_print_missing</u>	" "	<p>Specifies the number of uncovered automatically generated cross bins that must be printed in the coverage report.</p> <p>By default, all of the uncovered automatically generated cross bins will be printed in the coverage reports.</p>

You can set these options:

- Inside the covergroup definition using the following syntax:

```
option.member_name = expression;
```

where

- `member_name` is the name of the option.
- `expression` is the value to be specified for the option. It can be constant expressions, parameters, or expressions that use covergroup arguments.

- Outside the covergroup definition (in the procedural code). For more details, see [Procedural Assignment of Covergroup Options](#).

i The `per_instance` option can be set only in the covergroup definition. In addition, the `auto_bin_max` and `detect_overlap` options can be set only in the covergroup or coverpoint definition.

The following table summarizes the syntactical level (covergroup, coverpoint, or cross) at which these options can be specified.

Option	Allowed in Syntactic Level		
	Covergroup Cross	Coverpoint	Cross
<code>at_least</code>	Yes	Yes	Yes
<code>goal</code>	Yes	Yes	Yes
<code>weight</code>	Yes	Yes	Yes
<code>name</code>	Yes	No	No
<code>comment</code>	Yes	Yes	Yes
<code>per_instance</code>	Yes	No	No
<code>auto_bin_max</code>	Yes	Yes	No
<code>cross_auto_bin_max</code>	Yes	No	Yes
<code>detect_overlap</code>	Yes	Yes	No
<code>cross_num_print_missing</code>	Yes	No	Yes

Note: You cannot specify values > 32 bits for these options.

Except for `goal`, `weight`, `comment`, and `per_instance` options, all other options set at the covergroup syntactic level act as the default value for corresponding coverpoints and crosses. For example, in

the following code, `auto_bin_max` is not specified for coverpoint `b`. Therefore, the `auto_bin_max` defined at the covergroup syntactic level, which is 5, will apply to coverpoint `b`.

```
covergroup cg @(posedge clk);
    option.auto_bin_max = 5;
    a : coverpoint a_var{
        option.auto_bin_max = 3;
    }
    b : coverpoint b_var;
endgroup
```

Consider another piece of code.

```
covergroup CG2 @(posedge clk);
    option.weight = 20;
    C: coverpoint c { bins b1[]={0,1,2,3,4,5,6,7,8}; }
    D: coverpoint d { option.weight = 4;
        bins b1[]={0,1,2}; }
endgroup
```

In the above code, `weight` is not defined for coverpoint `C`. Therefore, default `weight` value, which is 1, is applied to coverpoint `C`.

Example: Using weight Option

Consider the following code:

```
covergroup CG1 @(posedge clk);
    type_option.weight = 10;
    A: coverpoint a { option.weight = 2;
        bins b1[]={0,1,2,3,4,5,6,7,8,9}; }
    B: coverpoint b { option.weight = 3;
        bins b1[]={0,1,2,3,4,5,6,7,8,9,10,11}; }
endgroup
...
covergroup CG2 @(posedge clk);
    type_option.weight = 20;
    C: coverpoint c { bins b1[]={0,1,2,3,4,5,6,7,8}; }
    D: coverpoint d { option.weight = 4;
        bins b1[]={0,1,2}; }
endgroup
...
```

```

CG1 g1 = new();
CG2 g2 = new();

...
    
```

Note: The weight set using the `option` keyword applies to the covergroup instance. The weight set using covergroup type as a whole is set using the `type_option` keyword. For more details, see [Example: Using weight Type Option](#).

The module-based report generated from the above code is:

Coverage	Count	Name	Line	Origin
0.41 (41/100)	9/22	CG1	8	covergroup CG1 @ (posedge clk);
0.40 (40/100)	4/10	CG1.A	10	A: coverpoint a { option.weight = 2;
	1(1)	b1[0]	11	bins b1[] = {0,1,2,3,4,5,6,7,8,9}; }
	1(1)	b1[1]	11	bins b1[] = {0,1,2,3,4,5,6,7,8,9}; }
	1(1)	b1[2]	11	bins b1[] = {0,1,2,3,4,5,6,7,8,9}; }
	3(1)	b1[3]	11	bins b1[] = {0,1,2,3,4,5,6,7,8,9}; }
	0(1)	b1[4] to [9]	11	bins b1[] = {0,1,2,3,4,5,6,7,8,9}; }
0.42 (42/100)	5/12	CG1.B	12	B: coverpoint b { option.weight = 3;
	1(1)	b1[0]	13	bins b1[] = {0,1,2,3,4,5,6,7,8,9,10,11}; }
	1(1)	b1[1]	13	bins b1[] = {0,1,2,3,4,5,6,7,8,9,10,11}; }
	1(1)	b1[2]	13	bins b1[] = {0,1,2,3,4,5,6,7,8,9,10,11}; }
	1(1)	b1[3]	13	bins b1[] = {0,1,2,3,4,5,6,7,8,9,10,11}; }
	2(1)	b1[4]	13	bins b1[] = {0,1,2,3,4,5,6,7,8,9,10,11}; }
	0(1)	b1[5] to [11]	13	bins b1[] = {0,1,2,3,4,5,6,7,8,9,10,11}; }
0.67 (67/100)	8/12	CG2	16	covergroup CG2 @ (posedge clk);
0.67 (67/100)	6/9	CG2.C	18	C: coverpoint c { bins b1[] = {0,1,2,3,4,5,6,7,8}; }
	1(1)	b1[0]	18	C: coverpoint c { bins b1[] = {0,1,2,3,4,5,6,7,8}; }
	1(1)	b1[1]	18	C: coverpoint c { bins b1[] = {0,1,2,3,4,5,6,7,8}; }
	1(1)	b1[2]	18	C: coverpoint c { bins b1[] = {0,1,2,3,4,5,6,7,8}; }
	1(1)	b1[3]	18	C: coverpoint c { bins b1[] = {0,1,2,3,4,5,6,7,8}; }
	1(1)	b1[4]	18	C: coverpoint c { bins b1[] = {0,1,2,3,4,5,6,7,8}; }
	1(1)	b1[5]	18	C: coverpoint c { bins b1[] = {0,1,2,3,4,5,6,7,8}; }
	0(1)	b1[6] to [8]	18	C: coverpoint c { bins b1[] = {0,1,2,3,4,5,6,7,8}; }
0.67 (67/100)	2/3	CG2.D	19	D: coverpoint d { option.weight = 4;
	1(1)	b1[0]	20	bins b1[] = {0,1,2}; }
	5(1)	b1[1]	20	bins b1[] = {0,1,2}; }
	0(1)	b1[2]	20	bins b1[] = {0,1,2}; }

In the above report, the coverage % for covergroup `CG1` is calculated as:

Coverage % for Covergroup CG1

$$\frac{\text{weight of A} * \text{hit_ratio of coverpoint A} + \text{weight of B} * \text{hit_ratio of coverpoint B}}{\text{weight of A} + \text{weight of B}}$$

where

`hit_ratio = number of hit bins of coverpoint / total_number_of_coverpoints`

$$\frac{4/10 * 2 + 5/12 * 3}{2 + 3} = .41$$

The coverage % for covergroup `CG2` is calculated as:

Coverage % for Covergroup CG2

$$\frac{\text{weight of C} * \text{hit_ratio of coverpoint C} + \text{weight of D} * \text{hit_ratio of coverpoint D}}{\text{weight of C} + \text{weight of D}}$$

where

hit_ratio = number of hit bins of coverpoint / total_number_of_coverpoints

$$\frac{6/9 * 1 + 2/3 * 4}{1 + 4} = .67$$

The summary report generated from the code is:

```
Coverage Summary Report, Instance-Based
=====
D = Data-oriented
C = Cumulative coverage for that coverage type including sub hierarchy
test
-----
D: 58% (1741/3000) DC: 58% (1741/3000)
```

The overall coverage % is calculated as:

Overall Coverage %

$$\frac{\text{weight of CG1} * \text{Coverage \% of CG1} + \text{weight of CG2} * \text{Coverage \% of CG2}}{\text{weight of CG1} + \text{weight of CG2}}$$

$$\frac{10 * .41 + 20 * .67}{10 + 20} = .58$$

The coverage computed above is then multiplied by 100, which gives 58%.

Example: Using per_instance Option

As per SystemVerilog LRM, the `per_instance` option is set as 0 and therefore, in an instance-based report, coverage for the covergroup instance is not reported. To save coverage of covergroup instances to the coverage database and to print coverage of instances in the instance-based report, set the `per_instance` option to 1.

Consider the following code:

```

covergroup cg @(clk) ;
    option.per_instance = 1;
    cp1 : coverpoint v1
    {
        bins b1[] = {[1:3]} ;
    }
endgroup
cg cg1 = new ;
initial begin
    v1 = 1 ;
    #3 $finish ;
end
    
```

The instance-based report generated from the above code is:

Automatically generated name				
Coverage	Count	Name	Line	Origin
0.33 (33/100)	1/3	cg1	8	covergroup cg @(clk) ;
0.33 (33/100)	1/3	cg1.cp1	10	cp1 : coverpoint v1
2(1)	b1[1]		12	bins b1[] = {[1:3]} ;
0(1)	b1[2] to [3]		12	bins b1[] = {[1:3]} ;

The above report prints the coverage results of covergroup instance `cg1` because the `per_instance` option is set to 1. If you do not set the `per_instance` option, then the coverage for the covergroup instances will not be stored and instance report will not list any instances. As the covergroup instance is not named explicitly, a name is automatically generated for covergroup instance `cg1`.

Note: You can also use the `set_covergroup -per_instance_default_one` command at elaboration to print coverage of instances in the instance-based report. However, if you use this command at elaboration, it applies only to those covergroups where `option.per_instance` is not explicitly set in the design.

Note: The `per_instance` option can be set in the covergroup declaration only. As a result, it applies only to the covergroup in which it is set. If a design includes multiple covergroups and you want to save and report coverage of instances of all of the covergroups in the design, you can do any of the following:

Set the `per_instance` option in all the covergroups, or

Use the `set_covergroup -per_instance_default_one` command in the coverage configuration file at elaboration

Note: The covergroup instance report is printed using the `-instance` option of the `report_detail`

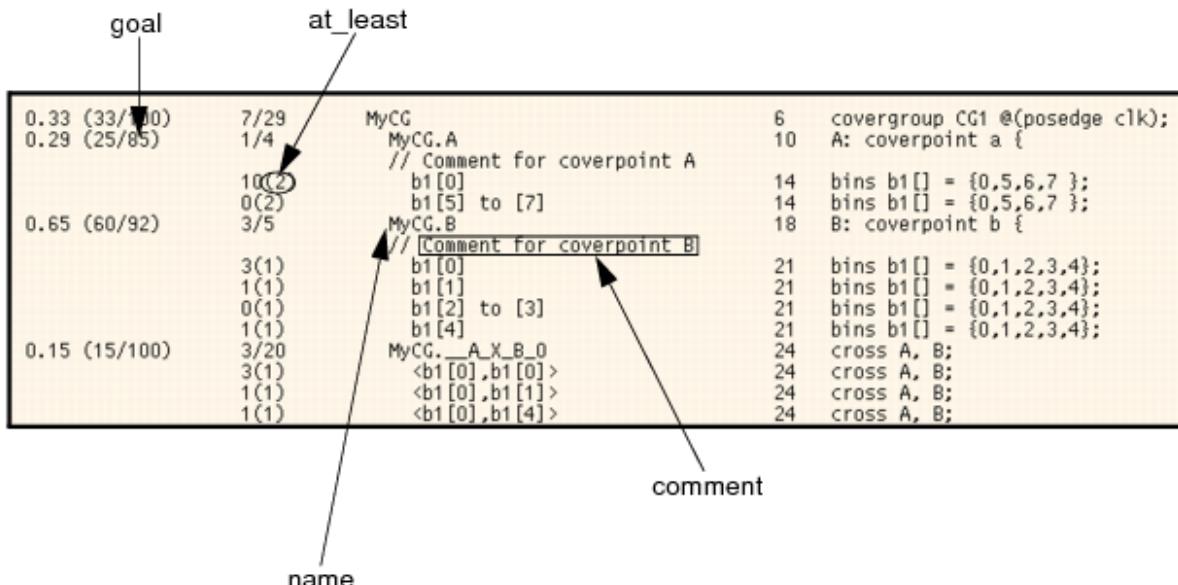
command and covergroup type report is printed using the `-module` option of the `report_detail` command.

Example: Using at_least, goal, name, and comment Options

Consider the following code:

```
covergroup CG1 @(posedge clk);
option.per_instance = 1;
option.name = "MyCG";
A: coverpoint a {
    option.goal = 85;
    option.comment = "Comment for coverpoint A";
    option.at_least = 2;
    bins b1[] = {0,5,6,7 };
}
B: coverpoint b {
    option.comment = "Comment for coverpoint B";
    option.goal = 92;
    bins b1[] = {0,1,2,3,4};
}
endgroup
```

The following instance-based report displays the impact of various options used in the code.



For coverpoint A, `at_least` option is defined as 2. The bins with a hit count 2 or more are considered

covered. In the code, the goal for coverpoint is defined as 85. The report displays (25/85), which indicates the expected goal was 85 out of which only 25 was achieved for the coverpoint.

Note: In the above example, the goal is set explicitly for the covergroup and coverpoints. If the goal is not explicitly set, then the default goal 100 is used (if the default goal is not overridden using the [set_covergroup -default_goal](#)).

Note: You can use a variable string to assign value to `option.name`. Consider the given example in which a variable string `instname`, which is declared inside a module is used to specify name of covergroup instance:

```
module top;
    reg clk;
    reg v1;
    string instname;
    covergroup cg @(clk);
        option.name = instname;
        coverpoint v1;
    endgroup
    cg cg_inst1;
    cg cg_inst2;
    initial
    begin
        instname = "cg_inst1_name";
        cg_inst1 = new;
        instname = "cg_inst2_name";
        cg_inst2 = new;
        ....
    end
endmodule
```

In the given example, the names of the two declared covergroups, `cg_inst1` and `cg_inst2`, are being set with the value of variable string `instname` specified in `option.name`. At the time of covergroup instantiation, variable `instname` will be assigned to covergroup instance name. The following instance-based report displays the results with the new covergroup names.

In the given example, the names of covergroups `cg_inst1` and `cg_inst2` will be set to `cg_inst1_name` and `cg_inst2_name`, respectively.

Instance name: top Module/Entity/Interface/Program name: top File name: option_name/test_basic.v Number of covered bins: 4 of 4				
cnt	coverpoint/bin	line	origin	description
1	cg_inst1_name	10	covergroup cg	
1	cg_inst1_name.v1	11	coverpoint v1;	
1	auto[0]	11	coverpoint v1;	
1	auto[1]	11	coverpoint v1;	
1	cg_inst2_name	10	covergroup cg	
1	cg_inst2_name.v1	11	coverpoint v1;	
1	auto[0]	11	coverpoint v1;	
1	auto[1]	11	coverpoint v1;	

Example: Using the cross_auto_bin_max Option

Consider the following example:

```
covergroup cg1 () @ clk;
    option.per_instance = 1;
    option.cross_auto_bin_max = 0;
    A: coverpoint a {
        bins b11[] = { [1:3] };
        bins b12[] = { [8:9] };
    }
    B: coverpoint b {
        bins b21[] = { [3:5] };
        bins b22[] = { [10:11] };
    }
    CR1: cross A, B {
        bins b1 = binsof (A) intersect {3} && binsof (B) intersect {6};
        bins b2 = binsof (A.b12) intersect {5};
        bins b3 = binsof (B.b22) intersect {11};
    }
endgroup
```

In the given example, the `cross_auto_bin_max` option is defined at the covergroup level. So, the value of `cross_auto_bin_max` at covergoup level will propagate to cross level and total number of cross autobins will be 0. The instance-based report generated from the given example will be as follows:

Coverage	Count	Name	Line	origin
0.64 (64/100)	9/13	cg1	11	covergroup cg1 () @ clk;
0.80 (80/100)	4/5	cg1.A	16	A: coverpoint a {
			17	// CG with cross_auto_bin_max=0
			0(1)	bins b11[] = {[1:3]};
			1(1)	bins b11[] = {[1:3]};
			1(1)	bins b11[] = {[1:3]};
			1(1)	bins b12[] = {[8:9]};
			1(1)	bins b12[] = {[8:9]};
0.80 (80/100)	4/5	cg1.B	20	B: coverpoint b {
			0(1)	bins b21[] = {[3:5]};
			1(1)	bins b21[] = {[3:5]};
			1(1)	bins b21[] = {[3:5]};
			1(1)	bins b22[] = {[10:11]};
			1(1)	bins b22[] = {[10:11]};
0.33 (33/100)	1/3	cg1.CR1	24	CR1: cross A, B {
			0(1)	bins b1 = binsof (A) intersect {3} && binsof (B) intersect {6};
			0(1)	bins b2 = binsof (A.b12) intersect {5};
			1(1)	bins b3 = binsof (B.b22) intersect {11};

The `cross_auto_bin_max` option can be set at the covergroup and cross level but not at the coverpoint level. As a result, it applies only to the covergroup or cross in which it is set. If a design includes multiple covergroups and you want to save only user-defined cross bins without saving automatically-generated cross bins in the design, you can:

Use the `cross_auto_bin_max` option in all covergroups, or

Use the `set_covergroup -cross_auto_bin_max_default_zero` command in the coverage configuration file at elaboration

Note: The procedural assignment of the `cross_auto_bin_max` option is not supported, and if used in a procedural assignment, an error is displayed.

Example: Using the detect_overlap Option

The `detect_overlap` option specifies if a warning must be generated when there is an overlap between the range list or the transition list of bins of a coverpoint.

By default, the value of this option is set as `0`, which indicates that no warning will be generated for overlapping range lists or transition lists. Setting the value as `1` causes the tool to report a warning in case of overlapping ranges.

Consider the following example:

```
covergroup cg1 @(posedge clk);
option.detect_overlap = 1;
c1: coverpoint opcode1{
    bins x = {0,1,2};
    bins y = {1,2,3};
    bins a1[] = {0=>1=>2};
    bins a2[] = {0=>1=>2};
```

```
    }
endgroup : cg1
```

As the `detect_overlap` option is set to true, warnings are reported to indicate that an overlap exists in bin ranges for bins `x` and `y`, and transition bins `a1` and `a2`.

Limitations

Currently, no warning is generated in following scenarios even if the `detect_overlap` option is set to true:

Partially overlapping transition bins

```
option.detect_overlap = 1;
a : coverpoint a_var{
    bins a1[] = {0 => 1 => 2};
    bins a2[] = {0 =>1}; //partial overlap -- No warning generated
}
```

Overlapping of bins if any of the bins is either `ignore_bins` or `illegal_bins`, as shown below:

```
option.detect_overlap = 1;
a : coverpoint a_var{
    bins a1[] = {0,1,2};
    ignore_bins ign_1[] = {0,1,2}; //overlaps with bin a1
    ignore_bins ign_2[] = {3,4};
    ignore_bins ign_3[] = {3,4}; //overlaps with bin ign_2
}
```

Overlapping in the case of wildcard bins

Example: Using the cross_num_print_missing Option

The `cross_num_print_missing` option is used to specify the number of uncovered automatically generated cross tuples/bins that must be printed in the coverage report.

Note: This option controls only the number of uncovered automatically generated cross tuples/bins that must be printed in the coverage detailed report. It does not control the number of bins that must be saved to the coverage database. In addition, this option controls printing of automatically generated cross bins and not the user-defined cross bins.

Consider the following code:

```
covergroup cg1 @(posedge clk);
    c1: coverpoint opcode;
    c2: coverpoint address;
    c3: cross c1,c2
```

```
{
    option.cross_num_print_missing = 2;
}
endgroup : cg1
```

The coverage results from the above code are:

Number of covered bins: 6 of 24					
Coverage	Count	Name		Line	Origin
0.37 (37/100)	6/24	cg1		7	covergroup cg1 @(posedge clk);
0.50 (50/100)	2/4	cg1.c1		8	c1: coverpoint opcode;
	0(1)	auto[0]		8	c1: coverpoint opcode;
	1(1)	auto[1]		8	c1: coverpoint opcode;
	1(1)	auto[2]		8	c1: coverpoint opcode;
	0(1)	auto[3]		8	c1: coverpoint opcode;
0.50 (50/100)	2/4	cg1.c2		9	c2: coverpoint address;
	0(1)	auto[0]		9	c2: coverpoint address;
	1(1)	auto[1]		9	c2: coverpoint address;
	1(1)	auto[2]		9	c2: coverpoint address;
	0(1)	auto[3]		9	c2: coverpoint address;
0.12 (12/100)	2/16	cg1.c3		10	c3: cross c1,c2
	0(1)	<auto[0],auto[0]>		10	c3: cross c1,c2
	0(1)	<auto[0],auto[1]>		10	c3: cross c1,c2
	1(1)	<auto[1],auto[1]>		10	c3: cross c1,c2
	1(1)	<auto[2],auto[2]>		10	c3: cross c1,c2

2 uncovered automatic bins reported

In the above report, only two uncovered automatically generated cross bins are reported because the value of the `cross_num_print_missing` option is set as 2. If the value of this option is not set (if the default value is not overridden using the [set_covergroup-default_cross_num_print_missing](#) command at elaboration), then all the uncovered cross bins are reported.

Example: Using Covergroup Input Arguments to Set Coverage Options

Consider a piece of code where covergroup input argument is used to specify value for covergroup option `auto_bin_max`.

```
covergroup cg ( input int size ) @(posedge clk);
    option.auto_bin_max = size+1;
    coverpoint cp1;
endgroup
cg cg_inst1
cg_inst1 = new(2) ;
```

In the above code, covergroup argument `size` is used to specify the value of covergroup option `auto_bin_max`. A value 2 is passed to covergroup argument `size`, and therefore the value of `auto_bin_max` will be 3. See [Example: Defining Covergroups with Arguments](#) for more information on defining covergroup with arguments.

Consider another example where covergroup `input` argument of type `string` is used to set covergroup option `comment`.

```
covergroup cg(input string instcomment) @(posedge clk);
    option.per_instance = 1;
    option.comment = instcomment;
    coverpoint v1;
endgroup
cg cg_inst1;
cg cg_inst2;
...
cg_inst1 = new ("This is cg_inst1.");
cg_inst1.set_inst_name("cg_inst1");
cg_inst2 = new ("This is cg_inst2.");
cg_inst2.set_inst_name ("cg_inst2");
...
```

In the above code, covergroup `input` argument `instcomment` of type `string` is used to set the value of covergroup option `comment`. In the code, two instances of covergroup `cg` are created. A value "This is cg_inst1." is passed to set the value of covergroup option `comment` for `cg_inst1` and value "This is cg_inst2." is passed to set the value of covergroup option `comment` for `cg_inst2`.

The instance-based report generated from the above code is:

Value set for comment option (for cg_inst1)				
Coverage	Count	Name	Line	Origin
0.00 (0/100)	0/2	cg_inst1 // This is cg_inst1.	10	covergroup cg(input string instcomment) @(posedge clk);
0.00 (0/100)	0/2	cg_inst1.v1 0(1) 0(1)	13	coverpoint v1;
		auto[0] auto[1]	13	coverpoint v1;
0.00 (0/100)	0/2	cg_inst2 // This is cg_inst2.	10	covergroup cg(input string instcomment) @(posedge clk);
0.00 (0/100)	0/2	cg_inst2.v1 0(1) 0(1)	13	coverpoint v1;
		auto[0] auto[1]	13	coverpoint v1;

Value set for comment option (for cg_inst2)

See [Example: Defining Covergroups with Arguments](#) for more information on defining covergroup with arguments.

Note: You cannot use covergroup arguments of type other than `string` to set covergroup options `name` and `comment`. This is because these options can take only `string` value. Other covergroup options, such as `weight`, `at_least`, `goal`, `per_instance`, and `auto_bin_max` can be assigned only a

numeric value. As a result, using a `string` type argument to set values for these options will result in an error.

Type-Specific Covergroup Options

The following table describes the type options specific to covergroup type as a whole.

Option Name	Default	Description
<u>comment</u>	""	Specifies a comment that appears with the covergroup type, or with a coverpoint, or cross of the covergroup type. The comment is saved to the coverage database and is included in coverage reports.
<u>weight</u>	1	If set at the covergroup level, it specifies the weight of that covergroup for computing the overall cumulative covergroup type coverage. If set at the coverpoint or cross level, it specifies the weight of the coverpoint or the cross for computing the cumulative coverage of the enclosing covergroup. The specified weight should be a non-negative integral value.
<u>goal</u>	100	Specifies target goal for a covergroup type or for a coverpoint or a cross of a covergroup type.
<u>strobe</u>	0	When true, all samples are taken at the end of the time slot, like the <code>\$strobe</code> system task. With the <code>strobe</code> type option, samples for the coverage values are taken in the Postponed region, so that only one sample is taken for each time slot. The <code>strobe</code> type option is Boolean.

<u>real_interval</u>	NA	<p>When specified, it sets the value range of vector bins as the value of this type option is used as precision in specification of bin ranges.</p> <p>The <code>real_interval</code> type option does not have a default value and can be specified as any non-zero positive real number. If its value is not specified, an error is reported.</p> <p>Note: The <code>real_interval</code> type option can be used only with the type <code>real</code> coverpoints.</p>
<u>merge_instances</u>	1	<p>When specified, type coverage is calculated as the union of all bins from all covergroup instances.</p> <p>The <code>merge_instances</code> type option is boolean and currently, can only be specified as 1.</p> <p>The behavior of the type option <code>merge_instances</code> can be turn on/off for all the covergroup types in the complete design using the <code>set_covergroup -merge_instances_on</code> or <code>set_covergroup -merge_instances_off</code> coverage configuration file commands, respectively at elaboration.</p> <p>When you use the <code>set_covergroup -merge_instances_off</code> command, all the type option <code>merge_instances</code> instances in the design are overridden. Using the configuration file commands takes precedence over the value of <code>type_option.merge_instances</code> as specified in specific covergroups.</p>

You can set these options:

Inside the covergroup definition using the following syntax:

```
type_option.member_name = constant_expression;
```

where

- `member_name` is the name of the option.

- `constant_expression` is the value to be specified for the option. These options can be initialized only using constant expressions.

Outside the covergroup definition (in the procedural code). For more details, see [Procedural Assignment of Covergroup Options](#).

Note: Different instances of a covergroup cannot assign different values to type options.

The following table summarizes the syntactical level (covergroup, coverpoint, or cross) at which different type options can be specified.

Option	Allowed in Syntactic Level		
	Covergroup Cross	Coverpoint	
comment	Yes	Yes	Yes
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
strobe	Yes	No	No
real_interval	Yes	Yes	No
merge_instances	Yes	NA	NA

Example: Using comment Type Option

Consider the following code:

```
covergroup cg1 @(posedge clk);
    c1: coverpoint opcode{
        type_option.comment = "Comment for covergroup type";
        option.comment = "Comment for instance";
    }
endgroup : cg1
cg1 inst1;
...
```

In the above code, the comment for covergroup type is set using the `type_option` keyword and the comment for the instance is set using the `option` keyword.

The module-based report generated from the above code is:

Value set for comment option (for covergroup type)				
Coverage	Count	Name	Line	Origin
0.00 (0/100)	0/8	c91	8	covergroup c91 @(posedge clk);
0.00 (0/100)	0/8	c91.c1 // Comment for covergroup type	10	c1: coverpoint opcode{
0(1)		auto[0]	10	c1: coverpoint opcode{
0(1)		auto[1]	10	c1: coverpoint opcode{
0(1)		auto[2]	10	c1: coverpoint opcode{
0(1)		auto[3]	10	c1: coverpoint opcode{
0(1)		auto[4]	10	c1: coverpoint opcode{
0(1)		auto[5]	10	c1: coverpoint opcode{
0(1)		auto[6]	10	c1: coverpoint opcode{
0(1)		auto[7]	10	c1: coverpoint opcode{

The above report is generated with the `-module` option of the `report_detail` command and therefore, it shows the comment set for the covergroup type.

Note: The comment set for the covergroup instance (using the `option` keyword) will appear in the instance-based report that can be generated using the `-instance` option of the `report_detail` command.

Example: Using weight Type Option

Consider the following code:

```
module top;
...
covergroup cg1 (int w) @(posedge clk);
    c1: coverpoint opcode
    {
        type_option.weight = 1;
        option.weight = 4;
    }
    c2: coverpoint address
    {
        type_option.weight = 1;
        option.weight = w;
    }
endgroup : cg1
cg1 cover_inst11;
initial begin
    clk = 0;
    cover_inst11 = new(6);
    cover_inst11.set_inst_name("cover_inst11");
```

```
...
end
endmodule
```

In the above code, the weight set using the `type_option` keyword is used to calculate the type coverage, and the weight set using the `option` keyword is used to calculate the instance coverage.

The report generated from the above code is:

Module-Based Report

Number of covered bins: 3 of 8			
Coverage	Count	Name	Line Origin
0.37 (37/100)	3/8	cg1	7 covergroup cg1 (int v) @(posedge clk);
0.50 (50/100)	2/4	cg1.c1	11 c1: coverpoint opcode
	0(1)	auto[0]	11 c1: coverpoint opcode
	1(1)	auto[1]	11 c1: coverpoint opcode
	1(1)	auto[2]	11 c1: coverpoint opcode
	0(1)	auto[3]	11 c1: coverpoint opcode
0.25 (25/100)	1/4	cg1.c2	16 c2: coverpoint address
	0(1)	auto[0]	16 c2: coverpoint address
	2(1)	auto[1]	16 c2: coverpoint address
	0(1)	auto[2]	16 c2: coverpoint address
	0(1)	auto[3]	16 c2: coverpoint address

Instance-Based Report

Number of covered bins: 3 of 8			
Coverage	Count	Name	Line Origin
0.35 (35/100)	3/8	cover_inst11	7 covergroup cg1 (int v) @(posedge clk);
0.50 (50/100)	2/4	cover_inst11.c1	11 c1: coverpoint opcode
	0(1)	auto[0]	11 c1: coverpoint opcode
	1(1)	auto[1]	11 c1: coverpoint opcode
	1(1)	auto[2]	11 c1: coverpoint opcode
	0(1)	auto[3]	11 c1: coverpoint opcode
0.25 (25/100)	1/4	cover_inst11.c2	16 c2: coverpoint address
	0(1)	auto[0]	16 c2: coverpoint address
	2(1)	auto[1]	16 c2: coverpoint address
	0(1)	auto[2]	16 c2: coverpoint address
	0(1)	auto[3]	16 c2: coverpoint address

In the above report, coverage count of covergroup instance `cover_inst11` is same as the coverage count of covergroup type `cg1`. However, the weighted coverage of the covergroup `cg1` is 37/100 and the weighted coverage of the covergroup instance `cover_inst11` is 35/100. This is because the computation of weighted coverage of covergroup type differs from the computation of weighted coverage of covergroup instances.

Note: In this example, the goal is not set explicitly, and therefore, 100 is assumed as the target goal.

The coverage of covergroup instances is computed as:

$$\frac{((\text{hit_ratio of cpt_1} * \text{option.weight of cpt_1}) + (\text{hit_ratio of cpt_2} * \text{option.weight of cpt_2}) * 100)}{\text{option.weight of cpt_1} + \text{option.weight of cpt_2}}$$

where,

hit_ratio = number of hit bins of coverpoint / total_number_of_coverpoints

Using the above formula, the coverage of instance `cover_inst11` is computed as:

$$\frac{((2/4) * 4) + (1/4) * 6) * 100}{4 + 6} = 35$$

The coverage computed above is then divided by the target goal to calculate the weighted coverage of the covergroup instance `cover_inst11`.

The coverage of the covergroup type as a whole is computed as:

$$\frac{((\text{merge_of_coverage of cpt_1} * \text{type_option.weight of cpt_1}) + (\text{merge_of_coverage of cpt_2} * \text{type_option.weight of cpt_2}) * 100)}{\text{type_option.weight of cpt_1} + \text{type_option.weight of cpt_2}}$$

where,

`merge_of_coverage` is the merge of `hit_ratio` of the coverpoint from all of the instances

Using the above formula, the coverage of the covergroup type `cg1` is computed as:

$$\frac{((2/4) * 1) + (1/4) * 1) * 100}{1 + 1} = 37$$

The coverage computed above is then divided by the target goal to calculate the weighted coverage of the covergroup type `cg1`.

Example: Using goal Type Option

Consider the following code:

```
module top;
...
covergroup cg1 @(`clk);
    type_option.goal = 40;
    c1: coverpoint opcode{
        type_option.goal = 50;
    }
    c2: coverpoint address;
endgroup : cg1
```

```
cg1 cover_inst;
initial begin
    cover_inst = new;
    clk = 0;
...
endmodule
```

In the above code, target goal set for covergroup `cg` is 40 and target goal set for coverpoint `c1` is 50. Target goal is not set for the coverpoint `c2` and therefore, default goal 100 will be applied to coverpoint `c2` (if the default goal is not overridden using the [set_covergroup-default_type_option_goal](#) command).

The module-based report generated from the above code is:

Goal used as set using type_option.goal

Number of covered bins: 5 of 12				
Coverage	Count	Name	Line	Origin
0.92 (33/40)	5/12	cg1	7	covergroup cg1 @(`clk);
1.00 (50/50)	4/8	cg1.c1	9	c1: coverpoint opcode{
	0(1)	auto[0]	9	c1: coverpoint opcode{
	1(1)	auto[1]	9	c1: coverpoint opcode{
	0(1)	auto[2]	9	c1: coverpoint opcode{
	1(1)	auto[3]	9	c1: coverpoint opcode{
	0(1)	auto[4]	9	c1: coverpoint opcode{
	1(1)	auto[5]	9	c1: coverpoint opcode{
	0(1)	auto[6]	9	c1: coverpoint opcode{
	2(1)	auto[7]	9	c1: coverpoint opcode{
0.25 (25/100)	1/4	cg1.c2	12	c2: coverpoint address;
	0(1)	auto[0]	12	c2: coverpoint address;
	1(1)	auto[1]	12	c2: coverpoint address;
	0(1)	auto[2]	12	c2: coverpoint address;
	0(1)	auto[3]	12	c2: coverpoint address;

Default goal used is 100

In the above report, goal applied to covergroup `cg` is 40, coverpoint `c1` is 50, and coverpoint `c2` is 100.

Note: In the above example, the goal is not explicitly set for coverpoint `c2`, and therefore default goal 100 is used. You can override this default goal using the [set_covergroup-default_type_option_goal](#) command in the coverage configuration file at elaboration.

If the default goal is set as 85 using the following command in the coverage configuration file, then the goal applied to the coverpoint `c2` will be 85 instead of 100.

```
set_covergroup -default_type_option_goal 85
```

The report generated after using the above command in the coverage configuration file is:

Number of covered bins: 5 of 12					
Coverage	Count	Name		Line	Origin
0.92 (37/40)	5/12	cg1		7	covergroup cg1 @(`clk);
1.00 (50/50)	4/8	cg1.c1		9	c1: coverpoint opcode{
	0(1)	auto[0]		9	c1: coverpoint opcode{
	1(1)	auto[1]		9	c1: coverpoint opcode{
	0(1)	auto[2]		9	c1: coverpoint opcode{
	1(1)	auto[3]		9	c1: coverpoint opcode{
	0(1)	auto[4]		9	c1: coverpoint opcode{
	1(1)	auto[5]		9	c1: coverpoint opcode{
	0(1)	auto[6]		9	c1: coverpoint opcode{
	2(1)	auto[7]		9	c1: coverpoint opcode{
0.29 (25/85)	1/4	cg1.c2		12	c2: coverpoint address;
	0(1)	auto[0]		12	c2: coverpoint address;
	1(1)	auto[1]		12	c2: coverpoint address;
	0(1)	auto[2]		12	c2: coverpoint address;
	0(1)	auto[3]		12	c2: coverpoint address;

Goal used is 85

In the above report, the goal applied to coverpoint c2 is 85 instead of 100 because the default goal is overridden using the `set_covergroup -default_type_option_goal` command.

