cadence®

ICC User Guide

Product Version 13.1

June 2013

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

Note: For the purpose of documentation, Incisive Coverage is the same as Incisive Comprehensive Coverage defined in the patents.

Contents

Preface

Cadence's Incisive Comprehensive Coverage (ICC) solution 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 the ICC solution and focuses on coverage data generation and data analysis using ICCR.

Note: Another tool, Incisive Metrics Center (IMC), is available for coverage data analysis. For more details on IMC, see the *Incisive Metrics Center User Guide*.

This preface discusses the following topics:

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 Cadence® NC-Verilog® simulator or the Cadence® NC-VHDL® simulator.

Related Documentation

- The *ICC Analysis User Guide* provides in-depth information on generating coverage reports using the ICCR GUI.
- The *ICC Quick Reference Guide* discusses the available commands for coverage data generation and analysis.
- The *ICC Quick Start Guide* discusses the key concepts for coverage data generation and analysis.
- The *Incisive Metrics Center User Guide* provides in-depth information on merging metrics data, displaying reports, marking metrics items, and analyzing metrics data using IMC.
- The *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
italic	Titles of books	See the *ICC Analysis Guide* for more information.
`literal`	Text that you must enter `literally` in source files or on the command line	`% ncvlog *.v`
<italic literal>	User-defined *arguments* for which you must substitute a name or a value.	`% ncvlog -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` *argument1* | *argument2*
[ ]	Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices. You can choose one argument from the list.	`command [`*argument1* | *argument2* `]`
{ }	Braces are used with OR-bars and enclose a list of choices. You must choose one argument from the list.	`command {`*argument1* | *argument2* `}`

. . .	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.	*argument*`...`
		`specify at least one`
		
		`[`*argument*`]...`
		`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:

```
% iccr -help
% ncvlog -help
% ncvhdl -help
% ncelab -help
% ncsim -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 *nchelp* utility to display extended help on the brief messages generated by the *compiler*, *elaborator*, and *simulator*.

Syntax:

```
% nchelp tool_name message_code
```

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

Examples:

```
% nchelp ncvlog BADCLP
% nchelp ncvlog badclp
% nchelp ncelab 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

1

ICC Overview

Cadence's Incisive Comprehensive Coverage (ICC) 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 ICC in the design flow.

1.1 Code Coverage

Code coverage in ICC 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 7, “FSM Coverage,” for more information.

1.2 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 ICC, 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.

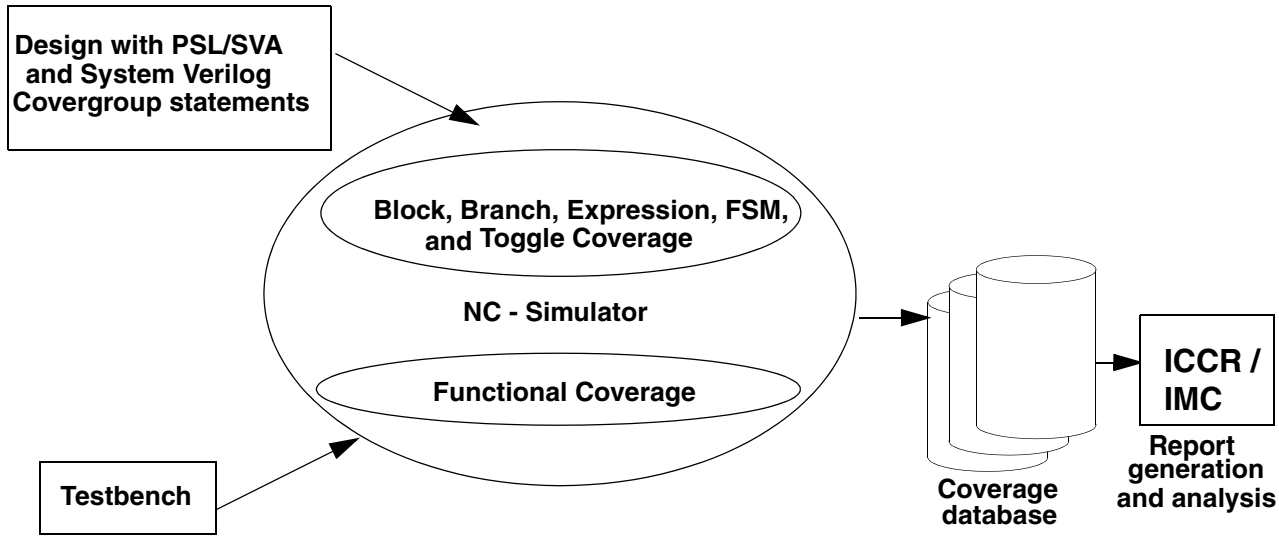
- Data-oriented functional coverage - Focuses on tracking data values. It includes coverage of variable values, binning, specification of sampling, and cross products. It helps design engineers to identify untested data values or subranges. In ICC, data-oriented functional coverage is specified using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

See Chapter 6, “Functional Coverage,” for more information.

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

1.3 ICC in the Design Flow

The ICC solution enables you to perform an analysis of code coverage and functional coverage in a single run, as shown below.



The design and testbench are passed to NC-Sim for coverage data generation. The coverage data is stored to a database, which later is analyzed using the coverage reporting tool, ICCR or IMC.

See Chapter 8, “Generating Coverage Data,” and Chapter 11, “Analyzing Coverage Data,” for detailed information on coverage data generation and analysis.

2

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.

2.1 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.

2.1.1 Blocks in Verilog/VHDL

A block is a statement or 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. ICC considers following procedural statements as a block:

- 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

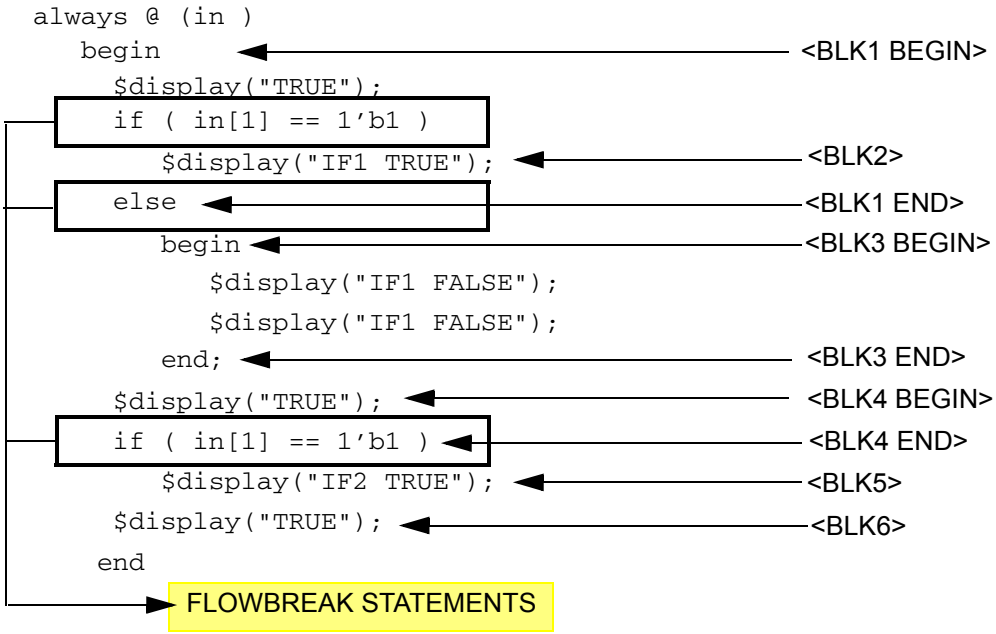
`if` statement	`case` item	`forever` statement
`repeat` statement	`while` statement	`for` statement
`wait` statement	delay control (`#`)	event control (`@`)
`disable` statement	tasks and function calls	

Flowbreak Statements in VHDL

`if` statement	`case` item	`loop` statement
`next` statement	`exit` statement	`return` statement
`wait` 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.



Note: Blocks are not defined for primitives.

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.

Note: Blocks in VHDL are not scored for a function defined inside a package when the function is called from inside VHDL generate block.

2.2 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.

2.2.1 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)

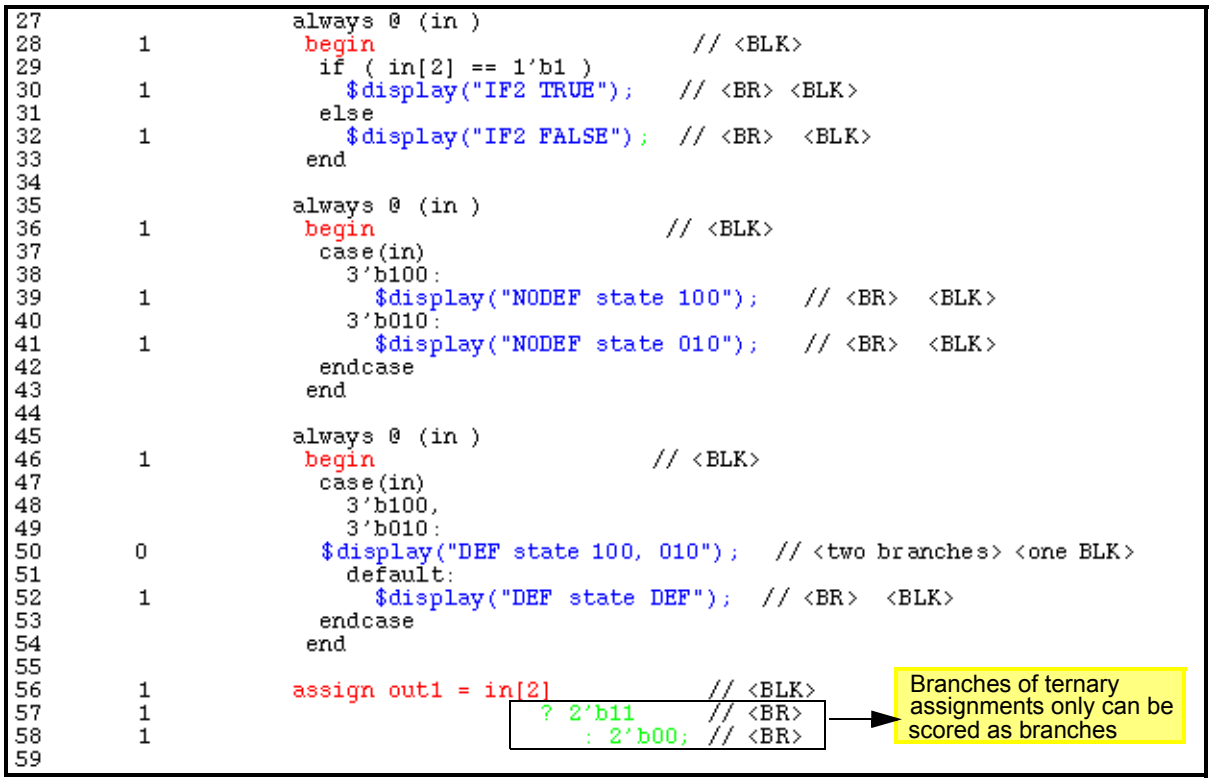
Branches in VHDL are generated due to:

- `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)

2.2.2 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 Chapter 11, “Analyzing Coverage Data,” for details.

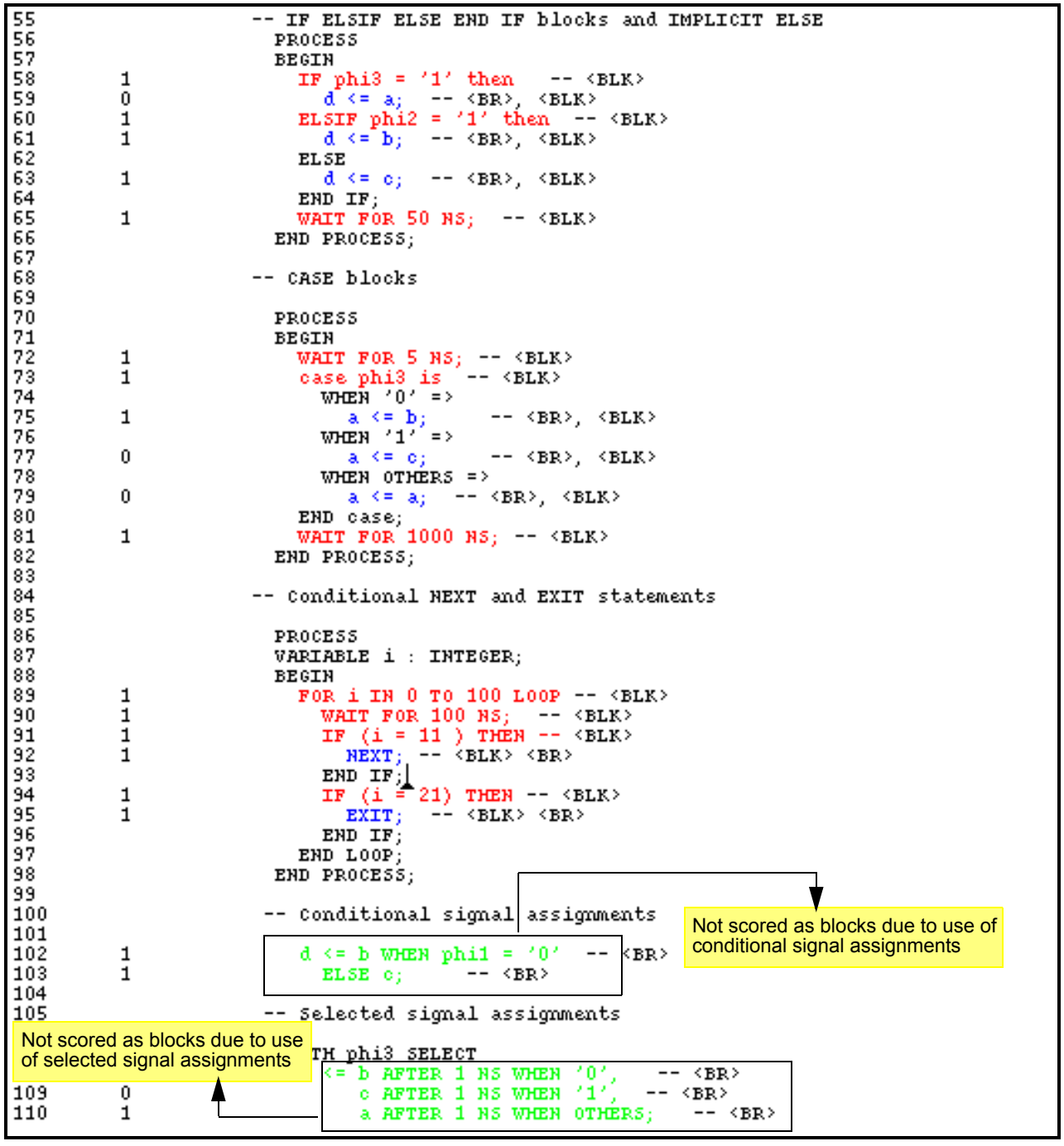


In the above figure, red indicates pure blocks, green indicates branches not scored as block, and blue indicates the code scored as block and branch.

Note: In a block detailed report (if branch scoring is enabled), the branches can be identified by descriptions next to them as ternary, true part of, false part of, a case item of, or implicit else. See Example: Block Coverage for a sample block detailed report.

2.2.3 VHDL Example

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



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.

2.3 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.

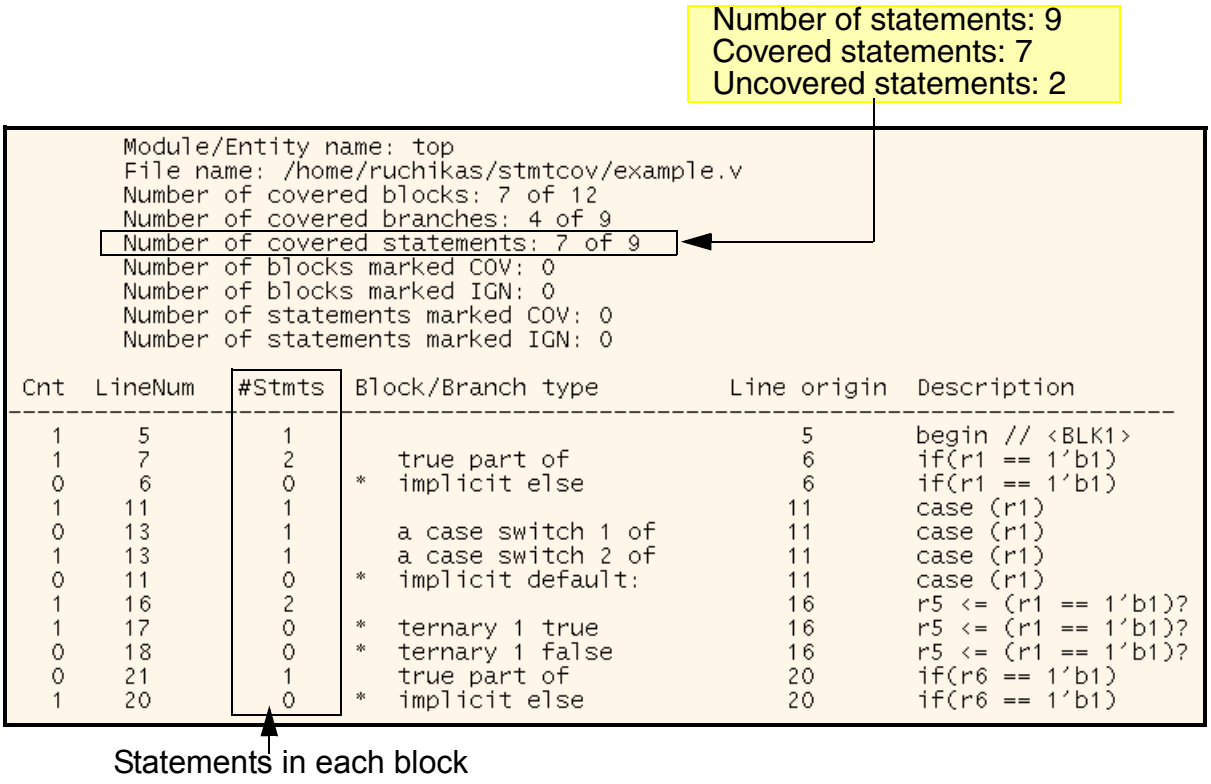
With this command, scoring of statements is enabled, and the statement coverage information is also included in the block coverage report.

Consider the following code.

```
 1  module top(r1, r6);
 2  input r1, r6;
 3  reg r2, r3, r4, r5;
 4  always@(r1 or r6)
 5  begin // <BLK1>
 6      if(r1 == 1'b1)        // implicit else <BLK3>
 7      begin // <BLK2>
 8          r2 <= 1'b0; r3 <= 1'b1;
 9     end
10
11      case (r1)             // <BLK4>, implicit default <BLK7>
12      1'b0, 1'b1 :
13          r4 <= 1'b1;       // <BLK5>, <BLK6>
14      endcase
15
16      r5 <= (r1 == 1'b1)?     // <BLK8>
17          1'b1: // <BLK9>
18          1'b0; // <BLK10>
19
20      if(r6 == 1'b1)          // implicit else <BLK12>
21          r4 <= ~r5;          // <BLK11>
22  end
```

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.



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

- ICC determines the number of statements 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.

- ICC does not ignore the statements enclosed within pragmas. 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
```

- ICC ignores the statements inside protected code.

2.4 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	■ Pass `-coverage block` to ncelab, or ■ Use the select_coverage command in the coverage configuration file and then pass this file using the `-covfile` option to ncelab.
Branch	Use set_branch_scoring command in the coverage configuration file and then pass this file using the `-covfile` option to ncelab.
Statement	Use set_statement_scoring command in the coverage configuration file and then pass this file using the `-covfile` option to ncelab.

Important

Branch coverage and statement coverage cannot be scored without enabling block coverage. See Chapter 8, “Generating Coverage Data,” for more details.

After simulation has dumped block, branch, and statement coverage data, you analyze it using the reporting tool ICCR. See Chapter 11, “Analyzing Coverage Data,” for details on report generation and analysis using ICCR.

3

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.
- 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.

Important

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. See Scoring of Operators for more details on scoring other 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 on page 207 for details on specifying scoring mode.

3.0.1 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)

Note: 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> <3>
-------------------------
   1 |    1 |  1   -   1
   1 |    1 |  1   1   -
   0 |    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.

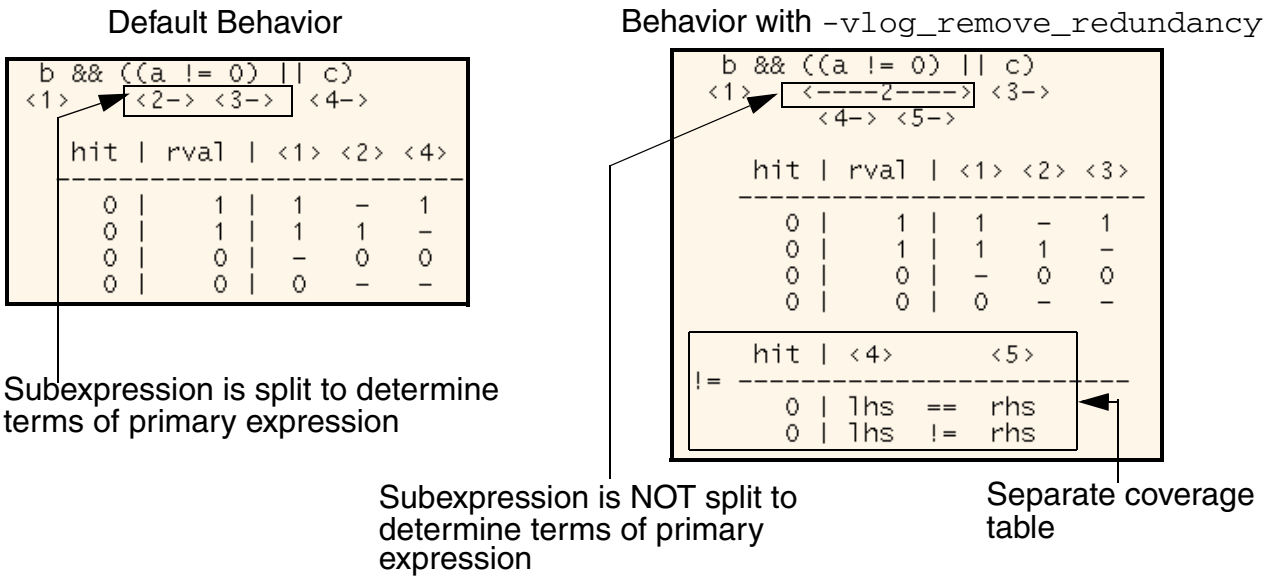
With this command, the terms of the primary expression are determined as:

```
 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.

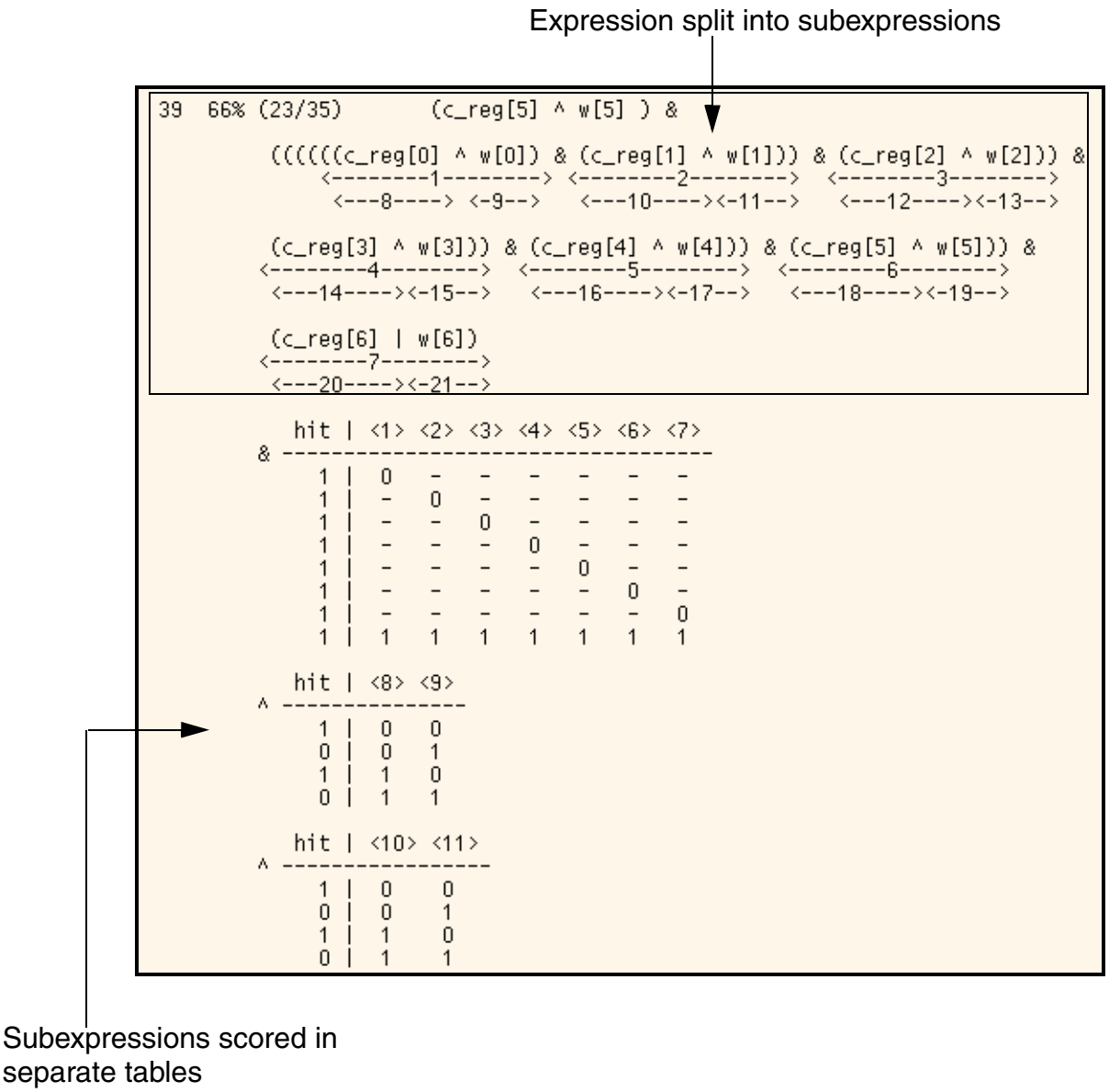


Example 4

ICC splits an expression 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

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

Note: 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.

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

Note: 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.

Consider the following SOP table for a VHDL `AND` operator.

```
 hit | <1> <2>
-------------
  0  |  0   -    <= ROW 1
  0  |  -   0    <= ROW 2
  0  |  1   1
```

By default, for an input combination of term `<1>` = `0` and term `<2>` = `0`, ICC will score Row 1 only 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, ICC will evaluate both term `<1>` and term `<2>`, and will score both Row 1 and Row 2.

3.0.2 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>
```

Note: 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)        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
```

3.0.3 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 input changes value, then the 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>`

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

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

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

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.

3.0.4 Vector Scoring

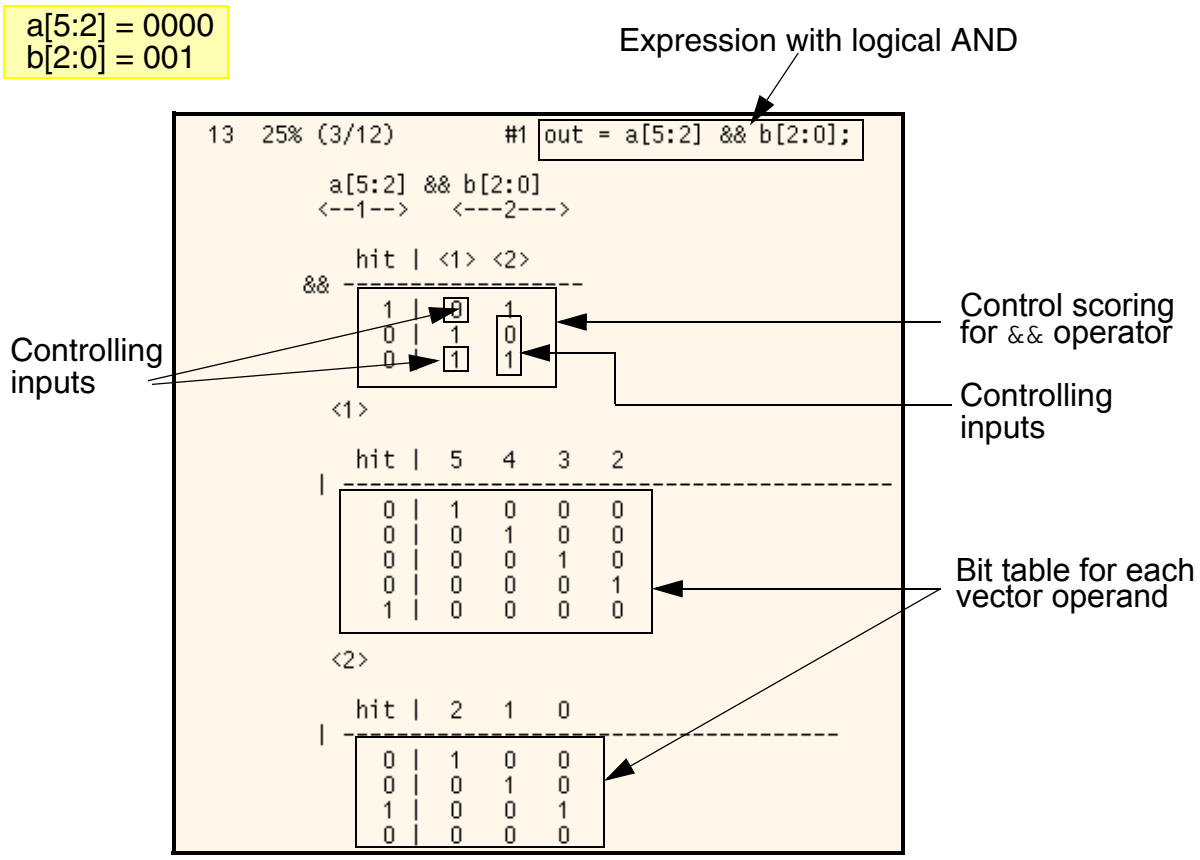
Vector scoring mode is an extension of control scoring mode. With vector scoring, each bit of a multi-bit signal is scored and reported separately and you have lots of data to analyze. The following Verilog code demonstrates how truth tables are created for vector scoring.

```
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.

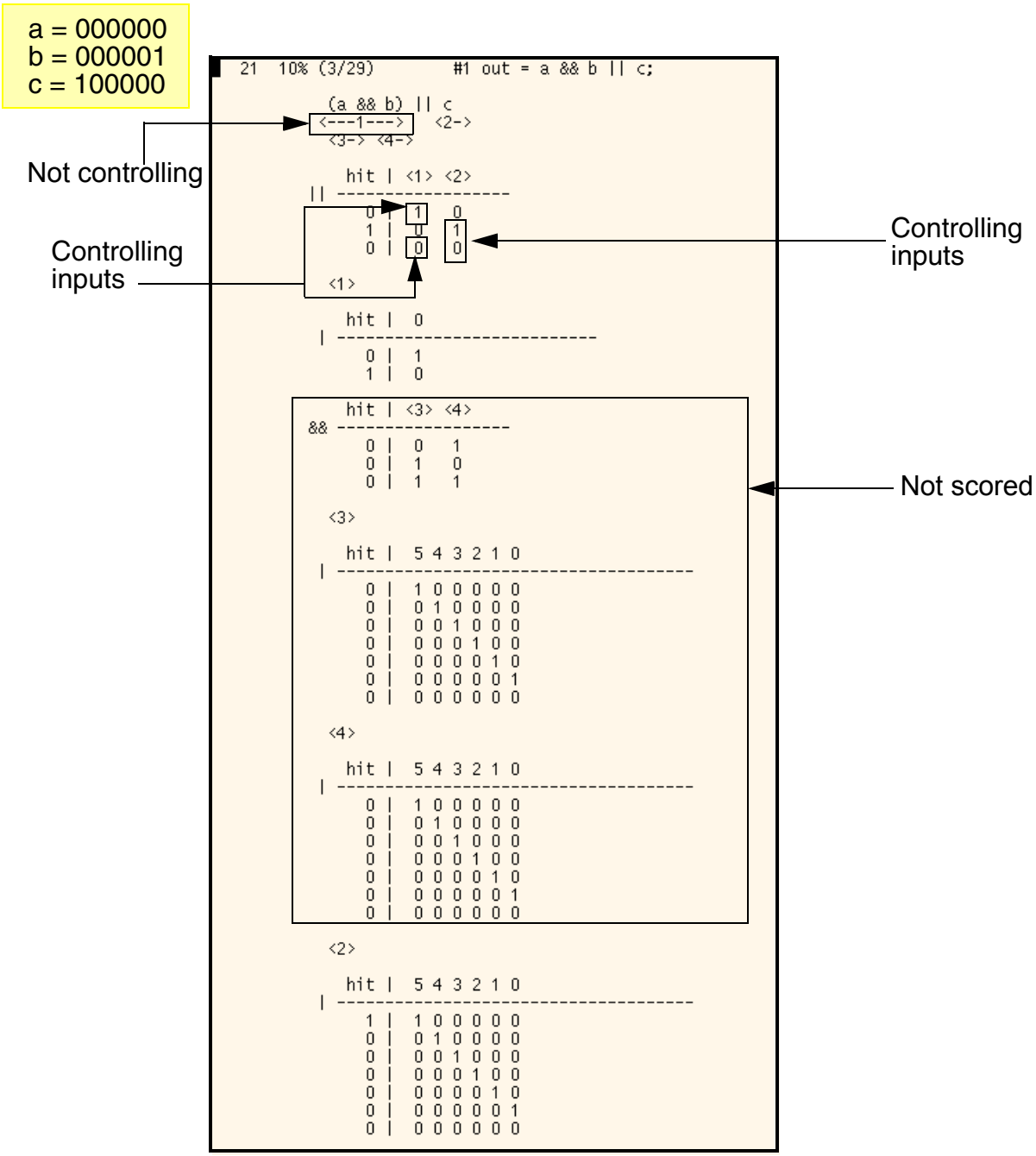
In a logical operator truth table, each operand is scored using an | reduction table, and the overall expression is scored using the same table as a scalar || or && operator.

The operands for operators || and && are first compressed into a single bit, such that if any bit is a 1, then the operand is a 1. This is effectively an | reduction (for both operators) and would be 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 a 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.

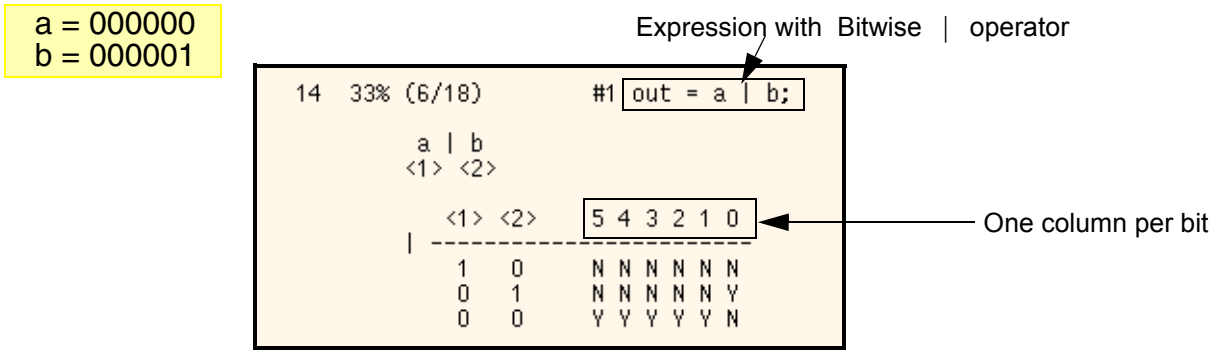
Consider another truth table for expression containing logical operators on Line 21.



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.

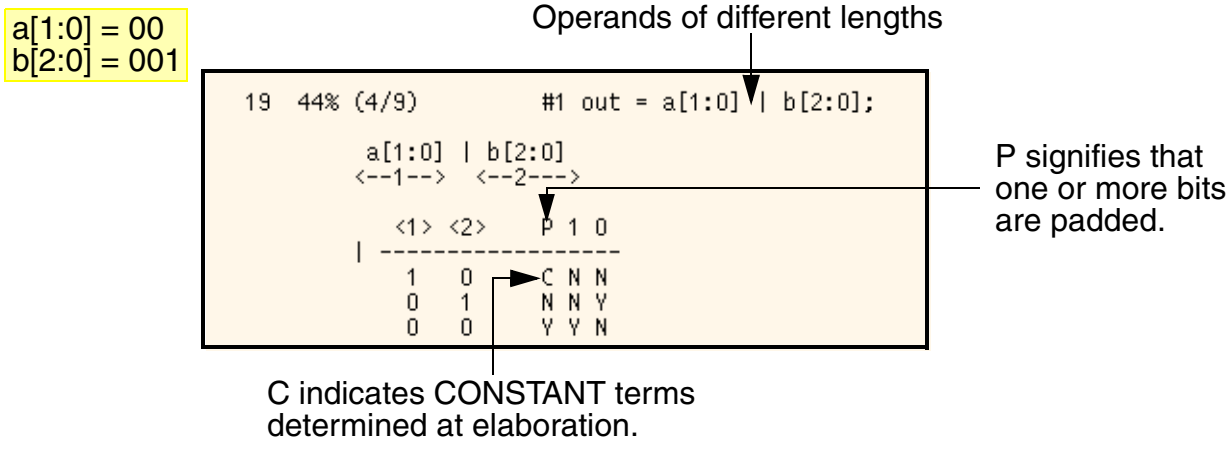
Scoring of bitwise operators

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



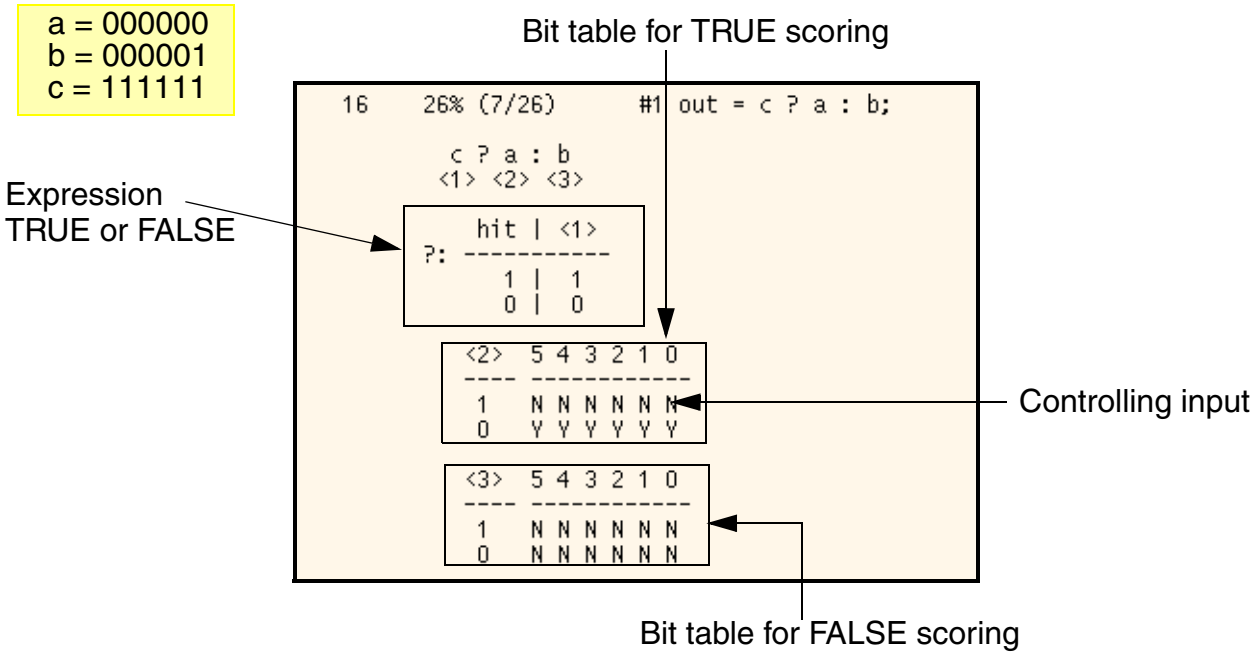
In a bitwise operator truth table, Y indicates that this bit combination has occurred. N indicates that this bit combination has not occurred. The scoring count is number of hits (Y) divided by number of scorable entries (Y+N), which in this case is 6/18.

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



Scoring of conditional operator

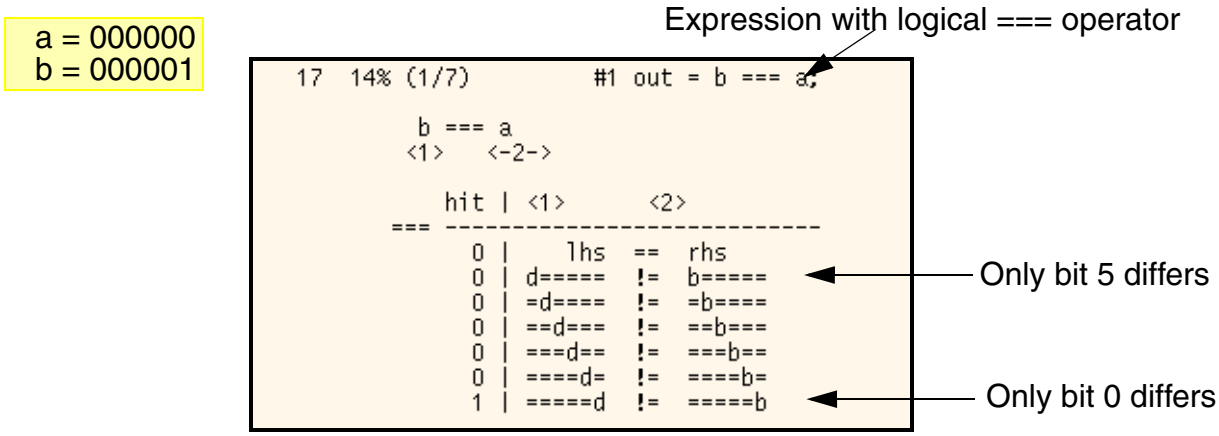
The truth table for expression containing a conditional operator on Line 16 is as follows.



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 `<2>` 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 number of the largest 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.

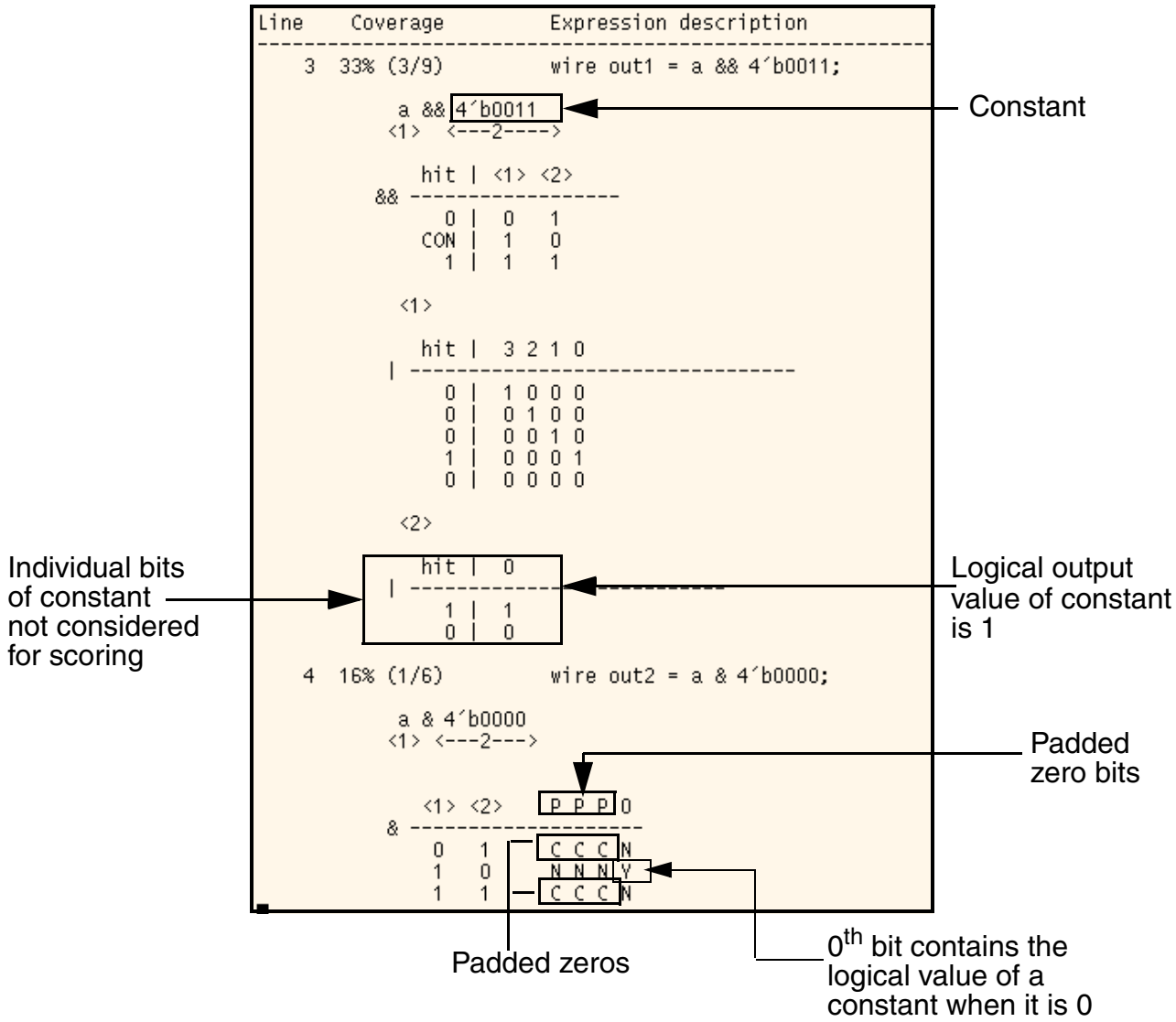
See Chapter 11, "Analyzing Coverage Data," for details on ICCR and how to generate reports for analyzing Expression coverage data.

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:



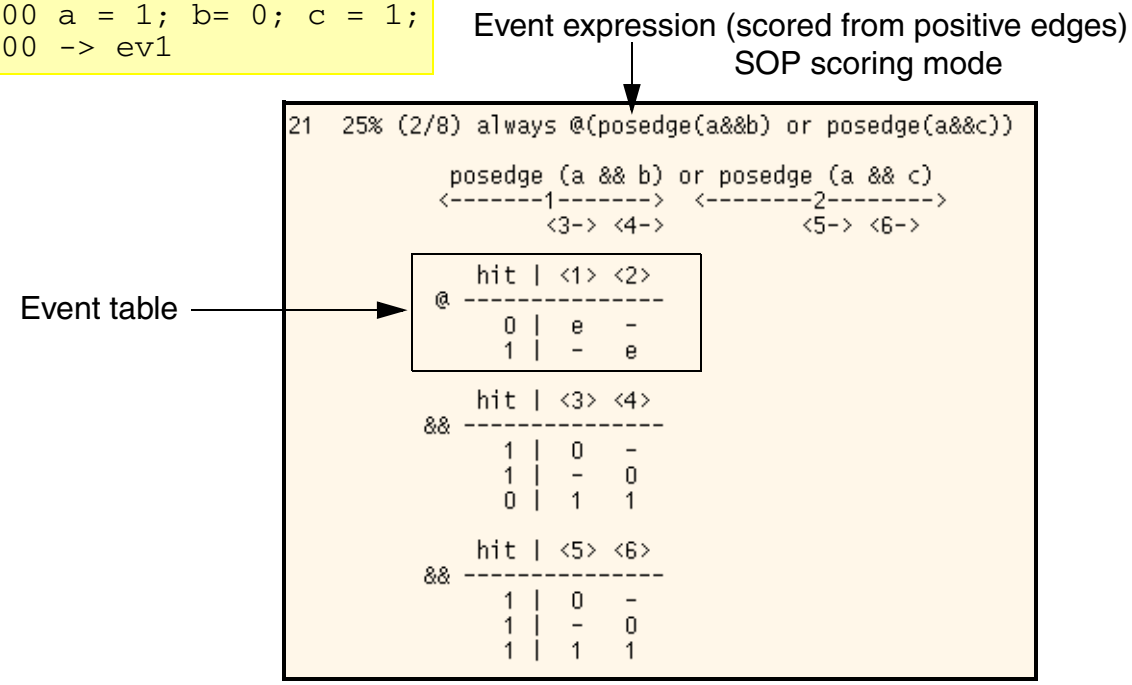
3.1 Scoring Events

Scoring events helps you examine the possible value changes that trigger the execution of an `always` block. By default, ICC 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.

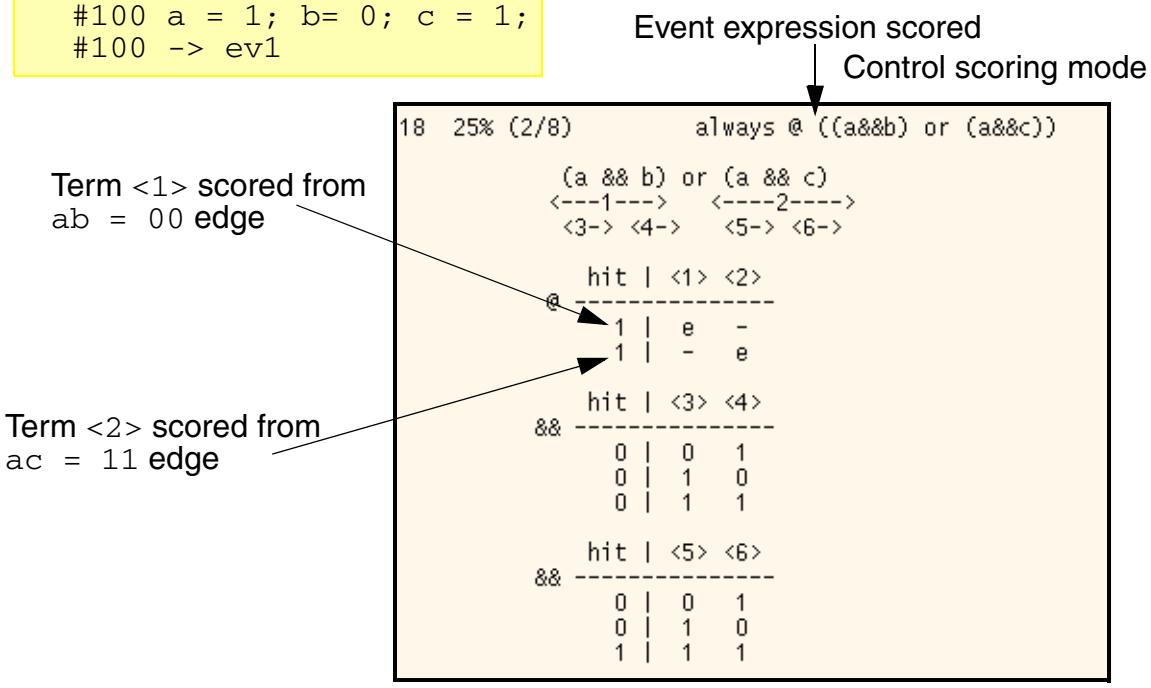
```
event ev1, ev2;
reg a, b, c, r;
initial begin
  #100 a = 0; b= 0; c = 0;
  #100 a = 1; b= 0; c = 0;
  #100 a = 1; b= 0; c = 1;
  #100 -> ev1
```



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.

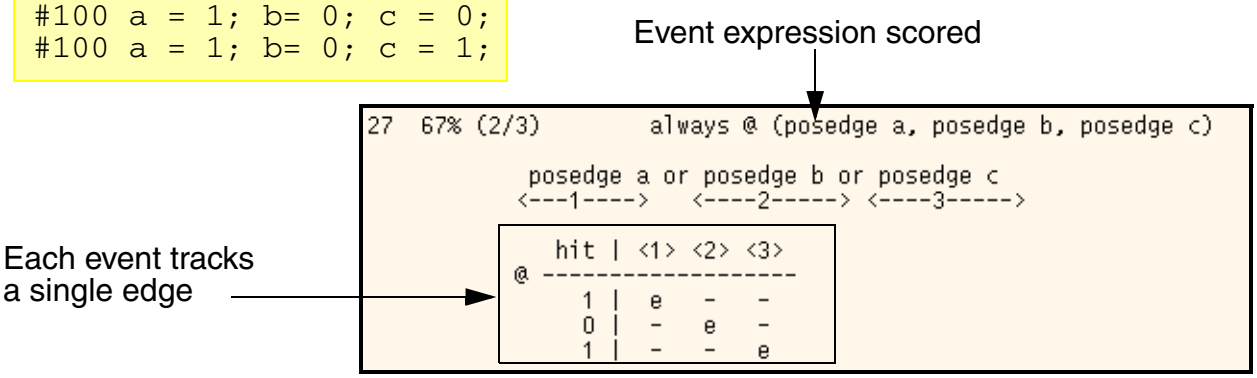
Consider another example.