Example: Using strobe Type Option

Consider the following code:

```
module top;
integer x;

covergroup cg @ (x);
    type_option.strobe= 1 ;
    option.per_instance = 1;
    coverpoint x {
        bins b1[] = {[0:10]};
    }
endgroup
cg cgi;

always @(x)
    $display("x = %d", x);
initial
begin
    cgi = new;
    cgi.set_inst_name("cgi");
    #1 x=4;
    #0 x=1; #0 x=3; x=0;
```

```
#0 x=1; #0 x=3;
x<=2; // this value of x is sampled
end

endmodule
```

In the above code, the value of `x` changes multiple times in a single time slot as shown in the given XMSIM output:

```
ncsim> run 1500 ns
x =        4
x =        1
x =        0
x =        1
x =        3
x =        2
```

When the `type_option.strobe` is specified as `1` for covergroup `cg`, only the last value of `x` is sampled. The type-based report generated from the given code with the strobe option enabled is:

Coverage	Count	Name	Line	Origin
0.09 (9/100)	1/11	cg	7	covergroup cg @ (x);
0.09 (9/100)	1/11	cg.x	10	coverpoint x {
0(1)		b1[0] to [1]	11	bins b1[] = {[0:10]};
1(1)		b1[2]	11	bins b1[] = {[0:10]};
0(1)		b1[3] to [10]	11	bins b1[] = {[0:10]};

The type-based report generated from the given code with the strobe option disabled is:

Coverage	Count	Name	Line	Origin
0.45 (45/100)	5/11	cg	7	covergroup cg @ (x);
0.45 (45/100)	5/11	cg.x	10	coverpoint x {
1(1)		b1[0]	11	bins b1[] = {[0:10]};
2(1)		b1[1]	11	bins b1[] = {[0:10]};
1(1)		b1[2]	11	bins b1[] = {[0:10]};
1(1)		b1[3]	11	bins b1[] = {[0:10]};
1(1)		b1[4]	11	bins b1[] = {[0:10]};
0(1)		b1[5] to [10]	11	bins b1[] = {[0:10]};

Example: Using real_interval Type Option

Consider the following code in which an embedded covergroup, `cg1`, has a coverpoint declared on a real variable, `x_r`. The coverpoint has a user-defined vector bin, `x1`, which contains two real literal values:

```
module top;
    logic clk;
```

```

reg [2:0] y_i;
real x_r;

covergroup cg1 @(clk);
type_option.real_interval = 0.1;
coverpoint x_r
{
    bins x1[] = {[1.1:2.1]};
}
endgroup : cg1
cg1 cover_inst = new;
initial begin
clk = 0;
$monitor("x_r = ", x_r);
x_r = 0;
#1 x_r = 1.1; clk=~clk;
#1 x_r = 1.3; clk=~clk;
#1 $finish();
end
endmodule

```

The type-based report generated from the given code is:

```

Instance name: top
Module/Entity/Interface/Program name: top
File name: /vobs/nc_test/test/vlog/coverage/covergroup/cg_real/bt/test1.v
Number of covered bins: 2 of 10

```

Coverage	Count	Name	Line	Origin
0.20 (20/100)	2/10	cover_inst	7	covergroup cg1 @(clk);
0.20 (20/100)	2/10	cover_inst.x_r	9	coverpoint x_r
	1(1)	x1[1.1:1.2)	11	bins x1[] = {[1.1:2.1]};
	0(1)	x1[1.2:1.3)	11	bins x1[] = {[1.1:2.1]};
	1(1)	x1[1.3:1.4)	11	bins x1[] = {[1.1:2.1]};
	0(1)	x1[1.4:1.5)	11	bins x1[] = {[1.1:2.1]};
	0(1)	x1[1.5:1.6)	11	bins x1[] = {[1.1:2.1]};
	0(1)	x1[1.6:1.7)	11	bins x1[] = {[1.1:2.1]};
	0(1)	x1[1.7:1.8)	11	bins x1[] = {[1.1:2.1]};
	0(1)	x1[1.8:1.9)	11	bins x1[] = {[1.1:2.1]};
	0(1)	x1[1.9:2)	11	bins x1[] = {[1.1:2.1]};
	0(1)	x1[2:2.1]	11	bins x1[] = {[1.1:2.1]};

Example: Using merge_instances Type Option

Consider the following code containing a cross with an automatic bin and user defined bin. In this example, auto cross bin tuples contributing to user defined bin is based on formal arguments of covergroup:

```
module top;
```

```
reg clk;
reg [2:0] v1, v2;

covergroup cg(int a, int b) @(clk);
    option.per_instance = 1;
    type_option.merge_instances = 1;
    CP1: coverpoint v1 {
        bins b1[] = {[0:1]} ;
    }
    CP2: coverpoint v2 {
        bins b2[] = {[0:1]} ;
    }

    CR1 : cross CP1, CP2
    {
        bins b1 = binsof(CP1) intersect {a}
        && binsof(CP2) intersect {[a:b]} ;
    }
endgroup

cg cg_inst1 = new(0,1);
cg cg_inst2 = new(1,0);

initial
begin
    cg_inst1 = new(0,1);
    #1 v1 = 0;      v2 = 0;
    #1 v1 = 0;      v2 = 1;
    #1 v1 = 1;      v2 = 0;
    cg_inst2 = new(1,0);
    #1 v1 = 1;      v2 = 1;
    ...
end
endmodule
```

The instance-based report of the covergroup instance, `cg_inst1`, is shown. In the given report, cross `CR1` has one user defined bin `b1` and two automatic cross bins `b1[1],b2[0]` and `b1[1],b2[1]`. Notice that the automatic cross bins `b1[0],b2[0]` and `b1[0],b2[1]` are not in the report as they are a part of the user defined bin `b1`.

```
Type name: top
File name: /vobs/nc_test/vlog/coverage/covergroups/test.v
Number of covered bins: 7 of 7
Number of uncovered bins: 7 of 7
Number of excluded cover bins: 0
Name          Grade   Line  Source Code
-----
cg_inst1      67%(7/7) 10    covergroup CG(int a, int b)...;
|--CP1         50%(1/2) 11    CP1 : coverpoint v1 {
|  |--b1[0]    100%(1/1) 12    bins b1[] ={{[0:1]}} ;
|  |--b1[1]    100%(1/1) 12    bins b1[] ={{[0:1]}} ;
|--CP2         100%(1/1) 11    CP2 : coverpoint v2 {
|  |--b2[0]    100%(1/1) 12    bins b2[] ={{[0:1]}} ;
|  |--b2[1]    100%(1/1) 12    bins b2[] ={{[0:1]}} ;
|--CR1         100%(3/3) 11    CR1 : cross A,B {
|  |--b1       100%(1/1) 12    CR1 : cross A,B {
|  |--b1[1],b2[0] 100%(1/1) 12    CR1 : cross A,B {
|  |--b1[1],b2[1] 100%(1/1) 12    CR1 : cross A,B {
```

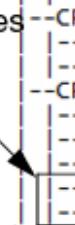
The instance-based report of the covergroup, `cg_inst2`, generated from the given code is shown:

```
Type name: top
File name: /vobs/nc_test/vlog/coverage/covergroups/test.v
Number of covered bins: 3 of 7
Number of uncovered bins: 3 of 7
Number of excluded cover bins: 0
Name          Grade   Line  Source Code
-----
cg_inst2      43%(3/7) 10    covergroup CG(int a, int b)...;
|--CP1         50%(1/2) 11    CP1 : coverpoint v1 {
|  |--b1[0]    0%(0/1) 12    bins b1[] ={{[0:1]}} ;
|  |--b1[1]    100%(1/1) 12    bins b1[] ={{[0:1]}} ;
|--CP2         50%(1/2) 11    CP2 : coverpoint v2 {
|  |--b2[0]    0%(0/1) 12    bins b2[] ={{[0:1]}} ;
|  |--b2[1]    100%(1/1) 12    bins b2[] ={{[0:1]}} ;
|--CR1         33%(1/3) 11    CR1 : cross A,B {
|  |--b1       100%(1/1) 12    CR1 : cross A,B {
|  |--b1[0],b2[0] 0%(0/1) 12    CR1 : cross A,B {
|  |--b1[0],b2[1] 0%(0/1) 12    CR1 : cross A,B {
```

As mentioned earlier, in the above example, the cross `CR1` of covergroup type contains five bins, two automatic bins from each covergroup instance and a common user defined bin. The type-based report of the covergroup, `cg`, generated from the given code is shown. Notice that auto cross bin tuples, `b1[0],b2[0]` and `b1[0],b2[1]`, are shown as uncovered in covergroup type even though these tuples are covered as a part of the user defined cross bin `b1` of covergroup instance `cg_inst1`.

Name	Grade	Line	Source Code
cg	77%(7/9)	10	covergroup CG(int a, int b)...;
--CP1	50%(2/2)	11	CP1 : coverpoint v1 {
--b1[0]	100%(1/1)	12	bins b1[] = {[0:1]} ;
--b1[1]	100%(2/1)	12	bins b1[] = {[0:1]} ;
--CP2	100%(2/2)	11	CP2 : coverpoint v2 {
--b2[0]	100%(1/1)	12	bins b2[] = {[0:1]} ;
--b2[1]	100%(2/1)	12	bins b2[] = {[0:1]} ;
--CR1	100%(3/5)	11	CR1 : cross A,B {
--b1	100%(2/1)	12	CR1 : cross A,B }
--b1[1],b2[0]	100%(1/1)	12	CR1 : cross A,B }
--b1[1],b2[1]	100%(1/1)	12	CR1 : cross A,B }
--b1[0],b2[0]	0%(0/1)	12	CR1 : cross A,B }
--b1[0],b2[1]	0%(0/1)	12	CR1 : cross A,B }

Cross tuples shown as uncovered



For automatically created cross bins, bin names are of the form <binname1, ..., binnameN>, where the bin names are derived from the crossed coverpoint bins. In this case, the instances sharing the same cross product bin name have overlapping bins.

When the `type_option.merge_instances=1` is specified in a covergroup, covergroup types saved to the database depend on the unique specialization of module, with a separate covergroup type saved for each unique specialization of module. Consider the given example:

```
module test;
  reg clk;
  reg [1:0] v1;
  parameter p =1 ;

  covergroup cg(int a) @(clk);
    option.per_instance = 1;
    type_option.merge_instances = 1;
    CP1: coverpoint v1 {
      bins b1 = {a};
    }
  endgroup

  cg cobj;

  initial
  begin
    cobj = new(0);
    ...
  end
endmodule
```

```
module top ();
    test T1();
    test T2();
    test #( .p(2) ) T3();
    test #( .p(2) ) T4();
endmodule
```

As shown in the given report, there are two specialization of module, and hence two separate covergroup types are saved to the database. To differentiate between covergroup types from different specialization of module, specialized module name is added as prefix to covergroup type name. Counts from covergroup instance belonging to a particular specialization of module are accumulated into its type.

```
Type name: test
File name: /vobs/nc_test/vlog/coverage/covergroups/test.v
Number of covered bins: 2 of 2
Number of uncovered bins: 0 of 2
Number of excluded cover bins: 0
```

Name	Grade	Line	Source Code
test#(1)::cg	100%(2/1)	10	covergroup CG @ (clk)
--CP1	100%(2/1)	11	CP1 : coverpoint v1 {
--b1	100%(2/1)	12	bins b1 = {a};
test#(2)::cg	100%(2/1)	10	covergroup CG @ (clk)
--CP1	100%(2/1)	11	CP1 : coverpoint v1 {
--b1	100%(2/1)	12	bins b1 = {a};

Specialized module names as cg type name prefix

The behavior of the type option `merge_instances` can be turned on/off for all the covergroup types in the complete design using the

`set_covergroup -merge_instances_on` or `set_covergroup - merge_instances_off` coverage configuration file commands, respectively at elaboration.

When you use the `set_covergroup -merge_instances_off` command, all the type option `merge_instances` instances in the design are overridden. Using the configuration file commands takes precedence over the value of `type_option.merge_instances` as specified in specific covergroups.

Limitations of `merge_instances`

- Covergroup types with and without `type_option.merge_instances=1` cannot be merged.
- Type option `merge_instances` is not supported for covergroups with real coverpoints and for

covergroups containing crosses with hierarchical coverpoints. In both of these cases, an error at elaboration time is reported.

Note: During merging, the covergroup types with `type_option.merge_instances=1` are merged based on their names but for these covergroup types, covergroup instances are merged based on their valid space.

Procedural Assignment of Covergroup Options

You can set covergroup options (both instance-specific and type-specific) through procedural assignment for covergroups, coverpoints, and crosses. The type-specific options are assigned values using the `type_option` keyword and instance-specific options are assigned values using the `option` keyword.

The following code demonstrates the procedural assignment of `goal` option.

```
covergroup cg @(clk);
    option.per_instance = 1;
    option.auto_bin_max = 4;
    A: coverpoint a{
    }
    B : coverpoint b{
        type_option.weight = 6;
    }
endgroup : cg
cg cgi= new ;
cg cgi2= new;
initial begin
    cgi.set_inst_name("cgi");
    cgi2.set_inst_name("cgi2");
...
#1 cg::type_option.goal = 40; //goal assigned to covergroup type cg is 40
#1 cgi.option.goal = 80; //goal assigned to covergroup instance cgi is 80
#2 cgi2.option.goal = 30; //goal assigned to covergroup instance cgi2 is 30
end
```

The instance-based report generated from the above code is:

Goal used -- 80(cgi.option.goal = 80)					
Coverage	Count	Name	Line Origin		
0.46 (37/80)	3/8	cgi	10	covergroup cg @clk;	
0.50 (50/100)	2/4	cgi.A	13	A: coverpoint a{	
0(1)		auto[8:11]	13	A: coverpoint a{	
0(1)		auto[12:15]	13	A: coverpoint a{	
0.25 (25/100)	1/4	cgi.B	15	B : coverpoint b{	
0(1)		auto[0:3]	15	B : coverpoint b{	
0(1)		auto[8:11]	15	B : coverpoint b{	
0(1)		auto[12:15]	15	B : coverpoint b{	
1.00 (37/30)	3/8	cgi2	10	covergroup cg @clk;	
0.50 (50/100)	2/4	cgi2.A	13	A: coverpoint a{	
0(1)		auto[8:11]	13	A: coverpoint a{	
0(1)		auto[12:15]	13	A: coverpoint a{	
0.25 (25/100)	1/4	cgi2.B	15	B : coverpoint b{	
0(1)		auto[0:3]	15	B : coverpoint b{	
0(1)		auto[8:11]	15	B : coverpoint b{	
0(1)		auto[12:15]	15	B : coverpoint b{	

Goal used -- 30 (cgi2.option.goal = 30)

In the above report, goal applied to covergroup instance `cgi` is 80, and covergroup instance `cgi2` is 30.

The module-based report generated from the above code is:

Goal used --40 (cg::type_option.goal = 40)					
Coverage	Count	Name	Line Origin		
0.70 (28/40)	3/8	cg	10	covergroup cg @clk;	
0.50 (50/100)	2/4	cg.A	13	A: coverpoint a{	
0(1)		auto[8:11]	13	A: coverpoint a{	
0(1)		auto[12:15]	13	A: coverpoint a{	
0.25 (25/100)	1/4	cg.B	15	B : coverpoint b{	
0(1)		auto[0:3]	15	B : coverpoint b{	
0(1)		auto[8:11]	15	B : coverpoint b{	
0(1)		auto[12:15]	15	B : coverpoint b{	

In the above report, goal applied to covergroup `cg` is 40.

- ⓘ The `per_instance` option can be set only in the covergroup definition. In addition, the `auto_bin_max` and `detect_overlap` options can be set only in the covergroup or coverpoint definition.

The following tables summarize the syntactical level (covergroup, coverpoint, or cross) at which different options and type options can be assigned procedurally.

A. Options

Option	Allowed in Syntactic Level		
	Covergroup Cross	Coverpoint	
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
name	Yes	NA	NA
comment	Yes	Yes	Yes
at_least	Yes	Yes	Yes
auto_bin_max	No	No	NA
cross_auto_bin_max	No	NA	No
per_instance	No	NA	NA
detect_overlap	No	No	NA
get_inst_coverage	No	NA	NA
cross_num_print_missing	Yes	NA	Yes

Hierarchical access to covergroup instance is allowed, and can now be used to:

- Assign values to options
- Call built-in methods (such as `start()` and `stop()`)
- Call `new()` to instantiate a covergroup and pass arguments
- Create crosses with some local members

Consider the given example, in which covergroup instance are accessed through module instance to assign values to options:

```
module top
  reg clk;
  int v1;
  covergroup cg@(clk);

```

```

coverpoint v1;
endgroup
cg cgi = new;
endmodule

module test;
top i1();
initial
    i1.cgi.option.goal = 5;
    i1.cgi.start();
endmodule

```

In the given example, covergroup instance, `cgi`, is accessed through module instance, `i1` to assign value to option goal. In addition, in the example, a hierarchical expression is used to call a built-in method, `start()`, of covergroup.

B. Type Options

Type Option	Allowed in Syntactic Level		
	Covergroup Cross	Coverpoint	
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
comment	Yes	Yes	Yes
strobe	No	NA	NA
real_interval	Yes	Yes	No
merge_instances	No	NA	NA

Note: Hierarchical access to type options is not supported.

Considerations When Assigning Values to Covergroup Options in the Procedural Code

When assigning values to covergroup options in the procedural code, remember that:

Use of type parameter or an object of type parameter in the hierarchical name is currently not supported.

```

parameter type T = c;
T o1 = new;

```

```
...
T::cg::type_options::goal = 5; // Not supported because T is a type parameter
o1.cg.option.goal = 80; //Not supported as o1 is an object of type parameter
```

Procedural assignment is not supported:

- As an RHS in an expression

```
a1::cg::type_option.weight = a2::cg::type_option.weight; // Not supported as RHS
```

- As parameters to functions or tasks

```
$display(cgi.option.goal); // Not supported as a parameter to function
```

- In conditional expressions

```
if (a2::cg::type_option.weight > 2) // Not supported in conditional expression
```

If coverage is dumped during simulation run and name is assigned to a covergroup instance in the procedural code after the coverage database is dumped, then incorrect coverage results might be observed in the report generated at the end of simulation.

The `type_option` of embedded covergroup cannot be set procedurally.

Covergroups in Classes

A class is a user-defined data type that can encapsulate data members and methods. Data members and methods are used together to define the functionality and characteristics of an object. By embedding a covergroup within a class, you can cover a subset of data members. This integration of coverage with classes provides a mechanism for defining the coverage model associated with a class.

A covergroup declaration within a class is an embedded covergroup declaration. Consider the following code where a covergroup is declared inside a class.

```
class xyz;
    bit [3:0] m_x;
    int m_y;
    bit m_z;
    covergroup cov1 @m_z; // embedded covergroup
        coverpoint m_x;
        coverpoint m_y;
    endgroup
    function new(); cov1 = new; endfunction
endclass
```

An embedded covergroup declaration declares an anonymous covergroup type and an instance

variable of the anonymous type. The covergroup identifier defines the name of the instance variable. In the above example, a variable `cov1` (of the anonymous coverage group) is implicitly declared. Data members `m_x` and `m_y` are covered using covergroup `cov1` and are sampled on every change of data member `m_z`. Also notice the instantiation of covergroup variable `cov1` in the `new` method.

Note: Use of packages having embedded covergroups is supported. For more information, refer to [Covergroups in Packages](#).

Note: For coverage sampling, it is important that the covergroup variable is explicitly instantiated in the `new` method.

You can also define a covergroup outside a class and create its instances within the class, as shown below:

```
covergroup cg @clk;
    coverpoint pc1.p1;
endgroup
class c1;
    cg cg1; // Covergroup variable of a non-embedded covergroup
    reg [4:0] p1;
    function new();
        p1 = 3;
        cg1 = new; //Covergroup instantiation inside a class
    endfunction
endclass
```

Covergroup declaration in the parent class and instantiation in the derived class

A covergroup can be declared in the parent class and instantiated in the derived class. In such a scenario, covergroup sampling is enabled only when an object of the derived class is created, as shown in the code below.

```
class xyz;
    bit [3:0] m_x;
    int m_y;
    bit m_z;
    covergroup cov1 @m_z; // embedded covergroup
        coverpoint m_x;
        coverpoint m_y;
    endgroup
endclass
class abc extends xyz;
    int m_a, m_b, m_c;
    function new(); cov1 = new; endfunction
```

```
endclass  
abc a1 = new; // enables sampling  
xyz x1 = new; // does not enable sampling
```

In the above code, the embedded covergroup `cov1` is instantiated in the `new` method of a derived class. As a result, covergroup sampling is enabled only when object `a1` of the derived class is created. Creating an object of the parent class will not enable sampling because the function `new` is not defined in it.

Covergroup with the same name in parent and derived class

You can define covergroups with the same name in the parent class as well as the derived class. In such a scenario, use the `super` keyword to refer to the members of the parent class, as shown in the code below.

```
class xyz;  
    bit [3:0] m_x;  
    int m_y;  
    bit m_z;  
    covergroup cov1 @m_z; // embedded covergroup  
        coverpoint m_x;  
        coverpoint m_y;  
    endgroup  
endclass  
class abc extends xyz;  
    int m_a, m_b, m_c;  
    covergroup cov1 @m_a; // embedded covergroup  
        coverpoint m_b;  
        coverpoint m_c;  
    endgroup  
    function new();  
        cov1 = new; // Instantiate covergroup of current class.  
        super.cov1 = new; // Instantiate covergroup of parent class.  
    endfunction  
endclass
```

In the above code, both the parent and the derived class include a covergroup with the identical name. You use the `super` keyword to instantiate the covergroup of the parent class. You can access only one level of parent class through the `super` keyword. Using more than one level (for example `super.super.cov1`) will result in a parse error.

Arrays of classes with class embedded covergroup

Coverage data collection for dynamic arrays, associative arrays, and queues of class objects with

embedded covergroup is supported. Consider the following code where an associative array of a class is created.

```

module test;
    ...
    class MyClass;
        bit [3:0] by;
        covergroup CG;
            option.per_instance = 1;
            A: coverpoint by;
            option.auto_bin_max = 4;
        endgroup // CG
        function new(string nm);
            CG = new();
            CG.set_inst_name({nm, ".CG"});
            ...
        endfunction // new
    endclass // MyClass
    MyClass o[string]; //associative array of class MyClass
    ...
    initial begin
        o["1st"] = new("1st");
        o["2nd"] = new("23rd");
        o["3rd"] = new("23rd");
    end
    ...
endmodule // test

```

The instance-based report from the above code is:

Number of covered bins: 6 of 8				
Coverage	Count	Name	Line	Origin
0.75 (75/100)	3/4	1st.CG	8	covergroup CG;
0.75 (75/100)	3/4	1st.CG.A	10	A: coverpoint by;
	1(1)	auto[0:3]	10	A: coverpoint by;
	2(1)	auto[4:7]	10	A: coverpoint by;
	2(1)	auto[8:11]	10	A: coverpoint by;
	0(1)	auto[12:15]	10	A: coverpoint by;
0.75 (75/100)	3/4	23rd.CG	8	covergroup CG;
0.75 (75/100)	3/4	23rd.CG.A	10	A: coverpoint by;
	6(1)	auto[0:3]	10	A: coverpoint by;
	2(1)	auto[4:7]	10	A: coverpoint by;
	2(1)	auto[8:11]	10	A: coverpoint by;
	0(1)	auto[12:15]	10	A: coverpoint by;

Coverage for 1st array element

Coverage merged for identically named array elements

In the above report, coverage for identically named instances is merged.

The module-based report from the above code is:

Coverage for covergroup CG

embedded in
MyClass

Number of covered bins: 3 of 4					
Coverage	Count	Name		Line	Origin
0.75 (75/100)	3/4	MyClass::CG		8	covergroup CG;
0.75 (75/100)	3/4	MyClass::CG.A		10	A: coverpoint by;
	7(1)	auto[0:3]		10	A: coverpoint by;
	4(1)	auto[4:7]		10	A: coverpoint by;
	4(1)	auto[8:11]		10	A: coverpoint by;
	0(1)	auto[12:15]		10	A: coverpoint by;

Note: Dynamic or static arrays of covergroup objects is not supported.

Covergroups in Classes: Use of constant Variable to Define Bin Range/Value

Consider an example where constant variables are used to define bin ranges for covergroups embedded within a class.

```
module top() ;
class cl ;
    const int unsigned MODEX = 1; //const var declared
    const int unsigned MODEY = 2; //const var declared
    int unsigned c_sig;
    covergroup cg ;
        cp : coverpoint c_sig {
            bins MODEX = {MODEX}; //const var used to define the bin range
            bins MODEY = {MODEY}; //const var used to define the bin range
        }
    endgroup
    ...
endclass
cl cinst = new(2) ;
...
endmodule
```

In the above code, instance constant variables `MODEX` and `MODEY` are used to define the bin ranges for bins defined within coverpoint `cp` of covergroup `cg`.

Covergroups in Parameterized Classes/Modules

Covergroups can be specified in parameterized classes and modules. Consider the code.

```
module mod_1 #(type t = logic, type t2 = logic);
reg clk;
class class_1;
    reg [2:0] vc;
endclass
```

```
class class_2;
    reg vc;
endclass

class class_3 #(type t=class_1, int size=2); //parameterized class
t v1;
logic[size:0] v2;
covergroup cg@(clk);
    coverpoint v1.vc;
    coverpoint v2{
        bins b1[] = {[0:size]};
    }
endgroup
function new;
    v1 = new;
    cg = new;
    v1.vc = 2;
endfunction
endclass

class_3 #(class_2, 7) obj = new;

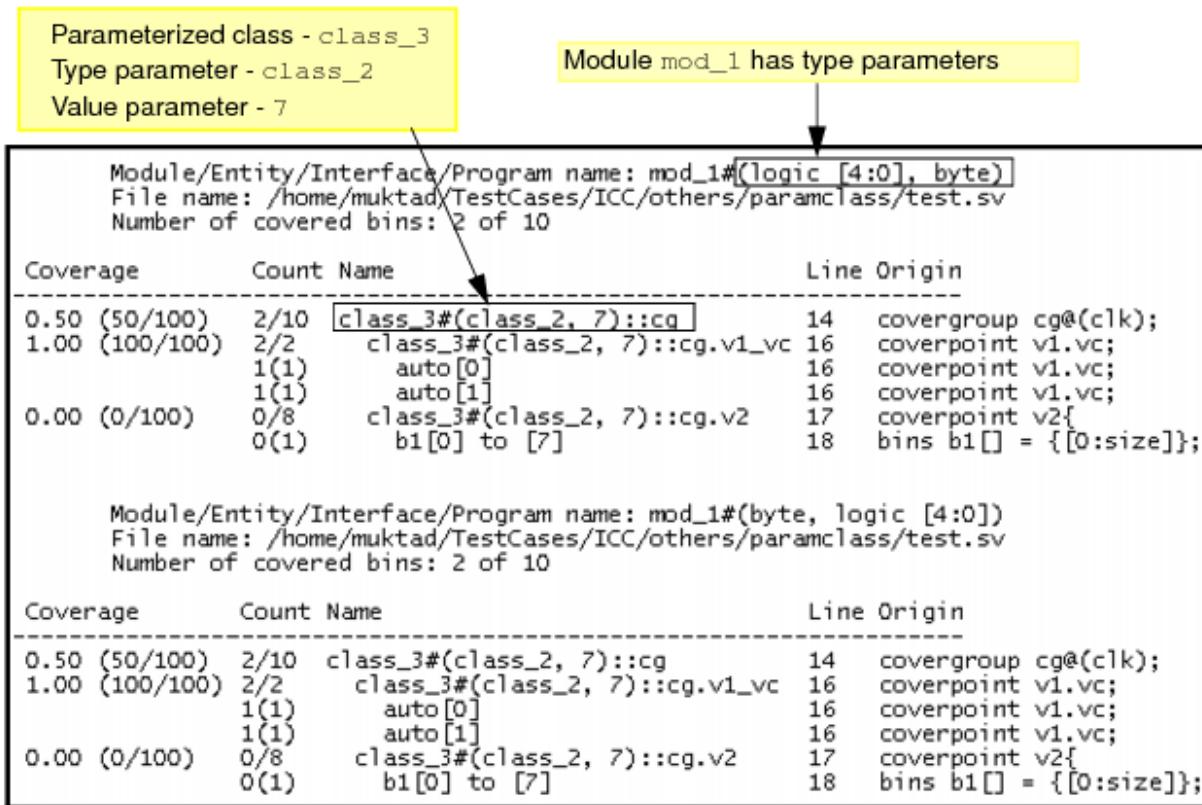
initial
begin
    clk = 0;
    obj.v1.vc = 5;
    #2 clk = 1;
    #3 obj.v1.vc = 8;
    #5 clk = 0;
    #20 $finish;
end
endmodule

module top;
    typedef logic[4:0] T;
    mod_1 #(T,byte) i1();
    mod_1 #(byte, T) i2();
endmodule
```

In the above code, class `class_3` is a parameterized class with `t` as the type parameter and `size` as the value parameter. The value passed to `t` is `class_2` and `size` is `7`. In this code, module `mod_1` is a

parameterized module with type parameters t and t_2 . For parameterized modules, the type name is appended to the module name in the report.

For details, refer to the given instance-based report:



As shown in the given instance-based report, for instance `i1`, module name is printed as `mod_1#(logic [4:0], byte)` and for instance `i2`, module name is printed as `mod_1#(byte, logic [4:0])`, as shown below.

For more on SystemVerilog parameterized classes, see the [SystemVerilog Reference Guide](#).

Covergroups in classes - Performance considerations

If a design includes class embedded covergroups and the design has many class objects, then the time taken to dump coverage data might increase significantly. To reduce the coverage database dumping time, use the `set_covergroup -optimize_dump` command at elaboration.

Covergroups in Compilation Units

Covergroups are not supported in a compilation unit scope. The only way to use a covergroup in a compilation unit scope is to embed it within a class, which is declared inside the compilation unit.

For more on SystemVerilog compilation units, see the [SystemVerilog Reference Guide](#).

Covergroups in Interfaces

An interface is a named bundle of nets or variables that encapsulates the connectivity between blocks. By declaring an interface, you can define a group of signals once in one modeling block. The interface can then be instantiated in the design and can be accessed through a module port as a single item. In addition to connectivity, an interface can encapsulate functionality. An interface can contain data type declarations, tasks and functions, initial and always blocks, continuous assignments, and so on.

Similar to defining a covergroup within a module, you can define a covergroup within an interface. The behavior of a covergroup within an interface is exactly same as a covergroup inside a module.

Consider the following code where the instance of an interface is passed in the port list of a module.

```
interface inf(input clk);
    logic [3:0] a;
    covergroup cg @(posedge clk) {
        A: coverpoint a {
            bins b1[] = {[1:10]};
        }
    endgroup
    cg cg_inst = new();
endinterface

module top;
...
inf interface_inst(clk); // interface instance
mod module_inst(interface_inst, clk); // module instance
endmodule

module mod(inf inf_inst , input clk);
initial
begin
    inf_inst.a = 4'b0001;
...

```

```
end  
endmodule
```

In the above code, the covergroup `cg` is defined within an interface `inf`. This interface is instantiated within the module `top` and passed as a parameter to the module `mod`. For the above code, bins `b1[1]`, `b1[3]`, and `b1[4]` are covered and rest are uncovered.

For more on SystemVerilog interfaces, see the [SystemVerilog Reference Guide](#).

Covergroups in Program Blocks

A program block, similar to a module, facilitates the creation of a testbench, but has special syntax and semantic restrictions. A program block:

- Provides an entry point to the execution of testbenches
- Acts as a scope for the data contained in the program block
- Provides a syntactic context that schedules events in the reactive region

Consider an example where a covergroup is defined in a program block.

```
module top;  
logic [3:0] I1, I2, O1, O2;  
logic CLK;  
test1 T1 (O1,O2,I1,I2,CLK);  
design DUT (O1,O2,I1,I2,CLK);  
endmodule  
  
module design (o1,o2,i1,i2,clk);  
output logic [3:0] o1,o2;  
input logic [3:0] i1,i2;  
input logic clk;  
// logic which decides the functionality of design.  
endmodule
```

```
program test1(o1,o2,i1,i2,clk);  
input [3:0] o1,o2;  
output [3:0] i1,i2;  
output clk;  
logic [3:0] i1,i2;
```

```
logic clk=0;
covergroup CG @(posedge clk);
    I1: coverpoint i1;
    I2: coverpoint i2;
    O1: coverpoint o1;
    O2: coverpoint o2;
    I1xI2: cross I1, I2;
    O1xO2: cross O1, O2;
endgroup
CG g = new;
initial forever #1 clk = ~clk;
initial begin
    i1 = 0;
    i2 = 0;
    @(negedge clk);
    i1 = 4;
    i2 = 8;
    @(negedge clk);
    // directed stimulus
end
endprogram
```

In the above code, the module `design` is verified using the program block `test1`. The wrapper module `top` is used to connect the module `design` with the program block. The program block generates the stimulus and samples the outputs well. The covergroup `CG` declared in program block `test1` samples and tracks the covered input and output combinations.

For more on the SystemVerilog program block construct, see the [SystemVerilog Reference Guide](#).

Covergroups in Generate Blocks

You can declare and instantiate covergroups in the generate block of your SystemVerilog code. A generate block can be defined in a module using the `generate` and `endgenerate` keywords, as shown below:

```
generate
    generate_item
endgenerate
```

Here, `generate_item` can be any of the following:

for loop

if block

case block

When defining covergroups within generate blocks, you can:

- Declare and instantiate the covergroup in same generate block, or
- Declare the covergroup in parent scope and instantiate it in the generate block

Declare and instantiate the covergroup in same generate block

In the following example, covergroup `cg` is declared and instantiated in the same `generate` block.

```
module test;
parameter p = 1;
reg clk,a;
always begin
#2 clk = ~clk;
#3 a = ~a;
end

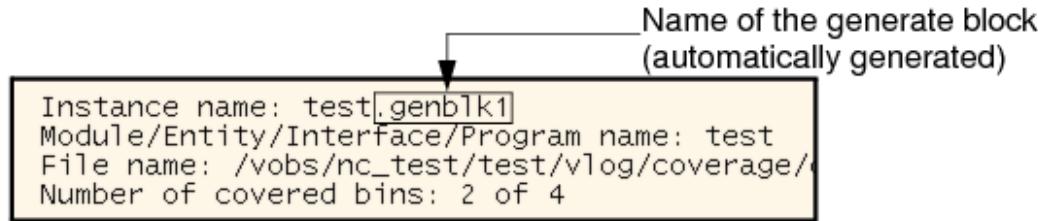
initial
begin
clk = 2;
a = 3;
#20 $finish;
end
generate
if(p > 0)
begin
covergroup cg @(posedge clk); //covergroup declaration
A: coverpoint a;
endgroup
cg cg_inst = new; //covergroup instantiation
initial
cg_inst.set_inst_name(" if_cginst");
end
endgenerate
endmodule
```

Declare the covergroup in parent scope and instantiate it in the generate block

In the following example, covergroup `cg` is declared in the module scope, and its instance `cg_inst` is created in the `generate` block.

```
module test;
parameter p = 1;
reg clk,a;
covergroup cg @(posedge clk); //covergroup declared in module scope
    A: coverpoint a;
endgroup
always begin
    #2 clk = ~clk;
    #3 a = ~a;
end
initial
begin
    clk = 2;
    a = 3;
    #20 $finish;
end
generate
if (p > 0)
begin
    cg cg_inst = new; //covergroup instantiated in generate block
    initial
        cg_inst.set_inst_name("inside_if");
end
endgenerate
endmodule
```

Note: When you define a covergroup in the `generate` block, then the instance name in the report includes the name of the `generate` block, as shown below:



If the `generate` block is not named explicitly, the tool automatically generates a name for the `generate` block, and displays it in the report (as shown in the above report). If the `generate` block is named explicitly, then that name is displayed in the report.

Unsupported scenarios for Covergroups in Generate Blocks

When defining covergroups in generate blocks, remember that:

- You cannot declare a covergroup in one generate block and instantiate it in another generate block.

```
module test;
...
generate
if(...)
begin : GENERATE_1
    covergroup cg @(posedge clk); //covergroup declaration
        A: coverpoint a;
    endgroup
    ...
endgenerate
...
generate
if(...)
begin : GENERATE_2
    cg cg_inst = new; //covergroup instantiation
    ...
endgenerate
endmodule
```

- You cannot instantiate a covergroup in one generate block and use that instance to call coverage methods in another generate block.

```
module test;
...
covergroup cg @(posedge clk); //covergroup declaration
    A: coverpoint a;
endgroup
generate
if(...)
begin : GENERATE_1
    cg cg_inst = new; //covergroup instantiation
    ...
endgenerate
...
generate
if(...)
begin : GENERATE_2
```

```
GENERATE_1.cg_inst.sample(); //coverage method called  
...  
endgenerate  
endmodule
```

Covergroups in Packages

System Verilog packages are used to share parameters, data, type, task, function, sequence, property and checker declarations across multiple modules, interfaces, programs, and checkers. A package can be defined using the `package` and `endpackage` keywords. You can declare and instantiate covergroups in packages. Consider the given example in which a covergroup is declared inside a package:

```
package tstpkg;  
    reg clk;  
    reg [4:0]p1;  
  
    covergroup cg @clk;  
        A:coverpoint p1{  
            bins x1 = {1};  
            bins x2 = {6};  
            bins x3[] = {[7:8]};  
        }  
    endgroup  
  
    class c;  
        covergroup cg @clk;  
            A:coverpoint p1{  
                bins x1 = {1};  
                bins x2 = {6};  
                bins x3[] = {[7:8]};  
            }  
        endgroup  
        function new;cg = new; cg.set_inst_name("embed_cg");endfunction  
    endclass  
endpackage  
  
module top;  
    import tstpkg::*;

    cg cgi = new;
    c cobj = new;
    initial
```

```

begin
#5 p1 = 6; clk = 0;
#5 p1 = 7; clk = 1;
#5 p1 = 8; clk = 0;
end
endmodule

```

In the given example, the covergroup, `cgi`, declared in a package, `testpkg`, is instantiated as `cgi` inside module `top`. In the given example, notice that the package, `testpkg`, also has an embedded covergroup.

For the given example, the instance-based report is shown below:

```

Instance name: top
Module/Entity/Interface/Program name: top
File name: /vobs/nc_test/test/vlog/coverage/covergroup/cg_pkg/tc/test1.v
Number of covered bins: 6 of 8

```

Coverage	Count	Name	Line	Origin
0.75 (75/100)	3/4	cgi	6	covergroup cg @clk;
0.75 (75/100)	3/4	cgi.A	7	A:coverpoint p1
0(1)		x1	9	bins x1 = {1};
1(1)		x2	10	bins x2 = {6};
1(1)		x3[7]	11	bins x3[] = {[7:8]};
1(1)		x3[8]	11	bins x3[] = {[7:8]};
0.75 (75/100)	3/4	embed_cg	16	covergroup cg @clk;
0.75 (75/100)	3/4	embed_cg.	17	A:coverpoint p1
0(1)		x1	19	bins x1 = {1};
1(1)		x2	20	bins x2 = {6};
1(1)		x3[7]	21	bins x3[] = {[7:8]};
1(1)		x3[8]	21	bins x3[] = {[7:8]};

Scope of Covergroup and Covergroup Instances

Covergroup instances created inside a module, interface, or program block remain alive throughout the simulation. As these instances are created once and remain alive throughout the simulation, these instances are known as static instances. The coverage information of these covergroup instances is dumped in the coverage database for post-process analysis. The coverage sampling of these instances remains enabled throughout the simulation.

Consider the following code:

```

module test;
logic clk=0;
bit [3:0] by;
covergroup CG @(posedge clk);

```

```

A: coverpoint by;
endgroup // CG
CG cg_mod = new(); // CG instance in module scope
initial begin
repeat(10) begin : B1
    CG cg_named; // CG instance in named block scope
    cg_named = new();
    cg_named.set_inst_name("named_1");
    @(negedge clk) by = $urandom;
end: B1
$finish;
end
initial begin
    cg_mod.set_inst_name("mod_1");
end
always #2 clk = ~clk;
endmodule // test
    
```

The coverage sampling for `cg_mod` remains enabled throughout the simulation. The instance `cg_named` is created ten times inside the block `B1`. Out of 10 instances nine got killed, and one is alive at the end of simulation. The following instance-based report demonstrates this.

Instance with module scope live throughout simulation run (static instance)

Number of covered bins: 9 of 32			
Coverage	Count	Name	Line Origin
0.50 (50/100)	8/16	mod_1	5 covergroup CG @(posedge clk);
0.50 (50/100)	8/16	mod_1.A	7 A: coverpoint by;
	2(1)	auto[0]	7 A: coverpoint by;
	1(1)	auto[1]	7 A: coverpoint by;
	1(1)	auto[4]	7 A: coverpoint by;
	1(1)	auto[5]	7 A: coverpoint by;
	1(1)	auto[9]	7 A: coverpoint by;
	1(1)	auto[11]	7 A: coverpoint by;
	1(1)	auto[12]	7 A: coverpoint by;
	2(1)	auto[13]	7 A: coverpoint by;
0.06 (6/100)	1/16	B1.named_1	5 covergroup CG @(posedge clk);
0.06 (6/100)	1/16	B1.named_1.A	7 A: coverpoint by;
	1(1)	auto[13]	7 A: coverpoint by;

Last instance in B1 live at simulation end (dynamic instance)

The instance within the named block is reported only once because only one instance was alive at simulation end. Currently, you cannot report coverage for destroyed instances.

Adding Covergroups in Verification Units

You can add covergroups to an existing design, without modifying the existing source code by writing them in a separate file, within a verification unit (vunit) and associate it with the relevant

portion of the design. This helps you experiment with covergroups before embedding them in the source file and also reuse covergroups from a previous design.

The separate file for writing verification code is referred to as a vunit file, which typically includes a `vunit` construct. The covergroups for verification code are written inside the `vunit` construct. The `vunit` file is passed along with the instrumented code to the simulator for generating functional coverage results. The code written in the `vunit` file is interpreted as part of the module and is inserted in the module, immediately before the end of the module. Using the `vunit` construct is based on designer's need and discretion. The scope of this manual is to describe different permutations and combinations in which designers can use a `vunit` construct.

Define a covergroup and create its instances in a `vunit` file, as:

```
vunit unit_top (top) {
    covergroup cg1 @ (negedge clk) ;
        coverpoint a;
    endgroup
    cg1 cg_inst = new();
}
```

In the above code, `unit_top` is the name of the `vunit` and `top` is the name of the module with which the covergroup statements mentioned in the `vunit` construct would be associated.

Define a covergroup in the design file and create its instances in the `vunit` file, as:

```
vunit unit_top (top) {
    cg1 cg_inst = new(); //only instances in the vunit file
}
```

Note: Covergroup definition in a `vunit` file and instantiation in the design file does not work because during compilation, the code written in the `vunit` file is included in the module, immediately before the end of the module. As a result, the statements related to covergroup instantiation will appear prior to covergroup definition. This will lead to an error.

Define a covergroup in one `vunit` file and create its instance in another `vunit` file, as:

Contents of vunit file 1

```
vunit unit_top1 (top) {
    covergroup cg1 @ (negedge clk) ;
        coverpoint a;
    endgroup
}
```

Contents of vunit file 2

```
vunit unit_top2 (top) {
cg1 cg_inst = new();
}
```

Both the vunits are associated with the module `top`. The vunit `unit_top1` includes the covergroup definition statements and `unit_top2` includes the covergroup instantiation statements.

Note: With multiple vunit files, ensure that during compilation, the vunit file containing covergroup definition statement is passed on the command-line prior to the vunit file containing instantiation statements. If the vunit file containing the instantiation statement is passed prior to the vunit file containing the covergroup definition, an error will occur. For more on the `vunit` construct, refer to the [Assertion Writing Guide](#).

When using a vunit file, use the following command to generate coverage data:

```
xrun -sv -propfile_vlog <vunit_file> <design_file>
```

In the above command,

- `propfile_vlog` indicates that the vunit file is being used.
- `<vunit_file>` specifies the name of the vunit file.
- `<design_file>` is the name of the file with which the vunit file needs to be associated.

Note: Code inside verification units is for verification purposes only (not part of the design). Therefore, code coverage items inside verification units are not scored, and are marked `IGN` (ignore) in reports. To stop dumping of code coverage items (inside the verification units), use the `-ignore_vunit_code_coverage` option with the [`set_code_fine_grained_merging`](#) command at elaboration.

Scoring Functional Coverage

To score functional coverage, you can either:

- Pass `-coverage functional` to `xmelab`.
- Use the `select_functional` command in the coverage configuration file and then pass this file using the `-covfile` option to `xmelab`.

See [Chapter 9, "Generating Coverage Data,"](#) for more details. After simulation has dumped functional coverage data, you analyze it using the reporting tool IMC. See the *Integrated Metrics*

Center User Guide in the Metric-Driven Verification (MDV) release for details on IMC.

Functional Coverage using Incisive Assertion Library

The Incisive Assertion Library (IAL) is collection of reusable verification components containing pre-defined assertions and coverage points. The IAL components (IAL-Cs) are designed for the functional verification of commonly used hardware structures. There are several ways you can use to add IAL components to add assertions and coverage points to a design. You can:

- Embed the IAL components directly in the HDL code.
- Place the IAL components in an external file (vunit) that is bound to a module.
- Include the IAL components in an external file in a standalone verification component called a monitor.

For detailed information on adding assertions using above mentioned ways, refer to the *Incisive Assertion Library Reference Guide*.

FSM Coverage

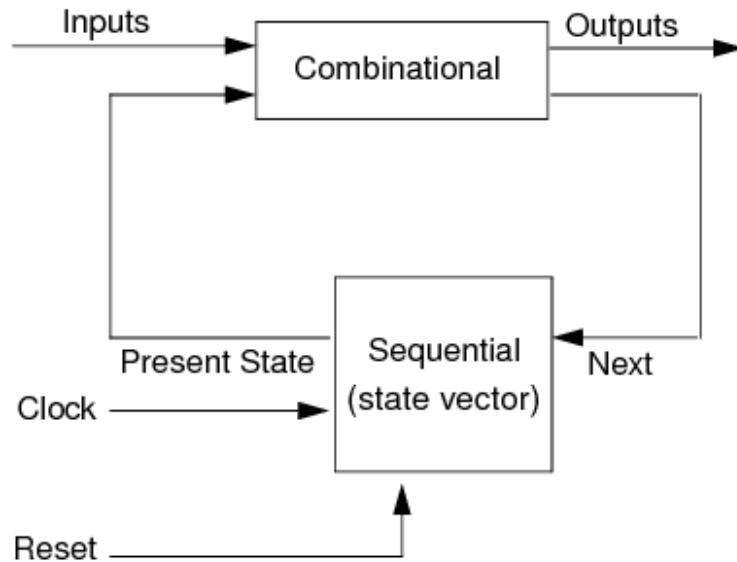
FSM coverage interprets the synthesis semantics of the HDL design and monitors the coverage of the FSM representation of control logic blocks in the design. It answers the question "Did I reach all the states and cover all possible transitions or arcs in a given state machine?"

This chapter covers the following topics:

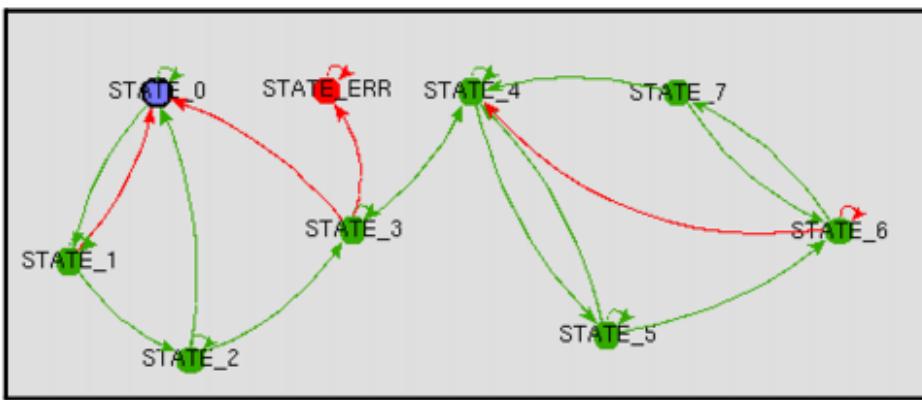
- [Basic FSM Model](#)
- [FSM Coverage Types](#)
 - [State Coverage](#)
 - [Arc Coverage](#)
- [Modeling Styles Supported](#)
 - [Two Process Modeling Style](#)
 - [Automatic FSM Extraction of Part-Select Signals](#)
 - [Present state or next state in LHS of a Continuous Assignment Statement](#)
 - [Rules for Specifying Pragmas](#)
 - [Limitations with Pragma-Based FSMs](#)
 - [Single Process Modeling Style](#)
 - [One-Hot Encoding Style](#)
- [FSM Extraction in ICC](#)
 - [Unsupported Scenarios for FSM Extraction](#)
 - [Unexpected FSM Results Scenarios](#)
- [Scoring FSM Coverage](#)

Basic FSM Model

An FSM consists of a combinational block that computes the next state for the FSM and a sequential block that preserves the present state (state vector) of the machine. The sequential block is a set of n flip-flops clocked by a single clock signal (hence synchronous state machine). The following figure displays a basic state machine structure.



Conceptually, FSM can be viewed as a state diagram that depicts a finite number of states and transitions between those states. The following figure displays a state diagram.



The diagram shows different states and transitions between these states. Using ICC, you can generate coverage reports for possible states in an FSM, transitions between these states, and conditions under which each transition occurs.

FSM Coverage Types

The following types of FSM coverage are available:

- State Coverage -- reports what states were visited
- Transition Coverage -- reports what transitions occurred
- Arc Coverage -- reports why each transition occurred

State Coverage

State coverage identifies all of the states in an FSM and reports on covered/uncovered states, the encoding value, the number of times each state is visited, and the number of times reset states are reached. The following figure displays a sample state coverage report.

All FSM Detail Report, Module/Entity-Based		
<hr/>		
Module/Entity name: spi		
State Register: present_state		
Number of covered states: 6 of 6		
Number of uncovered states: 0 of 6		
State Coverage :		
<hr/>		
State	Encoding	Num of Visits
State_1	001	45904
State_0	000	17
State_2	010	833
State_3	011	6375
State_7	111	17
<hr/>		
Reset State	Num of Resets	
State_1	1	

States

Visit count for each state

Reset state

A state is considered as a reset state if transition to that state is not dependent on the current state of the FSM. For example, in the following code, states `s0`, `s1`, and `s2` are inferred as reset states because transition to these states is independent of current state `present_state`.

```
case(instr)
I0: nxt_state = s0;
I1: nxt_state = s1;
I2: nxt_state = s2;
endcase
```

```
case(present_state)
```

```
...
```

See [Chapter 9, "Generating Coverage Data,"](#) for detailed information on coverage data generation.

 In IMC, source information is not displayed for a state that does not occur in any transition as a from-state.

Transition Coverage

Transition coverage identifies all of the transitions in an FSM, and reports on the number of covered/uncovered transitions/reset transitions, and the number of times each transition/reset transition occurred. The following figure displays a sample transition coverage report.

Module/Entity name: spi State Register: present_state Number of covered transitions: 6 of 11 Number of uncovered transitions: 5 of 11 Inputs = (-)			
Transition Coverage :			
P-State	N-State	Inputs	Num of Visits
State_1	State_0	-	17
State_0	State_1	-	0
State_2	State_2	-	17
State_3	State_3	-	17
State_7	State_1	-	17

P-State	Reset State	Num of Resets
State_1	State_1	1
State_0	State_1	0
State_2	State_1	0
State_3	State_1	0
State_7	State_1	0

See [Chapter 9, "Generating Coverage Data,"](#) for detailed information on coverage data generation.

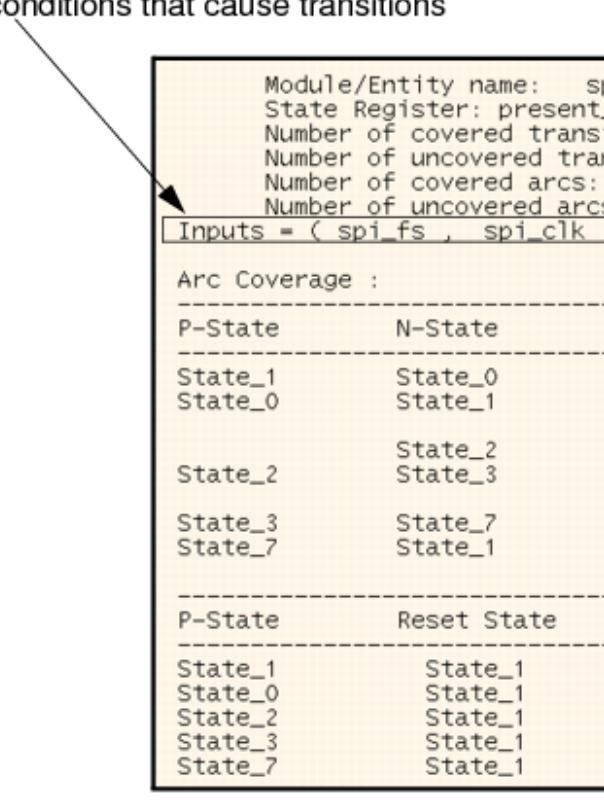
Arc Coverage

Arc coverage reports each transition with all possible input conditions under which the transition can take place. Each possible input condition is scored separately.

By default, arc coverage scoring is turned off. To enable arc coverage scoring, specify the

`set_fsm_arc_scoring` command in a coverage configuration file. The following figure displays a sample arc coverage report.

Input conditions that cause transitions



Module/Entity name: spi			
State Register: present_state			
Number of covered transitions: 6 of 11			
Number of uncovered transitions: 5 of 11			
Number of covered arcs: 6 of 13			
Number of uncovered arcs: 7 of 13			
Inputs = (spi_fs , spi_clk , not_full , full)			
Arc Coverage :			
P-State	N-State	Inputs	Num of Visits
State_1	State_0	10--	17
State_0	State_1	-1--	0
		0---	0
State_2	State_2	10--	17
	State_3	-1--	17
State_3	State_7	--0-	0
	State_1	-0-1	17
State_7			17
P-State	Reset State	Num of Resets	
State_1	State_1	1	
State_0	State_1	0	
State_2	State_1	0	
State_3	State_1	0	
State_7	State_1	0	

Input conditions reported as SOPs

Note: In Verilog, arcs are not extracted for FSMs that have the next state computation in a function or a task. However, transitions are extracted and reported for such FSMs.

Extraction of Inputs

- If an expression includes only logical operators, bit-wise operators, reduction operators, or concatenation operators, then each expression term is considered as a single input, as shown below.

Expression 1	(a && b) c
Inputs	a b c

Note: If a term is a vector, individual bits of the vector are listed as inputs.

- If an expression has a relational operator or an arithmetic operator, then the entire relational (sub) expression or arithmetic (sub) expression is considered as a single input, as shown

below.

Expression 1	a && b (c < d)
Inputs	a b c < d
Expression 2	((a b) < (c && d)) ((e && f) > (g h))
Inputs	((a b) < (c && d)) ((e && f) > (g h))
Expression 3	((a + b) && c) ((d + e) && f)
Inputs	(a+b) c (d+e) f

- If an expression has a function call, then the function call is considered as a separate input.

Modeling Styles Supported

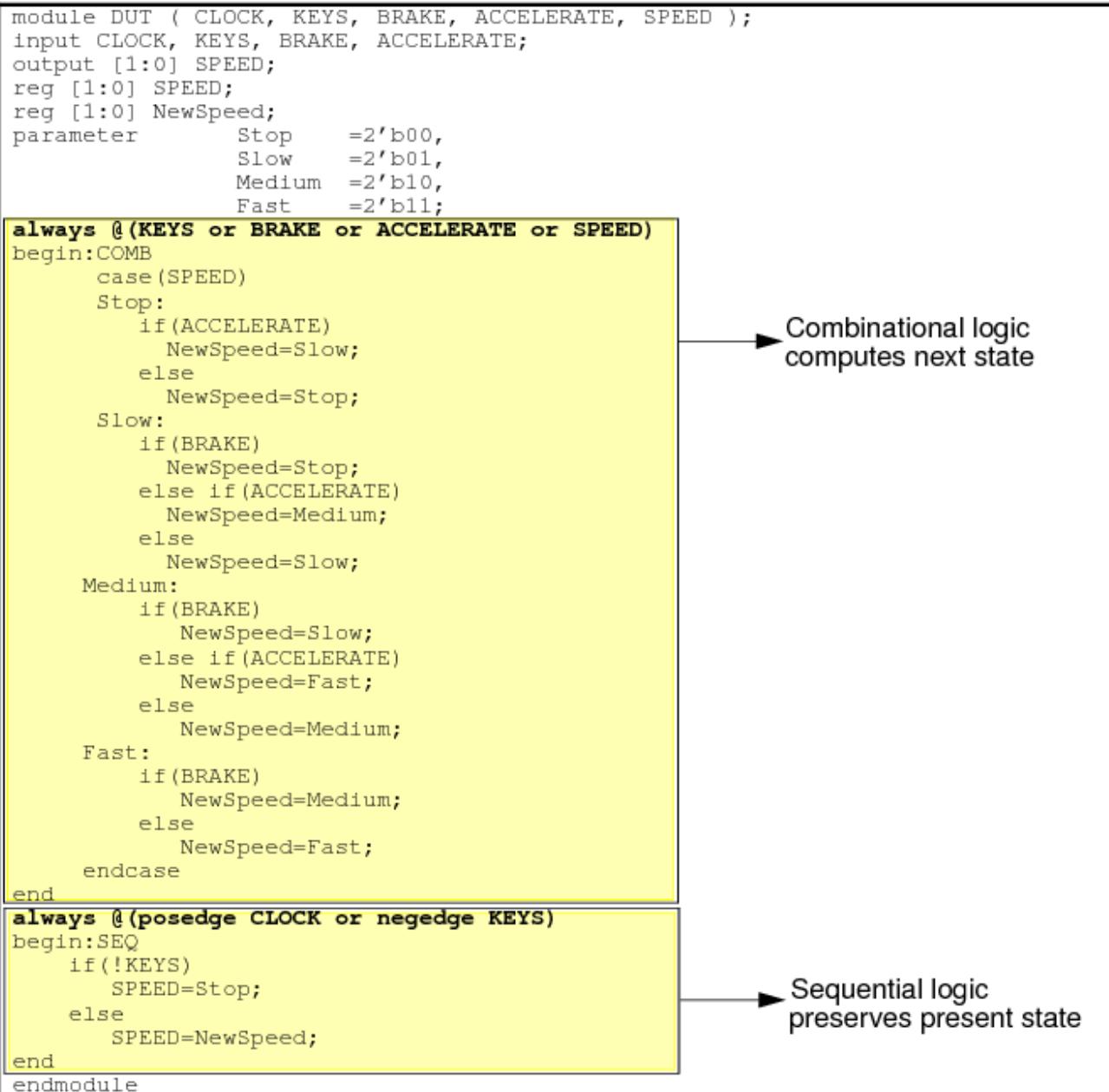
ICC extracts FSMs in a design if they are written in any of the following styles and not subject to any of the documented limitations:

- Two Process Modeling Style
- Single Process Modeling Style
- One-Hot Encoding Style

Two Process Modeling Style

In a two process modeling style, the combinational logic and sequential logic are specified as separate processes. The combinational logic block computes the next state for the FSM and is sensitive to all signals (including the state register) that are read in this block. The sequential logic block preserves the present state and is sensitive to the clock edge event and all asynchronous resets (if any).

The following figure displays a two process FSM.



Automatic FSM Extraction of Part-Select Signals

In two process modeling for Verilog designs FSM, coverage is supported for a state-register that are part-select of a signal. For such part-select state registers, following two templates are supported:

Styles Supported:

Example 1:

Consider the given example, in which different combinational blocks for various part-select of the signal form state registers, and a single sequential block containing a `ps=ns` statement is included. In this example, `ps[0:1]` and `ps[2:3]` state registers are identified.

```
reg [0:3] ps;
    reg [0:3] ns;
    parameter [0:1] S0 = 2'b00,
                  S1 = 2'b01,
                  S2 = 2'b10;

    always @ (posedge clk or posedge rst)
    begin
        if (rst == 1'b1)
            ps = 4'b0000;
        else
            ps = ns;
    end
    always @ (ps[0:1])
    begin
        case (ps[0:1])
            S0:
                ns[0:1] = S1;
            ....
    end
    always @ (ps[2:3])
    begin
        case (ps[2:3])
            S0:
                ns[2:3] = S1;
            ....
    end
```

Example 2:

Consider the following example, in which `ps[0:1]` and `ps[2:3]` state registers are identified.

```
reg [0:5] pst;
reg [0:1] nst1;
reg [0:1] nst2;

parameter [0:1] S0 = 2'b00,
            S1 = 2'b01,
            S2 = 2'b10,
            S3 = 2'b11;
always @ (posedge clk or posedge rst)
begin
    if (rst == 1'b1) begin
        pst[0:1] = S0;
        pst[2:3] = S1;
    end
    else
        pst[0:1] = nst1;
        pst[2:3] = nst2;
end
always @ (x or pst[0:1])
begin
    case (pst[0:1])
        S0: begin
            if (z == 1)
                nst1 = S1;
        ....
    end
    always @ (pst[2:3])
begin
    case (pst[2:3])
        S0: begin
            if (z == 1)
                nst2 = S2;
        ....
    end
end
```

Example 3:

Consider the given example with multiple sequential and combinational blocks for different part-select of the signal forming state registers.

```
reg [0:7] ps;
    reg [0:7] ns;
    parameter [0:1] S0 = 2'b00,
                  S1 = 2'b01,
                  S2 = 2'b10;

    always @ (posedge clk or posedge rst)
    begin
        if (rst == 1'b1)
            ps[0:3] = 4'b0000;
        else
            ps[0:3] = ns[0:3];
    end
    always @ (posedge clk or posedge rst)
    begin
        if (rst == 1'b1)
            ps[4:5] = 2'b00;
        else
            ps[4:5] = ns[4:5];
    end

    always @ (ps[0:1])
    begin
        case (ps[0:1])
            S0:
                ns[0:1] = S1;
            ...
    end
    always @ (ps[2:3])
    begin
        case (ps[2:3])
            S0:
                ns[2:3] = S1;
            ...
    end
    always @ (ps[4:5])
    begin
        case (ps[4:5])
            S0:
                ns[4:5] = S1;
            ...
    end

```



- Part-select of a signal can be identified as a state register only for two-process FSM style.
- FSM Extraction of part-select state registers is not supported for one-hot encoding style.
- Different FSMs modeled in a single state register must not have overlapping bits. For

example, `ps[0:2] = ns1` and `ps[2:4] = ns2` are invalid.

- There must be only one next case computation logic in one combinational block. The given code is invalid and FSM will not be recognized for state registers `ps[0:1]` and `ps[2:3]`.

```
always @ (ps)
begin:comboblk1
  case (ps[0:1])
    S0:
      ns [0:1] = S1;
    ....
    case (ps[2:3])
      S0:
        ns [2:3] = S1;
  end:comboblk1
```

- If next-state computation logic for part-select state register is defined in a function/task, the current state should be defines as a full-state vector in a function/task. Consider the given example:

```
assign NSTATE[2:0] = fncNSTATE(PSTATE[2:0]);

always @ (posedge clk or posedge rst)
begin
  if (rst == 1'b1)
    PSTATE[3:0] = 4'b0000;
  else
    PSTATE[3:0] = NSTATE[3:0];
end

function reg[2:0] fncNSTATE (input reg [2:0] PSTATE);
begin
  case(PSTATE)      /*Present State (PSTATE) is used as a full-state vector inside
function*/
    2'b00: begin
      if(RED_t1) begin
        t1 = RedState;
      end
    ....
  end
end
```

Present state or next state in LHS of a Continuous Assignment Statement

In the given example, a single continuous assign statement, where Current State is assigned to another register, such that Current State acts as buffer, and a single combinational block, where Case/If statement uses that state register being written in the continuous assign statement. In this example, the LHS of the Continuous Assignment Statement is assigned a temporary variable Speed_temp.

```
always @*
  case (Speed)
    stop:
      NewSpeed = slow;
    slow:
      NewSpeed = stop;
    ...
    assign Speed = Speed_temp;

  always @ (posedge clk or posedge rst)
    if (rst)
      Speed_temp <= 1'b0;
    else
      Speed_temp <= NewSpeed;
```

Pragma-Based FSM Extraction

In two process modeling, use pragmas to identify the FSMs that are not extracted automatically. Pragma-based FSM extraction is applicable only for Verilog. An FSM can be identified through pragma for a state-vector declared inside a module scope. Currently, this pragma support has been added for two process FSM with sequential process in a sub-module. Any additional CCF command is not required to enable this support. The syntax of the pragma is:

```
/*pragma state_vector cs_signal_name next_state_vector ns_signal name [enum
enumeration_name]
```

where,

- `pragma`, `state_vector`, `next_state_vector`, and `enum` are the keywords.
- `cs_signal_name` is the name of a current-state vector of the FSM.
- `ns_signal_name` is the name of the next-state vector of the FSM associated with the `current-state`.

- The signal `cs_signal_name` must be declared in the same module scope as the pragma. It should be an identifier that can only be:
 - a wire, reg, logic, or bit
 - vector of wire, reg, logic, or bit
 - byte, shortint, int, longint, integer, or an enum
- The signal `ns_signal_name` should be an identifier. Its declaration should be in the same scope as `cs_signal_name` declaration. It can only be:
 - reg, logic, or bit
 - vector of reg, logic, or bit
 - byte, shortint, int, longint, integer, or an enum



The `next_state_vector` should be directly driven through a single combinational always block.

It is mandatory to specify `next_state_vector`.

- `enumeration_name` is the unique name to identify a Pragma identified FSM. It will be used for reporting. Specifying an `enumeration_name` is optional. If `enumeration_name` is not specified, then the FSM is reported by `cs_signal_name`.

Consider the given example of a two-process FSM, in which a sequential process is included in a sub-module:

```
module fsm_submod (clk, rst, x);
/* pragma state_vector cs enum myFSM */

    input clk, rst;
    input x;

    parameter SIZE = 2;
    parameter [1:0] S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

    reg [SIZE-1:0] cs;
    reg [SIZE-1:0] ns;

    always @ (cs, s_rst, x)
    begin : FSM_COMBO

        if(en && s_rst)
            ns = S0;
```

```

        case(cs)
S0 : if(x) ns = S1;
      S1 : if(x) ns = S2; else ns = S1;
S2 : ns = S3;
default : ns = S0; // Explicit Transition S3->S0
endcase
end

//-----Seq Logic-----
FF #( .width(SIZE), .default_state_value(S0))
  u_FF(clk, rst, ns, cs);

endmodule

//-----Separate FF MODULE-----
module FF(clk, rst, state_in, state_out);
    parameter width = 1;
    parameter [width-1:0] default_state_value = 'b0;

    input clk;
    input rst;
    input [width-1:0] state_in ;
    output [width-1:0] state_out;

    reg [width-1:0] tmp ;

    assign state_out = tmp ;

    always@(posedge clk or posedge rst)
        begin : FSM_SEQ
            if (rst == 1'b1) begin
                tmp <= default_state_value ;
            end else begin
                tmp <= state_in ;
            end
        end
    end
endmodule

```

In the above example, the states that are reported are S0, S1, S2, and S3. The explicit transitions reported are

S0->S1
S1->S1, S2
S2->S3
S3->S0

Notice that among these states S1->S1 is a hold transition state, and will be reported only when the [set_fsm_scoring -hold_transition](#) command is specified.

Rules for Specifying Pragmas

- Single-line as well as multi-line comments are supported.
- Include pragma specification directly inside the module scope. Specification of pragma outside the module scope is ignored.
- Specify pragma in the same module as the state_vector and next_state_vector.
Consider the following examples for a valid usage of pragmas:

Example:Pragma specified before declaration of state_vector “cs”:

```
/* pragma state_vector signal_name next_state_vector ns */  
logic [1:0] cs;
```

Example:Pragma specified after declaration of state_vector “cs”:

```
logic [1:0] cs;  
/* pragma state_vector signal_name next_state_vector ns */
```

- If a pragma that specifies a state_vector, a next_state_vector, or an enumeration_name already used in an earlier pragma, then the latter pragma is ignored with a warning. For example, for the following specifications, the second pragma will be ignored:

```
/* pragma state_vector cs next_state_vector ns enum myFSM */  
  
/* pragma state_vector cs next_state_vector ns */.
```



- The state_vector should be directly driven from the sub-module and not through any intermediate combinational logic or buffers.
- state_vector should be an identifier, and it cannot be a bit-select, part-select, concatenation, struct, or its member.
- next_state_vector should be an identifier, and it cannot be a bit-select, part-select, concatenation, struct, or its member.
- An automatic FSM identification is not performed for the state_vector specified in Pragma.

Limitations with Pragma-Based FSMs

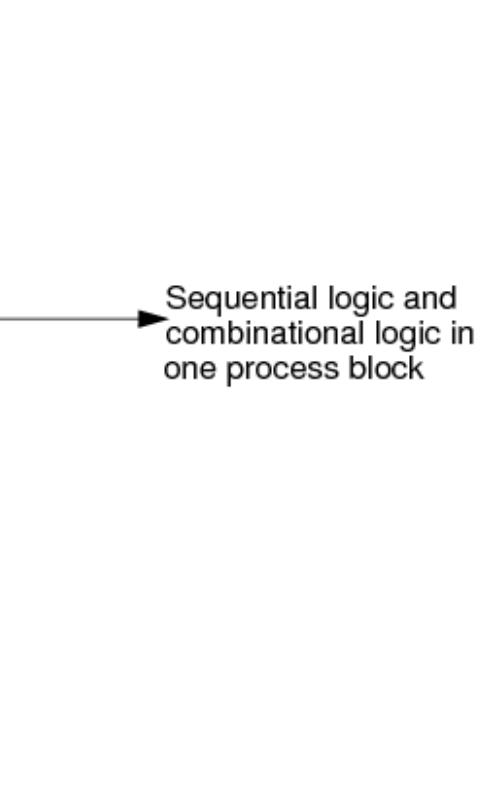
- Reset transition coverage is not supported.
- FSM Arc coverage is not supported.
- FSM tagging for selection/deselection through CCF is not supported.

Single Process Modeling Style

In a single process modeling style, the entire logic is described in a single process sensitive to the edge of clock and asynchronous resets, as shown below.

Figure 8-1 Single Process FSM

```
module DUT ( CLOCK, KEYS, BRAKE, ACCELERATE, SPEED );
  input CLOCK, KEYS, BRAKE, ACCELERATE;
  output [1:0] SPEED;
  typedef enum bit[1:0] {Stop=2'b00, Slow=2'b01, Medium=2'b10, Fast=2'b11} states;
  states SPEED;
  always @ (posedge CLOCK or negedge KEYS)
  begin
    if (!KEYS)
      SPEED=Stop;
    else
      case (SPEED)
        Stop:
          if (ACCELERATE)
            SPEED=Slow;
          else
            SPEED=Stop;
        Slow:
          if (BRAKE)
            SPEED=Stop;
          else if (ACCELERATE)
            SPEED=Medium;
          else
            SPEED=Slow;
        Medium:
          if (BRAKE)
            SPEED=Slow;
          else if (ACCELERATE)
            SPEED=Fast;
          else
            SPEED=Medium;
        Fast:
          if (BRAKE)
            SPEED=Medium;
          else
            SPEED=Fast;
      endcase
  end
endmodule
```



Sequential logic and combinational logic in one process block

One-Hot Encoding Style

In one-hot encoded style, states are defined using parameters or literals. This style requires a flip-flop for each state in the design and only one flip-flop (representing the current state) is set at a time, as shown below.

Figure 8-2 One-Hot FSM

```

module example (a, rst, clock);
input a, rst, clock;
parameter [1:0] S0=2'b00,
    S1=2'b01,
    S2=2'b10;
reg [2:0] present_state, next_state;
always @ (posedge clock or posedge rst)
begin
    if (rst)
        begin
            present_state = 0;
            present_state[S0] = 1'b1;
        end
    else
        present_state = next_state;
end
always @ (present_state or a)
begin
    next_state = 0;
    case (1'b1)
        present_state[S0]:
            begin
                if (a == 1'b1)
                    next_state[S1] = 1'b1;
                else
                    next_state[S0] = 1'b1;
            end
        present_state[S1]:
            begin
                casex (a)
                    1'b1:
                        begin
                            next_state[S2] = 1'b1;
                        end
                    default:
                        begin
                            next_state[S0] = 1'b1;
                        end
                endcase
            end
        present_state[S2]:
            begin
                if (a == 1'b1)
                    next_state[S2] = 1'b1;
                if (a == 1'b0)
                    next_state[S0] = 1'b1;
            end
    endcase
end
endmodule

```

Index used instead
of state encoding

- i If a state register has assignment statements to unknowns (X's), then the unknowns (X's) are ignored and FSM is extracted for the state register.

FSM Extraction in ICC

FSMs for following `if/if-else/case` models are extracted:

- Top-level `case` construct

```
case (present_state)
    State0 :
    State1 :
    State2 :
    ...
    ...
```

- Primary `if-else` branch

```
if (present_state == State0)
    ...
else if (present_state == State1)
    ...
else if (present_state == State2)
    ...
...
...
```

- `case` statement in the final `else` branch

```
if (present_state == State0)
    ...
else if (present_state == State1)
    ...
else case (present_state)
    ...
...
...
```

Internal state names are generated in the format `State_<state_value>` if you:

- Use constant literals as states, as shown below.

```

typedef enum bit [1:0] {Stop=2'b00, Slow=2'b01, Medium=2'b10, Fast=2'b11}states;
states SPEED;
...
case(SPEED)
  Stop:
  ...
  Slow:
  ...
  2'b10: ──────────► State name generated as State_2
  2'b11: ──────────► State name generated as State_3

```

- Use `define instead of parameters/enums to define states, as shown below.