```
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
```



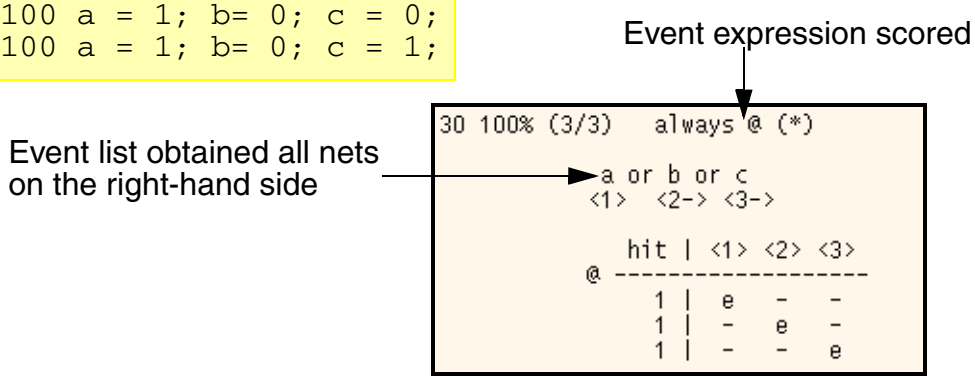
Consider another example that uses a sensitivity list.

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



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;
```



3.2 Scoring of Operators

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.

- To score coverage for all operators wherever expressions are scored, use the set_expr_coverable_operators -all command in the coverage configuration file, and then pass it to `ncelab` using the `-covfile` command-line option.

  Note: The `set_expr_coverable_operators -all` command will enable scoring of all operators in condition expressions except for the VHDL `not` operator. To enable scoring of VHDL `not` operator in the sop mode, use the set_expr_scoring `-vhdl_not_as_operator` command in the coverage configuration file at elaboration. The VHDL not operator is not scored in any other mode.

- To score coverage for all operators, and for expressions in conditions as well as assignment statements, specify the set_expr_scoring `-all` command in the coverage configuration file, and then pass it to `ncelab` using the `-covfile` command-line option.

The following table lists the operators that are scored when the `set_expr_scoring -all` command is used.

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

VHDL Operator	SOP Table Created	Control Table Created	Vector Table Created	FCC Table Created
Miscellaneous: `**`, `abs`, `not` **Note:** You can enable scoring of VHDL `not` operator using the set_expr_scoring -`vhdl_not_as_operator` command in the coverage configuration file at elaboration.	No	No	NA	No

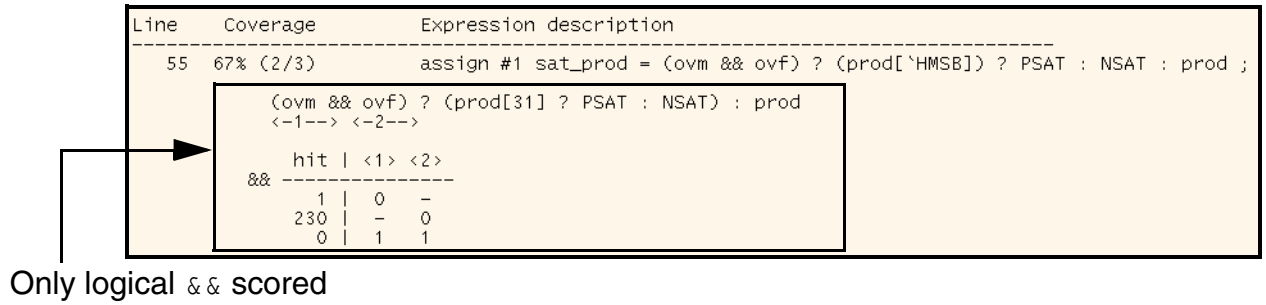
Note: Vector scoring is not supported for VHDL design units.

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

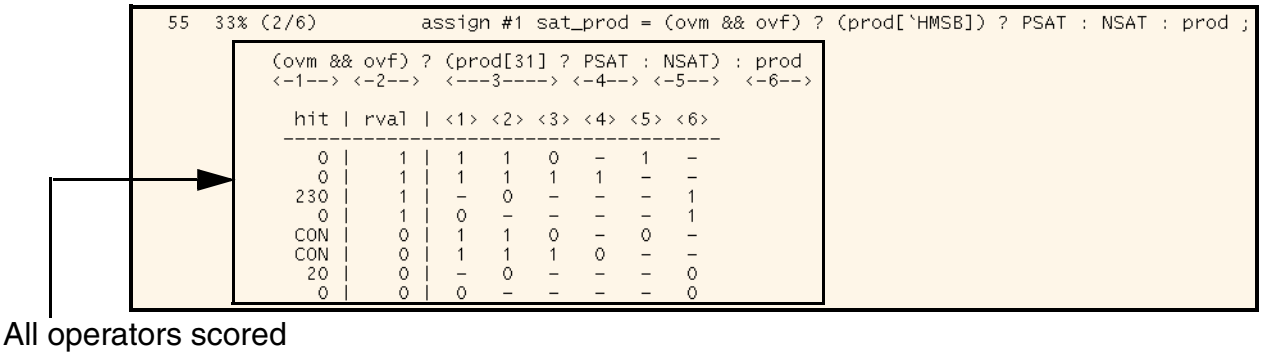
The following expression coverage report displays a sample expression which includes multiple operators. In the default mode, coverage is scored only for Verilog logical operator `&&`, as shown below:



Only logical `&&` scored

Expression Coverage Report -- All operators scored

When the same expression is scored with the `set_expr_scoring -all` command in the CCF, all the operators are scored, as shown below:



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)

3.3 Scoring Expression Coverage

To score expression coverage, you can either:

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

See Chapter 8, “Generating Coverage Data,” for more details.

After simulation has dumped expression coverage data, you analyze it using the reporting tool ICCR. See Chapter 11, “Analyzing Coverage Data,” for details on ICCR and how to generate reports for analyzing Expression coverage data.

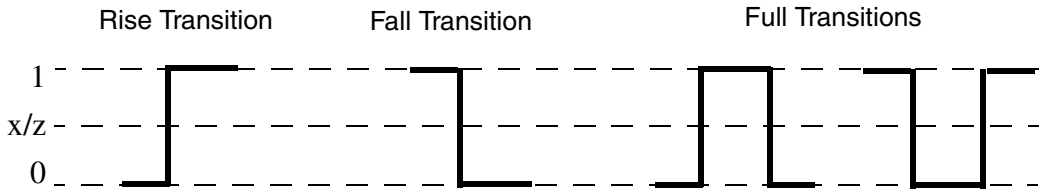
4

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 ICC.

4.1 Types of Transitions

Change in signal value during simulation run is a transition. ICC records rise, fall, and full transitions, 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.

Note: In case of enum variables, transition is measured based on only the final state, irrespective of the initial state of the object. Consider the given example:

```
typedef enum logic [1:0]
{
  S1 = 2'b00,
  S2 = 2'b01,
  S3 = 2'b11,
} enum_state;
```

In the given example, transition will be counted for both the transitions from `S1` to `S3` and `S2` to `S3`, and the toggle count will be scored for the transition to `S3` independent of the initial state.

Toggle coverage scores these transitions 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.

Important

Currently, ICC does not score toggle coverage for:

- Real numbers or integers.
- `supply0` and `supply1` nets.

4.2 Signal Behavior Recognized in ICC

The default behavior of toggle coverage in ICC 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_excludefile 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 in 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.

See Chapter 8, “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.

4.3 Scoring Toggle Coverage

To score toggle coverage, you can either:

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

See Chapter 8, “Generating Coverage Data,” for more details.

After simulation has dumped toggle coverage data, you analyze it using reporting tool ICCR. See Chapter 11, “Analyzing Coverage Data,” for details on ICCR and how to generate reports for analyzing toggle coverage data.

5

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 of ICC enables you to automatically detect many un-exercisable coverage items in the design, mark them, and ignore them from coverage figures. ICC does 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 be identified and manually marked. This chapter discusses automatic marking of coverage items using COM.

Note: ICC also provides a mechanism for manual marking of coverage items through pragmas and the ICCR `mark` command. See Chapter 12, “Manual Marking of Coverage,” for details on manual marking.

5.1 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 ncelab.

- 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.

5.1.1 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, ICC marks the following toggle objects.

Verilog	■ `wire` ■ `reg`
VHDL	■ signal (`std_ulogic`, `std_logic`, `std_ulogic_vector`, `std_logic_vector`, `bit`, `bit_vector`, `boolean`) ■ variable (`std_ulogic`, `std_logic`, `std_ulogic_vector`, `std_logic_vector`, `bit`, `bit_vector`, `boolean`)

COM analysis for toggle coverage involves identifying and marking objects `IGN` that never toggle during simulation run.

5.1.2 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 by ICC.

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

Verilog	**VHDL** (from `standard` and `std_logic_1164` packages)
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 ICC 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.

5.1.3 COM and Block Coverage

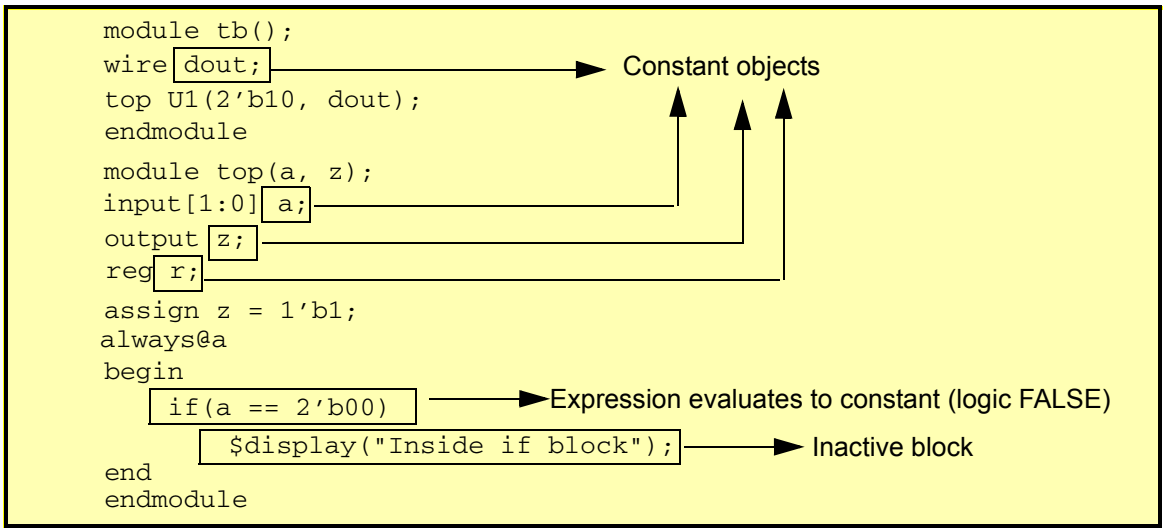
Verilog and VHDL blocks that are controlled by `if`, `if-else`, and `case` expressions are marked if the corresponding controlling condition can be marked. Blocks that are immediate children of a control statement (`if` or `case`) are marked as `IGN` if the controlling expression determines that this 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.

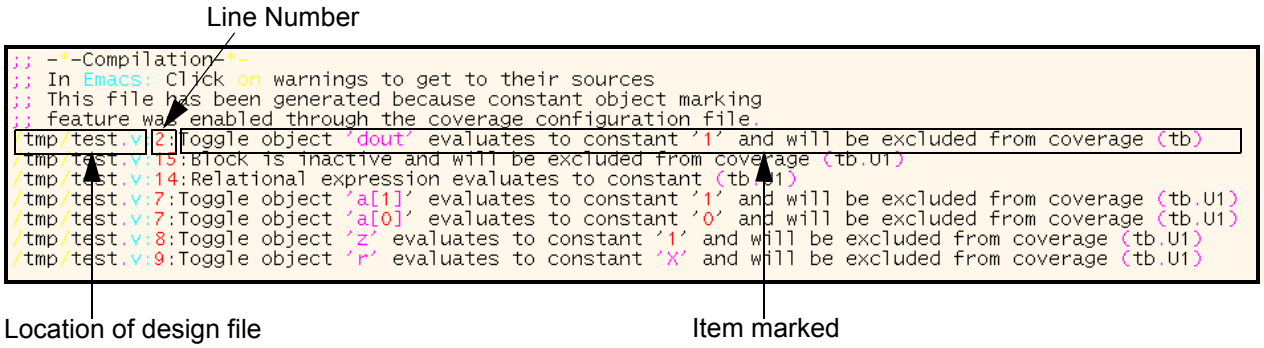
```
assign w = (1`b1) ? a:b;
```

5.1.4 COM and Coverage Calculation

Consider the following Verilog code.