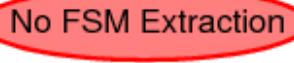
<code>`define Stop 2'b00</code>	► State name generated as State_0
<code>`define Slow 2'b01</code>	► State name generated as State_1
<code>`define Medium 2'b10</code>	► State name generated as State_2
<code>`define Fast 2'b11</code>	► State name generated as State_3

States and transitions are extracted for constant tags only. Variable tags are ignored for FSM extraction, as shown below.

```

module top ( y, x, CLOCK, KEYS, BRAKE, ACCELERATE, SPEED );
  input CLOCK, KEYS, BRAKE, ACCELERATE;
  output [1:0] SPEED;
  parameter Stop    =2'b00,
            Slow     =2'b01,
            Medium   =2'b10,
            Fast     =2'b11;
  case(SPEED)
    x:
    y:
    Medium:
    Fast:

```



Transitions are extracted for constant assignments to state registers. Variable assignments to state registers are ignored for extraction, as shown below.

```

wire [1:0] var;
case(SPEED)
  Stop:
    begin
      if (BRAKE)
        NewSpeed=var; ──────────► Transition Ignored (Stop → var)
      else if (ACCELERATE)
        NewSpeed=Slow; ──────────► Transition Reported (Stop → Slow)
    end

```

Single-bit FSMs are extracted if explicit state tags of value 0 or 1 are found and/or explicit

transitions from state 0 to state 1 and from state 1 to state 0 are found, as shown below.

```

always @(posedge CLK or negedge RST)
begin
    if (RST == 1'b0)
        begin
            i_count <= 1'b0 ;
        end
    else
        begin
            case (i_count)
                1'b0: i_count <= 1'b1;
                1'b1: i_count <= 1'b0;
            endcase
        end
end

```

In a VHDL design, FSMs are extracted only for locally static state expression assignments to the state register.

```

-- Start VHDL subcode
constant C1: std_logic_vector(1 downto 0) := (others => '0'); -- globally static
constant C2: std_logic_vector(1 downto 0) := "00"; -- locally static
signal S1: std_logic_vector(1 downto 0) := "00"; -- not static
...
L1 : process
begin
    ...
    current_state <= C1; → Ignored because C1 is globally static
    ...
    current_state <= C2; → Considered because C2 is locally static
    ...
    current_state <= "00"; → Considered because 00 is locally static
    ...
    current_state <= S1; → Ignored because no locally static assignment to S1
    ...
end process;
-- End VHDL subcode

```

FSM are extracted for VHDL states of only following discrete types or discrete arrays:

- std_ulegic
- std_ulegic_vector
- std_logic
- std_logic_vector
- unsigned
- signed

- bit
- bit_vector
- boolean
- integer
- enumerated
- character

Unsupported Scenarios for FSM Extraction

- FSM extraction is not supported if FSM is coded in multiple sequential `if/else if/else` blocks, that check for module input signals instead of present state, as shown below.

```

module DUT ( KEYS, BRAKE, ACCELERATE, SPEED );
  input KEYS, BRAKE, ACCELERATE;
  output [1:0] SPEED;
  reg [2:0] SPEED;
  parameter
    Stop=2'b00,
    Slow=2'b01,
    Medium=2'b10,
    Fast=2'b11;
  wire CLOCK;
  always @ (posedge CLOCK or negedge KEYS)
    begin:FSM1
      if (!KEYS)
        SPEED=Stop;
      else if (ACCELERATE)
        case(SPEED) // synopsys full_case
          Stop:SPEED = Slow;
          Slow:SPEED = Medium;
          Medium:SPEED = Fast;
          Fast:SPEED = Fast;
        endcase
      else if (BRAKE)
        case(SPEED) // synopsys full_case
          Stop:SPEED = Stop;
          Slow:SPEED = Stop;
          Medium: SPEED = Slow;
          Fast:SPEED = Medium;
        endcase
      else
        SPEED = SPEED;
    end
endmodule

```

FSM inside multiple
if / else if blocks

No FSM Extraction

- Bit select is supported only for one-hot encoding style. In other encoding styles, FSM

extraction does not happen for a Verilog state variable if the present state or the next state is assigned as bit select.

- If the FSM description for a state register differs across instances of a module, then the FSM extraction is not done for that state register in instances where the description differs from the first instance. This can occur when parameters are used to determine state values, and the module is instantiated with different parameter values.
- FSM extraction does not happen for a VHDL state variable if present state or next state is an element of a record or a multi-dimensional array, or a slice or index of a vector.
- FSM extraction with a clock or a reset that is an element of a record is not supported.
- FSM extraction does not happen for a VHDL state variable if next state computation logic is within a function or a procedure.

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
entity rbguisrc is
end entity rbguisrc;
architecture behaviour of rbguisrc is
function NEXT_GRAYCODE (A: in STD_ULOGIC_VECTOR(3 downto 0)) return
STD_ULOGIC_VECTOR is
variable Yv: STD_ULOGIC_VECTOR(3 downto 0);
begin
    case A is
        when "0000" =>
            Yv := "0001";
        when "0001" =>
            Yv := "0011";
        when "0011" =>
            ...
            ...
            ...
        when "1001" =>
            Yv := "1000";
        when "1000" =>
            Yv := "0000";
        when others =>
            Yv := "XXXX";
    end case;
    return (Yv);
end NEXT_GRAYCODE;
signal MGCWRPTR : STD_ULOGIC_VECTOR(3 downto 0);
signal NEXT_MGCWRPTR : STD_ULOGIC_VECTOR(3 downto 0);
signal iMCLK : BIT;
begin
    NEXT_MGCWRPTR <= NEXT_GRAYCODE (MGCWRPTR);
    uMFLOPS: process
    begin
        wait until iMCLK'event and iMCLK = '1';
        MGCWRPTR <= NEXT_MGCWRPTR;
    end process;
end behaviour;

```

→ Next state computation

No FSM extraction

```

process(clk)
procedure compute_next_state is
begin
    case ps is
        when "00" => ps <= "01";
        when "01" => ps <= "10";
        when "10" => ps <= "11";
        when "11" => ps <= "00";
    end case;
end procedure compute_next_state;
begin
    if(clk'event and clk='1') then
        compute_next_state;
    endif;
end process;

```

→ Next state computation

No FSM extraction

Unexpected FSM Results Scenarios

FSM coverage results reported might be unexpected in the following scenarios:

- As with any race condition in Verilog, you might observe unexpected FSM results if input signals change at the same edge/phase as the state register. Consider the example of an FSM given below:

```
module fsm1(clk, rst, in1, in2);  
  
    input clk, rst, in1, in2;  
    reg [1:0] curr, next;  
  
    always @(posedge clk) begin  
        if(rst) curr <= 2'b00;  
        else curr <= next;  
    end  
  
    always @(curr, in1, in2)  
        case (curr)  
            2'b00 : if(in1) next = 2'b01; else next = 2'b00;  
            2'b01 : if(in2) next = 2'b10; else next = 2'b01;  
            2'b10 : if(in1) next = 2'b11; else next = 2'b01;  
            2'b11 : if(in2) next = 2'b00; else next = 2'b11;  
        endcase  
    endmodule
```

In the above FSM, the state register, `curr` updates at the posedge of the signal `clk`. If the following testbench is given to the above FSM, incorrect coverage results are generated:

```
module test;  
reg clk, rst, in1, in2;  
initial begin  
    clk = 1'b0; rst = 1'b1; in1 = 1'b0; in2 = 1'b1;  
    #10 rst = 1'b1;
```

```
#100 $finish;  
  
end  
  
always #5 clk = ~clk;  
  
  
always @(posedge clk) begin  
    in1 = ~in1;  
    in2 = .....;  
end  
  
endmodule
```

In the above testbench, input signals `in1` and `in2` and the state register `curr` change at the posedge of `clk`. In this case, a race begins between `clk`, `in1`, and `in2` in the generated FSM model, because the generated FSM model requires input signals to change at a different phase/time from the clock. As a result, it is recommended that input signals should not change at the same phases/times as the clock signal.

- Differences between simulation results and FSM coverage can occur due to the presence of delays in state assignment statements, as shown below:

```
always @(posedge clk or posedge rst)  
begin  
    if (rst)  
        ps <= #DEL s0;  
    else  
        ps <= #DEL ns;
```

FSM coverage records the value of the state register upon clock edges only. Delays in state assignment statement, if any, are ignored.

Note: If it is certain that the delays have no impact on the arc coverage results, you can disable checking for delays using the [set_fsm_arc_scoring -no_delay_check](#) command.

- You might observe a difference in number of visits and some transition arcs not being covered when `set_fsm_arc_scoring` is enabled. This is because some arcs for a transition cannot be shown when an input's value is `x`. Consider the following code.

```
case(ps)  
2'b00:
```

```

if(in1 == 1'b1)
  ps <= 2'b01;
else
  ps <= 2'b10;

```

In this case, transition from `State_0` to `State_2` can occur even if `in1` is `1'bX`. However, when arc coverage is enabled, this transition (`State_0` to `State_2`) is considered uncovered because the input's value is `x` and the input value to be matched is `0`, as shown below.

Arc coverage not enabled

Transition Coverage :					
P-State	N-State	Inputs	Num of Visits		
State_0	State_1	-	<table border="1" style="display: inline-table;"><tr><td>0</td></tr><tr><td>2</td></tr></table>	0	2
0					
2					
	State_2	-	<table border="1" style="display: inline-table;"><tr><td>2</td></tr></table>	2	
2					

Covered

Arc coverage enabled

Arc Coverage :							
P-State	N-State	Inputs	Num of Visits				
State_0	State_1	<table border="1" style="display: inline-table;"><tr><td>1</td></tr><tr><td>0</td></tr></table>	1	0	<table border="1" style="display: inline-table;"><tr><td>0</td></tr><tr><td>0</td></tr></table>	0	0
1							
0							
0							
0							
	State_2	<table border="1" style="display: inline-table;"><tr><td>0</td></tr></table>	0	<table border="1" style="display: inline-table;"><tr><td>0</td></tr></table>	0		
0							
0							

Input value to match

Uncovered

Scoring FSM Coverage

To score FSM coverage, you can either:

- Pass `-coverage fsm` to `xmelab`.
- Use the `select_fsm` command in the coverage configuration file and then pass this file using the `-covfile` option to `xmelab`.

See [Chapter 9, "Generating Coverage Data,"](#) for more details.

After simulation has dumped FSM coverage data, you analyze it using the reporting tool IMC. For details on coverage data analysis with IMC, see the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

In one-hot FSM, if a state register has assignment statements to unknowns (X's), then FSM is not extracted for the state register.

Consider the given example:

```
case (1'b1)
  present_state[0]:
    begin
      if(rst && in)
        next_state[1] = 1'b1;
      else
        next_state[2] = 1'b1;
    end
  .....
  default:
    next_state = 3'hX;
```

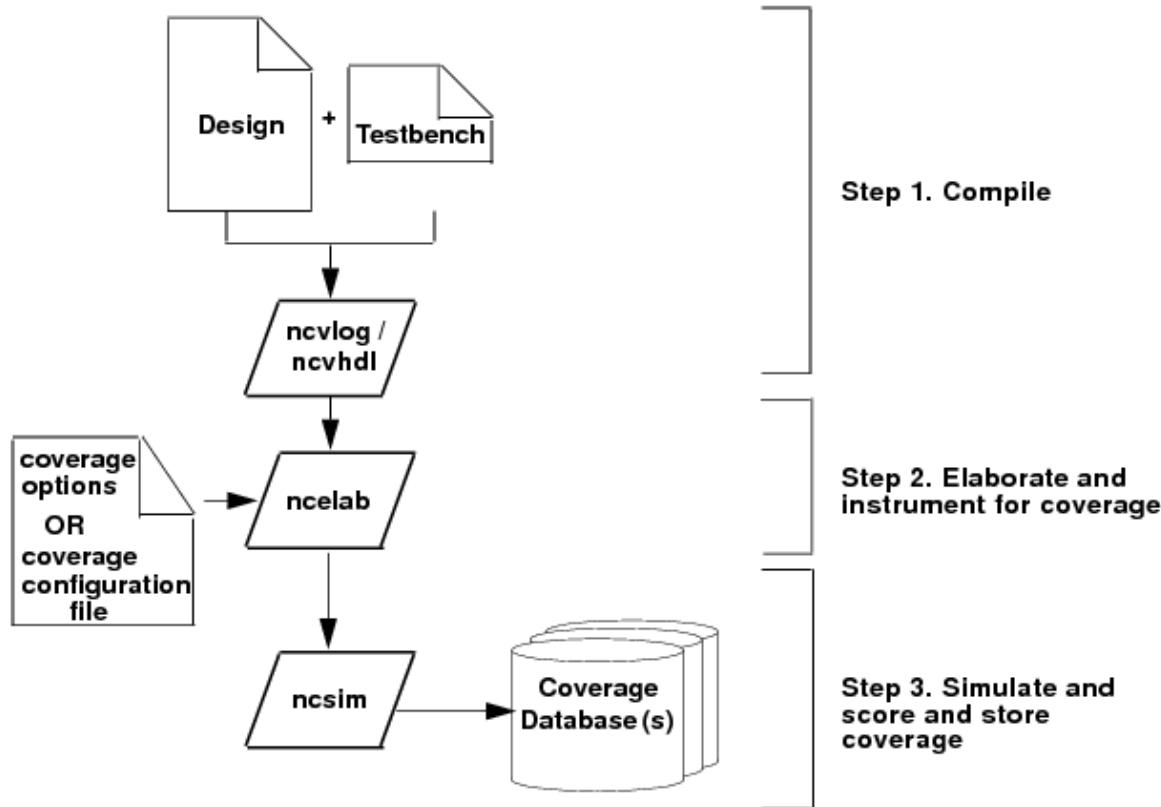
In the given example, the FSM is not extracted for the state register `present_state` as the `next_state` is assigned as unknown (`3'hX`).

Generating Coverage Data

This chapter explains how to generate coverage data using multi-step simulation and single-step simulation.

Generating Coverage Data using Multi-Step Simulation

The following diagram illustrates the multi-step process of generating coverage data.



As shown in the diagram, generating coverage data using multi-step simulation involves:

- **Compiling the Design** (`xmvlog/xmvhdl/xmsc/xmsc_run/xrun`)

- Elaborating the Design (`xmelab`)
- Simulating the Design (`xmsim`)

Compiling the Design

To compile the design, use:

```
% xmvllog [compilation_options] source_file [source_file]...
```

or

```
% xmvhdl [compilation_options] source_file [source_file]...
```

For details, see the *Xcelium Multi Core Simulator User Guide*.

Elaborating the Design

At the elaboration stage, you specify the coverage to be instrumented and scored and the scope of instrumentation. To elaborate the design to collect coverage data, use:

```
% xmelab [-coverage <coverage_types> | -covfile <coverage_configuration_file>]
          [-covdut <DUT_module>] [cov_cgsample] [other_elaboration_options]
[lib.]cell[:view]
```

The coverage options while elaborating the design are:

- `-coverage <coverage_types>`
- `-covfile <coverage_configuration_file>`
- `-covdut <DUT_module>`
- `-cov_cgsample`

Note: Coverage can also be enabled in the Multi-Snapshot Incremental Elaboration (MSIE) flow. For more details, see [Coverage in Multi-Snapshot Incremental Elaboration Flow](#).

-coverage <coverage_types>

Enables coverage data generation for all of the compiled modules. `<coverage_types>` can be:

- **B lock** - For enabling block coverage

- **E**xpr - For enabling expression coverage
- **F**sm - For enabling fsm coverage
- **T**oggle - For enabling toggle coverage
- **fU**nctional - For enabling functional coverage
- **A**ll - For enabling all supported coverage types

You can specify more than one coverage type by separating the coverage types with a colon (:). The argument(s) to the `-coverage` option are not case-sensitive. Substrings or single character names (marked in bold) of coverage types are also supported.

Note: Expression coverage by default is scored only for Verilog logical operators (|| and &&) and VHDL logical operators (OR, AND, NOR, and NAND), and is scored only in condition expressions. To score expression coverage for other operators, and for expressions in assignment statements, specify the [set_expr_scoring -all](#) command in the coverage configuration file.

-covfile <coverage_configuration_file>

Using this option, you pass xmelab a configuration file to control instrumentation. You can either include all of the commands in one configuration file or create separate configuration files for each type of coverage. For separate configuration files, you specify multiple `-covfile` options and the contents of different configuration files are combined before the instrumentation is actually done.

For a list of coverage configuration commands, refer to Chapter B, "[Coverage Configuration File \(CCF\) Commands](#)".

Selecting Coverage Using `-coverage` and `-covdut` and `-covfile` options

If you select coverage using both `-coverage` and `-covdut` and `-covfile` options, then take special care to understand how coverages are added as the command-line switches are processed and then selection is modified using commands in the CCF. Typically, with a coverage configuration file, you start with nothing and add desired instrumentation. Adding `-coverage` overrides this and instruments everything, and the coverage configuration file instead has to remove instrumentation. Mixing the use of `-coverage` and a coverage configuration file can be instead confusing. By keeping all control in only one place, you can be sure what is instrumented.

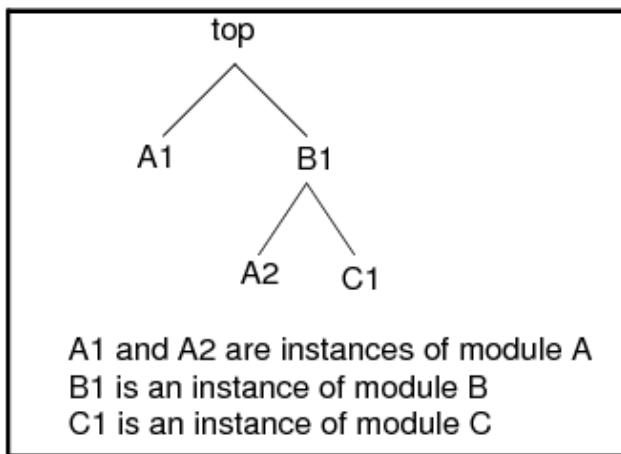
Use of `-coverage` together with setting basic options in CCF is okay. You should use the CCF to control instrumentation, when:

- The DUT should not be instrumented for the same coverage types.

- Additional coverage options are desired, such as `set_branch_scoring`.
- Less than default coverage is desired, such as `set_implicit_block_scoring`.
- Specific scoring options are desired to control hit limits and glitches.

-covdut <DUT_module>

Limits instrumentation as well as coverage database storing of selected coverage to an instance of `<DUT_module>` and its sub hierarchy. When using the `-covdut` option, it is important that coverage is enabled using `-coverage` or `-covfile` options. In the absence of these options, a warning is reported to indicate that even if `-covdut` is specified, it will have no effect. The `-covdut` option can appear more than once on the command line. Consider the following design hierarchy:



With this hierarchy, if you specify the `<DUT_module>` as `B`, coverage instrumentation is limited to `B1`, `A2`, and `C1`. In the absence of `-covdut`, by default, the top-level DUT, which in this case is `top` is considered for instrumentation.

When specifying a DUT, ensure that the DUT has a single top-level instance in the design. If you specify the DUT as `A`, an error is reported because `A` has multiple top-level instances. To avoid this:

- Specify the DUT as `top` or
- Specify both `A` and `B` as the DUT

The DUT specified with the `-covdut` option is considered the absolute DUT for coverage data generation. If you select/deselect any module-instance outside the DUT, using the [\(de\)select coverage](#) command, it is ignored. With the design hierarchy above, if you use:

```
% xmelab -covdut B -covfile cov.ccf worklib.top:v
```

and the coverage configuration file includes the following command

```
select_coverage -block -module A
```

then a warning is reported because instance `A1` of module `A` is outside the DUT specified using the `-covdut` option.

A warning is not generated if functional coverage is enabled using the `select_functional` command in the CCF file. However, the coverage will apply only to the DUT specified using the `-covdut` option.

-cov_cgsample

When the functional coverage is not enabled, the covergroups are not sampled. As a result, covergroup built-in

methods `get_coverage()`, `get_inst_coverage()`, `get_hit_count` and `get_inst_hit_count()` return zero coverage. The `-cov_cgsample` elaboration time option returns the correct coverage for these methods by enabling sampling those covergroups for which these methods are used.

Coverage in Multi-Snapshot Incremental Elaboration Flow

Coverage can also be enabled in the Multi-Snapshot Incremental Elaboration (MSIE) flow. In the MSIE flow, the design is partitioned into two sections: one that contains the large, stable part of the design, and one that contains the part of the design that is changing. The two sections are elaborated separately into two snapshots called the primary snapshot and the incremental snapshot.

The primary snapshot is created first, followed by the incremental snapshot. During the simulation step, you specify the incremental snapshot. This automatically loads the primary snapshot, and the two are joined into a single model at simulation time 0. Simulation proceeds as if the whole design had been elaborated into a single snapshot.

For more details, see the [Multi-Snapshot Incremental Elaboration](#) guide.

Note: During an incremental elaboration, if a subprogram defined in a VHDL package is called in both the primary and incremental parts of the design, the VHDL package is instrumented only for those coverage types (block/expression) which are applicable on the package from the primary part of the design.

For a list of CCF commands supported/unsupported in the MSIE flow, refer to [CCF Commands Supported in the MSIE Flow](#) and [CCF Commands Unsupported in the MSIE Flow](#).

When enabling coverage in the MSIE flow, remember that:

- Coverage configuration file cannot be specified at the time of creating the incremental snapshot.
- Functional coverage cannot be enabled at the time of generating the incremental snapshot if it

is not enabled at the time of generating the primary snapshot. In addition, if functional coverage is enabled at the time of generating the primary snapshot, it automatically applies to the incremental snapshot.

- Other types of coverage (block, expression, toggle, FSM) can be enabled when elaborating either the primary or incremental snapshot. You can specify a coverage configuration file (CCF) with the `-covfile` option during primary or incremental elaboration. For multiple primary snapshots, all coverage specifications provided during primary elaboration must be the same for all the primaries or an error will be generated during incremental elaboration. However, global settings, such as branch scoring, hit count limit, expression coverage mode (SOP/control/vector), and so on, specified through the CCF at the time of primary snapshot creation also apply to the incremental snapshot.

Simulating the Design

At the simulation stage, whatever is instrumented is stored and dumped to the coverage database. You can simulate the design in the following modes:

- Noninteractive mode
- Interactive mode

Noninteractive mode

To simulate in noninteractive mode, use:

```
xmsim <design_snapshot> [-covoverwrite] [-covworkdir <workdir>]
[-covscope] [-covtest <test>] [-covbaserun <test>] [-covnomodeldump]
[-covmodeldir <modeldir>] [-covfirstbinmatch] [-cov_debuglog]
[-cov_gen_merge] [-covcleanworkdir]
```

where

- `-covoverwrite` enables overwriting of coverage output files and directories. By default, the coverage output data is written to the directory `cov_work scope/test`. If this path already exists, the output directory is not overwritten and the simulation terminates. Using this option, you overwrite the coverage data.
- `-covworkdir <workdir>` specifies the basename for the work directory. By default, the work directory is `cov_work`. Use this option to specify a different work directory.

- `-covscope` specifies an alternate directory for storing coverage model files. By default, the model files are stored in the `./cov_work/cope` directory.
- `-covtest <test>` specifies the run directory name for the current run. By default, the coverage data files for the current run are stored in `./cov_work/cope/test`. For more details, see [Coverage Data Storage](#).
- `-covbaserun <test>` appends seed number to the name of the run directory. When you use the `-covbaserun` option and simulate a design multiple times using different seed values, the simulator automatically creates a new test directory based on the seed number for each run and all the results are automatically saved. For instance, if you specify `-covbaserun <test_directory_name>`, the final test directory name is `<test_directory_name>_sn <seed>` and `<test_directory_name>_sv <seed>` for Specman and IUS, respectively. Here, `<seed>` is the seed number. If both Specman and IUS are used, the final test directory name is `<test_directory_name>_sn <seed>_sv <seed>`.
- `-covnomodeldump` disables dumping of the coverage model file. This option is provided for the Enterprise Manager flow. Refer Enterprise Manager documentation for details. See [Coverage Data Storage](#) for details on the coverage model file.
- `-covmodeldir <modeldir>` enables saving or reusing coverage model (`.ucm`) from a specified directory. The absolute/relative path of the directory is passed as an argument to this command. If `xmsim` is able to reuse from or, if needed, save coverage model in this directory, a soft link to this coverage model is created in the coverage scope directory, which is by default `./cov_work/cope`. This also enables avoiding a violation of default coverage directory structure that is expected by coverage analysis tools.

Note: If the specified directory does not exist, `xmsim` creates it, and if `xmsim` is unable to create it, an error is reported and coverage model is not saved. Further, if the directory exists but `xmsim` is unable to reuse or save coverage model, an error is reported.

Note: If `xmsim` is invoked from `EMGR` and `-covmodeldir` is specified, a warning stating that `-covmodeldir` will not have an effect is reported and coverage model is saved in the directory specified by `EMGR`.

- `-covfirstbinmatch` enables the sampling of the first hit in the cross bin during a simulation cycle. By default, all the cross bins which get hit during a simulation cycle are sampled. Consider the given example:

```

module test;
covergroup cg ;
option.per_instance =1 ;

A: coverpoint a ;
B: coverpoint b ;
CR: cross A,B {
    bins b1 = binsof (A) intersect {1}; // NOTE :: bins b1 and b2 contains
                                         same cross tuples
    bins b2 = binsof (A) intersect {1} ;
}

endgroup;
endmodule;

```

The cross report with the `-covfirstbinmatch` option is shown.

```

0.28 (28/100) 4/14  cg.CR 15  CR: cross A,B {
    1(1)      b1   16  bins b1 = binsof (A) intersect {1} ; // bin b1 got hit because it
    0(1)      b2   17  bins b2 = binsof (A) intersect {1} ; // was the first significant bin
                                                               // bin b2 was not hit because it
                                                               // has same cross tuples as in bin b1

```

The cross report without the `-covfirstbinmatch` option is shown.

```

0.36 (36/100) 5/14  cg.CR 15  CR: cross A,B {
    1(1)      b1   16  bins b1 = binsof (A) intersect {1} ; // bin b1 and b2 both got hit
                                                               //because they share the same
    1(1)      b2   17  bins b2 = binsof (A) intersect {1} ; // cross tuples

```

Note: For accurate results and performance gains, use this option when the cross bins defined are mutually exclusive, such that there are no common cross tuple between user defined cross bins.

- `-cov_debuglog` is an internal option that prints the coverage-related information for different phases of simulation. By default, logs are not printed for this information.
- `-cov_gen_merge` while generating sop unqualification number of an instance, considers the sopable expressions inside elaborated generates blocks of the instance along with remaining sopable expressions of the instance. Consider the given example:

```

module top();
parameter P =1;
reg r2, r3;

```

```
wire r1;
assign r1 = r2^r3;
generate
  if(P==1)
    begin:TRUE
      assign r1=r2&r3;
    end
  else
    begin:FALSE
      assign r= r2||r3;
    end
  endgenerate
endmodule

module tb();
  top #(0) I1();
  top #(1) I2();
endmodule
```

In the given example, while generating sop uniquification with the `-cov_gen_merge` option, the expressions `r2^r3` and `r2||r3` will be considered for instance `I1`, and the expressions `r2&r3` and `r2|r3` will be considered for instance `I2`.

However, while generating sop uniquification without the `-cov_gen_merge` option all the expressions `r2^r3`, `r2&r3`, and `r2||r3` are considered for both the instances.

Note: The `-cov_gen_merge` option has been made redundant and has been deprecated.

- `-covcleanworkdir` removes coverage working directory (by default `cov_work`), which includes coverage model file (`.ucm`) and coverage data file (`.ucd`), and `icc.com`, at the beginning of the simulation.

Interactive mode

You use the `xmsim coverage` command to generate coverage data for a simulation run in interactive mode. Following is the BNF for the `coverage` command.

Setting up Coverage Variables

```
coverage -setup [-workdir <workdir>]
    [-scope <scope>]
    [-testname <test>]
    [-dut <dut_instance>]
```

Resetting Code Coverage Counts

```
coverage -code -reset [-after <time> [-absolute | relative]]
    [-instance <instance> [-depth <n> | all | to_cells]]
```

Resetting FSM Coverage Counts

```
coverage -fsm -reset [-after <time> [-absolute | relative]]
```

Resetting Toggle Coverage Counts

```
coverage -toggle -reset [-after <time> [-absolute | relative]]
```

Selecting Functional Coverage Points

```
coverage -functional -(de)select <assertion_selector>
    [-cover] [-assert] [-assume] [-boolean] [-immediate]
    [-depth {<n> | all | to_cells} [-filter <wildcard>]]
    -list
```

Turning off Coverage Storing

```
coverage -off
```

Dumping Currently Scored Coverage

```
coverage -dump <test>
```

Launching Coverage Analysis Tool

```
coverage -analyze
```

Setting up Coverage Variables

To set up global coverage variables, use:

```
coverage -setup [-workdir <workdir>] [-scope <scope>] [-testname <testname>] [-dut
<dut_instance>]
```

where:

- `-workdir <workdir>` specifies an alternate directory for coverage output files. By default, the

coverage files are stored in the `cov_work` directory created in the current working directory.

- `-scope <scope>` specifies an alternate directory for storing coverage model files. By default, the model files are stored in the `./cov_work/scope` directory.
- `-testname <test>` specifies the run directory name for the current run. By default, the coverage data files for the current run are stored in `./cov_work/scope/test`. For more details, see [Coverage Data Storage](#).
- `-dut <dut_instance>` defines the hierarchy reference for reporting. This option tells Xcelium™ Single Core simulator to store scored coverage in the hierarchy below `<dut_instance>` with paths that start at `<dut_instance>`. This makes databases insensitive to any hierarchy above that level.

Consider an example of two simulations with `mic` instantiated as `mic_0`.

Simulation 1:

```
xmsim> coverage -setup -dut tb_sys.sys.mic_0
```

Simulation 2:

```
xmsim> coverage -setup -dut tb_mic.mic_0
```

Both the simulations will store and report coverage for module `mic` and all the components instantiated within `mic`.

Note: The DUT can also be specified using the xmelab command line option `-covdut <DUT_module>`. Using `-covdut` to define a DUT is not allowed when the module is instantiated multiple times at the top level. To score multiple copies of the same DUT, use a virtual DUT above the desired ones or use the default top level.

Resetting Code Coverage Counts

To reset code coverage counts to 0, use:

```
coverage -code -reset [-after <time> [-absolute | relative]]  
[-instance <instances> [-depth <n> | all | to_cells]]
```

where

- `-after` resets the counts:
 - At the specified `<time>` if the `-absolute` option is used

- After the specified <time> if the -relative option is used
If -absolute or -relative is not specified, -relative is assumed.
- -instance <instances> specifies the instance scope to be reset.

Note: Generate blocks in a module-instance are considered as part of the module-instance. You cannot specify generate block scope with the -instance option.

- -depth specifies the scope levels to descend for resetting coverage counts. It can be:
 - <n> which is an integer value that specified the number of scope levels to descend. For example, 1 indicates that only the specified scope will be included, 2 indicates that the specified scope and its sub-scopes be included, and so on. The default value is 1.
 - all selects all scopes in the hierarchy below the specified scope.
 - <to_cells> includes all scopes except the modules marked with `celldefine or VITAL entities with the VITAL Level0 attribute.

Note: In block coverage, when you reset the coverage counters to remove the initial simulation effects, the simulation does not reenter the process and the block counter remains zero after the reset.

Resetting FSM Coverage Counts

To reset the FSM coverage counts to 0, use:

```
coverage -fsm -reset [-after <time> [-absolute | -relative]]
```

See [Resetting Code Coverage Counts](#) for syntax description.

Resetting Toggle Coverage Counts

To reset the toggle coverage counts to 0, use:

```
coverage -toggle -reset [-after <time> [-absolute | relative]]
```

See [Resetting Code Coverage Counts](#) for syntax description.

Note: When resetting toggle counts, counts for rise/fall transitions can be off by one if `set_toggle_strobe` is used in the CCF during elaboration.

Specifying Functional Coverage Points

To select/deselect functional coverage options while simulating the design, use:

```
coverage -functional - (de)select <assertion_selector>
           [-cover] [-assert] [-assume] [-boolean] [-immediate]
           [-depth {<n> | all | to_cells} [-filter <wildcard>]]
           -list
```

where

- <assertion_selector> specifies the names of the assertions for functional coverage analysis.
- -cover selects the cover directive.
- -assert selects the assert directive.
- -assume selects the assume directive.

Note: By default, all the directives get selected as functional coverage points.

- -boolean selects boolean type assert and assume directives.
- -immediate enables scoring of SVA immediate assertions.

Note: System Verilog immediate assertions inside a class declared in a module and instance-based coverage of immediate assertions inside a class are not scored.

- -depth specifies the scope levels to descend for selecting/deselecting assertions. See [Resetting Code Coverage Counts](#) for details on this option.
- -filter enables selection of directives that match <wildcard>. This option is used along with the -depth option and if used, should follow immediately after the -depth option.

For example, to select all the scopes in `Monitor_ABV` and with coverage point names starting with `m`, use:

```
xmsim> coverage -functional -select Monitor_ABV -depth all -filter m*
```

Note: You should use wildcards to match objects and not the scope. As a result, a wildcard specified with an intention of selecting scopes will not work. The following command will not match any scopes and hence select nothing:

```
xmsim> coverage -functional -select top.* -depth all
```

To include all of the scopes in the module `top`, use:

```
xmsim> coverage -functional -select top -depth all
```

- `deselect` omits or deselects coverage points at any time during the functional coverage analysis process. Selection and deselection commands are sequentially applied and build the final set of coverage points to be scored and stored.

To disable covering of immediate assertions inside classes in packages, use the given tcl command during simulation:

```
xmsim> coverage -functional -deselect <assertion_path>
```

For example, to deselect all assertions inside a package "pkg" use:

```
xmsim> coverage -functional -deselect pkg.
```

- `-list` lists the coverage points being processed for generating the functional coverage data. It is helpful to list points to score after doing complex selections and deselections. You cannot list the coverage points related to SystemVerilog CoverGroup using this option.

Note: Coverage is scored only for assertions that are enabled at the end of simulation. If any assertion is disabled prior to simulation exit, then coverage for that assertion is not dumped to the coverage database. For more information on enabling/disabling assertions, see [Xcelium Simulator tcl Command Reference](#).

Turning off Coverage Storing

To turn off storing of coverage results, use:

```
coverage -off
```

However, scoring of all instrumented coverages still happens in simulation.

Dumping Currently Scored Coverage

To immediately dump all of the currently scored coverages, use:

```
coverage -dump <test>
```

The above command dumps all of the currently scored coverages into `<workdir>/<scope>/<test>`. If `<test>` exists and the `-covoverwrite` option is used, then the `test` directory is overwritten; otherwise an error is reported and database dumping fails.

Note: When debugging a snapshot in SimVision, use the `-covoverwrite` option to ensure that a new database is stored in each rerun.

Launching Coverage Analysis Tool

To dump coverage data, and launch the coverage analysis tool, use:

```
coverage -analyze
```

By default, the above command dumps coverage data and launches *Integrated Metrics Center* (IMC) as the coverage analysis tool. If IMC is not found in the path from where NC is launched, an error is reported. Currently, IMC can be launched only from <install-dir>/bin. For more details on IMC, see the *Integrated Metrics Center User Guide* in the Metric-Driven Verification (MDV) release.

Coverage Data Storage

By default, coverage data is stored as:



where,

- `cov_work` is the default coverage working directory, which can contain multiple scope directories.
- `scope` is the default verification scope directory. It contains configuration of the design (DUT) or the testbench, for example the model of the DUT (`.ucm`) that should be shared across all runs of the scope. The verification scope directory also contains the run directories specific to that scope.
- `test` is the name of the run directory that stores run specific verification data (`.ucd`), for example, the sampled data.

The run directory, by default, is named as `test` if an alternate name is not specified using the `xmsim -covtest` option or the `-covbaserun` option.

Note: If the `-covtest` option or the `-covbaserun` option is not specified, the run directory is named based on the file name in the `-ovmtest` option. However, if even the `-ovmtest` option is missing, the run directory is named based on the file name in the `-ovmtop` option. For example, if the given xrun command is used:

```
xrun -NOCOPYRIGHT -NOCOPYRIGHT -access +RWC -covfile /vobs/pv_ncvlog/etc/cov10_2.ccf
-coverage all -nowarn ECSSDM -nowarn COVDCL -covfile /vobs/pv_ncvlog/etc/cov10_2.ccf
dut.sv -ovmtest SV:test1 -ovmtop SV:uvc1_env
```

The run directory is named as `SV:test1`. If the `xrun` command does not contain the `- ovmttest` option:

```
xrun -NOCOPYRIGHT -NOCOPYRIGHT -access +RWC -covfile /vobs/pv_ncvlog/etc/cov10_2.ccf
-coverage all -nowarn ECSSDM -nowarn COVDCL -covfile /vobs/pv_ncvlog/etc/cov10_2.ccf
dut.sv -ovmtop SV:uvc1_env
```

The run directory is named as `SV:uvc1_env`.

Note: If `-covtest` or `-covbaserun` is not specified and randomization is done using seed value, then the run directory is named as `test_sv<seed>`, where `<seed>` is random value assigned to the run.

For a single run, one model file and one data file are created. For subsequent runs on the same design, the model file is reused and a different data file is created. The model file across runs for the same design changes if:

Different instances/ modules are selected/ deselected in different runs through CCF file at elaboration.	Run 1 - CCF <code>select_coverage -block -instance dut</code> Run 2 - CCF <code>select_coverage -block -instance *...</code>
Different DUT name is specified to the xmelab command-line argument <code>-covdut</code> .	Run 1 <code>xmelab -covdut B -covfile cov.ccf worklib.top:v</code> Run 2 <code>xmelab -covdut A -covfile cov.ccf worklib.top:v</code>
Different coverage metrics are instrumented in different runs at elaboration.	Run 1 <code>xmelab -coverage b worklib.top:v</code> Run 2 <code>xmelab -coverage b:e worklib.top:v</code>

Simulation runs of different tests are run with different settings of DUT.

Run 1