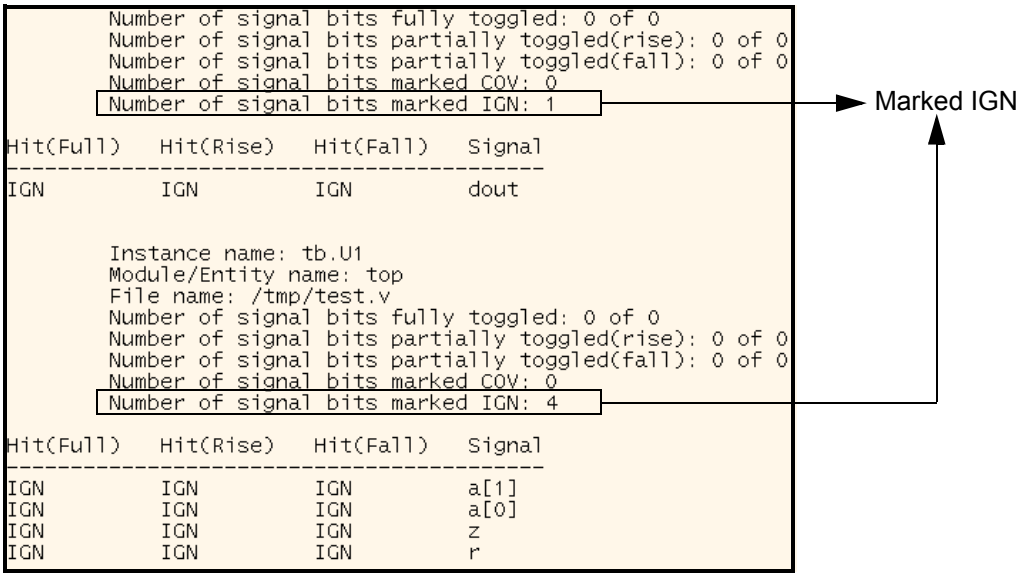
If COM is turned ON (using `set_com` in coverage configuration file) and block, expression, and toggle coverage is enabled, then the following `icc.com` file is generated.



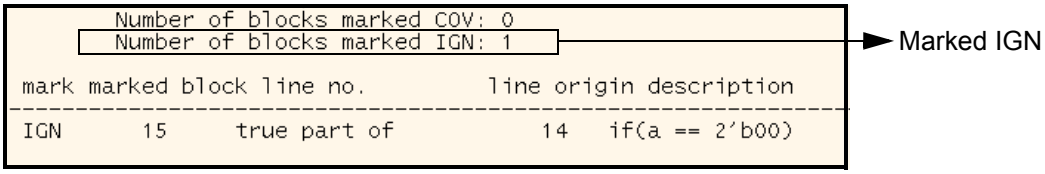
This file details the coverage items marked due to COM. If you are using an editor like emacs, then the items in this file link directly to the related source files.

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`.

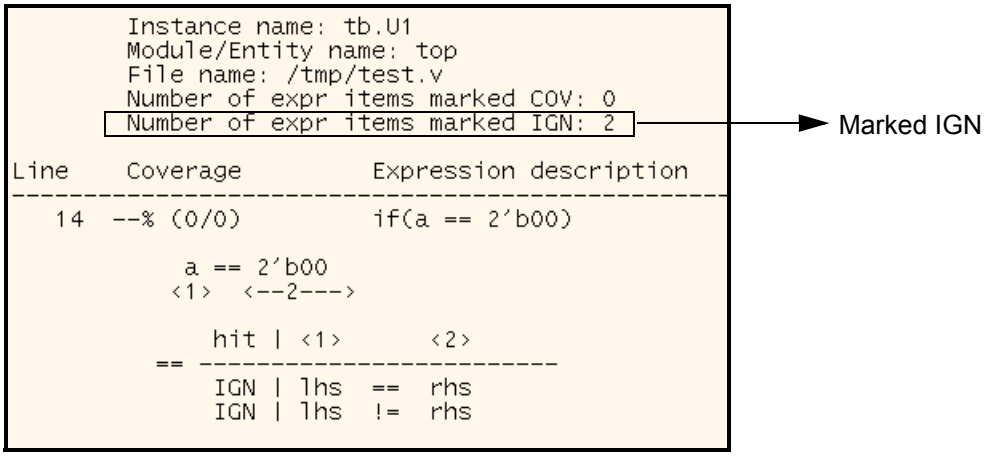
Un-exercised toggle objects are reported using the `-marked` option in the detailed report, as:



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:



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



See Chapter 11, “Analyzing Coverage Data,” for details on Coverage reports.

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.

5.2 Evaluation of Objects by ICC

ICC evaluates an object 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).

Note: 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 8, “Generating Coverage Data,” for more details.

6

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.

Note: To access the functional coverage capabilities, you need an Incisive license.

Functional coverage can be:

- Control-Oriented Using PSL/SVA Statements
- Data-Oriented Using SystemVerilog Covergroup

6.1 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 ICC, 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*.

6.1.1 Functional Coverage Points in PSL

ICC 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

6.1.1.1 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.

6.1.1.2 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 `assertion -disable` 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 ICCR.

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 *NC-SC Simulator Reference, Chapter 5 - Using SystemC PSL*.

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

6.1.2 Functional Coverage Points in SVA

ICC 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.

6.1.2.1 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);
```

6.1.2.2 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.

6.1.2.3 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*.

Important

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*.

6.2 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

6.2.1 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
```

Important

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.

6.2.1.1 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
```

6.2.1.2 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_inst1 = new(2); // value passed to formal argument size during instantiation
```

In the above code, covergroup `cg` has a formal argument `size` of type `input`. A value `2` is passed to the formal argument during covergroup instantiation.

Note: Covergroup arguments can be of type `input` or `ref`.

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.

Important

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 `ref` 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 on page 86.
- Example: Covergroup ref Argument to Specify Coverpoint Expressions on page 87.

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.
- Covergroup arguments cannot be used to specify ranges for wildcard bins.

```
covergroup cg (input reg[2:0] val) @(clk);
   coverpoint cp1 {wildcard bins b1[] = {val};}
endgroup
cg cg_inst = new(3'b11x); //passing a x value -- Not supported
```

- Specifying default values for covergroup arguments is currently not supported.

```
covergroup cg (input int i = 5) @(clk); //Not supported
   ...
endgroup
cg cg_inst = new;
```

- Covergroup `input` 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)

Note: 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 on page 165.

6.2.2 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_identifer :] 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}
     [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
```

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. If an identifier is not specified, the variable name is used as the identifier for the coverpoint. 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 on page 87

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 that 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

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

6.2.2.1 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:

```
bins bin_identifier [ [ [expression] ] ] = {open_range_list}
```

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 number (determined by the 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`"

Range	Description
`[$ : $]`	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.

Note: 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:

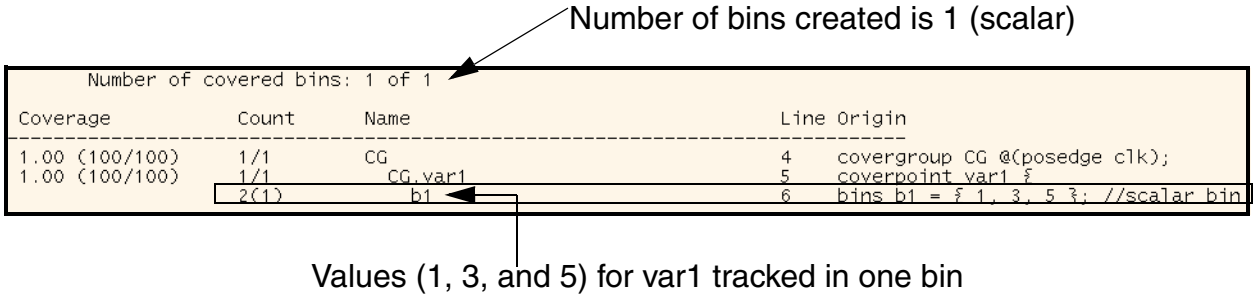
```
bins b1 = binsof(A) intersect{v2};
```

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`.

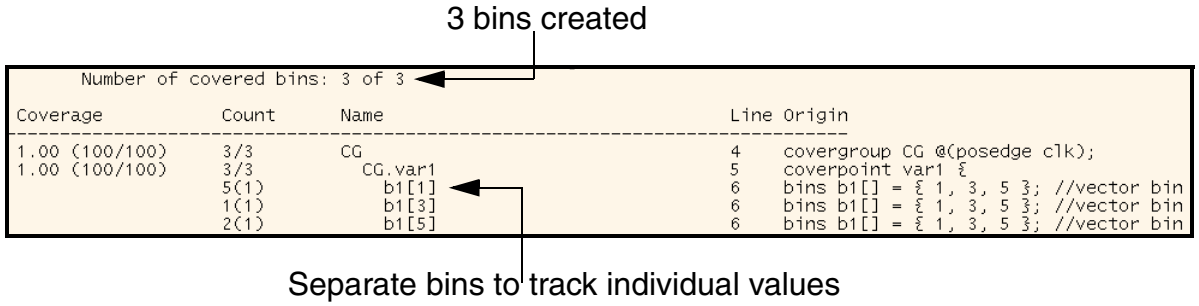


With a scalar bin, a single bin is created and you cannot identify how many times variable `var1` was assigned a particular value from the given range list.Example: Creating a Vector Bin

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`.

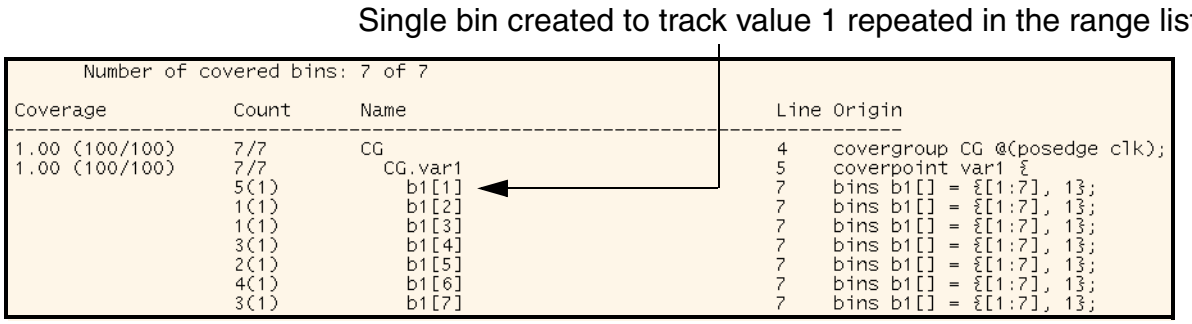


In this example, count is incremented individually for each of these bins depending on the number of times the value of `var1` was 1, 3, or 5 during simulation run.

Note: 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:



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.

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

```
Instance name: test
Module/Entity/Interface/Program name: test
File name: /home/muktad/TestCases/ICC/others/32_sizedvector/test.sv
Number of covered bins: 4 of 5

 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'h0000000000000000
                                                                      :64'h0000000000000003]};
                    0(1)             b2[1]         13   bins b2[2] = {[64'h0000000000000000
                                                                      :64'h0000000000000003]};
```

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

Important

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, see Chapter 11, “Analyzing Coverage Data”.

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};
  bins b3 = {[1.5:1.3)};
  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
```

Note: 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 on page 119.

Note: 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

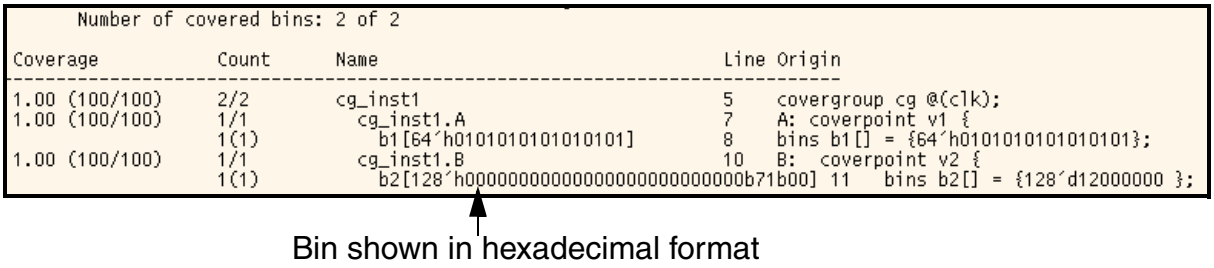
When creating coverpoints on variables with size greater than 32 bits, remember that:

- 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:



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'h0001010101010101]          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 `32` because the bin value is more than the size of the coverpoint variable.

6.2.2.2 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.

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.

Important

ICCR expands transition vector bins to report all possible transitions under a given definition. Consider the following code:

```
module top;
reg [7:0] a=0;
```

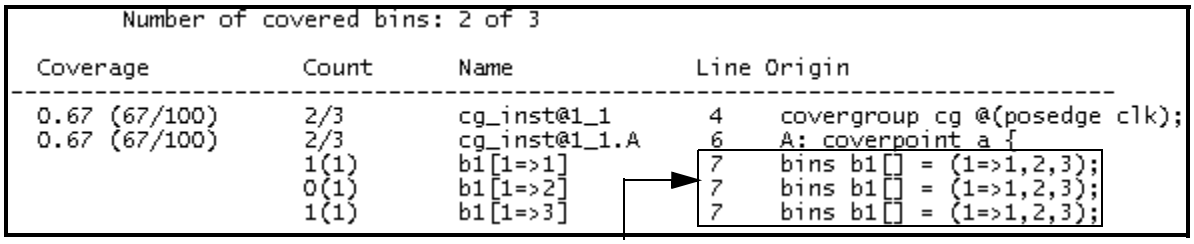
```
reg clk=0;
covergroup cg  @(posedge clk);
   A: coverpoint a {
   bins b1[] = (1=>1,2,3) ;
   }

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:



All transitions reported

Important

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]);
```

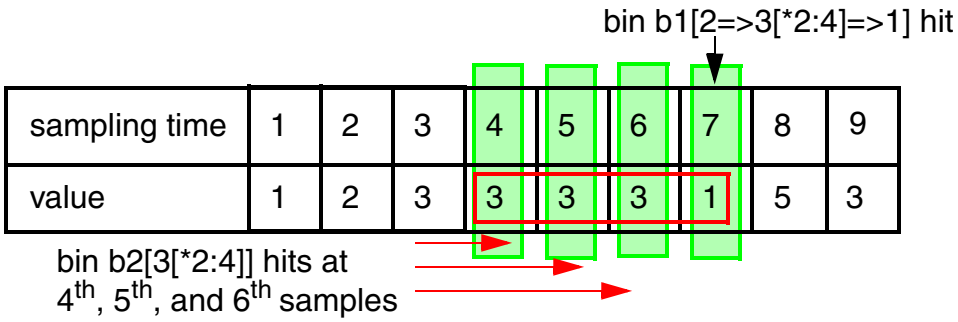
is same as:

```
(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



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.

Note: 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:

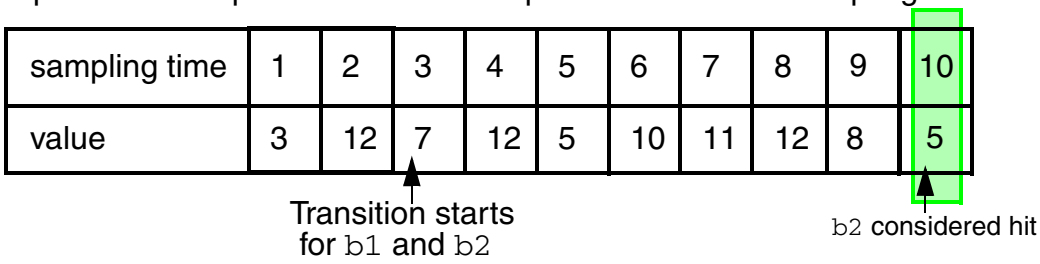
```
b1 = ...7=>12=>...=>12=>5
b2 = ...7=>12=>...=>12=>...=>5
```

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



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

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

6.2.2.3 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

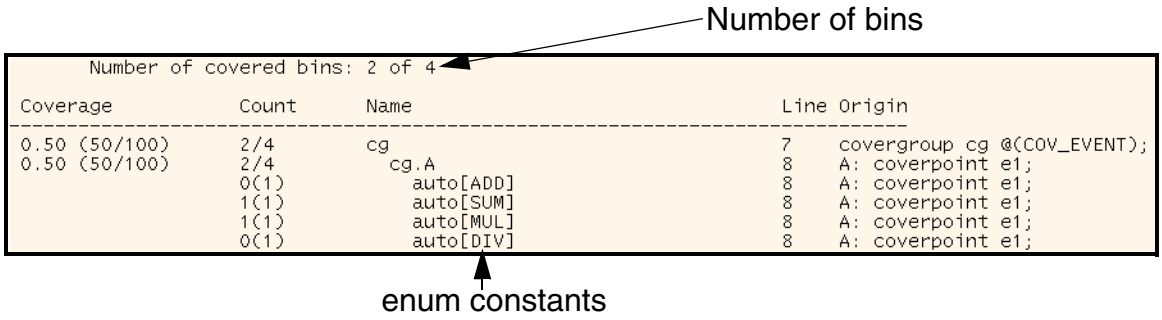
1	SystemVerilog calculates the number of bins (N) as minimum of 2^M and value of `auto_bin_max` option, where M is the number of bits needed to represent the coverpoint.	Code snippet
		`reg[0:3] a;`
		`covergroup cg @(posedge clk);`
		`  coverpoint a{`
		`  option.auto_bin_max = 3;`
		`  }`
		`endgroup`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

2	SystemVerilog determines how the possible values (V) will be stored across number of bins (N) as:
	■ If N < V, then the values are evenly distributed among N bins.
	■ If V is not divisible by N, then the last bin includes the remaining values.
		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:
		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:



For more on reporting, see Chapter 11, “Analyzing Coverage Data”.

6.2.2.4 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.

Note: 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`.

6.2.2.5 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
```

Note: 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.

6.2.2.6 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.

Important

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 1: 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.

Note: 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 2: 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 3: 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.

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

Example 4: 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 5: 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        cg1_inst1      6    covergroup cg  @(negedge clk);
0.67 (67/100)  2/3          cg1_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 6: 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=>5=>1` occurs at the 11th sample, it is never hit because transition `2=>5` is specified as `ignore_bins`. Here, bin `b1[1=>4=>3]` is incremented at the 6th and 13th samples.

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

Example 7: 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 on page 106.

Example 8: 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 9: 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 10: 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 non 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.

6.2.2.7 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.

6.2.3 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.

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_identififer::=
     {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_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
```

Note: 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 are not trackable through system tasks or methods. In addition, these coverpoints do not participate in coverage calculations and are not available in post-processing reports in ICCR.

6.2.3.1 Associating Labels with a Cross

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.

If no label is associated with a cross, the tool internally generates a label as:

```
__<label of first coverpoint>_X_<label of second coverpoint>_X_..._X_<label of last coverpoint>_n
```

where, n is an internally generated positive integer. This integer is initialized to 0 and is incremented for each new internally generated label. The following table describes how the tool generates a label for cross in different scenarios.

Scenario	Example	Cross Label Generated
All coverpoints are labeled	A: coverpoint a; B: coverpoint b; cross A, B;	__A_X_B_0
Not all the coverpoints are labeled	coverpoint a B: coverpoint b; cross a, B;	__a_X_B_0
Coverpoints are not labeled and variables are used in cross definition	reg [1:0] a; reg [3:0] b; covergroup cg @(posedge clk); coverpoint a; coverpoint b; **cross a, b;** endgroup	__a_X_b_0
Hierarchical references to coverpoints in cross declaration	covergroup cg@clk; CR_1: cross cg1_inst.A, cg1_inst.B; endgroup	__cg1_inst_A_X_cg1_inst_B_0

When defining a cross:

- Do not use a cross definition to define another cross.

```
AXB: cross A, B;
AxBxC: cross AXB, c; // Cannot use cross definition to define cross
```

6.2.3.2 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.
- Using OOMRs to access the coverpoints in cross declaration is not supported in this release. Consider the given example in which OOMR is used to access coverpoints in a cross declaration is not supported:

```
covergroup cg;
    cross I1.obj1.cg1.A, I2.obj2.cg2.B;  // I1 and I2 are module instances
endgroup
```

- 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
```

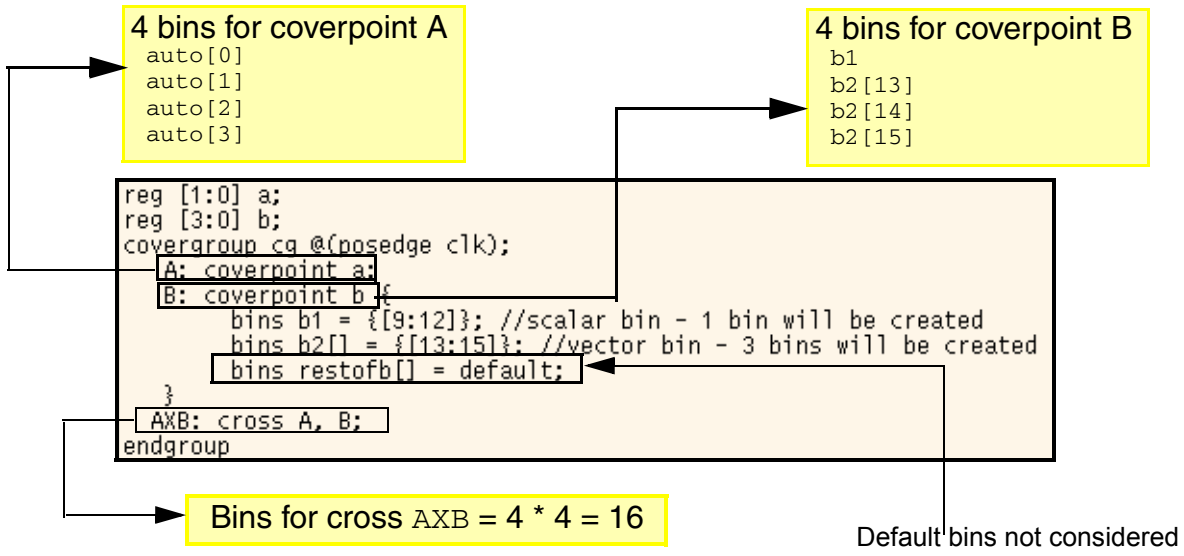
6.2.3.3 Automatic Cross Bins

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:

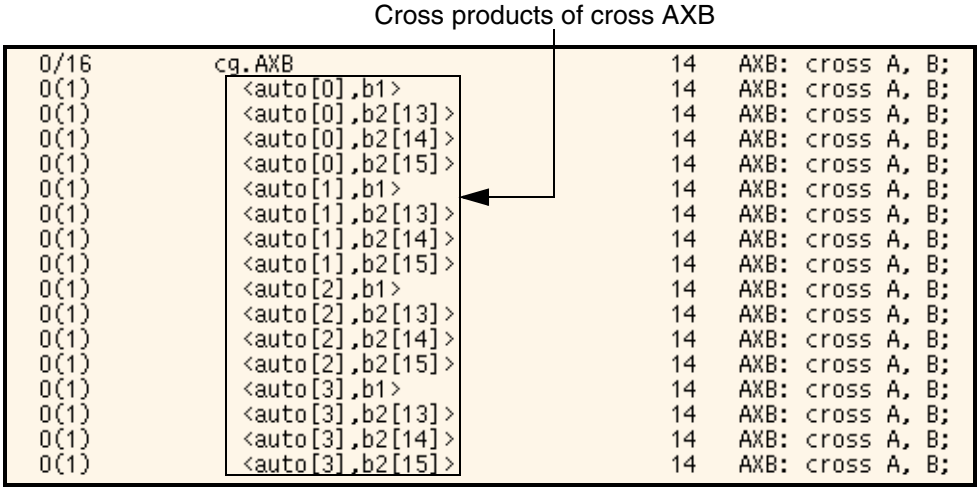
```
Bins for cross= Bins of coverpoint_1 * Bins of coverpoint_2 * Bins of coverpoint_N
```

Note: 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:



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.

Note: 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.

6.2.3.4 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}];
}
```

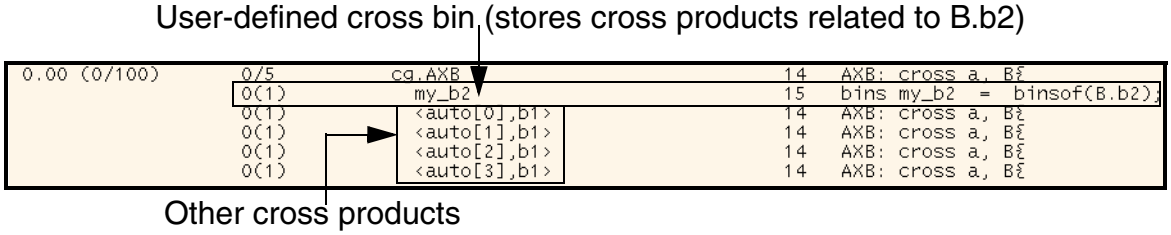
In the above definition,

- `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:

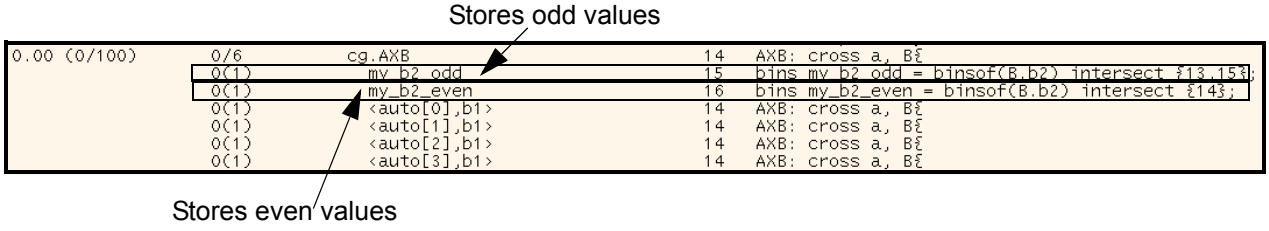


With the `intersect` construct, you can further refine the cross products reported with the `expression` stated in the `binsof` construct.

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:



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.

Note: 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);
}
```

6.2.3.5 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.

Note: 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`.

6.2.4 Predefined Coverage Methods

The following coverage methods are available for a covergroup.

Method	Can be called on			Description
	Covergroup	*Coverpoint*	*Cross*	
stop()	Yes	Yes	Yes	Stops sampling if default sampling condition is specified
start()	Yes	Yes	Yes	Starts sampling if default sampling condition is specified
sample()	Yes	No	No	Triggers sampling of covergroup irrespective of the `start` or `stop` method or the default sampling condition
set_inst_name(string)	Yes	No	No	Assigns name to a covergroup instance
get_coverage()	Yes	Yes	Yes	Returns cumulative coverage number (0...100)
get_inst_coverage()	Yes	Yes	Yes	Returns coverage number (0...100)
get_hitcount()	Yes	Yes	No	Returns hit count for coverpoint bins in covergroup type.

Method	Can be called on			Description
	Covergroup	*Coverpoint*	*Cross*	
get_inst_hitcount()	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.

  Note: 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

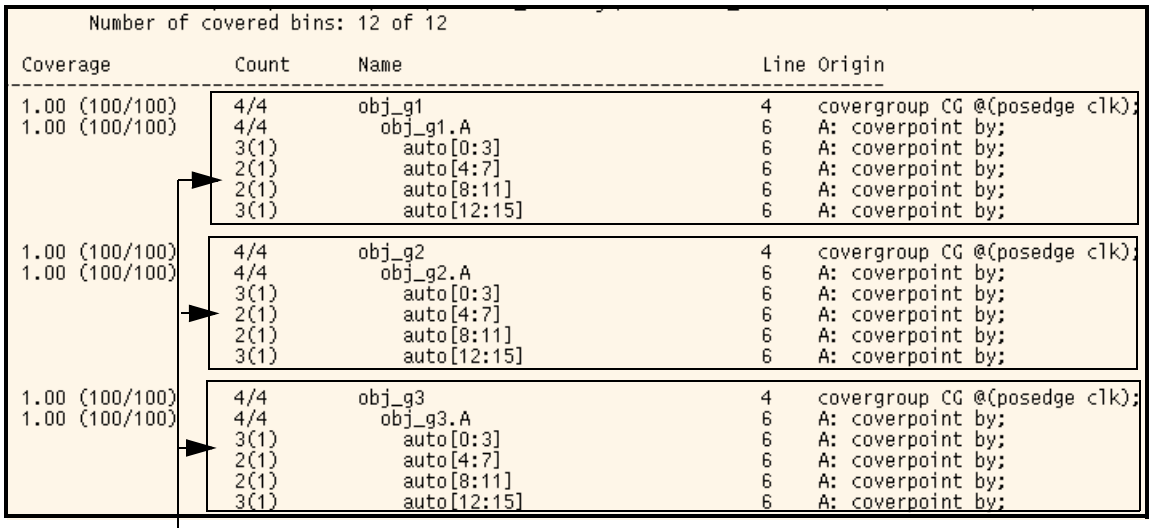
The `set_inst_name(string)` method assigns a name to a covergroup instance. The string supplied as an argument is the name assigned to the covergroup instance.

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:



The above report is an instance-based report generated using the `-instance` option of the `report_detail` command. The instances are reported only if the `per_instance` covergroup option is set to 1 or the `set_covergroup -per_instance_default_one` command is used in the coverage configuration file at elaboration.

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 ICC automatically generates a unique name for it. The covergroup name generated by ICC 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};
      }
      function new();
        CG = new;
      endfunction
    endgroup
  endclass

  C1 cobj;

  initial
  begin
    cobj = new;
    ...
  end
endmodule
```

The instance-based report from the above code is:

```
Instance name: top
Type name: top
File name: /vobs/nc_test/vlog/coverage/covergroups/test.v
Number of covered bins: 1 of 1
Number of uncovered bins: 0 of 1
Number of excluded cover bins: 0

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 a *Type-Based report*.

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.

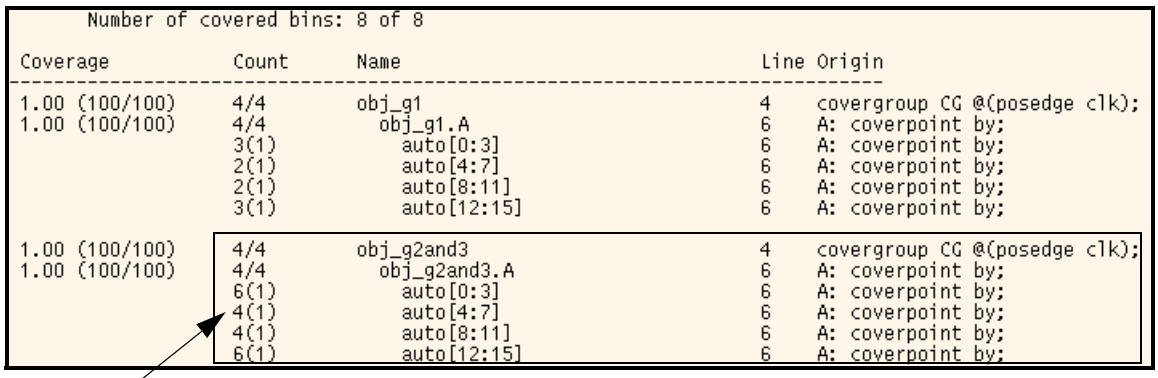
Prior to this release, type-based information was reported along with the instance-based information in the covergroup report using the `-instance` option of the `report_detail` command. With this release, type-based information is reported if a module-based report is generated with the `-module` option of the `report_detail` command.

For more details on reporting, see Chapter 11, “Analyzing Coverage Data”.

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:



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.

6.2.4.1 Limitations while Using Predefined Methods

- Out of module references to predefined methods is not supported.

6.2.5 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 `$set_coverage_db_name(`*`filename`*`)` system task perform same function as the `coverage -setup -test <`*`testname`*`>` command and the `-covtest` ncsim/irun command-line option. If the coverage database name is set using either the `coverage -setup -test <`*`testname`*`>` command or the `-covtest` option, and the `$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;
  real type_cov1, type_cov2, all_type_cov, d_get_cov;
  covergroup cg1 @(opcode1);
    type_option.weight = 2;
    c1: coverpoint opcode1;
  endgroup : cg1

    cg1 cg1_inst1 = new;

  covergroup cg2 @(opcode2);
    type_option.weight = 3;
    c1: coverpoint opcode2;
  endgroup : cg2

    cg2 cg2_inst1 = new;

  initial begin
    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", d_get_cov);

  end
endmodule
```

6.2.6 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)

6.2.6.1 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
at_least	1	Minimum number of hits for each bin to be considered covered.
goal	100	Specifies target goal for a covergroup instance or for a coverpoint or a cross of an instance.
weight	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.
name	Unique name	Specifies the name for the covergroup instance. If not specified, the tool automatically generates it.
comment	" "	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.
per_instance	0	Specifies if each instance contributes to the overall coverage. Value 1 indicates that each instance of the covergroup is saved in the coverage database and included in the coverage report. Default value is 0, which indicates that the coverage of the covergroup instances should not be saved to the coverage database.
auto_bin_max	64	Maximum number of automatically created bins, when no bins are explicitly defined for the coverpoint.

Option Name	Default	Description
cross_auto_bin_max	NA	Specifies if the information about all the cross autobins will be reported. When this option is specified, the information about only significant cross bins will be reported and all of the cross autobins will not be reported. The value of this option can only be specified as 0. If this option is not specified, all cross autobins will be reported.
detect_overlap	0	If true, generates a warning when there is an overlap between the range list (or transition list) of bins of a coverpoint. Default value is 0, which indicates no warning will be generated for overlapping range lists.
cross_num_print_missing	" "	Specifies the number of uncovered automatically generated cross bins that must be printed in the coverage report. By default, all of the uncovered automatically generated cross bins will be printed in the coverage reports.

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 on page 149.

Important

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	*Coverpoint*	*Cross*
at_least	Yes	Yes	Yes
goal	Yes	Yes	Yes
weight	Yes	Yes	Yes
name	Yes	No	No
comment	Yes	Yes	Yes
per_instance	Yes	No	No
auto_bin_max	Yes	Yes	No
cross_auto_bin_max	Yes	No	Yes
detect_overlap	Yes	Yes	No
cross_num_print_missing	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

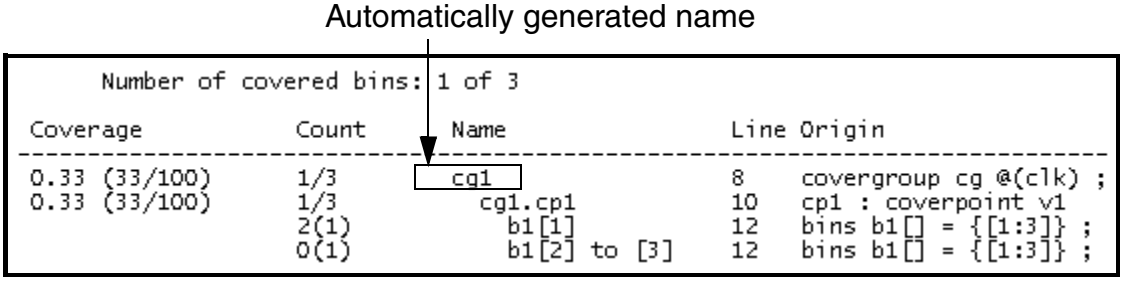
As per SystemVerilog LRM, the `per_instance` option is set as `0` and therefore, in an instance-based report, coverage for the covergroup instance is not reported. To save coverage of covergroup instances to the coverage database and to print coverage of

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:



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, ICC automatically generates a name 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.

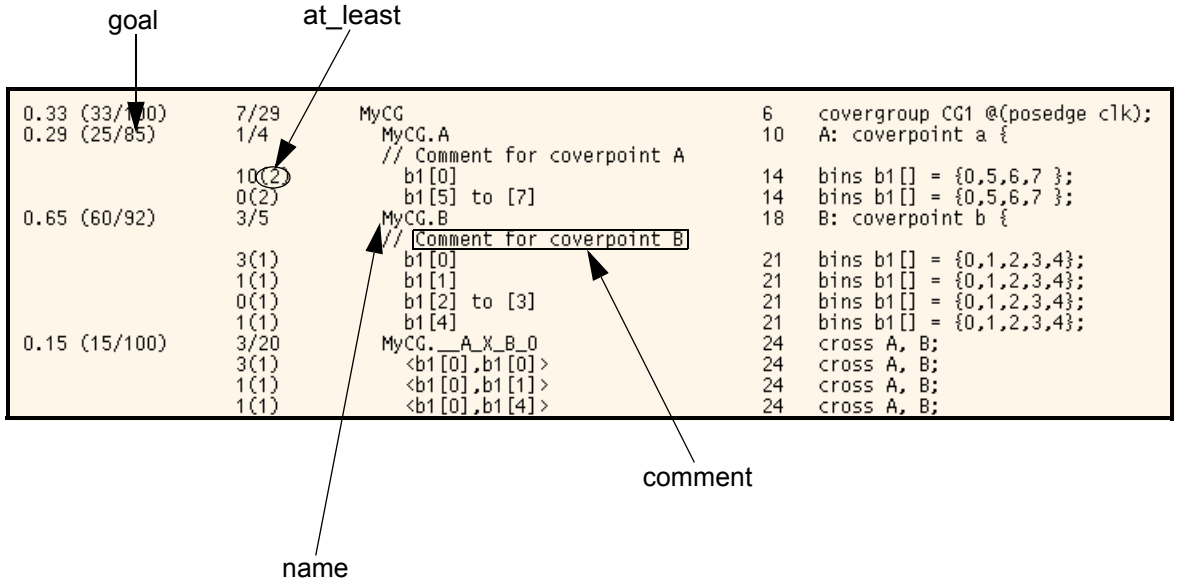
Example: Using at_least, goal, name, and comment Options

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
-----------------------------------------------------------
        cg_inst1_name       10     covergroup cg
        cg_inst1_name.v1    11     coverpoint v1;
1       auto[0]             11     coverpoint v1;
1       auto[1]             11     coverpoint v1;
        cg_inst2_name       10     covergroup cg
        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

The `cross_auto_bin_max` option specifies if all the cross autobins are saved in the coverage report.

When specified, you can set the value of this option only as `0`, which indicates that only significant cross bins will be saved and all of the cross autobins will not be saved. If this option is not specified, all the significant bins along with auto crossbins will be saved in the database.

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 {
                           // CG with cross_auto_bin_max=0
                 0(1)        b11[1]    17   bins b11[] = { [1:3] };
                 1(1)        b11[2]    17   bins b11[] = { [1:3] };
                 1(1)        b11[3]    17   bins b11[] = { [1:3] };
                 1(1)        b12[8]    18   bins b12[] = { [8:9] };
                 1(1)        b12[9]    18   bins b12[] = { [8:9] };
  0.80 (80/100)  4/5      cg1.B       20   B: coverpoint b {
                 0(1)        b21[3]    21   bins b21[] = { [3:5] };
                 1(1)        b21[4]    21   bins b21[] = { [3:5] };
                 1(1)        b21[5]    21   bins b21[] = { [3:5] };
                 1(1)        b22[10]   22   bins b22[] = { [10:11] };
                 1(1)        b22[11]   22   bins b22[] = { [10:11] };
  0.33 (33/100)  1/3      cg1.CR1     24   CR1: cross A, B {
                 0(1)        b1        25   bins b1 = binsof (A) intersect {3} && binsof (B) intersect {6};
                 0(1)        b2        26   bins b2 = binsof (A.b12) intersect {5};
                 1(1)        b3        27   bins b3 = binsof (B.b22) intersect {11};
```