```
coverage -setup -dut tb_mic.mic_0
```

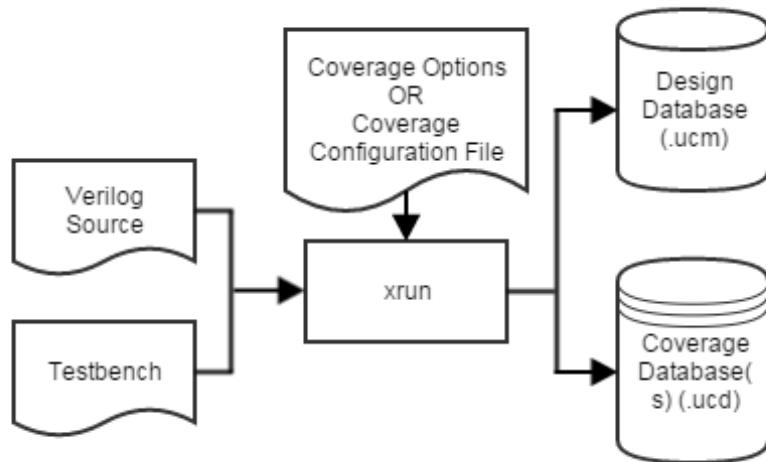
Run 2

```
coverage -setup -dut tb_mic.mic_0 fifo_0
```

Note: The model file across runs should be reused to allow merging of results, loading of tests, and re-using of marks file.

Generating Coverage Data using Single-Step Simulation

The following diagram illustrates the single-step process of generating the coverage data using XRUN:



To generate coverage data using single-step simulation, you use *xrun*. The syntax for coverage specific options in *xrun* is:

```
xrun [-coverage <coverage_types> |
       -covfile <coverage_configuration_file> ]
      -covdut <DUT_module> [<DUT_module>]
      -covworkdir <workdir>
      -covscope <scope>
      -covtest <test>
      -covmodeldir <modeldir>
      -covoverwrite
      -tcl <tclfile>
```

```
<other_xrun_options>
```

where,

- `-coverage <coverage_types>` enables coverage data generation for all of the compiled modules. For more details, see [-coverage <coverage_types>](#).
- `-covfile <coverage_configuration_file>` specifies the configuration file to be used for code coverage instrumentation. For more details, see [-covfile <coverage_configuration_file>](#).
- `-covdut <DUT_module>` specifies a design under test for coverage. For more details, see [-covdut <DUT_module>](#).
- `-covworkdir <workdir>` specifies the basename for the work directory. By default, the work directory is `cov_work`. Use this option to specify a different work directory.
- `-covscope <scope>` specifies an alternate directory for storing coverage model files. By default, the model files are stored in the `./cov_work scope` directory.
- `-covtest <test>` specifies the run directory name for the current run. By default, the coverage data files for the current run are stored in `./cov_work scope/test`. For more details, see [Coverage Data Storage](#).
- `-covmodeldir <modeldir>` enables saving or reusing coverage model (`.ucm`) from a specified directory. The absolute/relative path of the directory is passed as an argument to this command. If `xmsim` is able to reuse from or, if needed, save coverage model in this directory, a soft link to this coverage model is created in the coverage scope directory, which is by default `./cov_work scope`. This also enables avoiding a violation of default coverage directory structure that is expected by coverage analysis tools.

Note: If the specified directory does not exist, `xmsim` creates it, and if `xmsim` is unable to create it, an error is reported and coverage model is not saved. Further, if the directory exists but `xmsim` is unable to reuse or save coverage model, an error is reported.

Note: If `xmsim` is invoked from `EMGR` and `-covmodeldir` is specified, a warning stating that `-covmodeldir` will not have an effect is reported and coverage model is saved in the directory specified by `EMGR`.

- `-covoverwrite` enables overwriting of coverage output files and directories. By default, the coverage output data is written to the directory `cov_work scope/test`. If this path already exists, the output directory is not overwritten and the simulation terminates. Using this option, you

overwrite the coverage data.

To generate block coverage data for in single-step mode, use:

```
% xrun design_top.vhd testbench.v -coverage b
```

Support for Mixed HDL-SystemC (SC) / Analog Mixed Signals (AMS) Designs

Currently, only functional coverage is supported for SC modules, and only assertion coverage is supported for AMS modules (if the [select_functional -ams_control](#) command is used at elaboration). The following table summarizes the coverage types supported for SC, AMS, and HDL modules.

Design Modules	Code	FSM	Assertion	Covergroup
SC modules	No	No	Yes	Yes
AMS modules	No	No	Yes	No
HDL modules	Yes	Yes	Yes	Yes

As a result, in a mixed-HDL SC/AMS design, if coverage is enabled for entire design, then NC dumps coverage data for HDL modules only. In addition, a warning is generated to indicate that code and FSM coverage for SC modules is not supported and no coverage is supported for AMS modules.

Note: By default, no coverage is scored for AMS modules. However, you can enable scoring of assertions inside AMS modules using the [select_functional -ams_control](#) command at elaboration.

The syntax for generating code coverage data for designs including SystemC modules is:

```
xmsc_run -xmelab_args, [+nccovfile+<coverage_configuration_file> | +nccoverage+<coverage_types> ] <other_xmsc_run_options>
```

In the above syntax,

- `-xmelab_args` specifies coverage options to `xmelab`.
- `+nccovfile` option passes a configuration file to `xmelab` for instrumentation.
- `+nccoverage` option enables coverage data generation for different coverage types.
- `<other_xmsc_run_options>` specifies options related to simulating the design.

To generate coverage data for block and expression coverage from a mixed HDL-SystemC design,

use:

```
xmsc_run -noscv -dynamic -INPUT run_intg.tcl -ACCESS +RWC -xmelab_args, +nccoverage+b:e  
model.cpp sctop.cpp DUT.v model.v vchild.v
```

Support for SystemVerilog Constructs

The following limitations apply to scoring in SystemVerilog code:

- Block/expression/toggle coverage is not supported for SystemVerilog constructs. The coverage engine will ignore any SystemVerilog constructs and generate coverage data for rest of the design.
- Block/expression/toggle coverage and data coverage with SystemVerilog covergroups are supported in the same run.
- FSM coverage is supported for designs and testbenches with SystemVerilog SVA and covergroup constructs but not with any other SystemVerilog constructs.

Note: For detailed information on SystemVerilog covergroup construct, refer to [Functional Coverage](#).

The given SystemVerilog constructs are supported:

- Toggle coverage of objects of allowed types (as specified in Toggle coverage section) inside system verilog interfaces.
- Toggle coverage of System Verilog structs, both packed and unpacked. Only the fields with allowed types as specified in Toggle coverage section are supported.
- Toggle coverage for enum objects is supported and can be enabled by using the [set_toggle_scoring](#) CCF command. When using this command, an enum variable is considered as toggled if it has been assigned all the values in the enum declaration.
- Block/branch coverage due to flow break statements by structs and enums. Support has been added for COM analysis of such blocks and branches.
- Expression coverage for expressions that have struct and its members, for both packed and unpacked structs, is supported with the [set_expr_scoring -struct](#) command.
- Block and expression coverage for union and its members.
- Expression coverage for enums for most of the use cases is supported.

Analyzing Coverage Data

This chapter introduces Integrated Metrics Center (IMC) as an analysis tool to analyze, merge, and report coverage data. When using IMC, you can:

- Launch IMC
- Load Runs
- Merge Runs
- Rank Runs
- Generate Reports
- Refine Metrics

Launching IMC

IMC can be launched in any of the following mutually exclusive modes:

- GUI mode (Opens the Graphical User Interface)
To launch IMC in GUI mode, use the `imc -gui` command. This is the default mode. If `-init <tcl_file>` option is used with the `imc -gui` command, the `<tcl_file>` will be executed before launching IMC in GUI mode.
- Command-line Interactive mode (Opens the command-line interface)
To launch IMC in command-line interactive mode, use the `imc -batch` command. If `-init <tcl_file>` option is used with the `imc -batch` command, the `<tcl_file>` will be executed before launching IMC in command-line interactive mode.
- Batch mode (Allows execution of a batch script and exits)
To launch IMC in batch mode, use the `imc -exec <command_file>` command. If `-init <tcl_file>` option is used with the `imc -exec <command_file>` command, then the `<tcl_file>` will be executed before executing the `<command_file>`.

Note: The behavior of `imc -batch` command has changed. The `-batch` option now invokes IMC in command-line interactive mode. To just execute a script and exit, use the `-exec` option.

In addition, the following commands are no longer a valid method of launching IMC:

```
imc script.tcl  
imc -batch script.tcl
```

Instead of above commands, following command can be used:

```
imc -exec script.tcl
```

The complete syntax for the `imc` command is:

```
imc [-batch | -gui | -exec <command_file> ] [options]
```

For more information, refer to the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

Loading Runs

After launching IMC, you can load a coverage run to perform various tasks. To load a coverage run, use:

```
load -run { [ [ <workdir> /] <scope> /] <test> }
```

Listing Currently Loaded Run

To view the currently loaded run, use:

```
show [-run]
```

Unloading Run

To unload the loaded run and associated model files, use:

```
unload [-run]
```

For more information on loading and unloading runs in IMC, refer to the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

Merging Runs

To merge the runs:

1. Set the merge configuration using the `merge_config` command (optional).
2. Merge the runs using the `merge` command.

For more information on merging runs in IMC, refer to the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

Ranking Runs

To rank the runs:

1. Set the ranking configuration using the `rank_config` command (optional).
2. Rank the runs using the `rank` command.

For more information on ranking runs in IMC, refer to the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

Generating Reports

The command-line interactive mode of IMC provides following two commands for generating reports:

- `report`
- `report_metrics`

For more information on generating reports using IMC, refer to the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

Refining Metrics Data

Refinements (exclude and un-exclude items) can be applied in IMC's GUI as well as batch mode. The `exclude` command allows you to exclude items from coverage analysis. The `unexclude` command allows you to un-exclude previously excluded items.

For more information on these commands, refer to the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

Exiting IMC

To exit from IMC, use:

```
exit
```

Supported and Unsupported Functionality

All coverages in general:

- Score in VHDL processes.
- Do not score in VHDL/Verilog generate statements (except for block, expression, and functional coverage assertions).
- Do not score code in protected modules or instances trees.
- Do not simultaneously score multiple top-level DUTs of the same name.
- Do not score code coverage in property/vunit files.
- Do not score code coverage with all SystemVerilog constructs, except toggle coverage for interfaces and `struct` System Verilog constructs. However, scoring of SystemVerilog enumerations and multidimensional static arrays and vectors is supported with the `set_toggle_scoring` command.
- Do not support code coverage (Block, Expression, Toggle, and FSM) in modules with type parameters. However, code coverage in such modules is supported with the `set_parameterized_module_coverage covfile` option.

Exceptions to above for specific coverage types are discussed below:

- [Block Coverage](#)
- [Expression Coverage](#)
- [Toggle Coverage](#)
- [FSM Coverage](#)
- [Functional Coverage](#)

Block Coverage

Block coverage scores:

- Verilog in initial and always blocks, tasks, and functions.
- VHDL in subprograms and in process and generate statements.
- Verilog generate `if`, `for` and, `case` blocks.
- Verilog/VHDL implicit else (turn off with `set_implicit_block_scoring` or pragma).
- Verilog implicit case default (turn off with `set_implicit_block_scoring` or pragma).
- Verilog/VHDL explicit case `default` (turn off with `set_explicit_block_scoring` or pragma).
- Verilog continuous assignments as blocks (with `set_assign_scoring`).
- Verilog ternary assignments and VHDL conditional and selected signal assignments (with `set_branch_scoring`).

Block coverage does not score:

- VHDL concurrent signal assignments.
- VHDL subprogram defined in a package when it is called from inside only a generate block.
- Event constructs `@`, `or`, `and` event (turn on with `set_expr_scoring -event`).
- Gate-level or user-defined primitives.

Expression Coverage

- Scores in Verilog functions and tasks.
- Does not score in VHDL subprograms (functions and procedures) inside packages.
- Scores expressions:
 - In continuous assignments
 - In if, case, case_item, for, while, `@`, `#` in always blocks
 - In procedural assignments
 - On input ports of module instances (Verilog only)
 - In Verilog/VHDL generate blocks.
- Scores the following VHDL operators:
 - Logical: and, or, nand, nor, xor, xnor

- Relational: `=, /=, <, <=, >, >=`
Note: By default, scores VHDL logical operators (OR, AND, NOR, and NAND) only in condition expressions.
- Scores the following Verilog operators:
 - Relational: `>, >=, <, <=`
 - Logical: `!, &&, ||, ==, !=`
 - Case: `====, !==`
 - Bit-wise: `~, &, |, ^, ^~, ~^`
 - Reduction: `&, ~&, |, ~|, ^, ~^, ^~`
 - Conditional: `:`
Note: By default, scores Verilog logical operators (`||` and `&&`) only in condition expressions.
- Does not score:
 - VHDL sensitivity list
 - Expressions that contain functions

Toggle Coverage

- Scores transitions (0->1 and 1->0) on:
 - Verilog scalar/vector nets and registers and ports.
 - VHDL `bit`, `std_ulogic/std_logic`, records, and vector versions of `bit` and `std_ulogic/std_logic` and `boolean` types.
 - Optionally scores transitions originating also in X or Z.
- Scores OOMR signals for toggle coverage, but does not consider these objects for constant object marking.
- Is not scored for `supply0` and `supply1` nets.
- Scores toggle objects inside SystemVerilog interfaces. In addition, it scores SystemVerilog structs, both packed and unpacked. Consider the following example with struct:

```
typedef struct
{
```

```

    logic signed [2:0] fld1;
    logic signed fld2;
}
StConf st;

```

The toggle report for the toggle object st is shown:

Hit(Full)	Hit(Rise)	Hit(Fall)	signal
0	0	1	st.fld1[2]
1	1	1	st.fld1[1]
0	1	0	st.fld1[0]
1	1	1	st.fld2

- Scores packed and unpacked multi-dimensional static arrays for reg, logic, bit, and wire SystemVerilog data types by using `set_toggle_scoring -sv_mda [<max_bits_base2_exponent>]`.
- Does not score in Verilog/VHDL generate blocks or named blocks.
- Does not score a toggle object if it is marked as deselected for an instance and selected for another, as in this case, toggle report for the module treats the toggle object as excluded. This is because in the current implementation a higher precedence is given to exclude as compared to uncovered resolution.
- Does not score multidimensional arrays of structures and multidimensional enums.
- Does not score type-parameterized modules.
- Does not score input ports connected by unpacked arrays.

FSM Coverage

- Scores synchronous behavior of FSMs that follow supported coding styles and operation:
 - Has proper reset of FSM to put model in a defined state.
 - Asynchronous code coverage of same FSM may differ based on races, glitches and intermediate next states visited.
 - Assumes that inputs for decisions are defined (X on input is treated as 0).
 - Does not score transitions for expression that statically evaluate to FALSE.

- Does not extract FSMs that use current state at lower levels.
- Does not score in Verilog/VHDL generate blocks or named blocks.
- Does not score FSMs in modules that include any protected code.

Functional Coverage

- Scores PSL assert, assume, and cover directives.
- Scores SVA assert, assume, and cover directives.
- Scores type-based coverage of System Verilog immediate assertions inside a class declared in a Package or a Compilation Unit.
- Does not score System Verilog immediate assertions inside a class declared in a module.
- Does not score instance-based coverage of immediate assertions inside a class.
- Does not score VHDL asserts.
- Does not score plain properties.
- Scores instantiated SystemVerilog covergroups.
- Supports use of IAL components.
- Scores coverage points written in property files.
- Scores PSL/SVA assertions and covergroups in Verilog/VHDL generate if, for and, case blocks.

Coverage Configuration File (CCF) Commands

Following is the list of configuration file commands for different coverage types.

Command	Applies to Coverage Type/s	Description
<code>include_ccf</code>	All	Enables specifying different configuration commands for different blocks of a design by including coverage configuration file(s) within another coverage configuration file.
<code>set_libcell_scoring</code>	All	Enables scoring of Verilog modules compiled with <code>-v/-y</code> or <code>-libcell</code> option but continues to disable the scoring of Verilog modules defined with the <code>'celldefine</code> compiler directive as well as VHDL VITAL cells.
<code>set_merge_with_libname</code>	All	Enables merging of two modules from two different coverage databases, iff they have the same library name and self name.
<code>set_parameterized_module_coverage</code>	All	Enables separate type-coverage reporting of uniquely different module configurations of the same module declaration based on the different value parameter combinations with which that module is instantiated.
<code>select_coverage</code>	Block, Expression, Toggle, FSM, Covergroup	Enables coverage metric(s) for one (or more) design unit (module, package, interface, program, architecture, etc.) or, one (or more) instances of a design unit.
<code>(de) select_coverage</code>	Block, Expression, Toggle, FSM, Covergroup	Disables coverage metric(s) for one (or more) design unit (module, package, interface, program, architecture, etc.) or, one (or more) instance of a design unit

<code>set_inheritance_aware_class_coverage</code>	Block, Expression, Toggle, FSM, Covergroup	Enables separate reporting of the list of coverage items scored by the objects of a derived class and the objects of a base class.
<code>set_com</code>	Block, Expression, Toggle	Turns on/off constant object marking analysis in design.
<code>set_com_interface</code>	Block, Expression, Toggle	Defines ports on <i><modules></i> as variable for COM analysis.
<code>set_glitch_strobe</code>	Block, Expression	Filters glitches shorter than time window.
<code>set_hit_count_limit</code>	Block, Expression	Enables you to set the upper limit of hit counts during simulation run.
<code>set_subprogram_scoring</code>	Block, Expression	Scores only the subprograms in VHDL which are used in design scopes with block or expression coverage enabled.
<code>set_code_fine_grained_merging</code>	Block, Expression	Enables fine-grained merging to add merge resilience at concurrent block level instead of default module/instance level.
<code>set_refinement_resilience</code>	Block, Expression	Enables support for full refinement resilience for block and expression coverage.
<code>set_implicit_block_scoring</code>	Block	Turns off/on scoring of implicit blocks in modules.
<code>set_explicit_block_scoring</code>	Block	Turns off/on scoring of explicit defaults in modules.
<code>set_assign_scoring</code>	Block	Scores continuous assignments with block coverage.
<code>set_branch_scoring</code>	Block	Scores branches together with block coverage.
<code>set_statement_scoring</code>	Block	Scores statements within a block.

<code>deselect_macro</code>	Block	Deselects coverage of blocks identified inside macros used in SystemVerilog classes when the block coverage of classes is enabled using the <code>select_coverage -b -class <list_of_classes></code> command.
<code>set_expr_scoring</code>	Expression	Enables you to customize the coverage of expression in a design.
<code>set_expr_coverable_operators</code>	Expression	Enables coverage of various Verilog and VHDL operators with different options.
<code>set_expr_coverable_statements</code>	Expression	Enables coverage of operators in various statements.
<code>set_toggle_strobe</code>	Toggle	Filters glitches shorter than time window.
<code>set_toggle_limit</code>	Toggle	Sets Maximum count limit for rise and fall transitions.
<code>set_toggle_includex</code>	Toggle	Records <code>x -> 0</code> and <code>x -> 1</code> transitions.
<code>set_toggle_includez</code>	Toggle	Records <code>z -> 0</code> and <code>z -> 1</code> transitions.
<code>set_toggle_noports</code>	Toggle	Disables Toggle scoring of module ports.
<code>set_toggle_portsonly</code>	Toggle	Enables Toggle scoring of only module ports (excluding internal signals).
<code>set_toggle_scoring</code>	Toggle	Enables scoring and reporting of SV enumerations, multidimensional arrays and toggle scoring inside SV generate blocks.
<code>set_toggle_excludedfile</code>	Toggle	Excludes specific signals for toggle scoring using a file.
<code>set_toggle_smart_refinement</code>	Toggle	Enables smart refinement of toggle coverage.
<code>set_fsm_attribute</code>	FSM	Gives a tag name to a specific FSM in a module.
<code>select_fsm</code>	FSM	Selects FSMs to instrument on a per module/tag basis.
<code>deselect_fsm</code>	FSM	Deselects FSMs to instrument on a per module/tag basis.

<code>set_fsm_reset_scoring</code>	FSM	Scores reset states and reset transition states, separately.
<code>set_fsm_arc_scoring</code>	FSM	Scores individual arcs with input conditions for transitions.
<code>set_fsm_arc_termlimit</code>	FSM	Sets maximum limit for arc extraction. Default is 131072.
<code>Set_fsm_scoring</code>	FSM	Scores FSM hold transitions (transitions to the current state).
<code>set_optimize</code>	Functional (Assertion Coverage)	Enables you to perform various optimizations that help improve your coverage performance.
<code>select_functional</code>	Functional	Enables functional scoring and storing for selected DUT.
<code>set_covergroup</code>	Functional (Covergroup Coverage)	Enables you to perform various customizations in covergroup coverage.

include_ccf

The `include_ccf` command enables you to specify different configuration commands for different blocks of a design by including a coverage configuration file(s) within another coverage configuration file. You can use this command to include a CCF with common options across different blocks of a design and also include other CCFs that contain options specific to the contents of the blocks.

Syntax:

`include_ccf <filename>`

Arguments:

`<filename>` is the name of the included CCF, and can be specified as:

- Simple name, such as `c.ccf`
- Relative or absolute path of the CCF, such as `./abc/c.ccf`
or `/abc/test/log/coverage/block/impulse/c.ccf`, respectively
- Path of the CCF with the environment variable, such as `$(MYFILE)/c.ccf`.

Example:

Consider the given example, with two configuration command files, `common_opts.ccf` and `block_cov_opts.ccf` with the given commands:

```
//common_opts.ccf
set_branch_scoring
set_glitch_strobe 1ns

//block_cov_opts.ccf
select_coverage -b -inst test.dut1...
```

When you create a new CCF, `new_ccf.ccf`, using the `include_ccf` command as shown:

```
//new_ccf.ccf
include_ccf common_opts.ccf
include_ccf block_cov_opts.ccf
```

With the `common_opts.ccf` and `block_cov_opts.ccf` files included, `new_ccf.ccf` will get translated as:

```
//new_ccf.ccf
set_branch_scoring
set_glitch_strobe 1ns
select_coverage -b -inst test.dut1...
```

set_libcell_scoring

By default, coverage for Verilog library cells and VHDL VITAL cells is not scored. The `set_libcell_scoring` command enables scoring of Verilog library cells (modules defined with the ``celldefine` compiler directive or modules compiled with `-v/-y` options) and VHDL vital cells.

Syntax:

```
set_libcell_scoring -enable_vy
```

Arguments:

`-enable_vy` - Enables scoring of **only** Verilog modules compiled with the `-v/-y` or `-libcell` option.

Order of Precedence

The following table details the order of precedence for the `set_libcell_scoring` (without any option) and `set_libcell_scoring -enable_vy` commands:

CCF command/option	Coverage of Modules Compiled with the <code>-v/-y</code> or <code>-libcell</code> Option	Coverage of Modules whose Definition Appears Between Compilers Directives <code>`celldefine</code> and <code>`endcelldefine</code>
--------------------	--	--

Not Specified	Disabled	Disabled
set_libcell_scoring	Enabled	Enabled
set_libcell_scoring -enable_vy	Enabled	Disabled
set_libcell_scoring set_libcell_scoring -enable_vy (Irrespective of order)	Enabled	Enabled

set_merge_with_libname

By default, two modules or instances with the same name from two different coverage databases will be merged irrespective of the different libraries, if any, in which the modules were compiled. This eases merging and loading data across runs. To merge two modules from two different coverage databases, iff they have the same library name and self name, use the `set_merge_with_libname` command.

Syntax:

```
set_merge_with_libname
```

Example:

Consider the given example displaying two designs, Design 1 and Design 2 . In this example, Design1 has been compiled into worklib1 and Design2 has been compiled into worklib2 :

```
--Design 1
entity entity_object is
end entity_object;

architecture arch_object of entity_object is
begin
drive : process
begin
report "In design1 process";
end process drive;
end arch_object;
```

```
--Design 2
entity entity_object is
end entity_object;
architecture arch_object of entity_object is
signal var2:std_logic;
begin
```

```

drive : process(var2)
begin
report "In design2 process";
end process drive;
end arch_object;

```

When you merge `worklib1.entity_arch` and `worklib2.entity_arch` in the default mode, the count is updated for block coverage as well as other coverage metrics enabled, if any, as shown below:

Design 1	Design 2	Merged Output
Type name: <code>worklib1.entity_object</code> (arch_object)	Type name: <code>worklib2.entity_object</code> (arch_object)	Type name: <code>worklib1.entity_object</code> (arch_object)
File name: <code>/servers/testcases/vhdl/test1/design1.vhd</code>	File name: <code>/servers/testcases/vhdl/test1/design2.vhd</code>	File name: <code>/servers/testcases/vhdl/test1/design1.vhd</code>
Number of covered blocks: 0 of 1	Number of covered blocks: 1 of 1	Number of covered blocks: 1 of 1
Number of uncovered blocks: 1 of 1	Number of uncovered blocks: 0 of 1	Number of uncovered blocks: 0 of 1
Number of excluded blocks: 0	Number of excluded blocks: 0	Number of excluded blocks: 0
Count Block Line Kind Origin Source Code	Count Block Line Kind Origin Source Code	Count Block Line Kind Origin Source Code
-----	-----	-----
<code>0 1 13 code block 11 report</code> "In design1 process";	<code>1 1 13 code block 11 report</code> "In design2 process";	<code>1 1 13 code block 11 report</code> "In design1 process"

Consider the given example in which the two different libraries are used. In this example, with the `set_merge_with_libname` command, the count is not updated after merging the designs as shown:

Design 1	Design 2	Union
----------	----------	-------

Type name: worklib1.entity_object (arch_object)	Type name: worklib2.entity_object (arch_object)	Type name: worklib1.entity_object (arch_object)
File name: /servers/testcases/vhdl/ test1/design1.vhd	File name: /servers/testcases/vhdl/ test1/design2.vhd	File name: /servers/testcases/vhdl/ test1/design1.vhd
Number of covered blocks: 0 of 1	Number of covered blocks: 1 of 1	Number of covered blocks: 0 of 1
Number of uncovered blocks: 1 of 1	Number of uncovered blocks: 0 of 1	Number of uncovered blocks: 1 of 1
Number of excluded blocks: 0	Number of excluded blocks: 0	Number of excluded blocks: 0
Count Block Line Kind Origin Source Code	Count Block Line Kind Origin Source Code	Count Block Line Kind Origin Source Code
-----	-----	-----
0 1 13 code block 11 report "In design1 process";	1 1 13 code block 11 report "In design2 process";	0 1 12 code block 10 report "In design1 process";

-  The coverage databases generated with and without the `set_merge_with_libname` command cannot be merged.

set_parameterized_module_coverage

By default, for SystemVerilog parameterized modules, type-based coverage is the representative of the first instance of the module. If (and only if) coverage objects for other instances are different from the first instance, for the same module declaration, then their type-based coverage may not be reported. This command impacts only type-coverage and does not impact instance-coverage.

Behavior with the `set_parameterized_module_coverage` Command

With the `set_parameterized_module_coverage` command separate type-coverage of uniquely different module configurations is reported corresponding to different parameter values combinations used to instantiate the module. The following table details the tool behavior for different combinations:

	Design with "Value Parameterized" Modules	Design with "Value Parameterized" Modules	Design with "Type Parameterized" Modules	Design with "Type Parameterized" Modules
Instance Coverage	All instances are reported	All instances are reported	All instances are reported	No coverage for any instance is reported
Type Coverage	All instances are accumulated for type coverage	Only the first instance is reported for type coverage	All instances are accumulated for type coverage	No coverage for any instance for type coverage

Syntax:

```
set_parameterized_module_coverage
```

Consider the following code:

```
module dut();

parameter type T = int;
parameter [3:0] P0 = 1;
reg a; reg b;
reg [P0:0] r;
generate
  if(P0 == 1) begin: TRUE
    assign w = a || b;
  end
  else begin: FALSE
    assign w = a &b;
  end
endgenerate

initial
begin
#10 a=1; b=1;
#10 a=0; b=0;
end

endmodule

module top();
```

```

dut #(logic,2'b11) U1();
dut #(bit, 1'b1) U2();
dut #(bit, 1'b1) U3();

endmodule

```

In the given example, module dut is a parameterized module which is instantiated in module top with two uniquely different parameter value combinations (logic, 2'b11) and (bit, 1'b1). Hence, two different modules with name dut#(logic, 4'h3) for instance U1 and dut#(bit, 4'h1) for instances U2 and U3 are created and reported.

Module/Entity name: dut#(logic,4'h3)
File name: /vobs/nc_test/test/vlog/coverage/parameter_module/t0/test.v
Number of signal bits fully toggled: 0 of 6
Number of signal bits partially toggled(rise): 0 of 6
Number of signal bits partially toggled(fall): 2 of 6
Number of signal bits marked COV: 0
Number of signal bits marked IGN: 0

Hit(Full)	Hit(Rise)	Hit(Fall)	Signal
0	0	1	a
0	0	1	b
0	0	0	r[3]
0	0	0	r[2]
0	0	0	r[1]
0	0	0	r[0]

Module/Entity name: dut#(bit,4'h1)
File name: /vobs/nc_test/test/vlog/coverage/parameter_module/t0/test.v
Number of signal bits fully toggled: 0 of 4
Number of signal bits partially toggled(rise): 0 of 4
Number of signal bits partially toggled(fall): 2 of 4
Number of signal bits marked COV: 0
Number of signal bits marked IGN: 0

Hit(Full)	Hit(Rise)	Hit(Fall)	Signal
0	0	1	a
0	0	1	b
0	0	0	r[1]
0	0	0	r[0]

Points to Remember

- This feature is applicable only to Verilog/SystemVerilog modules.
- A design with a parametrized module whose type name changes across primary and secondary runs cannot be merged when ran with/without parametrization support.
- Toggle coverage is not supported for type-parameterized data type.

select_coverage

The `select_coverage` command enables you to select the design units, including modules, instances, packages, interfaces, compilation unit scope, and program blocks, that are scored for a coverage type.

Syntax:

```
select_coverage
(<metrics> [[(-design_unit|-module)] <list_of_design_units>]) |
(<metrics_covergroup> [[-module] <list_of_design_units>] [-class <list_of_classes>] [-cg_name <list_of_covergroups>]) |
(<metrics_code> {(-instance <list_of_instances>) | (-file <file_list>) | (-filelist <file_name>) | -sysv_bind_modules}) |
(<metrics_block> [<design_unit>] <list_of_design_units> [-class [<list_of_classes>]])
```

where,

```
metrics ::= [-block] [-expression] [-toggle] [-fsm] [-covergroup] [-all] [-betfc]
metrics_covergroup ::= [-covergroup] [-c]
metrics_code ::= [-block] [-expression] [-toggle] [-fsm] [-all] [-betfc]
<list_of_design_units> ::= <module_interface_program_package_compilation_unit_identifier>
[<list_of_design_units>]
<list_of_classes> ::= <class_path> [<list_of_classes>]
<class_path> ::= <class_identifier> [::<class_path>]
<list_of_covergroups> ::= <covergroup_identifier> [<list_of_covergroups>]
```

 [-all] does not include -covergroup.

Arguments:

- <metrics> specifies the type of coverage to score for the -module option.
- <metrics_covergroup> is used to specify covergroups at either design-unit level or a finer level that is

either at the class-level or by specifying the names of the covergroups. When you use the `-covergroup` or `-c`, it applies to the specified covergroup types. This option applies only to type-based selection and does not apply to instance-based selection.

- `<metrics_code>` specifies the type of coverage for `-instance` and `-file` options.
- `<metrics_block>` is used to specify block coverage at the class level. This option applies only to type-based selection and does not apply to instance-based selection for a class.
- `<list_of_design_units>`, `<list_of_classes>`, `<list_of_covergroups>`, `<list_of_instances>`, and `<file_list>` specifies the list of design units, classes, covergroups, instances, and files, respectively, to which the selected coverage applies. The following wildcards can be used in these lists:

Wildcard	Description	Example
*	Matches any text from current location until next delimiter or end of string	<pre>select_coverage -block -module test_mod*</pre> <p>Matches all names beginning with <code>test_mod</code>, such as <code>test_mod1</code>, <code>test_module</code>, and so on.</p> <pre>select_coverage -all -file test*</pre> <p>Matches all filenames beginning with <code>test</code> that include valid design modules.</p>
?	Matches any single character	<pre>select_coverage -all -module test_mod?</pre> <p>Matches all names beginning with <code>test_mod</code> followed by a single character such as <code>test_mod1</code> and <code>test_mod2</code>.</p>
...	Matches that module/instance/file path and all its descendants	<pre>select_coverage -all -module mod...</pre> <p>Matches all instances of <code>mod</code> and all of their descendants.</p> <pre>select_coverage -all -file /home/master/...</pre> <p>Matches all files under <code>/home/master/</code> and all the folders below <code>/home/master/</code>.</p>

- `-class` (de)selects type-based block coverage in specific classes. When specified with the `-cg_name` option, it (de)selects class-embedded covergroup types in specific classes and when specified with the `-b` option, it (de)selects type-based block coverage in classes. When you use the `-class` option with the `-b` option, blocks inside all kinds of methods, including static, virtual, local, protected, and public methods that have an implementation are scored. In addition, the blocks inside both non-parameterized

and specialized parameterized classes are scored. Blocks are scored separately for parameterized classes separately in each of its unique specialization.

- <list_of_classes> specifies the list of classes to be scored. If <list_of_classes> is not specified after -class, "*..." is assumed and all classes in the design unit are selected for coverage.
- <class path> specifies nested classes.
- -cg_name (de)selects specific covergroup types and all of their instances. If only the -cg_name option is specified and class is not specified, non-embedded covergroups are deselected.
- -filelist <filename> specifies the file that includes a list of design files to which the selected coverage must apply.

The files in <filename> must be listed one per line. To include comments in the file <filename>, begin the line with # .

For a Verilog/SystemVerilog design, specify the module names in the <list> as:

```
<module> | <library.module>
```

For a VHDL design, specify the module names in the <list> as:

```
<entity> | <library>.<entity>(<architecture>)
```

To select coverage for all instances of Verilog modules or all VHDL entity/architecture in a given library, use:

```
select_coverage <coverages> -module <library>.*
```

- -sysv_bind_modules enables/disables selected coverage types on all instances that are binded using SystemVerilog bind construct.
- Environment Variables can be used in the `select_coverage` command to specify the name or path of the DUT as follows:

```
setenv MYDUT_MODULE proj_dut  
select_coverage -block -expr -toggle -module ${MYDUT_MODULE}
```

Points to Remember

- When using the `select_coverage -b -class <list_of_classes>` command:
 - Any other argument and option cannot be specified.
 - COM analysis is not supported for class types.
 - Class type cannot be specified explicitly with `set_implicit_block_scoring` and `set_explicit_block_scoring` commands. However, wildcard '*' matches with the class types.
 - With the `deselect_coverage -remove_empty_instances` command, if class type is selected

for a module all its instances will be dumped even though they are empty.

- When specifying the files in <list> or in the file <filename>, remember that file names can be:
 - Absolute paths (for example, /home/master/design/ test.v)
 - Relative paths (for example, design/test.v, ./test.v)
 - Filename (for example test.v)

If you specify just the filename, then there is a possibility that the file matches multiple files in different paths. Consider the following compilation command: xmvlog <path1>/test.v <path2>/test.v

If the CCF command includes just the filename, as:

```
select_coverage -block -file test.v
```

then the selection command will apply to both <path1>/test.v and <path2>/test.v. The filename containing .../ is currently not supported. In addition, non-design filenames or files that contain the path of the design file cannot be specified in the file included in the filename. For example, if you include a file <icc> that contains <alu.v> and <memory.v> in the filename, then you cannot include <icc_1> that further contains <RAM.v> and <ROM.v> in <icc>. In this case, to include <RAM.v> and <ROM.v>, these need to be included directly in <icc> .

- The files can include environment variables, as shown below:
 - Example 1: (SRC_AREA is the environment variable.)

```
select_coverage -block -file /home/master/${SRC_AREA}/actual_file.v
```
 - Example 2: (SRC_AREA is the environment variable.)

```
select_coverage -block -filelist myfilelist
```

where contents of myfilelist are:
`./${SRC_AREA}/myfile1.v
/export/home/myuser/${SRC_AREA}/myfile2.sv`
- If none of the options (-module, -instance, -file, -filelist) is specified, -module is assumed as the option.
- When using the select_coverage command, the -module* and the - instance*... options have the same effect. Using any of these options, generates type as well instance coverage reports for block, expression, toggle, and FSM coverage.

! An error is displayed if the given conditions are violated:

- Options `-class` and `-cg_name` have to be specified with `-covergroup`.
- Options `-instance`, `-file`, `-filelist`, `-sysv_bind_modules`, and `-remove_empty_instances` cannot be specified with `-covergroup`.
- Wildcard character `...` cannot be used with options `-module` and `-cg_name` when used with coverage metric `-covergroup`.

Example:

Consider the given example:

```
reg clk=0;
reg rstn;
reg a = 1;

bind tb master master1 ( clk, rstn, a, b );
bind tb master master2 ( clk, rstn, b, c );

module tb;
    always #5 clk = ~clk;
    always #3 a = ~a;
    initial
    begin
        rstn = 0;
        #10 rstn = 1;
        #50 $finish;
    end
    ....
endmodule
```

With the `select_coverage -sysv_bind_modules -tbe` CCF command, the instance-based report is generated as follows:

```

Instance name: tb
Module/Entity name: tb
File name: /vobs/pv_ncvlog/test/vlog/automatic/coverage/
sv_covergroup/SWITCHES/sysV_bind/bind_sg/tb.v
Number of covered blocks: 5 of 5
Number of blocks marked COV: 0
Number of blocks marked IGN: 0

cnt covered block line no.          line origin description
-----
 1    11                      11  always #5 clk = ~clk;
 1    12                      12  always #3 a = ~a;
 1    14                      14  begin
 1    16                      16  #10 rstn = 1;
 1    17                      17  #50 $finish;

```

The bind report is generated as follows:

```

Instance name: tb.master1.driver1
Module/Entity name: driver
File name: /vobs/pv_ncvlog/test/vlog/automatic/coverage/
sv_covergroup/SWITCHES/sysV_bind/bind_sg/driver.v`include files aaa.
/vobs/pv_ncvlog/test/vlog/automatic/coverage/sv_covergroup
/SWITCHES/sysV_bind/bind_sg/both.ps1
Number of covered blocks: 3 of 3
Number of blocks marked COV: 0
Number of blocks marked IGN: 0

cnt covered block line no.          line origin description
-----
 1    4                      4  begin
 1    6      true part of      6  if ( !rstn ) b <= 0;
 1    7      false part of     6  if ( !rstn ) b <= 0;

```

(de)select_coverage

The (de) select_coverage command enables you to (de)select the design units, including modules, instances, packages, interfaces, CU scope, and program blocks, that are scored for a coverage type.

Syntax:

```

deselect_coverage
(<metrics> [ -module ] <list_of_design_units>) |
(<metrics_covergroup> [ [-module] <list_of_design_units> ] [-class <list_of_classes>] [-cg_name
<list_of_covergroups>]) |
(<metrics_code> {(-instance <list_of_instances>) | (-file <file_list>) | (-filelist <file_name>)
| -sysv_bind_modules}) |
-remove_empty_instances

```

where,

```

<metrics> ::= [-block] [-expression] [-toggle] [-fsm] [-covergroup] [-all] [-
betfc]

<metrics_covergroup> ::= [-covergroup] [-c]

```

```
<metrics_code> ::= [-block] [-expression] [-toggle] [-fsm] [-all] [-betf]  
<list_of_design_units> ::=  
<module_interface_program_package_compilationunit_identifier>  
[<list_of_design_units>]  
  
<list_of_classes> ::= <class_path> [<list_of_classes>]  
<class_path> ::= <class_identifier> [::<class_path>]  
  
<list_of_covergroups> ::= <covergroup_identifier> [<list_of_covergroups>]
```

 [-all] does not include -covergroup.

Arguments:

- -remove_empty_instances removes the instances from the design hierarchy where no coverage is found. For more information, see [Removing Empty Instances From the Coverage Hierarchy](#).

See [select_coverage](#) for details on syntax description.

Points to Remember

- If there are multiple `select_coverage` and `deselect_coverage` commands in a configuration file, then each command builds on the previous command. This means that the functionality is a union of all the select and deselect commands in the configuration file.
- When you use the `deselect_coverage -covergroup` command:
 - If the coverage methods `get_coverage()`, `get_inst_coverage()`, `get_hitcount()`, or `get_inst_hitcount()` refer to a deselected covergroup, these coverage methods will not take effect.
 - If predefined coverage methods, such as `sample()`, `start()`, and `stop()`, refer to deselected covergroups, a simulation time note is displayed to indicate that these methods are not effective for deselected covergroup types.
 - If any procedural `option/type_option` refers to a deselected covergroup, it will be ignored.
 - If an instance of a deselected covergroup type is used in a cross declaration, then coverage collection and dumping for the cross will not be done and an elaboration time warning will be displayed.
 - If none of the options is specified with `-covergroup`, all the non-class embedded covergroup types in the design will be (de)selected.
- Generate blocks in a module/instance are considered as part of the module/instance in selection and deselection commands used at elaboration. You cannot apply selection/deselection on generate blocks explicitly.
- If the `deselect_coverage -remove_empty_instances` command is used in any of the coverage configuration files passed during elaboration, it applies to the complete design. Note that the `-remove_empty_instances` option cannot be specified with the `select_coverage` command.

Example:

Consider the following example:

```

module m1;
covergroup cg1; ... ; endgroup
class c1;
    covergroup cg2; ... ; endgroup
    class c2;
        covergroup cg3; ....; endgroup
    endclass
endclass
endmodule
module m2;
covergroup cg1; ... ; endgroup
class c1;
    covergroup cg2; ... ; endgroup
    class c2;
        covergroup cg3; ....; endgroup
    endclass
endclass
generate if(1) begin: if_gen
covergroup cg4; ... ; endgroup
end
endgenerate
endmodule

```

For the given example, the following table elaborates the commands that can be used in the configuration file and their impact:

Command	Impact	Deselected Covergroup/s
(de) select_coverage -covergroup -module m2	(De)selects all non-embedded covergroups declared directly in the top-level design unit m2 and in generate block declared in m2.	m2::cg1, m2::cg4
(de) select_coverage -covergroup -class c1	(De)selects all class-embedded covergroups declared directly in the class c1.	m1::c1::cg2, m2::c1::cg2
(de) select_coverage -covergroup -cg_name cg1	(De)selects all non-embedded covergroups named "cg1" in the design	m1::cg1, m2::cg1
(de) select_coverage -covergroup -module m1 -class c1	(De)selects all class-embedded covergroups directly under the class c1 whose immediate parent scope is m1.	m1::c1::cg2
(de) select_coverage -covergroup -module m1 -cg_name cg1	(De)selects non-embedded covergroup named cg1 directly under m1.	m1::cg1

(de) select_coverage -covergroup -class c2 -cg_name cg3	(De)selects class embedded covergroups named cg3 directly under the class c2.	m1::c1::c2::cg3, m2::c1::c2::cg3
(de) select_coverage -covergroup -module m1 -class c1 -cg_name cg2	(De)selects class-embedded covergroup named cg2 directly under the class c1 whose parent scope is m1.	m1::c1::cg2
(de) select_coverage -covergroup -class c1::c2	(De)selects all class embedded covergroups declared in nested class c1::c2.	m1::c1::c2::cg3, m2::c1::c2::cg3
(de) select_coverage -covergroup -class c1 c2 -cg_name cg2 cg3	(De)select class embedded covergroup, cg2 and cg3, from the class c1 and c2. For unmatched covergroups, warning will be thrown at elaboration.	m1::c1::cg2, m2::c1::cg2, m1::c1::c2::cg3, m2::c1::c2::cg3 In this case, a warning will be issued for c1::cg3, c2::cg2.

Additional Information:

Typical use cases scenarios of the (de)select command are shown:

Command	Impact
(de) select_coverage -covergroup -class *...	(De)selects all covergroups from all classes.
(de) select_coverage -covergroup -class class_name	(De)selects all class embedded covergroups from a specified class.
(de) select_coverage -covergroup -module ubus_pkg	(De)selects all non-embedded covergroups included from the package ubus_pkg, such as those in monitors, drivers and so on.
(de) select_coverage -covergroup -cg_name <covergroup_name> -class <class_name>	(De)selects specific covergroup embedded in a specific class.
(de) select_coverage -covergroup -module ubus_pkg -class *...	(De)selects all class embedded covergroups for all classes included from the package ubus_pkg.

select_coverage -bet -instance *...	Selects the entire hierarchy for block, expression, and toggle coverage
deselect_coverage -be -instance top:vhdlttest:mctl...	Deselects hierarchy under the instance named <code>top:vhdlttest:mctl</code> . As a result, block and expression coverage will be ignored for the hierarchy under this instance.
select_coverage -e -instance top:vhdlttest:mctl:mctl:mem8x256	Enables only expression coverage for the instance named <code>top:vhdlttest:mctl:mctl:mem8x256</code>
deselect_coverage -b -module memctl	Deselects all instances of module <code>memctl</code>

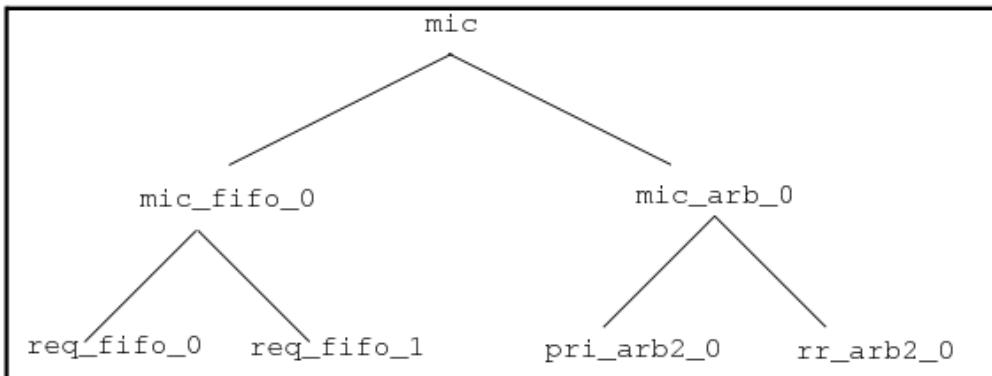
Removing Empty Instances From the Coverage Hierarchy

By default, ICC generates the complete design hierarchy even if the coverage for a few items is not scored and is not available. If there are many instances with no coverage, then the time taken to dump coverage data might increase significantly. You can reduce the coverage database dumping time by removing the instances from the design hierarchy where no coverage is found. To do so, use the following command in the coverage configuration file at elaboration:

```
deselect_coverage -remove_empty_instances
```

The above command removes the instances where self coverage and cumulative coverage is 0/0.

For example, consider the following hierarchy:



Assume that only functional coverage is scored and is available in all instances except `req_fifo_0` and `req_fifo_1`. For these instances, self coverage and cumulative coverage is 0/0. In this case, by default, the complete hierarchy will be dumped during simulation run. If the `deselect_coverage -remove_empty_instances` command is used in the coverage configuration file at elaboration, empty instances (`req_fifo_0` and `req_fifo_1`) will be removed while dumping the coverage hierarchy.

In the above example, assume that expression coverage is also scored along with functional coverage. If expression coverage is available in instance `req_fifo_0`, then only `req_fifo_1` will be removed while

dumping the coverage hierarchy.

set_inheritance_aware_class_coverage

A derived class inherits the methods (and therefore the blocks in the methods) of its base class. When you use the given command, the base class coverage items are reported in the base class as well as in its derived class(es) but are scored as per the reporting class objects.

With this command, blocks of the derived class as well as the inherited base class are reported. However, the coverage numbers reported for the inherited blocks correspond to the objects of the derived class only.

Syntax

set_inheritance_aware_class_coverage [-on]

Arguments

- -on turns on the separate reporting of the list of coverage items scored by the objects of a derived class and the objects of a base class. -on is the default behavior and an optional argument.

Example

Consider the given example:

```
// file package_base.sv
package package_base;
...
class packet;
    int payload [];
    int crc;
    int header;
    function do_crc();
        for (int i=0;i<payload.size();i++)
            begin
                crc= crc * payload[i];
            end
    endfunction

    function do_header(int src, int dest);
        header = src + dest;
    endfunction
endclass
...
endpackage

// file package_derived.sv
package package_derived;
...
class vlan_packet extends packet;
    int vlan_tag;
    virtual function do_vlan_tagging(int src);
        vlan_tag=src;
    endfunction
endclass
...
endpackage
```

Integrated Coverage User Guide

Coverage Configuration File (CCF) Commands--set_inheritance_aware_class_coverage

In the above example, the report for the derived class `vlan_packet` includes the blocks declared in `vlan_packet` and also the blocks inherited from the base class `packet`, as shown:

The screenshot displays two windows of the Cadence IMC tool interface, both titled "IMC (64) [Analysis - Code: package_derived::vlan_packet]".

Top Window: Shows the code for a CRC function:3 class packet;
4 int payload [];
5 int crc;
6 Int header;
7
8 function do_crc();
9 for (int i=0;i<payload.size();i++)
10 begin
11 crc = crc * payload[i];
12 end
13 endfunction

Bottom Window: Shows the code for a VLAN tagging function:19
20 class vlan_packet;
21 int vlan_tag;
22
23
24 virtual function do_vlan_tagging(int src);
25 vlan_tag = src;
26 endfunction
27 endclass

Points to Remember

When you use the `set_inheritance_aware_class_coverage` -on command:

- Merging of databases generated with and without this command is not allowed and an error is reported
- Refinement for a class generated from without this option onto a database generated with this option is not allowed and an error is reported.

set_com

COM identifies constant coverage items and marks them to be ignored. You can use the `set_com` command to mark the constant objects such as blocks, expressions, expression terms, and signals within the simulated DUT during elaboration or to apply COM to specific modules/instances of the DUT.

When you enable COM, `xmsim` marks the constant objects in the coverage database, and dumps this information in the `<workdir>/<scope>/<test>/icc.com` file. If this file already exists, the file is overwritten, except if the `-logreuse` option is specified.

Syntax:

`set_com`

In specific cases, to apply COM only to a subset, use:

```
set_com [-on | -off] [<coverages>] [-log | -logreuse] [-nounconnect] [[-module] <list> | -instance <list>]
```

where

```
coverages ::= [-block] [-expression] [-toggle] [-bet]
```

Arguments:

- `-on | -off` turns COM on or off. The default value is `-on`.
- `<coverages>` turns COM on/off for specified coverage types. If coverage types are not specified, the command applies to all supported coverage types for this command.
- `-log` ensures that the `icc.com` file is created. With multiple `set_com` commands in CCF, if the `-log` option is specified with one of the `set_com` commands, it applies to all; and the `icc.com` file is created for the complete design.
- `-logreuse` enables the reuse of the COM logfile, when specified in an existing `set_com` command. When you use this option for multiple simulation runs of the same UCM, the COM logfile will not be overwritten and will be reused, even if the contents of the COM file get changed.

The `-logreuse` option dumps the COM logfile with the UCM file at the `<cov_work>/<scope>` path, and the name of the COM file is the same as the name of the UCM file.

When the `-covmodeldir <modeldir>` option is also specified on the `xmsim` command-line, the COM file, similar to UCM, will be generated at the path specified by `-covmodeldir`. The link to this path will be available in `<cov_work>/<scope>/`.

- `-nounconnect` prevents marking of unconnected nets (in Verilog) and signals (in VHDL) as constants during COM analysis. With this option, unconnected nets and signals in a design are not treated as a

constant (treated as variables), and therefore are:

- not dumped to the [icc.com](#) file.
- excluded from toggle reports generated with the `-marked` option.

When multiple `set_com` commands are specified in CCF and if the `-nounconnect` option is specified with one of the commands, it applies to all.

- `-module` | `-instance` specifies if COM applies to all or specific modules and instances in the design. If not specified, `-module` is assumed.
- `<list>` specifies the list of modules/instances to which the command applies. If `<list>` is not specified, command applies to all modules in the design.

For example, to enable marking of signals in modules `mod1` and `mod2`, use:

```
set_com -on mod1 mod2
```

Points to Remember

- When you enable COM using the `set_com` command, it does not work with vector expression scoring that is enabled using `set_expr_scoring -vector`.
- When you use the `-logreuse` option, it is not mandatory to specify the `-log` option as the `-logreuse` option enables COM logfile generation as well as enables the reuse of the logfile.
- A toggle object can be reported as constant even if it is not in an instance for which COM is enabled through `set_com`. That would happen if the object has driver(s) common to the constant object in an instance for which COM is enabled.
- It is recommended to specify reuse option for Constant Object Marking (COM) logfile when you are sure that the multiple simulations are run on the same snapshot and commands in the elaboration configuration command file, especially the `set_com` and `set_com_interface` commands, are not changed. This is because a change in these commands can change the constant objects and COM content, while the logfile is not overwritten and may contain incorrect content. Therefore, you should cleanup all the COM logfiles when either the snapshot or elaboration configuration command file is changed.

 The `-nounconnect` option has been made default.

Exceptions:

The `set_com` command is not supported for the following datatypes:

Coverage Type	Datatype
Toggle	Structures
Toggle	Enum

set_com_interface

The `set_com_interface` command specifies the list of modules for which all ports are considered as variable. This command **must be used** when the DUT used for coverage is not the true DUT. This happens when only coverage is scored on a subset of DUT (less) or when coverage is also collected on part of the testbench (more).

Syntax:

```
set_com_interface <modules>
```

You can set top-level DUT for coverage using the `-covdut` option during elaboration. For COM analysis, all ports of the instances of DUT modules will be marked as variables. If `set_com_interface` and `covdut` modules are not top-level modules, all drivers coming from scopes in the design which are outside the scope of these module instance and its subtree are considered as variables.

If you specify both `set_com_interface` and `COVDUT`, preference will be given to the `set_com_interface` command, and ports of `COVDUT` modules will not be marked as variables by default.

Example:

Consider the given example:

Design file

```
module MODA(in1, in2, z);
  input in1, in2;
  output z;
  reg z;
  always@(in1 or in2)
  begin
    if(in1 == 1'b0)
      z <= 1'b1; // <BLK_I>
    if(in2 == 1'b1)
      z <= 1'b1; // <BLK_J>
  end
endmodule
module MODB(in1, in2, z);
  input in1, in2;
  output z;
  reg z;
  always@(in1 or in2)
  begin
    if(in1 == 1'b0)
      z <= 1'b1; // <BLK_K>
    if(in2 == 1'b1)
      z <= 1'b1; // <BLK_L>
  end
endmodule
module dut(a, z);
  input a;
  output z;
  reg r;
  wire w = 1'b0;
  MODA U1(a, w, z);
  MODB U2(a, w, z);
  always@(w)
    if(w)
      r <= 1'b1; // <BLK_M>
endmodule
module tb();
  reg r1;
  wire w1;
  initial
    r1 = 1'b0;
  dut U1(r1, w1);
endmodule
```

Coverage Configuration File

```
select_coverage -block
set_com -instance tb.U1.U1
set_com_interface dut
```

In the given example, block coverage will be reported for all of the design; that is, for `tb` and its hierarchy. COM analysis will be done only for `tb.U1.U1`. None of the blocks outside this scope will be analyzed.

The `set_com_interface` is specified as `dut`. As a result, the input `tb.U1.a` driven by `tb.r1` (outside `set_com_interface dut`), will be treated as a variable. COM analysis for instance `tb.U1.U1` will be

done as:

- <BLK_I> is controlled by expression (`in1 == 1'b0`) . Here, `in1` is driven from `tb.U1.a` which is a variable. Therefore, <BLK_I> will be reported for coverage.
- <BLK_J> is controlled by expression (`in2 == 1'b0`) . Here, `in2` is driven from `tb.U1.w` which is a constant (`1'b0`). Therefore, COM will report <BLK_J> as always inactive and will not be reported (ignored) in coverage reports.
- <BLK_K> , <BLK_L>, and <BLK_M> will not be analyzed because they are not specified under `set_com` . These will appear in coverage reports.

Consider specifying `-COVDUT MODA` in the `xmelab` command line, and remove the `set_com_interface` command from the CCF. This will cause both `tb.U1.U1.in1` and `tb.U1.U1.in2` to be treated as variables and no blocks will be reported as inactive by COM. COM analysis will be done as:

- <BLK_I> and <BLK_J> are controlled by variable expressions and will appear in coverage reports.
- <BLK_K> , <BLK_L> , and <BLK_M> will not be analyzed and will not appear in coverage reports as they are outside COVDUT.

If you specify `-COVDUT MODA` and keep `set_com_interface` also as DUT, then `-COVDUT` will not have any effect on COM.

- <BLK_I> will be reported as inactive and ignored from coverage reports.
- <BLK_J> controlled by variable expression will appear in coverage reports.
- <BLK_K> , <BLK_L> , and <BLK_M> will not be reported in coverage reports as they are outside COVDUT.

If both `set_com_interface` and `covdut` modules are not specified, ports of all top-level modules will be marked as variables.

i The `set_com_interface` command has no impact in the absence of the `set_com` command.

set_glitch_strobe

The glitch rejection time is used to analyze if a process is stable. A process is considered stable if it is not executed again before the specified glitch rejection time. Coverage counts for blocks and expressions within a process are incremented only if the process has been stable for more than the specified glitch rejection time. The `set_glitch_strobe` command enables you to specify a glitch rejection time interval.

Once specified, coverage simulation uses this information to determine when something has glitched. Events that are stable for at least the de-glitch time are recorded as coverage events and increment coverage counts. This helps in filtering signal glitches that can lead to artificially high coverage counts. A de-glitch time of one simulation base-unit does delta-delay de-glitching. The `set_glitch_strobe` command specified without any

arguments results in delta cycle de-glitching.

Syntax:

```
set_glitch_strobe [<time_number> <time_unit>]
```

Arguments:

- <time_number> is the numerical value of the glitch rejection time.
- <time_unit> is the unit of the glitch rejection time.

Example:

To set the glitch rejection time as 5 ns, use:

```
set_glitch_strobe 5 ns
```

When the process is executed, the coverage counts are recorded for blocks and expressions. If the process is executed again before 5 ns, then the process is not considered stable. Hence, the recorded coverage counts are ignored. If the process is executed after 5 ns, the coverage counts are incremented. By default, glitch rejection is turned off.

set_hit_count_limit

The `set_hit_count_limit` enables you to set the upper limit of hit counts during simulation run. When you specify the upper limit of hit counts and the hit count limit is reached during simulation, no more hits are added to the score counts.

Syntax:

```
set_hit_count_limit [<limit>]
```

Arguments:

< limit > is a numeric value in the range 1 - 240. The default limit is 1.



Points to Remember

An object is considered covered if hit count is at least 1.

set_subprogram_scoring

Verilog:

By default, coverage is scored for all Verilog subprograms (used and unused). The `set_subprogram_scoring` command disables the scoring of unused Verilog subprograms.

Syntax:

```
set_subprogram_scoring {-all | -used} <modules>
```

Arguments:

- -all enables scoring for both used and unused subprograms in modules specified using the <modules> option.
- -used enables scoring for only the used subprograms in modules specified using the <modules> option.
- <modules> lists modules to which the command applies. You can use the wildcard * to specify all modules in the design.

i An unused subprogram is a subprogram that is defined but never called. A subprogram that is used but within another unused subprogram is considered as unused.

VHDL:

By default, coverage is not scored for VHDL subprograms which are used in design scopes with block or expression coverage enabled. The `set_subprogram_scoring` command enables this scoring. This command covers only the subprograms defined in user-defined packages or design hierarchy instrumented for block or expression coverage. Further, it covers the subprograms that are called in the covered block or expression part of the design.

When you use this command, subprogram calls are searched for in all the design scopes marked for block or expression coverage and the subprograms that are defined in user-defined packages are reported for the coverage items that were applied on the calling object scope. These include subprogram calls from other subprograms.

Further, the subprograms that are called from different architectures are reported for all the coverage types applied on each of those scopes.

The format of the reported coverage data for the packages is similar to the format of reported data for architecture with the coverage items shown as a part of the package. The coverage data type for each subprogram in a package will be the sum of the coverage types of all calling scopes.

Syntax:

```
set_subprogram_scoring -vhdlpackage -used
```

Points to Remember

- In the current implementation, user cannot select or deselect a specific package for coverage instrumentation. If the user has specified only some entities or design hierarchy for coverage, the coverage data will be sum of the scoring from different parts of the design which may or may not have been specified for coverage. This is because no separate instance of the subprogram is created for each subprogram call. For example, if `func1()` has been called from architectures `arch1` and `arch2`, and the user specifies coverage for only `arch1`, the scoring for `func1` will be the sum of scoring from both `arch1` and `arch2`.
- When you merge multiple coverage databases, the data for a package present in a coverage database will be merged only if the data for that package is present in all the coverage databases being merged.
To better explain this, if a subprogram is used in design A and is not used in another design B, then the data for the subprogram will be reported in the database for design A and will not be reported in the database for design B. This will make both the databases structurally different and hence, the two databases cannot be merged.
- In incremental elaboration flow:
 - A package is instrumented only for the coverage types that are applied to the calling scopes in the primary part of the design. Further, if any extra coverage types are applied on a scope in the incremental part and a subprogram is called in this scope, the package will not be covered for these coverage types.
 - Only subprograms which have calls in scopes with block or expression coverage are reported, similar to the `set_subprogram_scoring-used` option for Verilog modules. So, it is not possible to generate a coverage report for all the subprograms in a package which has some unused subprograms

set_code_fine_grained_merging

The `set_code_fine_grained_merging` command provides merge resilience at concurrent block-level instead of the default module-instance-level. This command enables fine-grained merging in ICC by which merge is skipped only for the modified concurrent blocks in an instance.

Syntax:

```
set_code_fine_grained_merging [-ignore_vunit_code_coverage]
```

Argument:

- `-ignore_vunit_code_coverage` disables dumping of code coverage items in the verification units. This

argument is optional. With this argument, code coverage items (blocks and expressions) inside the verification units are not dumped to the coverage database.

Points to Remember

- When merging, the coverage databases generated with the `set_code_fine_grained_merging` command cannot be merged with one generated without the command.
- In the absence of the `set_code_fine_grained_merging` command (default behavior), if there is any change in the HDL code that affects the block or expression, then blocks or expressions are not merged for that instance.

set_refinement_resilience

The `set_refinement_resilience` command enables the support for refinement resilience at concurrent block (initial, always, process, and so on) level for block and expression coverage. By default, refinement resilience is supported at module/instance level for block and expression coverage.

With the `set_refinement_resilience` command, IMC continues to apply coverage refinements on blocks and expressions in those concurrent blocks that have not changed, even if other concurrent blocks in the same module/instance may have changed.

Syntax:

```
set_refinement_resilience
```

set_implicit_block_scoring

You can use the `set_implicit_block_scoring` command to disable the scoring of implicit `else` and default `case` blocks. By default, these blocks are scored in ICC.

Syntax:

```
set_implicit_block_scoring -off
```

In specific cases, to apply implicit scoring to specific modules, use:

```
set_implicit_block_scoring {-on | -off} [-if] [-case] [<modules>]
```

Arguments:

- `-on | -off` turns scoring ON or OFF of implicit else or default case blocks.
- `[-if] [-case]` enables/disables scoring of implicit else or default case blocks.
- `<modules>` specifies the name of the modules/entities to which implicit block scoring applies. If not specified, the command applies to the entire design.

Points to Remember

The `set_implicit_block_scoring` command has no effect on the `set_explicit_block_scoring` command.

set_explicit_block_scoring

You can use the `set_explicit_block_scoring` command to disable scoring of explicit Verilog `case default` and VHDL `case others`. By default, ICC scores all explicit Verilog case default and VHDL case others.

Syntax:

```
set_explicit_block_scoring -off
```

In specific cases, to apply explicit scoring to specific modules, use:

```
set_explicit_block_scoring {-off | -on} [<modules>]
```

Arguments:

- `-on` | `-off` turns scoring ON or OFF of explicit Verilog `case default` and VHDL `case others`.
- `<modules>` specifies the name of the modules/entities to which explicit block scoring applies. If not specified, the command applies to the entire design.

set_assign_scoring

The `set_assign_scoring` command enables you to score Verilog continuous assignments, which is by default disabled.

Syntax:

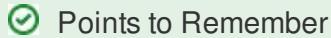
```
set_assign_scoring
```

set_branch_scoring

The `set_branch_scoring` command enables you to score individual branches, which is by default disabled. If the coverage configuration file includes this command, Verilog ternary assignment statements, VHDL conditional assignment statements, and VHDL selected signal assignment statements are also scored.

Syntax:

```
set_branch_scoring
```



Points to Remember

Branch coverage cannot be scored without enabling block coverage.

set_statement_scoring

The `set_statement_scoring` command enables statement coverage scoring. With this command, the

information related to number of statements within a block is also printed in the block coverage report.

Syntax:

set_statement_scoring

Points to Remember

Statement coverage cannot be scored without enabling block coverage.

deselect_macro

The `deselect_macro` command deselect coverage of blocks identified inside macros used in SystemVerilog classes.

Syntax:

```
deselect_macro -class [-name <list_of_macro_identifiers>]
```

where,

```
<list_of_macro_identifiers> ::= {space separated list of macro_text_identifier}
```

Arguments:

- `-class` is a mandatory option and specifies that macros are to be deselected inside SystemVerilog class scopes only.
- `-name` specifies the list of macros to be excluded from the block coverage.
- `<list_of_macro_identifiers>` specifies the list of macros to be deselected, matched by the macro text identifier. The following wildcards are supported:

Wildcard	Description	Example
*	Matches any text from current location until next delimiter or end of string	<code>deselect_macro -class -name SET_*</code> Matches all names beginning with <code>SET_</code> , such as <code>SET_CLK</code> , <code>SET_VAR1</code> , and so on.
?	Matches any single character	<code>deselect_macro -class -name OPCODE_?</code> Matches all names beginning with <code>OPCODE_</code> followed by a single character such as <code>OPCODE_1</code> and <code>OPCODE_2</code> .

Points to Remember

- The `deselect_macro` command is effective only if the block coverage of classes is enabled using the `select_coverage -b -class <list_of_classes>` command.
- The `deselect_macro` command is valid only for SystemVerilog classes.
- Currently, there is no corresponding `select_macro` command for the Coverage CCF file in `xmelab`.
- Multiple instances of the `deselect_macro` command can be specified in the same/different CCF files.
- When you use the `deselect_macro` command:
 - An error is reported by `xmelab` if the option `-class` is missing.
 - If the `-name` option is not specified, all the macros referenced inside classes will be deselected.
 - If the list of macro names is not specified with the `-name` option, `xmelab` exits with an error.

Examples:

The typical use case scenarios for the `deselect_macro` command are shown:

Command	Impact
<code>deselect_macro -class -name MY_MACRO_5</code>	Deselects block coverage for all blocks inside <code>MY_MACRO_5</code> in classes.
<code>deselect_macro -class -name *</code> <code>deselect_macro -class</code>	Deselects block coverage for all blocks in classes, which are inside a macro.
<code>deselect_macro -class -name SET_* GET_*</code>	Deselects block coverage for all blocks in classes, which are inside macros whose names begin with <code>SET_</code> or <code>GET_</code> .

set_expr_scoring

The `set_expr_scoring` command enables the coverage of expression in a design at various levels as specified using the different arguments.

Syntax:

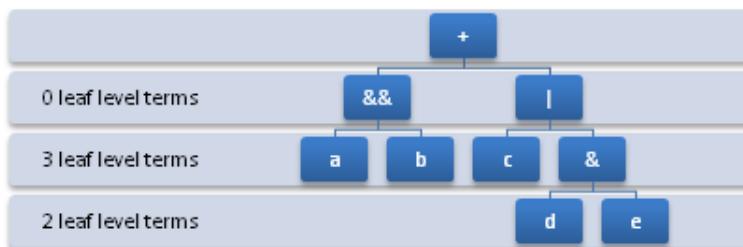
```
set_expr_scoring [-all] [-sop | -control | -vector | -fcc] [-event]
[-struct] [-no_vhdl_shortcircuit] [-vhdl_noreduce_table] [-vlog_remove_redundancy]
```

```
[ -vhdl_not_as_operator] [-max_expr_terms <num>] [-ungradeable] [-max_terms_sop < num >] [-  
max_terms_fcc <num>]  
[<module> | *]
```

Arguments:

- `-all` enables scoring of all Verilog and VHDL operators in conditions and assignments. See [Scoring of Operators in Different Modes](#) for details on operators scored with and without this option. If the `-all` option is used with any `set_expr_scoring` commands in the CCF, it applies to the complete design. It is recommended to use `set_expr_coverable_operators -all` and `set_expr_coverable_statements -all` commands to score all Verilog and VHDL operators in conditions and assignments.
- `-sop` enables SOP scoring of expressions. This is the default scoring mode. If no scoring mode is specified, SOP scoring is enabled.
- `-control` enables control scoring of expressions.
- `-vector` enables vector scoring of expressions.
- `-fcc` enables [FCC scoring](#) of expressions.
- `-max_expr_terms <maxlimit>` disables scoring of expressions with number of terms more than the user specified limit. Valid values of `maxlimit` are 16, 32, 64, 128, 256, 512, 1024, and 2048, and if any other value is specified a warning is reported. The `-max_expr_terms <maxlimit>` option applies to all the scoring modes - SOP, FCC, Control, and Vector.

The number of terms in an expression is the total number of leaf-level terms of the expression. Consider the expression $(a \&\& b) + (c | d \& e)$, which can be hierarchically represented as:



In the given example, the expression will be considered as an expression with 5(3+2) terms. The number of terms depends only upon the hierarchy of the expression irrespective of the option provided to score a particular operator.

Note: If `-max_expr_terms` and `-max_terms_sop` or `-max_terms_fcc` options are specified in the CCF file, the `-max_expr_terms` option will be the first-level check, and the `-max_terms_sop` or `-max_terms_fcc` option will be applied only to the expressions that are found to be scorable by `-max_expr_terms`.

- `-ungradeable` enables marking and reporting of ungradeable expressions. Ungradeable expressions are the expressions within a module and all instances of the module in which the number of terms in the expression is more than the maximum limit as specified in the `max_expr_terms` option. When you use the `-ungradeable` option, the ungradeable expressions are marked with `grade : UNG-Expr` in IMC report.
- `-event` enables scoring of Verilog events.

Note: The `-event` option has been deprecated and will not be supported in the future releases. To score Verilog events, use `set_expr_coverable_operators - event_or` and `set_expr_coverable_statements - event_control` commands

- `-struct` enables scoring of expressions containing union and struct datatypes.
- `-no_vhdl_shortcircuit` disables short-circuit evaluation of VHDL AND/NAND and OR/NOR operators for BIT / BOOLEAN types. If this option is used with any of the `set_expr_scoring` commands in the CCF, it applies to the complete design. For more details on short-circuit evaluation, see [Evaluation of VHDL AND/NAND and OR/NOR operators of BIT and BOOLEAN Types](#).
- `-vhdl_noreduce_table` creates new expression coverage tables for AND/OR short circuit operators in the SOP scoring mode. These tables are amenable for coverage hole analysis and UNR flow. For example, the VHDL AND and OR operator tables without using this option are shown:

```
((CYCLECOUNT /= "111") and (CYCLECOUNT /= "011")) and (CYCLECOUNT /= "100")
<-----1-----> <-----2-----> <-----3----->
```

index	hit	<1>	<2>	<3>	
2.1.1	1	0	-	-	'and'
2.1.2	1	-	0	-	
2.1.3	1	-	-	0	
2.1.4	1	1	1	1	

```
((CYCLECOUNT /= "111") or (CYCLECOUNT /= "011")) or (CYCLECOUNT /= "100")
<-----1-----> <-----2-----> <-----3----->
```

index	hit	<1>	<2>	<3>	
2.1.1	1	1	-	-	'or'
2.1.2	1	-	1	-	
2.1.3	1	-	-	1	
2.1.4	1	0	0	0	

However, with the `-vhdl_noreduce_table` option the VHDL AND and OR operator tables appear as:

```
((CYCLECOUNT /= "111") and (CYCLECOUNT /= "011")) and (CYCLECOUNT /= "100")
<-----1-----> <-----2-----> <-----3----->
```

index	hit	<1>	<2>	<3>	
					'and'
2.1.1	1	0	-	-	
2.1.2	1	1	0	-	
2.1.3	1	1	1	0	
2.1.4	1	1	1	1	