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);
```

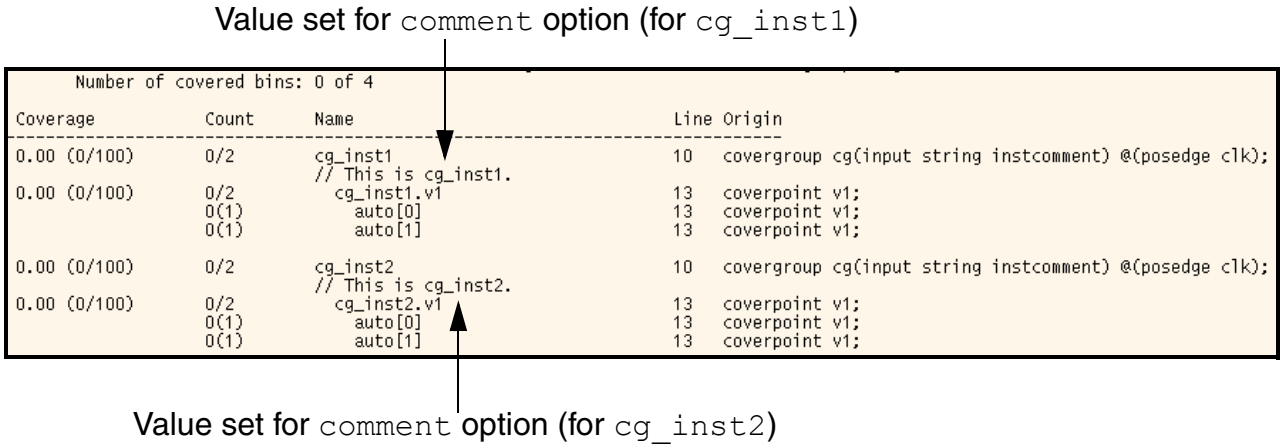
In the above code, covergroup argument `size` is used to specify the value of covergroup option `auto_bin_max`. A value `2` is passed to covergroup argument `size`, and therefore the value of `auto_bin_max` will be 3. See Example: Defining Covergroups with Arguments for more information on defining covergroup 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:



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.

6.2.6.2 Type-Specific Covergroup Options

The following table describes the type options specific to covergroup type as a whole.

Option Name	Default	Description
comment	" "	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.

Option Name	Default	Description
weight	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.
goal	100	Specifies target goal for a covergroup type or for a coverpoint or a cross of a covergroup type.
strobe	0	When true, all samples are taken at the end of the time slot, like the `$strobe system task`. With the `strobe` 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 `strobe` type option is Boolean.
real_interval	NA	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. The `real_interval` 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. **Note:** The `real_interval` type option can be used only with the type real coverpoints.

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 on page 149.

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	*Coverpoint*	*Cross*
`comment`	Yes	Yes	Yes
`weight`	Yes	Yes	Yes
`goal`	Yes	Yes	Yes
`strobe`	Yes	No	No
`real_interval`	Yes	Yes	No

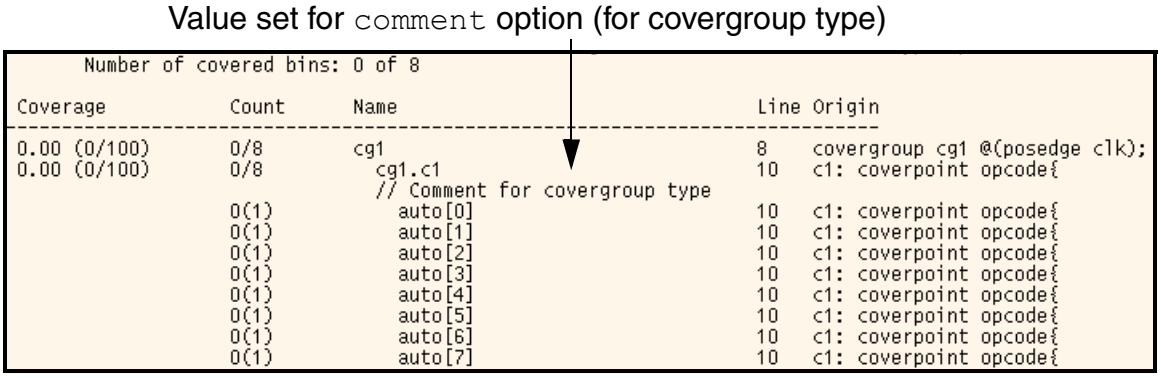
Example: Using comment Type Option

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:



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 w) @(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 w) @(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

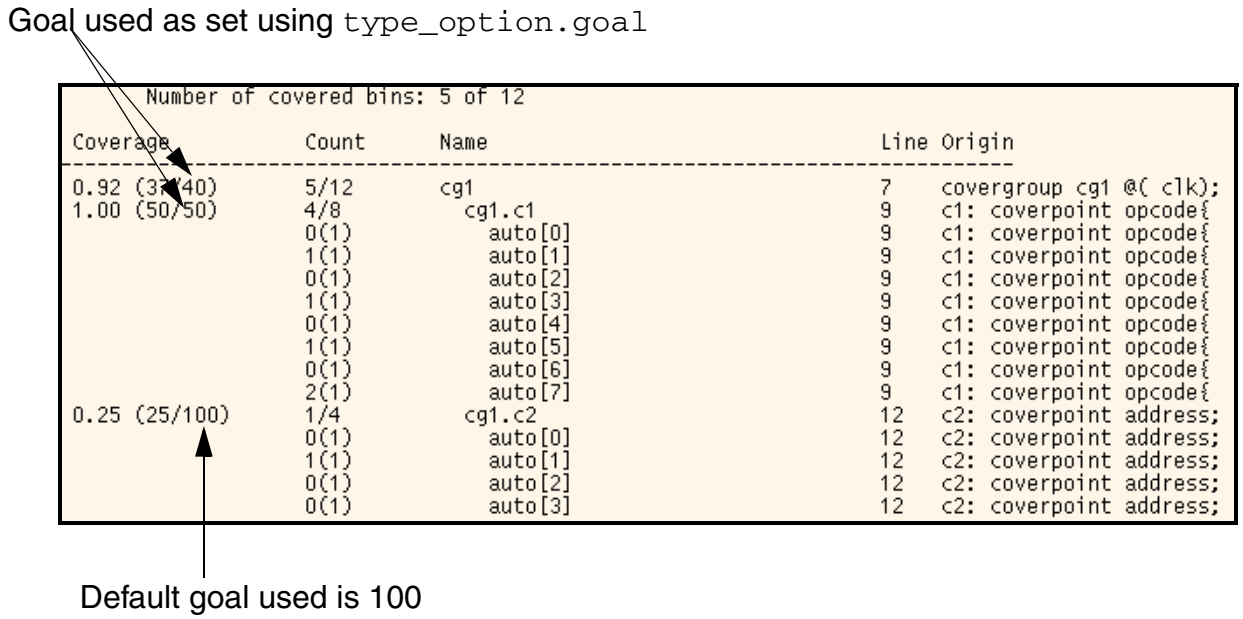
Consider the following code:

```
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:



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 NCSIM 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]};
```

6.2.6.3 Procedural Assignment of Covergroup Options

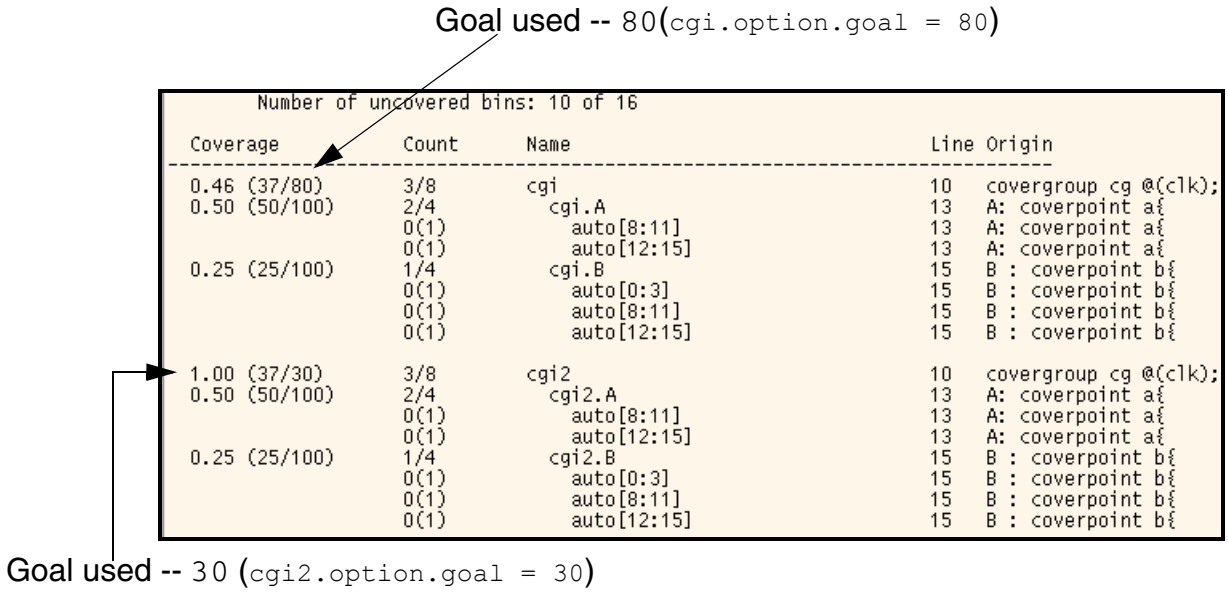
You can set covergroup options (both instance-specific and type-specific) through procedural assignment for covergroups, coverpoints, and crosses. The type-specific options are assigned values using the `type_option` keyword and instance-specific options are assigned values using the `option` keyword.

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;
```

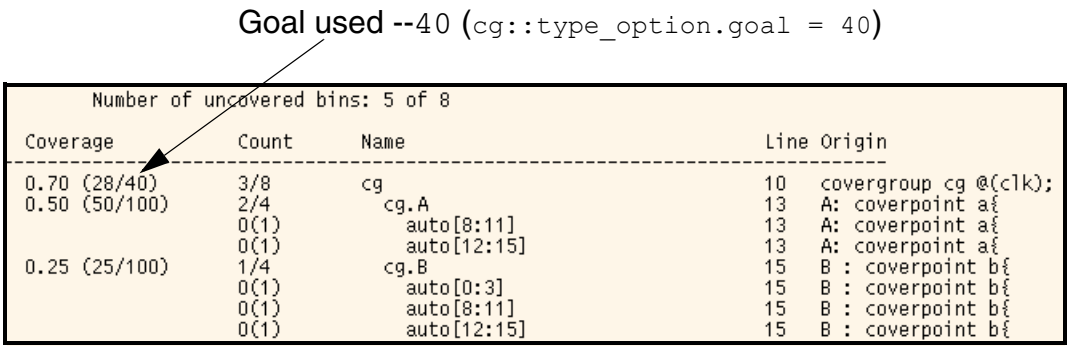
```
   }
endgroup : cg
cg cgi= new ;
cg cgi2= new;
initial begin
   cgi.set_inst_name("cgi");
   cgi2.set_inst_name("cgi2");
...
   #1 cg::type_option.goal = 40; //goal assigned to covergroup type cg is 40
   #1 cgi.option.goal = 80; //goal assigned to covergroup instance cgi is 80
   #2 cgi2.option.goal = 30; //goal assigned to covergroup instance cgi2 is 30
end
```

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



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:



In the above report, goal applied to covergroup `cg` is `40`.

Important

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	***Coverpoint***	***Cross***
`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

Note: The given table A.Options is also true for hierarchical procedural assignment of covergroup options.

B. Type Options

Type Option	Allowed in Syntactic Level		
	Covergroup	*Coverpoint*	*Cross*
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
comment	Yes	Yes	Yes
strobe	No	NA	NA
real_interval	Yes	Yes	No

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.

6.2.7 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
```

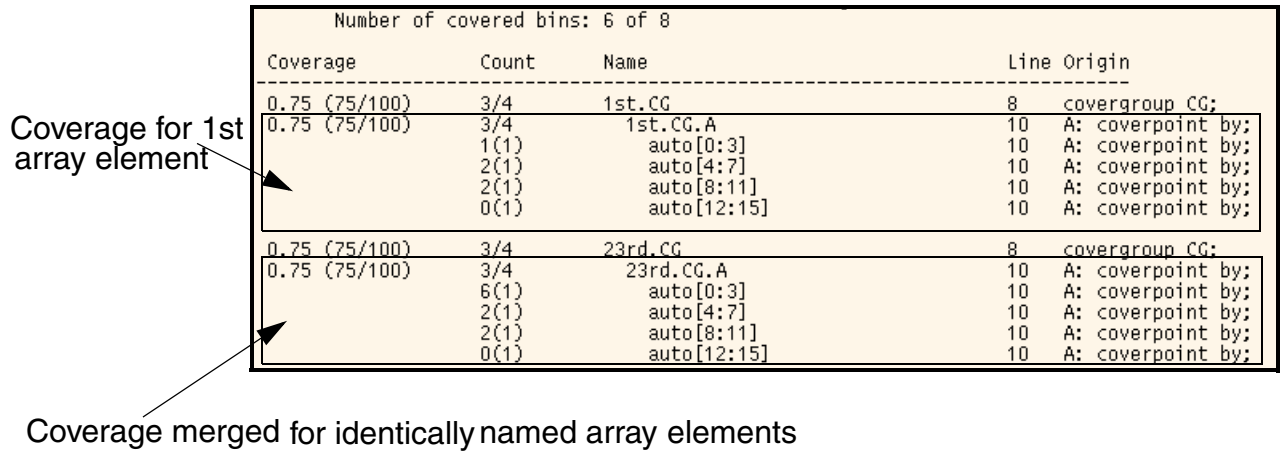
In the above code, both the parent and the derived class include a covergroup with the identical name. You use the `super` keyword to instantiate the covergroup of the parent class. You can access only one level of parent class through the `super` keyword. Using more than one level (for example `super.super.cov1`) 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:



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

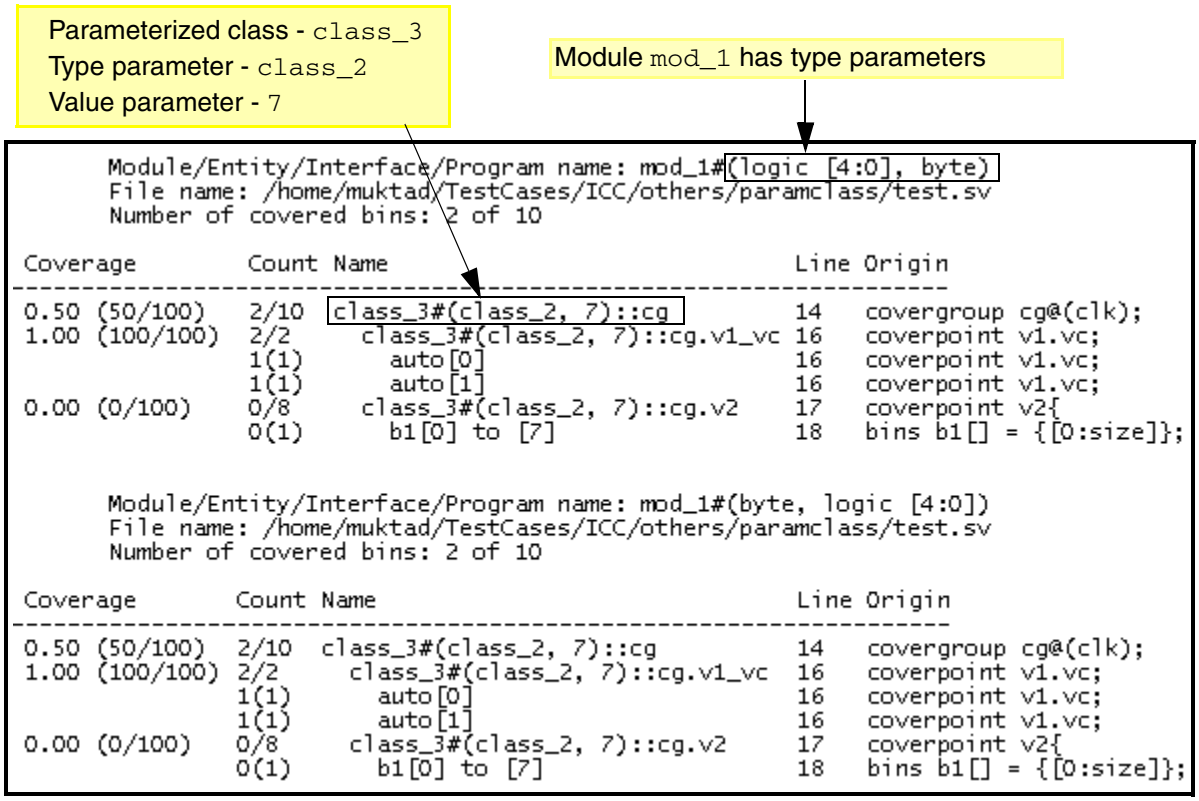
class_3 #(class_2, 7)  obj = new;

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 `t2`. For parameterized modules, the type name is appended to the module name in the report.

For details, refer to the given module-based report:




```
      Module/Entity/Interface/Program name: mod_1#(logic [4:0], byte)
      File name: /home/muktad/TestCases/ICC/others/paramclass/test.sv
      Number of covered bins: 2 of 10

Coverage        Count Name                                  Line Origin
-------------------------------------------------------------------------
0.50 (50/100)   2/10  class_3#(class_2, 7)::cg              14   covergroup cg@(clk);
1.00 (100/100)  2/2     class_3#(class_2, 7)::cg.v1_vc     16   coverpoint v1.vc;
                1(1)      auto[0]                          16   coverpoint v1.vc;
                1(1)      auto[1]                          16   coverpoint v1.vc;
0.00 (0/100)    0/8     class_3#(class_2, 7)::cg.v2        17   coverpoint v2{
                0(1)      b1[0] to [7]                     18   bins b1[] = {[0:size]};


      Module/Entity/Interface/Program name: mod_1#(byte, logic [4:0])
      File name: /home/muktad/TestCases/ICC/others/paramclass/test.sv
      Number of covered bins: 2 of 10

Coverage        Count Name                                  Line Origin
-------------------------------------------------------------------------
0.50 (50/100)   2/10  class_3#(class_2, 7)::cg              14   covergroup cg@(clk);
1.00 (100/100)  2/2     class_3#(class_2, 7)::cg.v1_vc     16   coverpoint v1.vc;
                1(1)      auto[0]                          16   coverpoint v1.vc;
                1(1)      auto[1]                          16   coverpoint v1.vc;
0.00 (0/100)    0/8     class_3#(class_2, 7)::cg.v2        17   coverpoint v2{
                0(1)      b1[0] to [7]                     18   bins b1[] = {[0:size]};
```

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.

6.2.8 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*.

6.2.9 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*.

6.2.10 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 output as 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*.

6.2.11 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

6.2.11.1 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
```

Note: If a covergroup declaration is inside the `generate` scope, then the module-based report (generated with the `-module` option of the `report_detail` command) does not print any covergroup items. In such scenarios, covergroup items are reported only if you generate an instance-based report.

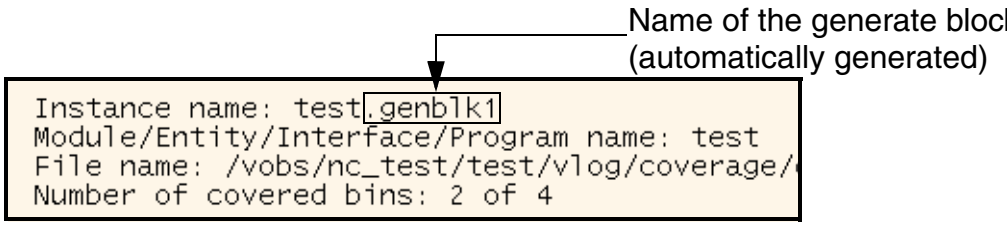
6.2.11.2 Declare the covergroup in parent scope and instantiate it in the generate block

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.

6.2.11.3 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
```

- Covergroup variable declaration and instantiation cannot reside in different scopes when one of the scope is `for-generate`. For example, the following scenarios will not work:
  - Covergroup variable declaration in the module scope, and covergroup instantiation in the `for-generate` block.
  - Covergroup variable declaration in the outer `for-generate` block, and covergroup instantiation in the inner nested `for-generate` block.

6.2.12 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, `cg`, 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]};
```

6.2.13 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.

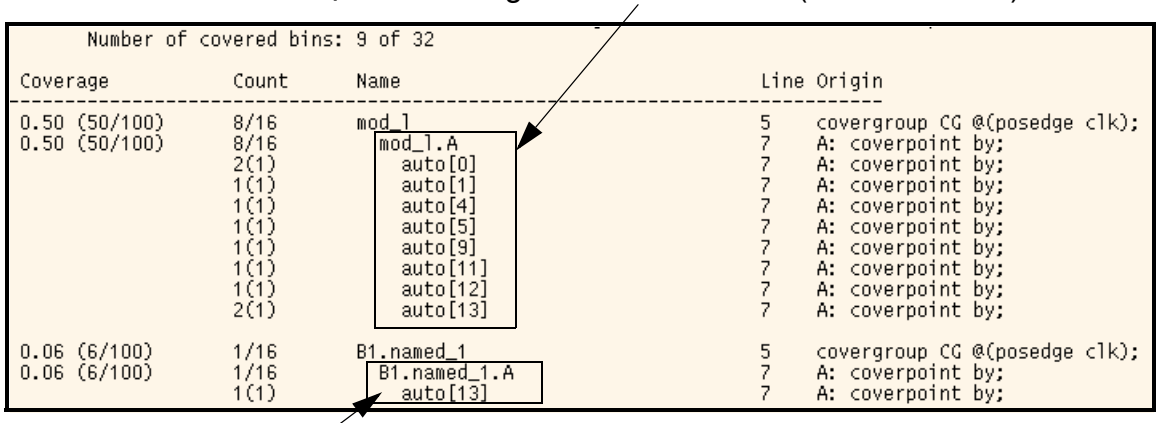
Consider the following code:

```
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.

Note: The covergroup instances created inside a class are alive until the object of the class is destroyed. In addition, covergroup instances inside a task, function, or unnamed block are not supported.

6.2.14 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:

```
irun -sv -propfile_vlog <vunit_file> -input <input.tcl> <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.

6.3 Scoring Functional Coverage

To score functional coverage, you can either:

- Pass `-coverage functional` to ncelab.
- Use the `select_functional` command in the coverage configuration file and then pass this file using the `-covfile` option to ncelab.

See Chapter 8, “Generating Coverage Data,” for more details. After simulation has dumped functional coverage data, you analyze it using the reporting tool ICCR. See Chapter 11, “Analyzing Coverage Data,” for details on ICCR and how to generate reports for analyzing functional coverage data.

6.4 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*.

7

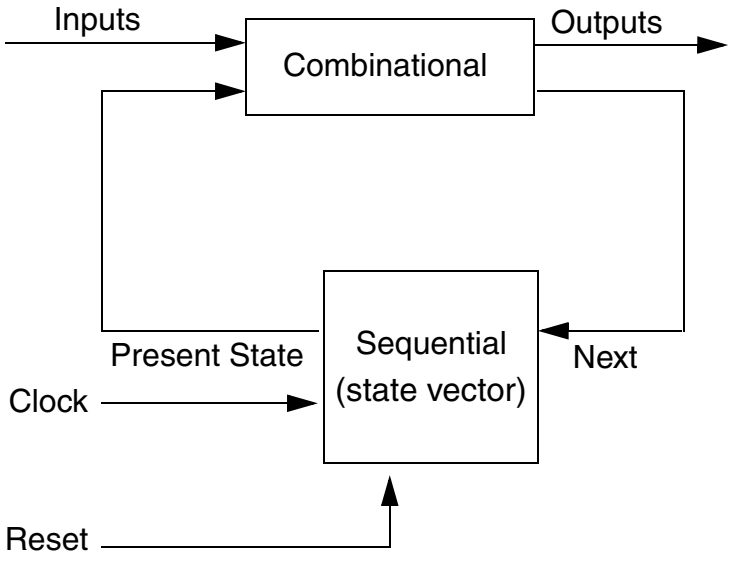
FSM Coverage

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?”

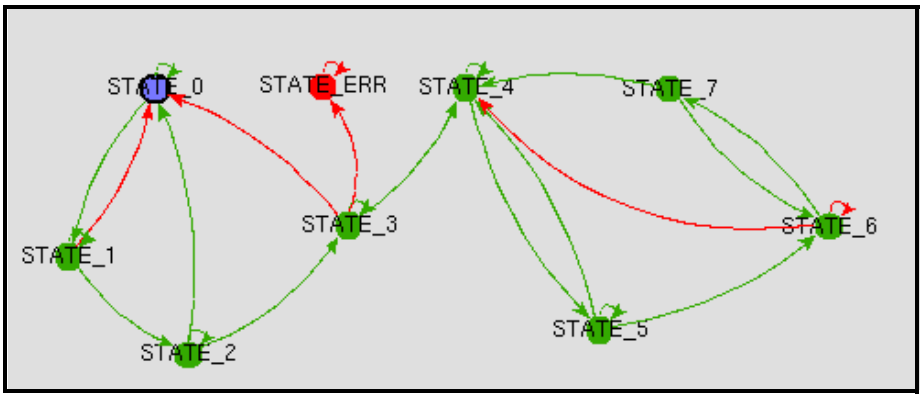
This chapter describes the basic FSM model, FSM coverage types in ICC, and FSM extraction in ICC.

7.1 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.

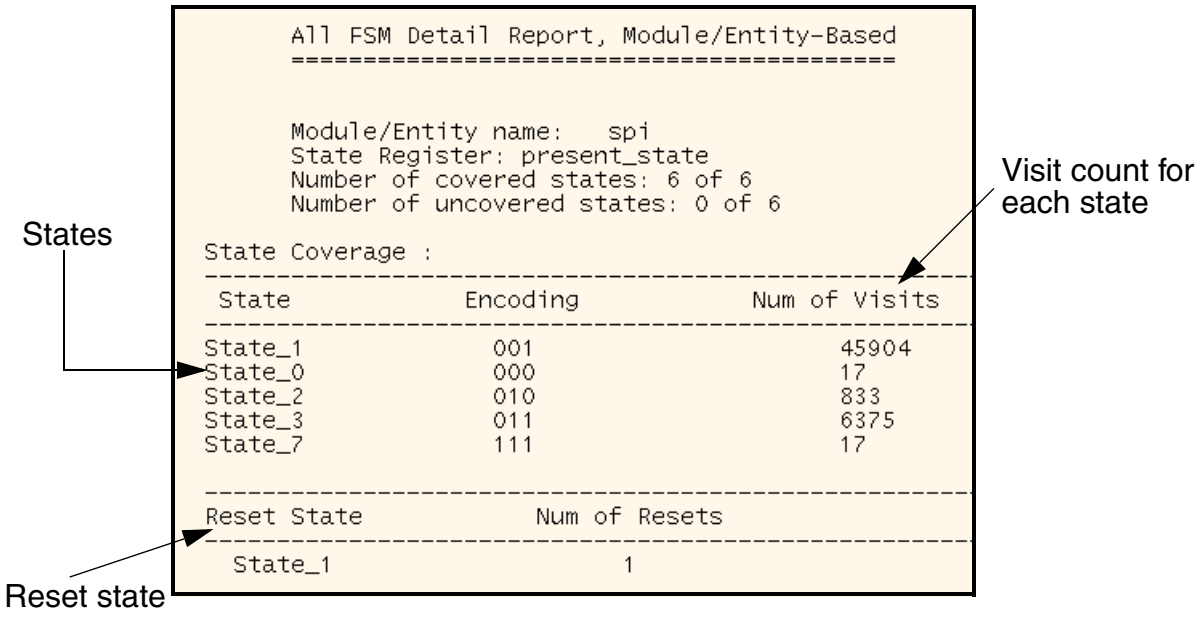
7.2 FSM Coverage Types

ICC offers the following types of FSM coverage:

- State Coverage — reports what states were visited
- Transition Coverage — reports what transitions occurred
- Arc Coverage — reports why each transition occurred

7.2.1 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.



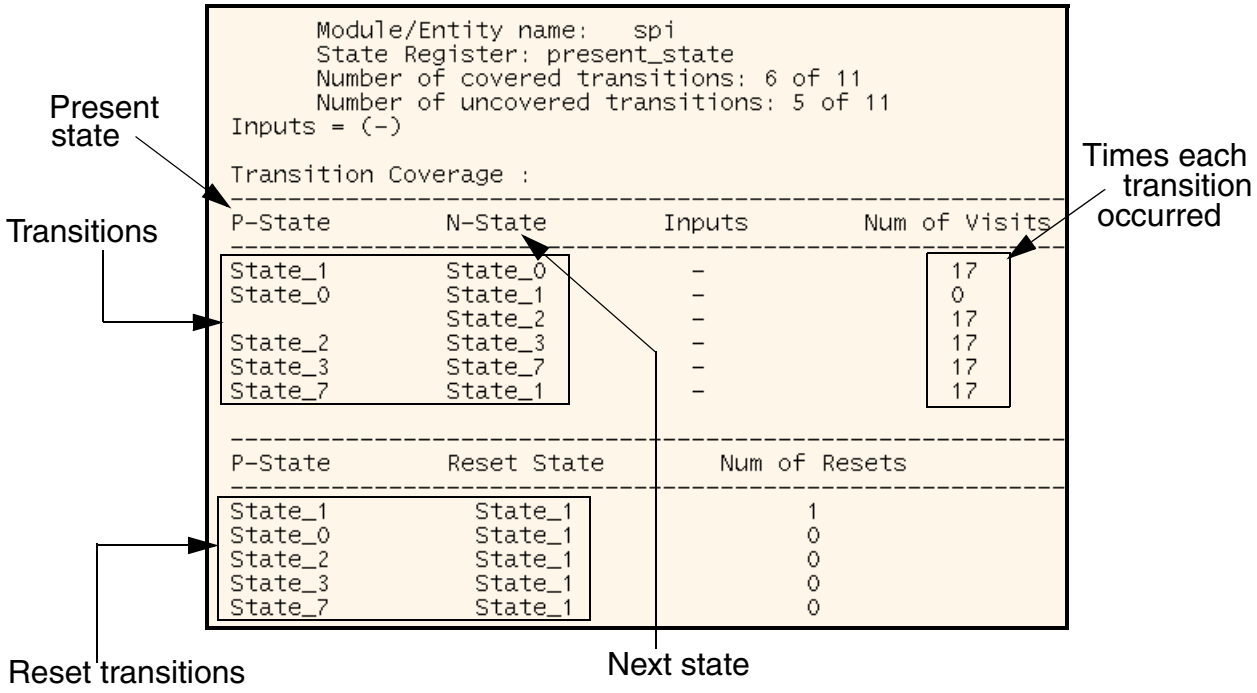
ICC considers a state as a reset state if transition to that state is not dependent on the current state of the FSM. For example, in the following code, states `S0`, `S1`, and `S2` are inferred as reset states because transition to these states is independent of current state `present_state`.

```
case(instr)
   I0: nxt_state = S0;
   I1: nxt_state = S1;
   I2: nxt_state = S2;
endcase
case(present_state)
   ...
```

See Chapter 11, “Analyzing Coverage Data,” for details on ICCR and how to generate reports for analyzing FSM coverage data.

7.2.2 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.

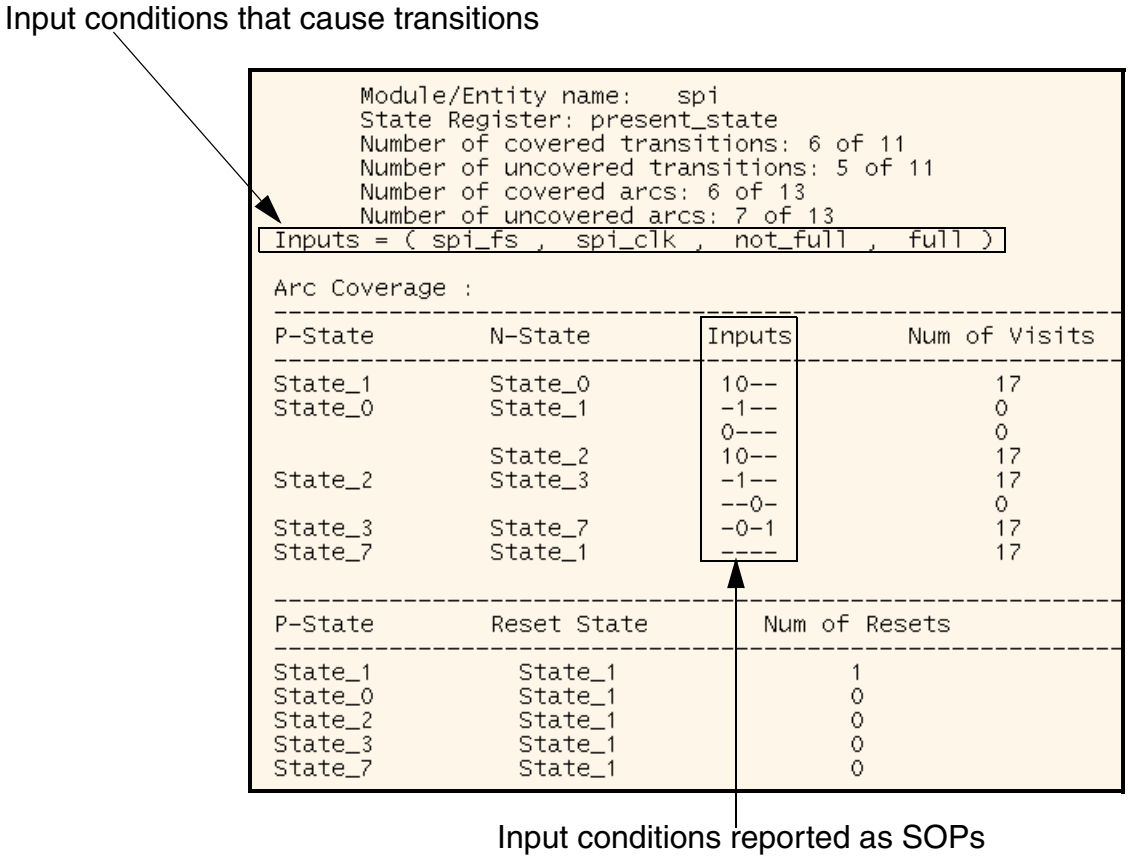


See Chapter 11, “Analyzing Coverage Data,” for details on ICCR and how to generate reports for analyzing FSM coverage data.

7.2.3 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.



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.

7.3 Modeling Styles Supported

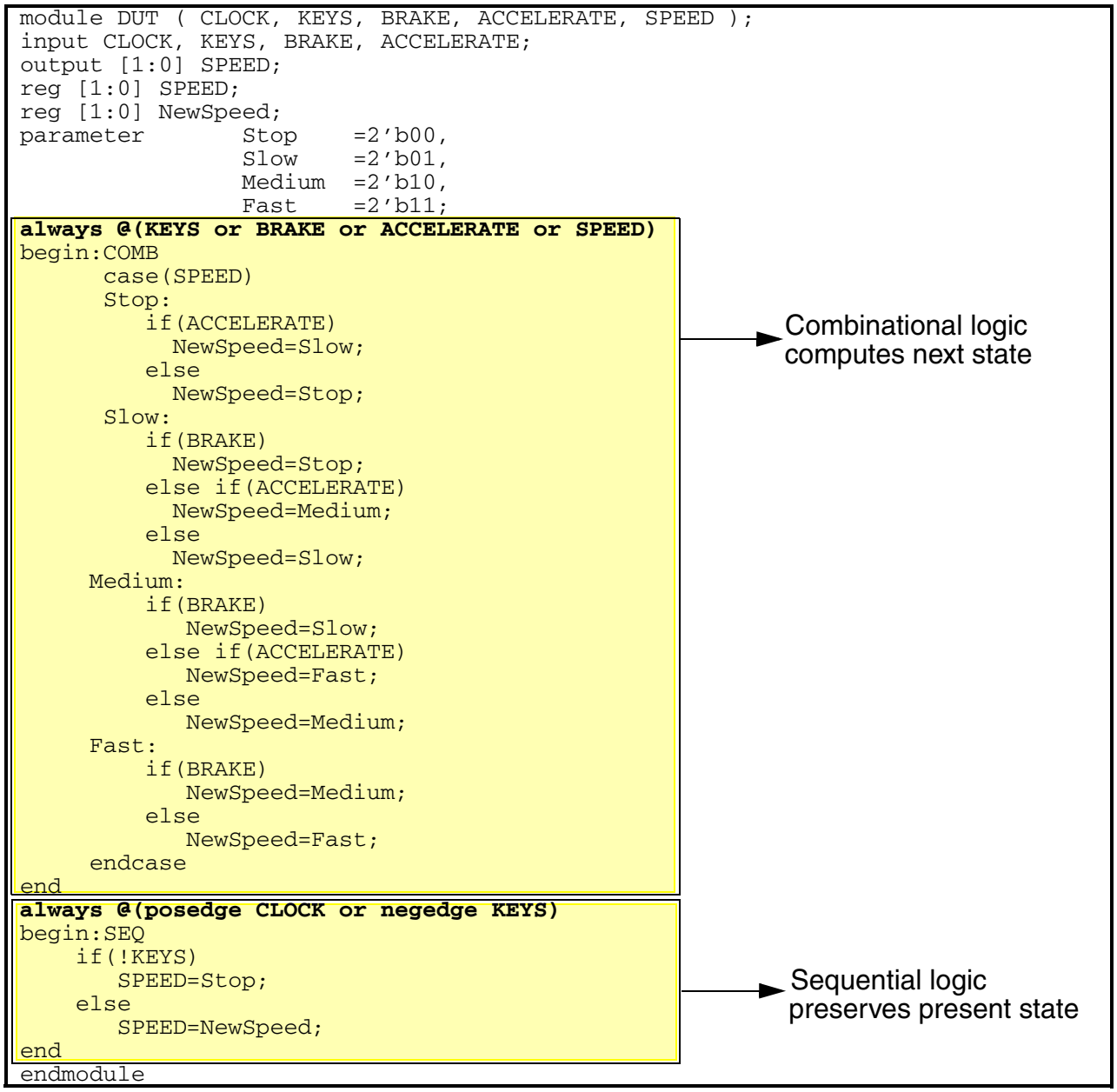
ICC extracts FSMs in a design if they are written in any of the following styles and not subject to any of the documented limitations:

- Two Process Modeling Style
- Single Process Modeling Style
- One-Hot Encoding Style

7.3.1 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

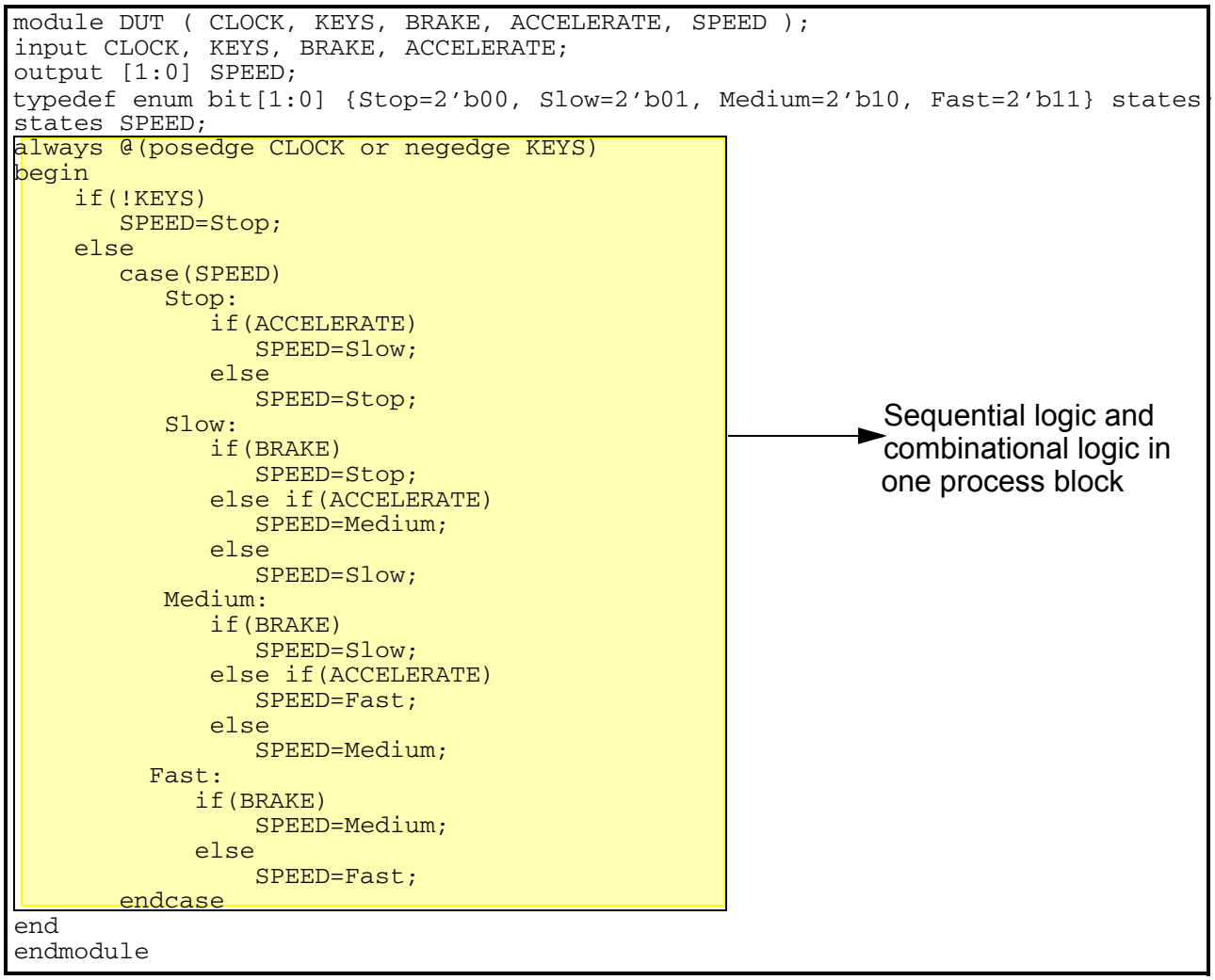
sequential logic block preserves the present state and is sensitive to the clock edge event and all asynchronous resets (if any). The following figure displays a two process FSM.



7.3.2 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.

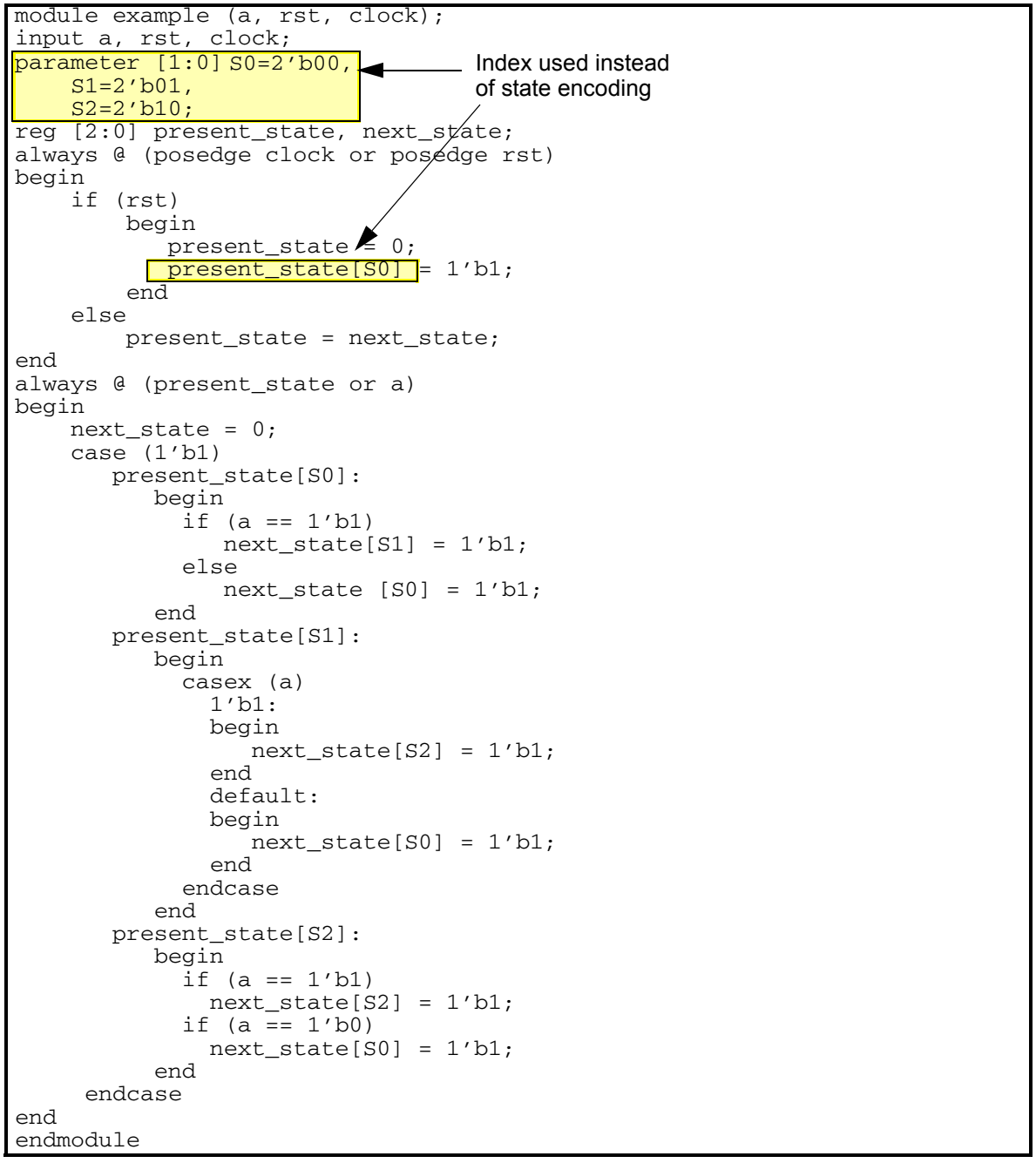
Figure 7-1 Single Process FSM



7.3.3 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 7-2 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
```

7.4 FSM Extraction in ICC

ICC extracts FSMs for following `if`/`if-else`/`case` models:

- 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)
   ...
...
```

ICC generates internal state names 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.