```
((CYCLECOUNT /= "111") or (CYCLECOUNT /= "011")) or (CYCLECOUNT /= "100")
<-----1-----> <-----2-----> <-----3----->
```

index	hit	<1>	<2>	<3>	
					'or'
2.1.1	1	1	-	-	
2.1.2	1	0	1	-	
2.1.3	1	0	0	1	
2.1.4	1	0	0	0	

- `-vlog_remove_redundancy` disables splitting of subexpressions that use logical equality (`==0`) or logical inequality (`!=0`) operators when determining the terms of the primary expression. It applies only to [SOP scoring mode](#). If this option is used with any of the `set_expr_scoring` commands in the CCF, it applies to the complete design.

Note: The `-vlog_remove_redundancy` option has been deprecated and will not be supported in the future releases.

- `-vhdl_not_as_operator` enables scoring of VHDL `not` operator. By default, VHDL `not` operator (and its operand) is treated as an expression term, and therefore, VHDL `not` operator is not scored. If this option is used with any of the `set_expr_scoring` commands in the CCF, it applies to the complete design.

Note: The behavior of the `-vhdl_not_as_operator` option will be made default in the future releases.

- By default, a maximum of 1024 terms are allowed in an expression marked for SOP coverage. You can change this limit using the `-max_terms_sop <num>` option. `-max_terms_sop <num>` sets the maximum number of terms in an expression marked for SOP coverage as `num`, where `num` is a positive integer. If this option is used with the `set_expr_scoring` command in the CCF, it applies to the complete design.

⚠ The `-max_terms_sop < num >` option has been deprecated and will not be supported in the subsequent release. It is recommended to use `-max_expr_terms <num>` to set the maximum number of terms in an expression.

- By default, a maximum of 10 subterms are reported in an expression marked for FCC coverage. You can change this limit using the `-max_terms_fcc <num>` option. `-max_terms_fcc` sets the maximum number of subterms in an expression marked for FCC coverage as `num`, where `num` is a positive integer. If this option is used with the `set_expr_scoring` command in the CCF, it applies to the complete design.
- Note:** For reduction operators `^`, `~^`, and `^~`, if the number of bits is more than the specified limit, only scalar scoring is done. You can increase this limit to a maximum of 20 terms.

- `<module> | *` specifies the modules on which the selected scoring mode is to be applied. If list of

modules is not specified, the command impacts all modules in the design.

Points to Remember

- By default, coverage is scored only for Verilog logical operators (`||` and `&&`) and VHDL logical operators (`OR`, `AND`, `NOR`, and `NAND`), and is scored only in condition expressions. This is regardless of the scoring mode specified, or event scoring enabled.
- Vector scoring is not supported for VHDL design units. If vector scoring is specified for VHDL design units, SOP scoring is used.
- When you use the `-vhdl_noreduce_table` option:
 - The number of minterms or rows or coverage space may increase for expressions containing a mix of `AND/OR` operators.
 - Coverage analysis operations, like merging and refinement, are not resilient for expression coverage of corresponding container instances across coverage databases generated with and without this option.
 - An error is reported if VHDL short-circuit evaluation (scoring) is turned-off using the `set_expr_scoring -no_vhdl_shortcircuit` command.
 - A warning is reported when non-SOP mode (control or FCC) is specified.
- The `-max_terms_sop < num >` option does not apply in incremental elaboration.
- If there are multiple `set_expr_scoring` commands in the configuration file, the command has a cumulative effect. Consider the following coverage configuration file commands:

```
select_coverage -module -expr *
set_expr_scoring -control test*
set_expr_scoring -sop test7
```

With the above commands, control scoring applies to all modules that start with `test` except for module `test7`.

set_expr_coverable_operators

You can use the `set_expr_coverable_operators` command with different options to enable coverage of various Verilog and VHDL operators. By default, coverage is scored only for Verilog logical operators (`||` and `&&`) and VHDL logical operators (`OR`, `AND`, `NOR`, and `NAND`).

Syntax:

```
set_expr_coverable_operators [-all] [-bitwise] [-relational] [-conditional] [-reduction] [-event_or]
[-logical_not]
```

Arguments:

The following table lists the various operators supported with different options of the `set_expr_coverable_operators` command:

Command Option	Verilog Operators	VHDL Operators
<code>-all</code>	All Verilog operators except events	All VHDL operators except VHDL NOT
<code>-bitwise</code>	<code>~, &, , ^, ^~, and ~^</code>	XOR and XNOR
<code>-relational</code>	<code>>, >=, <, ===, !==, ==, !=, and <=</code>	<code><, <=, >, >=, =, and /=</code>
<code>-conditional</code>	<code>?</code>	N/A
<code>-reduction</code>	<code>&, ~&, , ~ , ^, ~^, and ^~</code>	N/A
<code>-event_or</code>	<code>@(,) or @(or)</code>	N/A
<code>-logical_not</code>	<code>!</code>	NOT

set_expr_coverable_statements

You can enable the coverage of operators in various Verilog and VHDL statements using the `set_expr_coverable_statements` command.

Syntax:

```
set_expr_coverable_statements [-all] [-procassign] [-contassign] [-event_control] [-misc]
```

Arguments:

- `-all` enables scoring of expressions in all supported statements.
- `-procassign` enables scoring of expressions in procedural statements for Verilog and in sequential statements for VHDL.
- `-contassign` enables scoring of expressions in continuous assignment for Verilog and in concurrent assignment for VHDL.
- `-event_control` enables scoring of expressions in event.
- `-misc` enables scoring of expressions in the remaining statements, including as bit select, rand case, and rand sequence.

set_toggle_strobe

Toggle coverage by default considers glitches for toggle recording. This can lead to artificially high coverage counts. The `set_toggle_strobe` command enables you to define a strobe interval for filtering glitches within the time window on nets. With glitch filtering enabled, a transition is recorded when a net transitions from one stable value to another stable value.

Syntax:

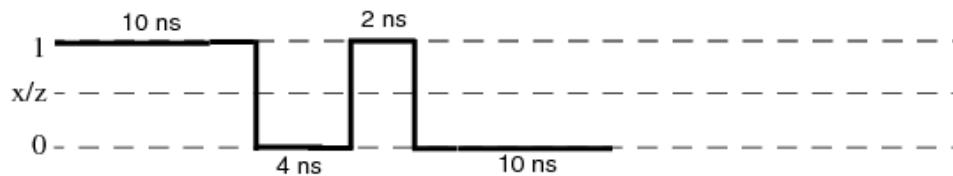
```
set_toggle_strobe <time_number> <time_unit>
```

Arguments:

- `<time_number>` is a positive integer to define the strobe interval (one or more time ticks).
- `<time_unit>` specifies the unit for the time specified in `<time_number>`. The `<time_unit>` can be `fs`, `ps`, `ns`, `us`, `ms`, and `s`.

Example:

Consider the following diagram:



If toggle strobe is specified as 3 ns then:

- The number of rise transitions is recorded as 0.
- The number of fall transitions is recorded as 1 (1 -> 0 at 10 ns).

If toggle strobe is specified as 1 ns then:

- The number of rise transitions is 1 (0 -> 1 at 14 ns).
- The number of fall transitions is 2 (1 -> 0 at 10 ns, 1 -> 0 at 16 ns).

set_toggle_limit

Toggle limit is the maximum count for rise and fall transitions reported by `xmsim`.

The `set_toggle_limit` command enables you to specify a toggle limit.

Syntax:

```
set_toggle_limit <limit>
```

Arguments:

- `<limit>` is a numeric value in the range of 1 and 2147483647. The default limit, if not specified, is 1.
However, note that the limit does not determine if an item is covered.
A count of 1 always is considered as covered.

set_toggle_includex

By default, ICC records transitions `1 -> 0` and `0 -> 1`. However, you can record transitions `x -> 0` and `x -> 1` using the `set_toggle_includex` command.

Syntax:

```
set_toggle_includex
```

Example:

Consider transition `0 -> x -> 1 -> 0 -> x -> 1`.

In the absence of the `set_toggle_includex` command:

- The number of rise transitions is 0.
- The number of fall transitions is 1 (`1 -> 0`).

With this command:

- The number of rise transitions is 2 (`x -> 1` and `x -> 1`).
- The number of fall transitions is 1 (`1 -> 0`).

set_toggle_includez

By default, ICC records transitions `1 -> 0` and `0 -> 1`. However you can record transitions `z -> 0` and `z -> 1` using the `set_toggle_includez` command.

Syntax:

```
set_toggle_includez
```

Example:

Consider transition $0 \rightarrow x \rightarrow z \rightarrow 1 \rightarrow 0$.

In the absence of `set_toggle_includez` and `set_toggle_includex` commands:

- The number of rise transitions is 0.
- The number of fall transitions is 1 ($1 \rightarrow 0$).

With the `set_toggle_includez` command:

- The number of rise transitions is 1 ($z \rightarrow 1$).
- The number of fall transitions is 1 ($1 \rightarrow 0$).

set_toggle_noports

You use the `set_toggle_noports` command to disable scoring of module ports, so that a net is scored only once within a hierarchy. When you use this command, signals are scored only at the highest level.

Syntax:

```
set_toggle_noports
```

Example:

Consider the following code:

```
module tb ();
    reg r;
    wire op;
    modA i1(r, op);
endmodule

module modA (in, op)
    input in;
    output op;
    ...
    reg op;
    ...
endmodule
```

With the `set_toggle_noports` command:

- `reg r` and `wire op` in module `tb` are scored.
- ports `in` and `op` of module `modA` are not scored.

set_toggle_portsonly

The `set_toggle_portsonly` command enables you to score only module ports when you are checking interfaces and connectivity of building blocks.

Syntax:

```
set_toggle_portsonly
```

Example:

Consider the code shown for the `set_toggle_noports` command. With the `set_toggle_portsonly` command:

- module ports `in` and `op` for module `modA` are scored.
- `reg r` and `wire op` in module `tb` are not scored.

 Points to Remember

The `set_toggle_noports` and `set_toggle_portsonly` commands are mutually exclusive.

set_toggle_scoring

The `set_toggle_scoring` command enables scoring and reporting of SystemVerilog enumerations and multi-dimensional static arrays and vectors that are not scored by default.

Syntax:

```
set_toggle_scoring ([(-sv_enum[-sv_struct_with_enum])] | [(-sv_mda [<max_bits_base2_exponent>] [-ungradeable] [-sv_mda_of_struct]) [-sv_generate])|(-regeot)
```

Arguments:

- `-sv_enum` enables scoring of enumerated data types.
- `-sv_enum [-sv_struct_with_enum]` enables the scoring of enumerated members declared inside SystemVerilog structures. When you use this option, the enumerations declared as members inside struct data types are scored independent of the states of other members in that structure.
- `-sv_mda [<max_bits_base2_exponent>]` enables scoring for packed and unpacked multi-dimensional vectors and static arrays for reg, logic, bit, and wire SystemVerilog data types inside modules and interfaces. In this command, `max_bits_base2_exponent` specifies the maximum numbers of bits that can be scored. It is an optional argument with a default value of 12. It can accept only 10, 12, 14, 16, 18, 20, 22, and 24 as its valid values, and specifying any other value leads to an error.
- `-ungradeable` enables reporting of ungradeable MDA signals. An MDA signal is considered ungradeable if its size exceeds the limit specified with the `-sv_mda` option. In this case, MDA member makes even its parent struct object ungradeable. When you use the `set_toggle_scoring -ungradeable` command for struct MDA signals with size greater than the allowed limit, only the struct objects in the model are reported as ungradeable and their members are not reported. However, for struct MDA signals with size lesser than the allowed limit, only the struct members are reported and the struct object is not reported.
- `-sv_mda_of_struct` enables scoring of members of packed and unpacked multidimensional static arrays (MDA) of struct declared inside modules and interfaces. With this option, the optional argument `-max_bits_base2_exponent` specified with `-sv_mda` also applies to `-sv_mda_of_struct`. When you use the `-sv_mda_of_struct` option, the number of bits for an MDA struct object is calculated as the sum total of number of bits of all the members that are supported for toggle coverage multiplied by array width of the struct object.

- `-sv_generate` enables toggle coverage of registers and nets inside all 'for-generate'/'if-generate'/'case-generate' blocks for the Verilog and SystemVerilog modules in which toggle coverage is enabled.
- `-regeot` samples register at the end of the simulation in toggle coverage mode.

Points to Remember

- COM analysis is not supported for SystemVerilog enumerated signals and multi-dimensional arrays.
- Toggle coverage for multidimensional signals with greater than “ $2^{\max_bits_base2_exponent}$ ” numbers of bits is not supported.
- If you initialize an enumerated variable or net to its default value at the beginning of the simulation, it will not be scored. This is because each bit of a SystemVerilog 2 state variable is assigned a default value of `1'b0` while each bit of a 4 state variable is assigned a value `2'bx`. An open net is assigned a value of `1'bz`.
- Reporting of toggle coverage of enumerated signals and multi-dimensional arrays is not supported in ICCR. It is currently supported only with IMC.
- In case of parameterized multi-dimensional arrays, if the number of bits of the signal exceeds the maximum limit in any of the instances of the module, toggle coverage of that signal is not supported across all instances of the module.
- If the size of the multi-dimensional array signal exceeds the limit in some instances and is within the limit in others, then for instances where the signal size is within the limit is supported with the `set_parameterized_module_coverage` CCF command.
- If the size of a multi-dimensional array in a struct exceeds the limit, toggle coverage of the entire struct across all instances of the module will not be supported.
- `-bit_exclude` option of CCF command `set_toggle_excludefile` is not supported for multi-dimensional arrays.

set_toggle_excludefile

By default, if toggle coverage is scored, all signals in the design are scored.

The `set_toggle_excludefile` enables you to exclude specific signals from the design for toggle coverage during simulation.

Syntax:

```
set_toggle_excludefile [-nolog] [-bitexclude] <file>
```

Arguments:

- `-nolog` ensures that no log file is generated to dump the list of signals that were excluded for toggle

coverage. In the absence of this option, a log file named [toggle_exclude.log](#) is generated.

- `-bitexclude` allows exclusion of bit selects of vector signals. With this option, the patterns in the exclude file are matched against the expanded names of each vector bit.
- `<file>` specifies the file that contains the list of signal names or patterns that should be excluded from toggle coverage. The `<file>` can be absolute paths (for example, `/home/ master/design/exfile`), relative paths (for example, `design/exfile`, `./exfile`), or just the filename (for example, `exfile`). It can also include environment variables. For example, in the following commands, environment variable `SRC_AREA` is used.

```
set_toggle_excludefile ${SRC_AREA}/exfile  
set_toggle_excludefile /home/master/${SRC_AREA}/exfile
```

Points to Remember

- The `-bit_exclude` option of the `set_toggle_excludefile` command is not supported for enums, and multi-dimensional packed and unpacked arrays.
- The `set_toggle_excludefile` command is supported only at uppermost struct object level of multi-dimensional packed and unpacked arrays of struct. Consider the following example:

```
typedef struct{  
    logic [1:0][1:0]a;  
    logic [1:0]b[0:1];  
}struct_t;  
  
typedef struct {  
    logic n_a[1:0][0:1];  
    struct_t n_st[1:0];  
}nested_struct_t;
```

For `struct_t st_one[1:0]` , `st_two[1:0][2:0]` , `nested_struct_t nest_st` in module top, refer to the following table:

Content in <file> provided to <code>set_toggle_excludefile</code>	Allowed
<code>module top.st_one</code> <code>module top.st_two</code>	Yes
<code>module top.st_one[1]</code> <code>module top.st_two[1][1]</code>	No
<code>module top.st_one[1].a</code> <code>module top.st_two[1][1].b</code> <code>module top.st_two[1][0].a</code>	No
<code>module top.nest_st.st</code>	No

Additional Information:

Formats for specifying signals in the exclude file:

Single Signal Specification

The signals can be specified in the exclude file by using either a single signal specification or a multiple signal specification. The format for single signal specification is:

```
[-ere] {instance|module} <signal_path>
```

Multiple Signal Specification

The formats for multiple signal specification are

[**-ere**] **module_signals** <*module_name*> <*siglist*>

or

[**-ere**] **instance_signals** <*instpath*> <*siglist*>

where

- **instance | module** specifies whether the signal to be excluded is from instance or module within the design.
- < **signal_path** > specifies the signal with module name or instance path. The formats for specifying signal path are:
 - Explicit signal path for one signal
 - Signal path with wildcards ? and *
 - Signal path with extended regular expressions (ERE)
- **-ere** keyword at the beginning specifies that signal paths are expressed as extended regular expressions. With EREs:
 - Special characters like [] () . * in the actual name must be escaped with \.
 - ? denotes a match to any character.
 - * repeats the previous character any number of times.

ERE searches will add a "^" at the beginning and a "\$" at the end of the specified ERE pattern, if these characters are not already there. This ensures that the description only matches complete path.

When defining patterns with the **-ere** keyword, VHDL portions of the pattern should be in lowercase regardless of how it is defined in the VHDL code. Verilog portions of the pattern should match with the one defined in the Verilog code. In the absence of the **-ere** keyword, VHDL portions of the pattern can be in upper or lower case.

- **module_signals** or **instance_signals** are keywords that are used to indicate if the signals are to be excluded from a module or an instance, respectively.
- < **module_name** > specifies the name of the module.
- < **instpath** > specifies hierarchical name of the instance.
- < **siglist** > specifies the list of signals in the specific module or instance.

Exclude Internal Signals

To exclude internal signals (signals that are not ports) from a module or instance, use:

```
[-ere] -internal [-skip_interfaces] {module|instance} <module_name|instance_path>
```

If you specify the `-internal` keyword with the `module` or `instance`, you must specify the `module_name` or `instance_path`, respectively. In this case, all the internal signals in the specified module or instance are excluded. You can also use `-skip_interfaces` to avoid exclusion of internal signals of SystemVerilog interfaces that might match the module/instance name specified in the given command.

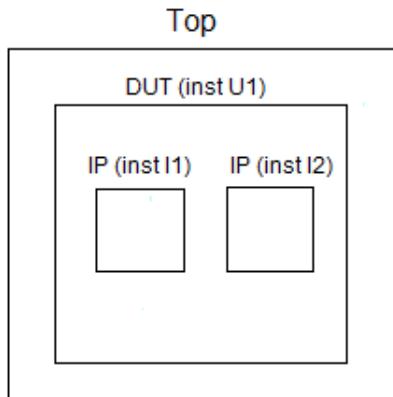
The syntax to exclude internal signals is valid for both Verilog and VHDL. When you use this syntax:

- Signals excluded through “`-internal`” will not cause exclusion of the net(s) that is(are) directly connected to the excluded net(s).
- If a net is excluded using the `-internal` keyword, then there is no effect if the same net is excluded explicitly. Consider the given example of commands included in the toggle exclude file:

```
-internal instance top.U1  
instance top.U1.sig
```

In the given example if `top.U1.sig` is an internal signal, then it is excluded by the first command. The second command does not have any effect, and this is reported in a warning. Consequently, the hierarchy connected to this signal is not excluded as it is excluded through `-internal`.

Consider the given example:



In the given example:

<code>-internal module DUT</code>	Excludes internal signals of module DUT
<code>-internal instance top.U1.I1</code>	Excludes internal signals of specific instance I1 of DUT (<code>top.U1</code>)

-internal module top...	Excludes all the internal signals in instance <code>top</code> , including internal signals of module <code>top</code> itself.
-ere -internal instance top\U1\..*	Excludes all the internal signals in hierarchy under instance <code>top.U1</code> (excluding <code>top.U1</code> itself)
-internal instance top.U1...	Excludes all the internal signals in hierarchy under instance <code>top.U1</code> (including <code>top.U1</code> itself)



- Specifying the `-internal` option in exclude file will have no impact if the `set_toggle_portsonly` command is specified in the CCF.
- If the `set_toggle_noports` command is specified in the CCF and the exclude file contains `-internal` option specified for `module|instance`, then no signal will be scored for that particular `module|instance`.

Exclude Signals Declared inside SystemVerilog generate Blocks

To exclude signals declared inside SystemVerilog generate blocks, use:

- `[-ere] {generate instance path}.<signal_name>`
- `[-ere] instance_signals<generate instance path><siglist>`

where

- `generate instance path` is the hierarchical name of the generate instance.



Type-based using the parent module/generate block name exclusion is not supported.

- ⓘ Use of regular expressions impacts performance. So, to exclude signals of a common module or instance, it is recommended to use one exclusion in multiple signal specification format instead of multiple exclusions in single signal specification format. For instance, a single command in the multiple signal specification format as follows:

```
-ere instance_signals tb\$.s0\.clt0\.clw\.cl\.pcw0\.pc\(\..*\)* .*pc_reset.*  
.shift_en.* .*clk_en_b.* .*tst_clk.* .*contention_block.*.*retain_b.*
```

will show better performance results than the given multiple commands in the single signal specification format:

```
-ere instance tb\$.s0\.clt0\.clw\.cl\.pcw0\.pc\(.*\.)*.*pc_reset.*  
-ere instance tb\$.s0\.clt0\.clw\.cl\.pcw0\.pc\(.*\.)*.*shift_en.*  
-ere instance tb\$.s0\.clt0\.clw\.cl\.pcw0\.pc\(.*\.)*.*clk_en_b.*  
-ere instance tb\$.s0\.clt0\.clw\.cl\.pcw0\.pc\(.*\.)*.*tst_clk.*  
-ere instance tb\$.s0\.clt0\.clw\.cl\.pcw0\.pc\(.*\.)*.*contention_block.*  
-ere instance tb\$.s0\.clt0\.clw\.cl\.pcw0\.pc\(.*\.)*.*retain_b.*
```

When you specify a signal to be excluded:

- All of the nets above or within its sub-hierarchy that are directly connected to this net are also excluded except enums.
- All of the wires that form a net will be excluded if any of the wires of the net are excluded.

Consider the given example:

```
module modA(input A);  
    wire wA = A;  
endmodule  
  
module top();  
    wire w1;  
    modA U1(.A(w1));  
endmodule  
  
//ccf  
set_toggle_excludefile ex  
  
// exclude file 'ex'  
instance top.w1  
  
//toggle_exclude.log  
top.w1  
top.U1.A
```

In the given example, `top.w1` and `top.U1.A` are excluded. However, `wire top.U1.wA` is not excluded.

- For enumerated nets, each individual net must be specified for exclusion.
- Individual bits of an enumerated net/variable cannot be excluded.

Verilog examples (Use "." as the separator)

instance top.i0.a	Excludes signal a in the instance top.i0 .
instance top.*...	Excludes all signals in instance top and its descendants.
instance A.B.C.*	Excludes all signals in instance A.B.C .
instance tp*tp3.a	Excludes signal a from all the instances starting with tp and ending with tp3 . For example, exclude tpmytp3.a and tp42tp3.a . It does not include instances like tp.tp1.tp3 .
module fifo.S*	Excludes all the signals starting with s from module fifo .
module mem.addr[1?]	Excludes the signals addr[10:19] of module mem .
instance test.mem*.data[*1]	Excludes the signals data[1] , data[11] , data[21] and so on of instances test.mem1 , test.mem2 , test.mem3 and so on.
-ere module fifo_.*\.S.*	Excludes all the signals starting with s of all the modules starting with fifo_ .
-ere instance tp.*tp3\.a	Excludes signal tp42tp3.a and tp.tp1.tp3.a . In addition, any hierarchical signal starting with tp and ending with tp3.a .
-ere module fifo_[2-4]\..*	Excludes all signals from module fifo_2 , fifo_3 , and fifo_4 .
-ere instance test\.mem\.data\[.\]	Excludes signals data[0:9] of instances test.mem1 , testmem2 , test.mem3 and so on.
-ere module mem\.addr\[1[2-6]\]	Excludes the signals addr[12:16] of module mem .
-ere instance tb\.mywire\[([1-7][0-9])\]	Excludes bits 10 to 79 .
-ere instance tb\.mywire\[(([0-9]) ([1-9][0-9]) (1[0-1][0-9]) (1[2][0-7]))\]	Excludes 0-127 bits.
-ere instance top\.crc\.crc_mmrs\.wreg3_0\.r_var\[((0) (2) (3))\]	Excludes bits 0 , 2 , or 3 of signal r_var .

- ⓘ The range given in the `ere` pattern works only in increasing order. For example, `[4-7]`. Decreasing the order does not work with the `ere` pattern. In addition, all `ERE` patterns work similar to the `egrep` command in UNIX.

VHDL examples (Use ":" as the separator)

instance :o*	Excludes all signals starting with <code>o</code> in top-level instance.
instance :Add:clock	Excludes <code>clock</code> from the VHDL hierarchy <code>:Add</code> .
instance :Add:*...	Excludes all signals in instance <code>:Add</code> and its descendants.
instance :tp*tp3:a	Excludes signal <code>a</code> from all the instances starting with <code>tp</code> and ending with <code>tp3</code> . For example, exclude <code>:tpmytp3:a</code> and <code>:tp42tp3:a</code> . It does not include instances like <code>:tp:tp1:tp3</code> .
module t1:*	Excludes all signals from module <code>t1</code> .
module lib3.fa(fa_str*:a	Excludes signal <code>a</code> from all architectures starting with <code>fa_str</code> of entity <code>fa</code> in library <code>lib3</code> .
module lib1.E1(*) :*	Excludes all signals of all architecture of entity <code>E1</code> in library <code>lib1</code> .
module MUX:SELECT(?1)	Excludes the signals <code>SELECT(11)</code> , <code>SELECT(21)</code> , <code>SELECT(31)</code> and so on of entity <code>MUX</code> .
-ere instance :11:Fa[13]:ha1:.*	This will not work because the pattern should be in lowercase regardless of how it is defined in the VHDL code. To exclude all signals from instance <code>:11:fa1:ha1</code> and <code>:11:fa3:ha1</code> , the pattern should be modified as: -ere instance :11:fa[13]:ha1:.*
-ere instance :tp.*tp3:a	Excludes signal <code>:tp42tp3:a</code> and <code>:tp:tp1:tp3:a</code> . In addition, any hierarchical signal starting with <code>:tp</code> and ending with <code>tp3:a</code> .
-ere module fa+:b	Excludes signal <code>b</code> from modules <code>faa</code> , <code>faaa</code> and so on (<code>a</code> should be present one or more times).
-ere instance :top:dut.*:sel\[4-7]\)	Excludes signals <code>sel(4)</code> to <code>sel(7)</code> of instances starting with <code>:top:dut</code> .

Mixed-Language examples (VHDL portion should be in lowercase and Verilog portion must be in actual case)

-ere instance :dkm_i:coin_counter_i.dimes\[[2- 6] \]	Excludes signals dimes[2] to dimes[6] of instance dkm_i:coin_counter_i .
-ere instance :dkm_i:dime_in	Excludes all signals in instance dkm_i:dime_in .
-ere instance :dkm_i.quarter_in	Excludes all signals in instance dkm_i.quarter_in . Note: Use a dot (.) to match any character. It also denotes a separator for the Verilog portion. Therefore, when using a dot to indicate a separator, it must be escaped with \.
instance :Add.clock	Excludes clock from hierarchy :Add .

With the `set_toggle_excludefile` command, a file named `toggle_exclude.log` gets created at the end of elaboration, which stores a list of full path of all of the signals that got excluded through `set_toggle_excludefile` command. This file gets created in the working directory from which `xmelab` is run. This file will not be created if the `set_toggle_excludefile` command does not result in any exclusions.

- i** For VHDL and mixed language, if you want to use full design along with library name use " . " as the separator, as shown below:

```
<library>.<entity>(<architecture>):<signal>
```

To exclude signals `a*` from entity-architecture pair `TRY_OUT(ARC_OUT)` in library `WORKLIB`, use:
`module WORKLIB.TRY_OUT(ARCH_OUT) :a*`

set_toggle_smart_refinement

The `set_toggle_smart_refinement` command enables IUS to dump connectivity information of nets into the coverage database. The information in the coverage database enables IMC to exclude all the nets that are connected to the net being excluded during analysis time. This command enables the IMC to exhibit the similar exclusion behavior for connected nets as that of `set_toggle_excludefile` CCF command during elaboration stage.

Syntax:

```
set_toggle_smart_refinement
```

Points to Remember

The `set_toggle_smart_refinement` command does not work for unpacked arrays of nets.

set_fsm_attribute

The `set_fsm_attribute` command enables you to tag an FSM state vector.

Syntax:

```
set_fsm_attribute -tag <tag> -module <module> -statereg <state_reg>
```

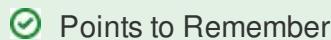
Arguments:

- `-tag <tag>` specifies a tag name, which acts as an identifier for the FSM. FSM tags are case-sensitive.
- `-module <module>` is the name of the module (design entity) containing the FSM.
- `-statereg <state_reg>` specifies the state vector that will be uniquely identified with this FSM. A state vector specified with the `-statereg` option must be physically declared in the design.

Example:

To define a tag named `FSM1` for a state vector `current_state` defined in the module `top`, use:

```
set_fsm_attribute -tag FSM1 -module top -statereg current_state
```



Points to Remember

You cannot extract unsupported FSM styles by tagging them using the `set_fsm_attribute` command.

select_fsm

The `select_fsm` enables you to select modules and tagged FSMs for FSM extraction. You can specify multiple `select_fsm` commands within a configuration file.

Syntax:

```
select_fsm [ -module <modules> | -tag <tags>]
```

Arguments:

- `-module <modules>` specifies the modules or design entities for which FSMs should be extracted. Currently, only wildcard `*` is supported. Wildcard `...` is not supported for inclusion of child instances.
- `-tag <tags>` specifies tags for one or more FSMs that should be extracted. You define a tag by using the `set_fsm_attribute` command.

Examples:

To specify modules `mod1` and `mod2` for FSM extraction, use:

```
select_fsm -module mod1 mod2
```

To specify all modules in the design for FSM extraction, use:

select_fsm

OR

select_fsm -module *

To specify module mod1 and all tags in the design for FSM extraction, use:

select_fsm -module mod1 -tag *

 **Points to Remember**

You can also use the `select_coverage` command to select FSMs. It is the preferred and easy to use command. The `select_fsm` command will be discontinued soon. In addition, you cannot use the (de) `select_coverage` command and the (de) `select_fsm` command together in the design, as shown below:

CCF contents:

```
select_coverage -fsm -module mod1 mod2  
select_fsm -module mod3
```

Such usage is prohibited. Use either the `select_coverage` command or the `select_fsm` command to make relevant selections.

deselect_fsm

The `deselect_fsm` command enables you to ignore modules and tagged FSMs from FSM extraction.

Syntax:

```
deselect_fsm [-module <modules> | -tag <tags>]
```

Arguments:

See [select_fsm](#) for syntax details.

Points to Remember

- You can also use the `deselect_coverage` command to deselect FSMs. It is the preferred and easy to use command. The `deselect_fsm` command will be discontinued soon. In addition, you cannot use the `(de) select_coverage` command and the `(de) select_fsm` command together in the design. See [select_fsm](#) for usage examples.
- You can also specify multiple `deselect_fsm` commands within a configuration file. If there are multiple `select_fsm` and `deselect_fsm` commands in a configuration file, then each command builds on the previous.

Examples:

To ignore modules `mod1` and `mod2` for FSM extraction, use:

```
deselect_fsm -module mod1 mod2
```

To exclude FSMs in all modules in the library `testbench`, use:

```
deselect_fsm -module testbench.*
```

set_fsm_reset_scoring

By default, ICC does not score reset states and transitions. The `set_fsm_reset_scoring` command enables scoring of reset states and transitions for identified FSMs.

Syntax:

```
set_fsm_reset_scoring
```

set_fsm_arc_scoring

By default, ICC does not score FSM arcs. The `set_fsm_arc_scoring` command enables scoring of arcs for identified FSMs.

Syntax:

```
set_fsm_arc_scoring [-on | -off] [ -module <modules> | -tag <tags>] [-no_delay_check]
```

Arguments:

- `-on | -off` enables/disables scoring of arcs. If not specified, scoring is `-on` by default.
- `-module <modules>` specifies the modules or design entities for which arc scoring should be enabled/disabled.
- `-tag <tags>` specifies tags for one or more FSMs for which arc scoring should be enabled/disabled. You define a tag by using the `set_fsm_attribute` command.
- `-no_delay_check` disables checking for delays. This option can be used for the cases when the delays do not impact the results.

The `set_fsm_arc_scoring` command can be specified multiple times in a CCF file. If this command is specified multiple times, then the processing happens in the order of their occurrence and a consolidated list of modules/tagged FSMs is created for which arc scoring should be enabled/disabled. To support per-module selection of arc coverage, FSM must be selected for instrumentation using the `select_fsm` command.

Points to Remember

- If both `-module` and `-tag` are used, `-module` overrides `-tag`.
- For backward compatibility, `set_fsm_arc_scoring` can be specified without any arguments. This will be equivalent to:
`set_fsm_arc_scoring -on -module *`

set_fsm_arc_termlimit

The FSM extraction engine uses a boolean engine to extract the SOP tables for the input expressions for each state transition. At times, the input expressions are very complex and result in the creation of many arc expressions. This affects elaboration performance and memory usage. To prevent performance degradation, a default limit 131072 has been set on the number of input terms that get created for an arc expression.

You can increase/decrease the term limit using the `set_fsm_arc_termlimit` command.

Syntax:

```
set_fsm_arc_termlimit <limit>
```

Arguments:

<limit> is a numeric value in the range of 1 to 2147483647. The default limit, if not specified, is 131072. Increasing the term limit beyond 131072 may degrade elaboration performance and/or memory usage.

 **Points to Remember**

Note that ICC reports a warning if the number of terms exceeds the default term limit or the one specified using the `set_fsm_arc_termlimit`. For such FSMs, arc coverage is disabled for the given state register and only transition coverage is reported.

set_fsm_scoring

By default, ICC does not score FSM hold transitions (`s0 -> s0`). The `set_fsm_scoring` command enables scoring of FSM hold transitions.

Syntax:

```
set_fsm_scoring -hold_transition
```

 **Points to Remember**

Scoring hold transitions might impact performance.

set_optimize

The `set_optimize` command enables you to perform various optimizations that help improve your coverage performance.

Syntax:

```
set_optimize {-vlog_prune_on | -ial_ovl_inst_asrt | -top_expr_non_scorable | -prune_covergroup}
```

Arguments:

- `-vlog_prune_on` - Improves the performance for expression coverage by collecting the expression coverage in an optimized manner. When you use the `-vlog_prune_on` option, the expression coverage runs in a mode cognizant of simulation options, and due to this expression coverage is collected only for the blocks that are triggered for a changing output during simulation cycles.

Consider the given example:

```
always @( a , b )
  If ( a && b )
    out <= a ;
```

In the given example, case `out` is dependent on `a` and `b` per sensitivity list. However for a given cycle, if "`a`" does not toggle and "`b`" toggles then "`out`" has no change, and hence always block does not trigger

in the optimize mode at all by the simulator. For this cycle, coverage is not collected for the expression. When you use the `set_optimize -vlog_prune_on` command, there is no additional impact on any other coverage functionality.

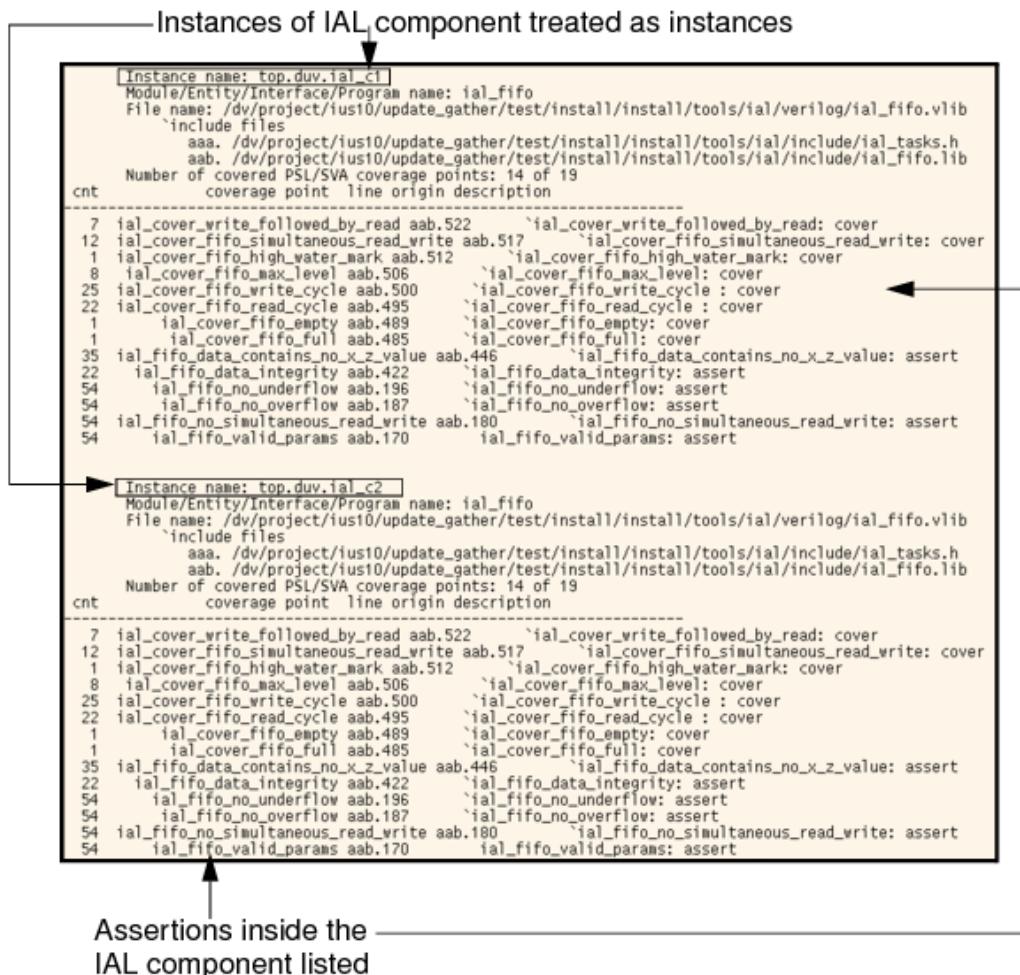
- `-ial_ovl_inst_asrt` - By default, instances of IAL and OVL components are treated as instances and assertions inside the IAL/OVL components are listed individually in the coverage report. If the design includes a huge number of IAL and OVL components (with a large number of assertions inside them), the coverage report becomes cluttered.

To reduce the clutter from the coverage report, you can use the following command in the coverage configuration file at elaboration:

```
set_optimize -ial_ovl_inst_asrt
```

With the above command, ICC will treat instances of IAL and OVL components as individual assertions. In addition, ICCR will report rolled-up count of all of the assertions inside the component along with the instance that is treated as assertion.

Consider the following coverage report.



In the above report, instances `ial_c1` and `ial_c2` of IAL component `ial_fifo` are considered as instances and all of the assertions inside `ial_fifo` are listed in the report. This is the default ICCR behavior.

If the `set_optimize -ial_oval_inst_asrt` command is used in the coverage configuration file at elaboration, then the coverage report will show instances `ial_c1` and `ial_c2` of IAL component `ial_fifo` as single assertion entities. The counts are shown as the cumulative coverage (rolled-up counts) of all of the assertions inside `ial_fifo` as shown below:

Instance name: top.duv Module/Entity/Interface/Program name: test_fifo File name: /home/ruchikas/WORK/Ruchika_Coverage/testcases_functionalcov/IAL_simple2/test.v Number of covered PSL/SVA coverage points: 2 of 2 coverage point line origin description				
cnt	ial_c1	36	ial_c1 {	
350	ial_c2	51	ial_c2 {	
350				

Instances of IAL component shown as single assertion entities

Cumulative coverage of all of the assertions

- **-top_expr_non_scorable** - The `set_optimize -top_expr_non_scorable` command optimizes the simulation run time and memory of a Verilog/SystemVerilog design if:
 - Expression coverage is enabled in control scoring mode using the `set_expr_scoring -control` CCF command and,
 - Expression coverage is NOT enabled for all operators in the design using the `set_expr_scoring -all` or `set_expr_coverable_operators -all` CCF command
- **-prune_covergroup** - The `set_optimize -prune_covergroup` command optimizes the `xmsim` CPU usage for SystemVerilog covergroup coverage. When you use this command, coverage sampling is triggered and sampled for the first time. Subsequently, the covergroup is sampled only if there is a change in any of the components (inputs) of coverpoint expressions in the covergroup.
 The optimizations are not enabled if:
 - Value of `option.at_least > 1` for any coverpoint/cross at the time of the construction of covergroup instance.
 - Covergroup contains any guard (`iff`) condition or any transition bin.
 - Coverpoint expression contains an automatic lifetime variable, such as a non-static member of a class or a covergroup argument.

Points to Remember

- Expression coverage of coverage databases saved with and without the `set_optimize -top_expr_non_scorable` CCF command/option cannot be merged.
- When you use the `set_optimize -prune_covergroup` command:
 - In IMC, if a bin is expected to be covered '*n*' times (where '*n*')>0) it is reported to be covered '*p*' times (where '*n*'>='*p*')>0).
 - There is no impact on the overall coverage of either bin or coverpoint or covergroup or rolled up coverage % numbers due to the fact that coverage % is a rollup of (covered / uncovered).
 - There is no impact on return values of the pre-defined `get_coverage/get_inst_coverage` methods.
 - Procedural assignment of `option.at_least` is not allowed for covergroup/coverpoint/cross eligible for this optimization, and a simulation time warning is displayed.

select_functional

The `select_functional` command enables scoring of all compiled assertions and covergroups. If `xmelab -covdut` switch is used to provide one or more major blocks for scoring, then functional coverage is scored only within these blocks.

Syntax:

```
select_functional [-ams_control | -imm_asrt_class_package]
```

Arguments:

- `-ams_control` - Enables scoring of assertions inside AMS modules, which are not scored by default.
- `-imm_asrt_class_package` - Enables reporting of immediate assertions inside a class in a package, which are not reported by default.

Points to Remember

- With the `select_functional -ams_control` command, assertion coverage is scored for AMS modules. Other coverage types (code, FSM, and covergroup) are not yet supported for AMS modules in the design hierarchy.
- Functional coverage does not score:
 - System Verilog immediate assertions inside a class declared in a module.
 - Instance-based coverage of immediate assertions inside a class.

set_covergroup

Syntax:

```
set_covergroup [-optimize_dump] [-show_uninstantiated] [-new_instance_reporting ] [ -  
default_cross_num_print_missing <num> ][-default_goal <num>][[-default_type_option_goal <num>]  
[-merge_instances_on|merge_instances_off][-cross_auto_bin_max_default_zero ][-  
per_instance_default_one][-detect_overlap][-bin_merge][-bin_name_merge][-merge_cp_by_name][-  
instant_sampling ]
```

Arguments:

- `-optimize_dump` - If a design includes class embedded covergroups and the design has many class objects, then the time taken to dump coverage data might increase significantly. The `-optimize_dump` option reduces the database generation time and generates data related to all covergroup instances in the design or testbench to the coverage database.
- `-show_uninstantiated` - Reports coverage for uninstantiated covergroups. By default, type-based coverage for uninstantiated covergroups is not reported. In this case, covergroups instantiated within conditional statements that are not executed at simulation time are also considered uninstantiated. For more details on type-based coverage, see [Chapter 6, "Functional Coverage"](#).
- `- new_instance_reporting` - Enables new naming conventions for covergroup objects, including:
 - **Embedded Covergroups Inside UVM Component**
The UVM Component hierarchy is reported as part of the IMC instance tree, which is rooted at `uvm_pkg`, and shown as:


Consider the given example:

```
module top;  
...  
class monitor extends uvm_monitor;  
    covergroup cov_trans @cov_transaction;
```

```
...
endgroup : cov_trans

function new (string name="monitor", uvm_component parent = null);
    super.new(name, parent);
    cov_trans = new();
endfunction // new
endclass // monitor

class env extends uvm_env;
    monitor monitors[];
    ...
function void build();
    super.build();
    monitors = new[2];
    monitors[0] = monitor::type_id::create("RX_mon",this);
    monitors[1] =
        monitor::type_id::create("TX_mon",this);
endfunction // void
...
endclass // env

class my_test extends uvm_test;
    env top_env;
    ...
function void build();
    super.build();
    top_env = env::type_id::create("my_env",this);
endfunction
...
endclass // my_test

initial
    run_test("my_test");
endmodule // top
```

In the given example, class monitor is a user defined class, which is derived from uvm_monitor and has embedded covergroups. There are two objects of this monitor class with UVM name as RX_mon & TX_mon. The instance-based report for the given example is shown. In this report, two covergroup objects with the name cov_trans are reported under UVM component uvm_pkg.uvm_test_top.my_env.RX_mon and uvm_pkg.uvm_test_top.my_env.TX_mon scope, respectively.

```
Instance name: uvm_pkg.uvm_test_top.my_env.RX_mon
Type name: uvm_pkg
```

Name	Average, Covered Grade	Line	Source Code
cov_trans	18.75%, 12.00% (3/24)	31	covergroup cov_trans @cov_transaction;

...

```
Instance name: uvm_pkg.uvm_test_top.my_env.TX_mon
Type name: uvm_pkg
```

Name	Average, Covered Grade	Line	Source Code
cov_trans	18.75%, 12.00% (3/24)	31	covergroup cov_trans @cov_transaction;

...

- o  Embedded Covergroups Inside Non-UVM Component within UVM Hierarchy
The non-UVM components within UVM hierarchy can be of two types:

- User-defined objects - These objects of User-Classes are instantiated inside UVM-Hierarchy.
- UVM-Objects - UVM-Objects are not part of the UVM Component hierarchy, though they have references from multiple UVM Components like sequencer, driver, and so on.

The covergroup objects are reported as a 3-level instance hierarchy under an IMC Instance tree, as shown below:

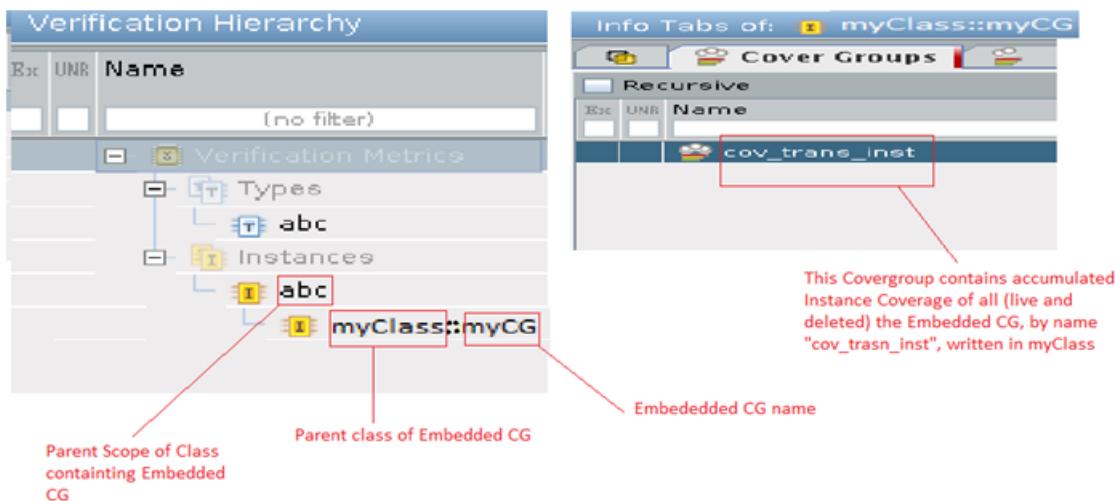
3-Level Instance Hierarchy

```
Level 1: Parent Scope Name of Class containing Embedded CG
|_ Level 2: Parent Class Name of Embedded CG::Embedded CG name
|_ Level 3: CG object Name
```

Consider the given example:

```
package abc;
  class myClass extends uvm_sequence_item; /* UVM Object */
    covergroup myCg;
      ...
    endgroup
  endclass
endpackage
```

In this example, instance coverage of all the covergroup objects of covergroup `myCg` are reported under the node `myClass::myCg` under the package node `abc` as shown:



- Instance coverage of multiple covergroup objects of same type and name will be accumulated and reported under a common name.
- Instance coverage of deleted covergroup objects (inside UVM Objects) will be merged with instance coverage of other covergroup objects of same type and name.

Consider the given example in which the embedded covergroup is inside UVM-object and the name of the embedded covergroup is same in all UVM-objects:

```

package arithmetic ;

class instruction extends uvm_sequence_item; /* UVM Object */
  typedef enum {PUSH_A,PUSH_B,ADD,SUB,MUL,DIV,POP_C} inst_t;
  rand inst_t inst;
  covergroup cg_instruction ;
    option.per_instance = 1 ;
    coverpoint inst ;
  endgroup
  `uvm_object_utils_begin(instruction)
    `uvm_field_enum(inst_t,inst, UVM_ALL_ON)
  `uvm_object_utils_end

  function new (string name ="instruction");
    super.new(name);
    cg_instruction = new ;
    /* set_inst_name() not done */
    // cg_instruction.set_inst_name("my_cg_instruction") ;
  endfunction
endclass

class operation_addition extends uvm_sequence #(instruction);
  instruction req;
  function new(string name="operation_addition");
    super.new(name);
  endfunction

```

```
'uvm_sequence_utils(operation_addition, instruction_sequencer)

virtual task body();
/* creating an object of sequence item i.e instruction */
req = instruction::type_id::create("req");
wait_for_grant();
assert(req.randomize() with {inst == instruction::PUSH_A; });
send_request(req);
wait_for_item_done();

/* creating an object of sequence item i.e instruction */
req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::PUSH_B;
send_request(req);
wait_for_item_done();
//get_response(res);

/* creating an object of sequence item i.e instruction */
req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::ADD;
send_request(req);
wait_for_item_done();

/* creating an object of sequence item i.e instruction */
req = instruction::type_id::create("req");
wait_for_grant();
req.inst = instruction::POP_C;
send_request(req);

wait_for_item_done();
endtask
endclass
endpackage : arithmetic
```

In the given example, “class instruction” is derived from `uvm_sequence_item` and it also contains an embedded covergroup “`cg_instruction`”. The objects of “`cg_instruction`” are created in the `body()` of a class `operation_addition` and transferred to other `uvm_component`. After the transfer of request response is complete, another transaction is initiated and a new object is created that results in the deletion of the previous object. Similarly, the objects of class instruction get created and destroyed throughout the simulation. In the given example, four objects of class instruction get created and by the end of simulation three of them get destroyed. So, by the end of simulation only one covergroup object with the name `cg_instruction` is left. So, by the end of simulation the instance coverage of three dead and one live covergroup objects is reported in an accumulated form of a single container.

The instance-based IMC report of all the covergroup objects is shown:

```

Level 1: arithmetic
|_ Level 2: instruction::cg_instruction
|_ Level 3: cg_instruction

Instance name: arithmetic.instruction::cg_instruction
Type name: arithmetic
Name          Average, Covered Grade    Line   Source Code
-----
cg_instruction 57.14%, 57.00%(4/7)  19 (sequence_item.sv)covergroup cg_instruction ;
|--A            57.00% (4/7)        21 (sequence_item.sv) A : coverpoint inst {
|--b[PUSH_A]    100.00% (1/1)      22 (sequence_item.sv) bins b[] = {[ $$ ]} ;
|--b[PUSH_B]    100.00% (2/1)      22 (sequence_item.sv) bins b[] = {[ $$:$ ]} ;
|--b[ADD]       100.00% (2/1)      22 (sequence_item.sv) bins b[] = {[ $$:$:$ ]} ;
|--b[SUB]       0.00% (0/1)        22 (sequence_item.sv) bins b[] = {[ $$:$:$:$ ]} ;
|--b[MUL]       0.00% (0/1)        22 (sequence_item.sv) bins b[] = {[ $$:$:$:$:$ ]} ;
|--b[DIV]       0.00% (0/1)        22 (sequence_item.sv) bins b[] = {[ $$:$:$:$:$:$ ]} ;
|--b[POP_C]     100.00% (2/1)      22 (sequence_item.sv) bins b[] = {[ $$:$:$:$:$:$:$ ]} ;

```

The default name of all the objects in the given example is `cg_instruction`. Any name that you specify will be used as covergroup object name in the report.

↳ Covergroups within Module/Instance Hierarchy

The covergroup objects within module/instance hierarchy can be:

- Non-embedded covergroup type declared inside module/package
- Embedded covergroup of a user-defined type

In covergroups within module/instance hierarchy:

- If the name of the covergroup object is not explicitly specified then the name of the covergroup object is *<name of the object/module hierarchy>. <name of the covergroup type>*.
- If a name is specified for the covergroup object, then the name of the covergroup object is *<name of the object/module hierarchy>. <user-specified name>*.

Consider the given example in which there are two covergroup objects, one inside module instance hierarchy and another inside object hierarchy with the names `cgi` and `ob.CG`, respectively. In this case, the scope for both the objects is test as both the objects are declared in module test.

```

module test;
  covergroup CG1;
  ...
endgroup : CG1

class abc ;
  covergroup CG;
  ...
endgroup : CG

function new ;
  CG = new ;
  endfunction
endclass : abc

abc ob = new; /* CG inside module */
CG1 cgi = new ; /* CG inside object hierarchy inside a module */
endmodule

```

The instance-based report of the example as it appears in IMC is shown:

Name	Average, Covered Grade	Line	Source Code
cgi	100.00%, 100.00% (1/1)	7	covergroup CG1
--CP1	100.00% (1/1)	9	CP1 : coverpoint v1 {
--b1	100.00% (11/1)	10	bins b1 = {0, 1};
ob.CG	100.00%, 100.00% (1/1)	7	covergroup CG
--CP1	100.00% (1/1)	9	CP1 : coverpoint v1 {
--b1	100.00% (11/1)	10	bins b1 = {0, 1};

- Additional Supported Scenarios

Some of the additional scenarios supported using the `set_covergroup -new_instance_reporting` command are discussed.

- Embedded Covergroup Inside Task and Functions

Consider the given example:

```
module test;
int i=0;
reg[3:0] v1=0;
class bottle;
bit v ;
covergroup CG1 ;
option.per_instance = 1;
    CP1 : coverpoint v{
        bins b1[] = {0, 1};
    }
endgroup : CG1
endclass : bottle
function bottle create_bottle();
    bottle ob = new; // declaration of object containing embedded CG
    return ob;
endfunction
bottle B ;
initial begin
    #1 create_bottle(); // call function to get the object containing CG
    #2 $finish();
end
endmodule
```

The instance-based report for the given example is shown:

Name	Average, Covered Grade	Line	Source Code
create_bottle.ob.CG1	100.00%, 100.00% (2/2)	8	covergroup CG1 ;
--CP1	100.00% (2/2)	10	CP1 : coverpoint v{
--b1[0]	100.00% (1/1)	11	bins b1[] = {0, 1};
--b1[1]	100.00% (1/1)	11	bins b1[] = {0, 1};

- Embedded Covergroups Inside Base-Derived Class Hierarchy

When derived class extends a base class with an embedded covergroup, covergroup object present in the base class is also inherited and is visible in the derived class. In this case the covergroup object is named with respect to the derived class. Consider the given example:

```

module test;
  reg [1:0] v1=0;
  reg clk =0;
  class Base ;
    int x;
    covergroup CG1@(clk);
      option.per_instance = 1;
      CP1 : coverpoint v1 {
        bins b1[] = {0, 1};
      }
    endgroup
    function new(int y);
      x = y ;
      CG1= new ;
    endfunction
  endclass : Base
  class Derived extends Base;
    int x;
    covergroup CG2 @ (clk);
      option.per_instance = 1;
      CP2 : coverpoint v1 {
        bins b1[] = {0, 1};
      }
    endgroup : CG2
    function new(int y);
      super.new(y-1);
      x = y;
      CG2 = new;
    endfunction
  endclass : Derived
  Derived D = new(2); // instantiates Covergroup CG1 & CG2
endmodule

```

The instance-based report for the given example is shown:

```

Instance name: test
Type name: test
File name: /servers/scratch03/irfanc/cg_full_path_name/testcase/non_uvm_base_derived_2/test1.v
Number of covered cover bins: 2 of 4
Number of uncovered cover bins: 2 of 4
Number of excluded cover bins: 0

Name          Average, Covered Grade   Line  Source Code
D.CG1          50.00%, 50.00% (1/2)   8     covergroup CG1@(clk);
| --CP1        50.00% (1/2)           10    CP1 : coverpoint v1 {
| | --b1[0]     100.00% (4/1)         11    bins b1[] = {0, 1};
| | --b1[1]     0.00% (0/1)          11    bins b1[] = {0, 1};
D.CG2          50.00%, 50.00% (1/2)   22    covergroup CG2 @ (clk);
| --CP2        50.00% (1/2)           24    CP2 : coverpoint v1 {
| | --b1[0]     100.00% (4/1)         25    bins b1[] = {0, 1};
| | --b1[1]     0.00% (0/1)          25    bins b1[] = {0, 1};

```

↳ Named Embedded Covergroup Object in Module/Instance Hierarchy

Consider the given example with an embedded covergroup object inside a module. In this example, the name specified is the complete logical name of the CG object, `ob.CG`. However, based on the naming rules, the cg-object name is replaced with the name explicitly specified instead of the complete logical name, so the name of the covergroup Object is reported as `ob.obj.cg_inst`. Therefore, the names of covergroup objects must be specified carefully to avoid any confusion.

```

module test;
class abc ;
covergroup CG;
...
endgroup : CG
    function new (string obj_name);
        CG = new ;
            CG.set_inst_name({obj_name, ".cg_inst"});
    endfunction
endclass : abc
abc ob = new ("obj"); /* CG inside module */
endmodule

```

The instance-based report for the given example is shown:

Instance name: test

Name	Average, Covered Grade	Line	Source Code
ob.obj.cg_inst	100.00%, 100.00% (1/1)	7	covergroup CG;
--CP1	100.00% (1/1)	9	CP1 : coverpoint v1 {
--b1	100.00% (1/1)	10	bins b1 = {0, 1};

- `-default_cross_num_print_missing` - Enables you to specify the default value of the `cross_num_print_missing` covergroup option.

By default, all the uncovered automatically generated cross tuples/bins are reported in the coverage detailed report. To restrict the number of cross tuples/bins that are reported, use the following command at elaboration:

`set_covergroup -default_cross_num_print_missing <value>`, where `value` must be a non-negative integral value.

The above command sets the default value to `<value>`, which applies to the all of the covergroups in the design.

For example, to set the default value of the `cross_num_print_missing` option to 5, use the following command:

`set_covergroup -default_cross_num_print_missing 5`

With the above command, five uncovered automatically generated cross tuples/bins will be printed in the coverage report whenever the `cross_num_print_missing` option is not set. If the `cross_num_print_missing` option is set, then the number of uncovered automatically generated cross tuples/bins printed in the report is controlled by the `cross_num_print_missing` option..

For more details on the covergroup options, see [Specifying Coverage Options](#).

- `-default_goal` - By default, the value of covergroup option `goal` is set to 100. The `-default_goal <value>` option enables you to override the default value of the covergroup option `goal`, where `value` must be a non-negative integer. The goal set using this option applies only to the covergroup items for which the goal is not explicitly set in the covergroup declaration.

For example, consider the following covergroup declaration:

```
covergroup cg;
option.goal = 90;
    cp1: coverpoint a;
    cp2: coverpoint b;
    crs_1: cross a,b;
endgroup
```

In the above code, the goal for the covergroup `cg` is set as `90`, and therefore `90` will be used as the target goal for calculating covergroup coverage. However, no goal is set for the coverpoints `cp1` and `cp2`, and cross `crs_1`. As a result, the default goal `100`, will be used as the target goal for coverpoints `cp1` and `cp2`, and cross `crs_1`. If `goal` is set as `80` using the command `set_covergroup -default_goal 80`, then `80` will be used as the target goal for calculating coverage of coverpoints `cp1` and `cp2`, and cross `crs_1`.

- ⓘ The goal used for calculating covergroup coverage will still be `90` because it is explicitly set in the covergroup declaration.

For more details on the covergroup options, see [Instance-Specific Covergroup Options](#).

- `-default_type_option_goal` - By default, the value of the covergroup type option `goal` is set to `100`. The `-default_type_option_goal <value>` option enables you to override the default value of the covergroup type option `goal`, where `value` must be a non-negative integral value. The `goal` set using this option applies to all the covergroup types for which the goal is not explicitly set in the covergroup declaration.

Consider the following covergroup declaration:

```
covergroup cg1 @(`clk);
type_option.goal = 40;
    c1: coverpoint opcode{
        type_option.goal = 50;
    }
    c2: coverpoint address;
endgroup : cg1
```

In the above code:

- Goal for the covergroup type `cg` is set as `40`, and therefore `40` will be used as the target goal for calculating covergroup coverage.
- Goal set for coverpoint `c1` is `50`, and therefore `50` will be used as the target goal for calculating coverage of coverpoint `c1`.
- No goal is set for coverpoint `c2`, and therefore the default goal `100` will be used for calculating coverage of coverpoint `c2`.

- If the goal is set as 85 using the `set_covergroup -default_type_option_goal 85` command, 85 will be used as the target goal for calculating coverage of coverpoint `c2`.

For more details on the covergroup type options, see [Type-Specific Covergroup Options](#).

- `-merge_instances_on` - Sets `type_option.merge_instances=1` for all covergroup types in the complete design. When specified, this option takes precedence over the value of the `merge_instances` type option as specified in specific covergroups.
- `-merge_instances_off` - Enables you to override `type_option.merge_instances=1` for all covergroup types in the complete design. When specified, this option takes precedence over the value of the `merge_instances` type option as specified in specific covergroups.
- `-cross_auto_bin_max_default_zero` - When coverage is computed for cross coverage bins, both user-defined cross bins and automatically-generated cross bins are included. The `-cross_auto_bin_max_default_zero` option enables you to exclude automatically generated cross bins from coverage computation for all the crosses in the design. With this option, automatically created cross bins are neither saved in the coverage database nor included in the coverage report.

Note: The `cross_auto_bin_max_default_zero` option is not a part of IEEE Std 1800™-2012.

- `-per_instance_default_one` - By default, covergroup option `per_instance` is set to 0. When the `per_instance` option is not specified or set to 0 in the covergroup declaration, instance coverage of the covergroup is not reported. To enable instance coverage of the covergroup, the `per_instance` option must be explicitly set to 1.

The `-per_instance_default_one` option enables you to change the default value of the option `per_instance` to 1 for all those covergroups where this option is not set explicitly inside the covergroup. When you use this option, the instance coverage of covergroups is reported for all the covergroups where `option.per_instance` is not set explicitly. If the value is set explicitly, instance coverage is reported as per setting of the option `per_instance`.

For more details on the `per_instance` option, see [Example: Using per_instance Option](#).

- `-detect_overlap` - By default, the value of `detect_overlap` is set to 0 and a warning is not generated for any overlap in the range list or the transition list of bins of a coverpoint. You can generate a warning for a coverpoint by setting the `detect_overlap` option to 1.

The `detect_overlap` option applies only to the coverpoint in which it is set. You can use the `set_covergroup -detect_overlap` command to enable the `detect_overlap` option for the entire design. With this command, a warning is generated for any overlap in range list or transition list in the entire design.

Note: The `detect_overlap` option traverses each coverpoint in the design to detect any overlap in the

range list or transition list of its bins. As a result, there might be performance overhead if the design contains a large number of bins that are present in a single coverpoint.

- **-bin_merge** - By default, the merge operation performs a merge at the coverpoint level. In this case, the merge of the complete coverpoint is ignored if any of the bins within that coverpoint differs across runs. Therefore, to merge coverpoints you can perform a merge at bin-level by using the following command in the coverage configuration file at elaboration:
- ```
set_covergroup -bin_merge
```

When you use this command, the bins with valid names and values in the secondary runs are merged to the bins target/resultant model. Note that the bins that have been modified across runs are not merged, and a merge conflict is reported along with the reason for conflict, such as change in bin description or missing bin in resultant model. For more information, refer to Merge Behavior if Bin-Level Merging is Enabled in the the *Integrated Metrics Center User Guide* in Metric-Driven Verification (MDV) release.

 The **-bin\_merge** option will be deprecated in a future release. It is recommended to use **-bin\_name\_merge** for name-based merging.

- **-bin\_name\_merge** - By default, the merge operation performs a merge at the coverpoint level. In this case, the merge of the complete coverpoint is ignored if any of the bins within that coverpoint differs across runs. Therefore, to merge coverpoints you can perform a merge at bin-level by using the following command in the coverage configuration file at elaboration:

```
set_covergroup -bin_name_merge
```

With the **-bin\_name\_merge** option, bins with the same name are merged irrespective of the value set that they represent and also irrespective of their bin definition. Consider the given example:

```
CP1 : coverpoint x {
 bins b1= {[low:high]} ; //scalar bin
 bins b2[size]= {1:10} ; //fixed size vector bin
}
```

In the given example for the scalar bin, if the value of `low:high` is `1:3` for run1 and `2:5` for run 2 with the same bin name, `b1`, bins are merged irrespective of the different value sets. In case of the given fixed size vector bin, if the value of size is `3` and `5` for run1 and run2, respectively, bin names that are common across run are merged as shown:

| <b>Run 1 Bins</b> | <b>Run 2 Bins</b> | <b>Merge</b> | <b>Data (Count)</b> |
|-------------------|-------------------|--------------|---------------------|
| b1                | b1                | Yes          | Both                |
| b2 [0]            | b2 [0]            | Yes          | Both                |
| b2 [1]            | b2 [1]            | Yes          | Both                |

|        |        |     |      |
|--------|--------|-----|------|
| b2 [2] | b2 [2] | Yes | Both |
| -      | b2 [3] | No  | NA   |
| -      | b2 [4] | No  | NA   |

When you use this option, name-based merging is done for all kinds of bins. During the merge, the bin names that are reported by the analysis tool are considered and the RTL names are ignored. In this case, model of the primary database is used as a reference, which means that only the sampled bins from secondary database that exist in primary database are merged.

- **-merge\_cp\_by\_name** - The **-merge\_cp\_by\_name** option enables merging coverpoints with the same name that were specified on different expression in two runs. Consider the given example:

### **Run1**

```
Covergroup cg;
CP: coverpoint a{
 Bins b = {1};
}
```

### **Run2**

```
Covergroup cg;
CP: coverpoint a+b{
 Bins b = {1};
}
```

In the given example, with the **set\_covergroup -merge\_cp\_by\_name** command, coverpoint **cp** will be merged even though the expression is different.

- **-instant\_sampling** - The **-instant\_sampling** option enables immediate sampling of covergroups when the clocking event of a covergroup is a SystemVerilog event. When you use this option, coverage points are sampled at the instant the clocking event takes place, as if the process triggering the event were to call the built-in **sample()** method. If the event is triggered multiple times in a step, then the sampling also happens multiple times. Consider the given example:

```
module top;
event e ;
Covergroup CG @(e) ;
A : coverpoint k
{
 bins b1[] = {[1:3]} ;
}
endgroup
.....
Initial begin
#1 k=1 ; ->e ; ->e; k=3 ; ->e ; k = 2 ;
end
endmodule
```

The sample report for the given example is shown:

| Name    | Average, Covered Grade | Line | Source Code          |
|---------|------------------------|------|----------------------|
| cg      | 66.66%, 66.66% (2/3)   | 7    | covergroup cg;       |
| --A     | 0% (2/3)               | 8    | A: coverpoint k{     |
| --b1[1] | 100% (2/1)             | 9    | bins b1 []= {[1:3]}; |
| --b1[2] | 0% (0/1)               | 9    | bins b1[] = {[1:3]}; |
| --b1[3] | 100% (1/1)             | 9    | bins b1[] = {[1:3]}; |

In the given example, value 1, 3 will be shown hit as the value of k is 1 and 3 before the first and third time event is triggered, respectively.

#### Points to Remember

##### set\_covergroup - optimize\_dump

- For UVM testbenches, logical object hierarchy will not be dumped. This will further impact logical instance-based flows. In addition, user will have to ensure unique names for each of the covergroup instances within UVM object hierarchy. This can be done by using `get_full_name()` while setting the name of the covergroup instance by `set_inst_name()` method.
- Covergroup instances may not be reported in the correct scope. For instance, consider the given example in which covergroup instances will not be reported in correct scope in module, package, or interface updation of covergroup instances in dynamic scope change.

```

package pkg ;
class C1 ;
covergroup cg ;
option.per_instance =1 ;
coverpoint 1;
endgroup

function new();
cg = new ;
endfunction
endclass
C1 Cobj ;
endpackage : pkg

module test;
import pkg::* ;
C1 Cobj1 ;
initial begin
Cobj = new ;
Cobj.cg.set_inst_name("inst1") ;
Cobj1 = Cobj ;
Cobj = null ;
end

```

```
endmodule
```

In the above example, covergroup instance inst1 will be reported under package pkg and not under module instance top even though its scope has been updated.

**set\_covergroup - new\_instance\_reporting**

- All covergroup objects under each scope will be reported alphabetically.
- Use of `set_covergroup -new_instance_reporting` command is not allowed with a design containing OVM.
- Specifying both `set_covergroup -optimize_dump` and `set_covergroup -new_instance_reporting` commands together is not allowed.
- Use of `set_covergroup -new_instance_reporting` command is allowed only with the design using Cadence installation UVM package.
- Merging of databases is not allowed for designs with or without command `set_covergroup -new_instance_reporting` command.

**set\_covergroup -bin\_merge**

- For a successful bin-level merge, both primary and secondary runs must be dumped with the `set_covergroup -bin_merge` command at elaboration.
- Bin-level merging is supported only for SystemVerilog covergroups and only with standard mode of merge.
- This command is not supported for real coverpoint.

**set\_covergroup -bin\_name\_merge**

- For a successful bin-level merge, both primary and secondary runs must be dumped with the `set_covergroup -bin_name_merge` command at elaboration.
- Bin-level merging is supported only for SystemVerilog covergroups and only with standard mode of merge. Union of bins is not supported, which means that a bin from the secondary database that does not exist in the primary database is not merged.
- This command is not supported for real coverpoint.

**set\_covergroup -optimize\_model**

- When you use the `set_covergroup -optimize_model` command, the support for the

cross of a cross is automatically enabled. For more information on cross of a cross, refer to [Declaring a Cross of a Cross](#).

- The behavior of the `-optimize_model` option has been made default.
- When you use the `-optimize_model` option in IMC:
  - In standard merge mode, coverage databases generated with and without this option are merged iff the primary database is generated with this option. However, the databases do not get merged if the primary database is generated without this option and the secondary database is generated with this option.
  - In union merge mode, coverage databases that are generated with and without this option cannot be merged.

#### `set covergroup -instant sampling`

- This option is valid if Covergroup is sensitive only on SystemVerilog events.

## Deprecated CCF Commands/Options

The following CCF commands/options have been deprecated and should be avoided:

- `select_fsm`
- `deselect_fsm`
- `set_expr_scoring -event`
- `set_expr_scoring -vlog_remove_redundancy`
- `set_expr_scoring -max_terms_sop`
- `set_expr_scoring -vhdl_shortcircuit`
- `set_expr_scoring -no_vhdl_control`
- `set_com -nolog`
- `set_ignore_library_name`
- `set_covergroup -optimize_model`

## Using CCF commands across simulations

The following CCF commands impact design checksum of the coverage database. These commands should be used in a consistent manner across simulations to allow merging of results, loading of tests, and re-using of refinements file.

- [select\\_coverage](#)
- [deselect\\_coverage](#)
- [set\\_implicit\\_block\\_scoring](#)
- [set\\_assign\\_scoring](#)
- [set\\_branch\\_scoring](#)
- [set\\_statement\\_scoring](#)
- [set\\_expr\\_scoring](#)
- [set\\_expr\\_coverable\\_operators](#)
- [set\\_expr\\_coverable\\_statements](#)
- [set\\_code\\_fine\\_grained\\_merging](#)
- [select\\_fsm](#)
- [deselect\\_fsm](#)
- [set\\_fsm\\_arc\\_scoring](#)
- [set\\_toggle\\_noports](#)
- [set\\_toggle\\_portsonly](#)
- [set\\_toggle\\_scoring](#)
- [set\\_optimize](#)
- [select\\_functional](#)
- [set\\_libcell\\_scoring](#)

- [set\\_fsm\\_scoring](#)
- [set\\_covergroup](#)
- [set\\_merge\\_with\\_libname](#)
- [set\\_parameterized\\_module\\_coverage](#)

## CCF Commands Supported in the MSIE Flow

Not all coverage configuration commands are supported in the MSIE flow. The following coverage configuration commands are supported and can be specified in the coverage configuration file passed at the time of primary snapshot creation:

- `include_ccf`
- `select_coverage`
- `(de)select_coverage` (Except the `-covergroup` option)
- `set_implicit_block_scoring`
- `set_explicit_block_scoring`
- `set_assign_scoring`
- `set_branch_scoring`
- `set_statement_scoring`
- `set_glitch_strobe`
- `set_hit_count_limit`
- `set_subprogram_scoring`
- `set_expr_scoring`
- `set_expr_coverable_operators`
- `set_expr_coverable_statements`
- `set_toggle_strobe`
- `set_toggle_limit`
- `set_toggle_includex`
- `set_toggle_includez`
- `set_toggle_noports`
- `set_toggle_portsonly`
- `set_toggle_scoring`
- `set_toggle_excludefile`

- `select_functional` (Except -ams\_control and -imm\_asrt\_class\_package options)
- `set_covergroup`
- `set_optimize` (Except -ial\_ovl\_inst\_asrt and -top\_expr\_non\_scorable options)
- `set_libcell_scoring`
- `set_merge_with_libname`

## CCF Commands Unsupported in the MSIE Flow

The following coverage configuration commands are unsupported in the MSIE flow and an error is generated if these commands are included in the CCF during elaboration:

- `(de)select_coverage -covergroup`
- `set_com`
- `set_com_interface`
- `set_code_fine_grained_merging`
- `set_fsm_attribute`
- `select_fsm`
- `deselect_fsm`
- `set_fsm_reset_scoring`
- `set_fsm_arc_scoring`
- `set_fsm_arc_termlimit`
- `set_fsm_scoring`
- `set_expr_scoring -struct`
- `set_refinement_resilience`
- `set_parameterized_module_coverage`
- `set_covergroup -bin_name_merge`