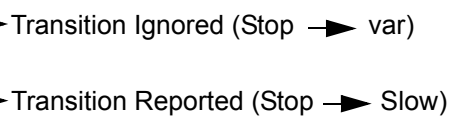
```
`define    Stop     2'b00   → State name generated as State_0
`define    Slow     2'b01   → State name generated as State_1
`define    Medium   2'b10   → State name generated as State_2
`define    Fast     2'b11   → State name generated as State_3
```

ICC extracts states and transitions 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:
```

ICC extracts transitions 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;
         else if (ACCELERATE)
            NewSpeed=Slow;
      end
```



ICC extracts single-bit FSMs 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, ICC extracts FSMs 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
```

ICC extracts FSM for VHDL states of only following discrete types or discrete arrays:

- `std_ulogic`
- `std_ulogic_vector`
- `std_logic`
- `std_logic_vector`
- `unsigned`
- `signed`
- `bit`
- `bit_vector`
- `boolean`
- `integer`
- `enumerated`
- `character`

7.4.1 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

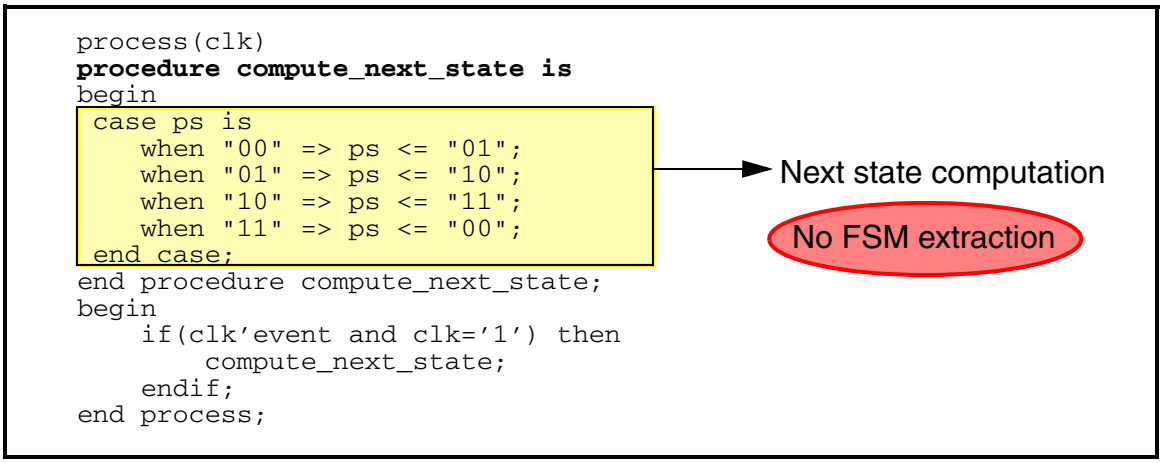
- FSM extraction does not happen for a Verilog state variable if the present state or the next state are assigned as part select or bit select. (Bit select support is applicable only if FSM is determined to be one-hot.)
- If the FSM description for a state register differs across instances of a module, 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 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



7.4.2 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)
   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.

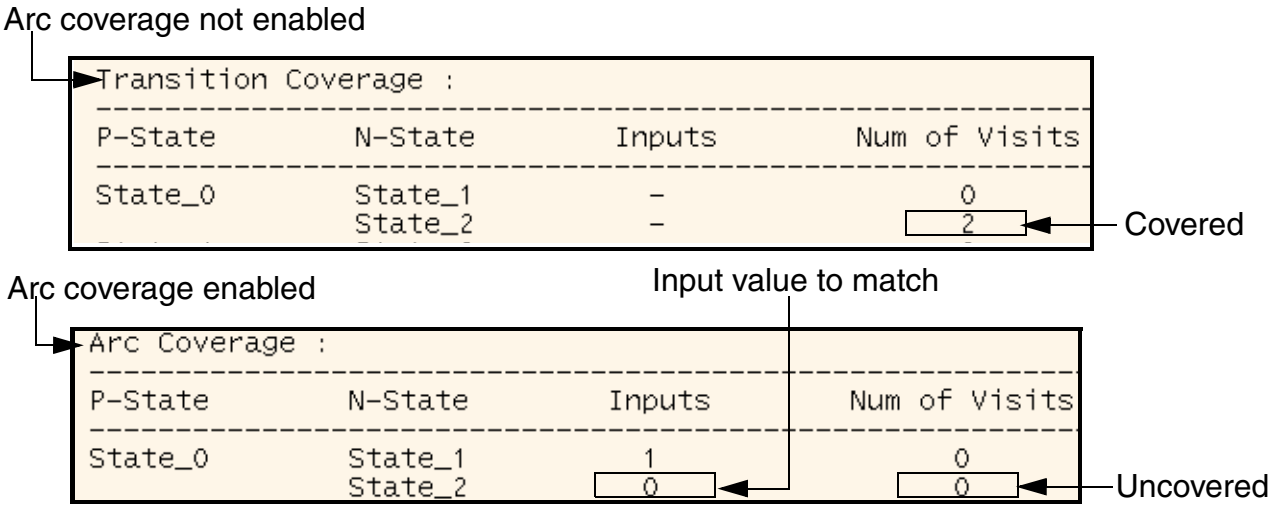
Note: If it is certain that the delays have no impact on the arc coverage results, you can disable checking for delays using the `set_fsm_arc_scoring -no_delay_check` command.

- 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.



7.5 Scoring FSM Coverage

To score FSM coverage, you can either:

- Pass `-coverage fsm` to ncelab.
- Use the `select_fsm` command in the coverage configuration file and then pass this file using the `-covfile` option to ncelab.

See Chapter 8, "Generating Coverage Data," for more details.

After simulation has dumped FSM coverage data, you analyze it using the reporting tool ICCR. See Chapter 11, "Analyzing Coverage Data," for details on ICCR and how to generate reports for analyzing FSM coverage data.

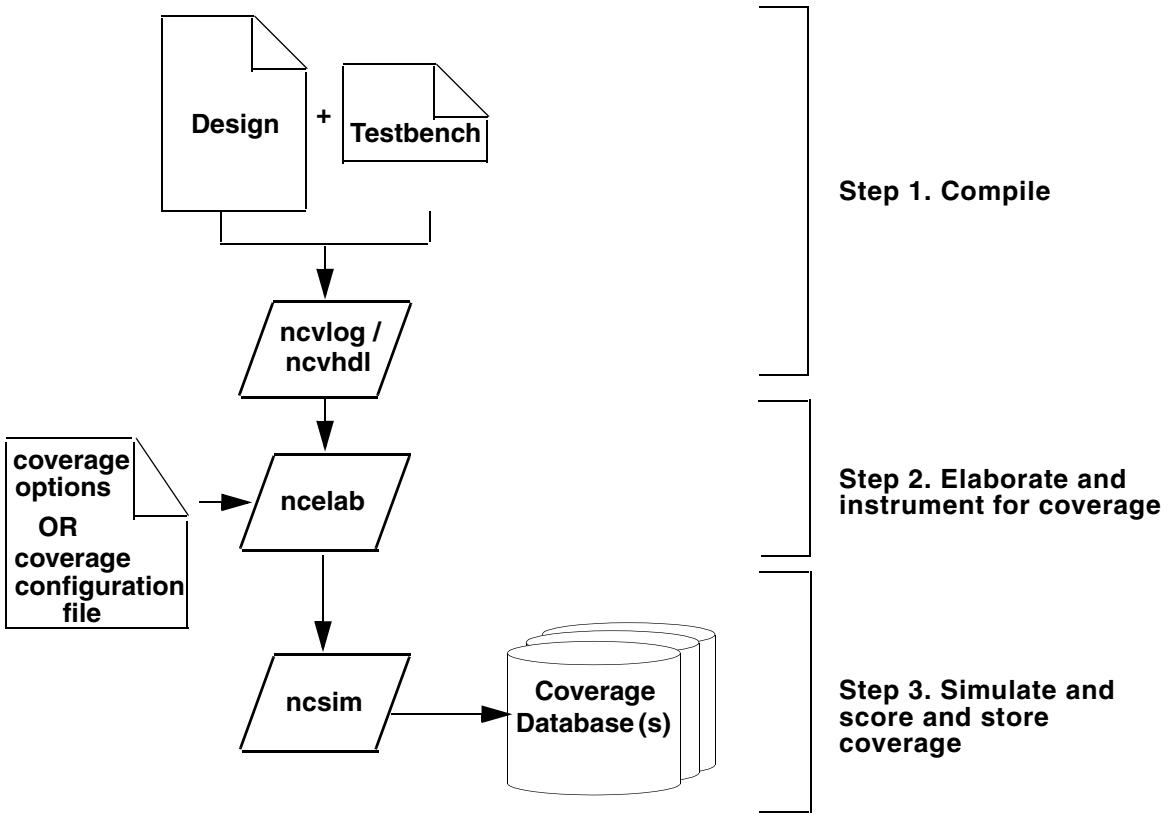
8

Generating Coverage Data

This chapter explains how to generate coverage data using multi-step simulation and single-step simulation.

8.1 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 (`ncvlog/ncvhdl/ncsc/ncsc_run/irun`)
- Elaborating the Design (`ncelab`)
- Simulating the Design (`ncsim`)

8.1.1 Compiling the Design

To compile the design, use:

```
% ncvlog [compilation_options] source_file [source_file]...
```

or

```
% ncvhdl [compilation_options] source_file [source_file]...
```

For details, see:

- *Compiling Verilog Source Files with ncvlog* of the *NC-Verilog Simulator Help*.
- *Compiling VHDL Source Files with ncvhdl* of the *NC-VHDL Simulator Help*.
- *Mixed Verilog/VHDL Simulation* of the *NC-Verilog Simulator Help*.

8.1.2 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:

```
% ncelab [-coverage <coverage_types> | k]
         [-covdut <DUT_module>]
         [other_elaboration_options]
         [lib.]cell[:view]
```

The coverage options while elaborating the design are:

- -coverage <coverage_types>
- -covfile <coverage_configuration_file>
- -covdut <DUT_module>

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 on page 239.

-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
- *f****U****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 ncelab 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.

Following is the list of configuration file commands for different coverage types.

Command	Applies to
select_coverage	Block, Expression, Toggle, FSM
deselect_coverage	Block, Expression, Toggle, FSM
set_implicit_block_scoring	Block
set_explicit_block_scoring	Block

Command	Applies to
set_assign_scoring	Block
set_branch_scoring	Block
set_statement_scoring	Block
set_vhd_case_branch	Block
set_glitch_strobe	Block, Expression
set_hit_count_limit	Block, Expression
set_subprogram_scoring	Block, Expression
set_com	Block, Expression, Toggle
set_com_interface	Block, Expression, Toggle
set_expr_scoring	Expression
set_expr_coverable_operators	Expression
set_code_fine_grained_merging	Block, Expression
set_fsm_attribute	FSM
select_fsm	FSM
deselect_fsm	FSM
set_fsm_reset_scoring	FSM
set_fsm_arc_scoring	FSM
set_fsm_arc_termlimit	FSM
set_fsm_scoring	FSM
set_toggle_strobe	Toggle
set_toggle_limit	Toggle
set_toggle_includex	Toggle
set_toggle_includez	Toggle
set_toggle_noports	Toggle
set_toggle_portsonly	Toggle
set_toggle_scoring -sv_enum	Toggle
set_toggle_excludefile	Toggle

Command	Applies to
set_optimize	Functional (assertion coverage)
select_functional	Functional
set_covergroup	Functional (covergroup coverage)
set_covergroup -detect_overlap	Functional
set_libcell_scoring	All
set_merge_with_libname	All
set_parameterized_module_coverage	All

select_coverage

To select the design units or instances to score for a coverage type, use:

```
select_coverage {<coverages>}
     [[-module] <list> | -instance <list> | -file <list> | -filelist <filename>]
     [-sysv_bind_modules]$ENV_VARIABLE
```

where

```
coverages ::= [-block] [-expression] [-toggle] [-fsm] [-all] [-betf]
```

In the above syntax,

- `<coverages>` specifies the type of coverage to score.
- `-module <list> | -instance <list> | -file` <list> specifies the list of modules, instances, and files to which the selected coverage applies. The following wildcards can be used in `<list>`:

Wildcard	Description	Example

*	Matches any text from current location until next delimiter or end of string	`select_coverage -block -module test_mod*`
		Matches all names beginning with `test_mod`, such as `test_mod1`, `test_module`, and so on.
		`select_coverage -all -file test*`
		Matches all filenames beginning with `test` that include valid design modules.

?	Matches any single character	`select_coverage -all -module test_mod?` Matches all names beginning with `test_mod` followed by a single character such as `test_mod1` and `test_mod2`.

...	Matches that module/ instance/file path and all its descendents	`select_coverage -all -module mod...`
		Matches all instances of `mod` and all of their descendents.
		`select_coverage -all -file /home/master/...`
		Matches all files under `/home/master/` and all the folders below `/home/master/`.

- `-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 `#`.

Important

When specifying the files in `<list>` or in the file `<filename>`, remember that:

- The filenames can be absolute paths (for example, `/home/master/design/test.v`), relative paths (for example, `design/test.v`, `./test.v`), or just the 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:

  ```
  ncvlog <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`.

  Note: 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
```

Note: If none of the options (-module, -instance, -file, -filelist) is specified, -module is assumed.

Important

The new support of FSM instrumentation through (de)select_coverage is the preferred and easier-to-use model for selecting/deselecting FSMs. The old (de)select_fsm will be discontinued soon. The two commands cannot be used together.

For a Verilog 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>.*
```

Note: 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.

- -sysv_bind_modules enables/disables selected coverage types on all instances that are binded using sysv 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}
```

deselect_coverage

To subtract from what is instrumented, use:

```
deselect_coverage {<coverages>}
     [ [-module] <list> | -instance <list> | -file <list> | -filelist <filename>
     | [-remove_empty_instances]] [-sysv_bind_modules]
```

where

```
coverages ::= [-block] [-expression] [-toggle] [-fsm] [-all] [-betf]
```

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 `deselect_coverage -sysv_bind_modules -tbe` CCF command, the given 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/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;
```

See `select_coverage` for details on syntax description. For details on `-remove_empty_instances`, see Removing Empty Instances From the Coverage Hierarchy on page 197.

If there are multiple `select_coverage` and `deselect_coverage` commands in a configuration file, then each command builds on the previous.

Consider the following commands in the configuration file and its impact:

Command	Impact
`select_coverage -bet -instance *...`	Selects the entire hierarchy for block, expression, and toggle coverage
`deselect_coverage -be -instance top:vhdltest:mctl...`	Deselects hierarchy under the instance named `top:vhdltest:mctl`. As a result, block and expression coverage will be ignored for the hierarchy under this instance.
`select_coverage -e -instance top:vhdltest:mctl:mctl:mem8x256`	Enables only expression coverage for the instance named `top:vhdltest:mctl:mctl:mem8x256`
`deselect_coverage -b -module memctl`	Deselects all instances of module `memctl`

Note: 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. However, ICCR commands allow you to directly specify generate blocks.

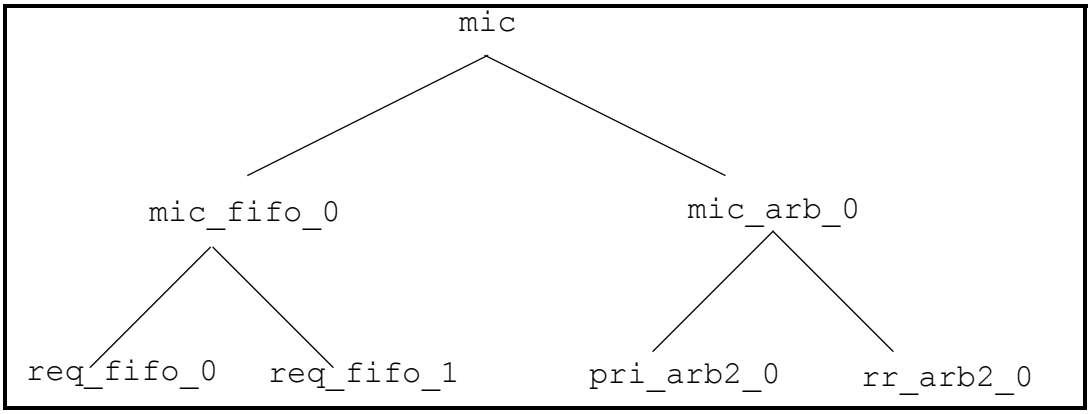
Removing Empty Instances From the Coverage Hierarchy

By default, ICCR dumps the complete design hierarchy even if the coverage for 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.

Note: 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. In addition, the `-remove_empty_instances` option cannot be specified with the `select_coverage` command.

Note: The `-remove_empty_instances` option will not remove empty generate blocks from the coverage database. Empty generate blocks are removed along with parent instance.

set_implicit_block_scoring

By default, ICC scores implicit `else` and default `case` blocks. To disable scoring of implicit `else` and default `case` blocks, use:

```
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>]
```

where

- `-on|-off` turns scoring of implicit `else` or default `case` blocks ON or OFF.
- `[-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.

Note: This command has no effect on the `set_explicit_block_scoring` command.

set_explicit_block_scoring

By default, ICC scores all explicit Verilog `case` default and VHDL `case` others. To disable scoring of explicit Verilog `case` default and VHDL `case` others, use:

```
set_explicit_block_scoring -off
```

In specific cases, to apply explicit scoring to specific modules, use:

```
set_explicit_block_scoring {-off | -on} [<modules>]
```

where

- `-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

By default, scoring of continuous assignments is disabled. To enable scoring of Verilog continuous assignments, use the `set_assign_scoring` command.

set_branch_scoring

By default, scoring of branches is disabled. To enable scoring of individual branches, use the `set_branch_scoring` command. Branch coverage cannot be scored without enabling block coverage.

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.

Note: Branch coverage cannot be scored without enabling block coverage.

set_statement_scoring

To enable statement coverage scoring, use the `set_statement_scoring` command. With this command, the information related to number of statements within a block is also printed in the block coverage report.

Note: Statement coverage cannot be scored without enabling block coverage.

set_vhd_case_branch

`set_vhd_case_branch` command enables branch scoring corresponding to case when statements in VHDL. This command enables branch scoring behavior for "VHDL case when" similar to that of "Verilog switch case".

Note: The behavior of the `set_vhd_case_branch` command has been made default.

set_glitch_strobe

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. A process is considered stable if it is not executed again before the specified glitch rejection time.

To specify a glitch rejection time interval, use:

```
set_glitch_strobe [<time_number> <time_unit>]
```

The glitch rejection time is set as 5 ns using:

```
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.

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.

set_hit_count_limit

Sets the upper limit of hit counts during simulation run. Once the hit count limit is reached, during simulation, no more hits are added to the score counts. To set the hit count limit, use:

```
set_hit_count_limit [<limit>]
```

where `<limit>` is a numeric value in the range of `1` and `240`. The default limit, if not specified, is `1`.

An object is considered covered if hit count is at least 1.

set_subprogram_scoring

By default, coverage is scored for all Verilog subprograms (used and unused). To disable scoring of unused subprograms, use:

```
set_subprogram_scoring {-all | -used} <modules>
```

where

- `-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.

Note: 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.

By default, coverage is not scored for VHDL subprograms which are used in design scopes with block or expression coverage enabled. To enable this scoring, use:

```
set_subprogram_scoring -vhdlpackage -used
```

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.

Note:

- Using this command will not enable used subprogram coverage for Verilog modules in the design. In this case, to enable used subprogram coverage use the given command again.

  ```
  set_subprogram_scoring -used <modules>
  ```

- 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_com

COM identifies items that can never happen and marks them to be ignored. To enable marking of constant objects such as blocks, expressions, expression terms, and signals within the simulated DUT during elaboration, use:

```
set_com
```

When you enable COM, `ncsim` 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.

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]
```

In the above syntax,

- `-on`|`-off` turns COM on or off. If `-on`/`-off` options are not specified, `-on` is assumed.
- `<coverages>` turns COM on/off for specified coverage types. If coverage types are not specified, the command applies to all coverage types.
- `-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 commands, it applies to all; and the `icc.com` file is created for the whole 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 ncsim 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>/`.

  Note: Using `-logreuse` command explicitly with the `set_com` command enables reuse of the COM file for the complete design.

Tip

When you use this 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.

Important

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.

- `-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, and therefore are:
  - not dumped to the `icc.com` file.
  - excluded from toggle reports generated with the `-marked` option.

  With the `-nounconnect` option, unconnected nets and signals are treated as variables in expressions that use them.

  Note: With multiple `set_com` commands in CCF, if the `-nounconnect` option is specified with one of the commands, it applies to all.
- `-module | -instance` specifies if an `-on` or `-off` switch 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 coverage applies. If `<list>` is not specified, command applies to all modules in the design.

To enable marking of signals in modules `mod1` and `mod2`, use:

```
set_com -on mod1 mod2
```

Note: 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.

Note: When you enable COM using the `set_com` command, it does not work with vector expression scoring that is enabled using `set_expr_scorng -vector`.

Note: The `-noregunconnect` option has been made default.

set_com_interface

The command `set_com_interface` must be used when the DUT used for coverage is not the true DUT. This happens when coverage only is scored on a subset of DUT (less) or when coverage also is collected on part of the testbench (more).

To change the definition of modules for which all ports are considered variable, use:

```
set_com_interface <modules>
```

You can set top-level DUT for coverage using `-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.

Consider the following example that explains this point:

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 this case, 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>` and `<BLK_L>` will not be analyzed because they are not specified under `set_com`. These will appear in coverage reports.
- `<BLK_M>` will not be analyzed because it is not specified under `set_com`. It will appear in coverage reports.

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.

Consider specifying `-COVDUT MODA` in the `ncelab` 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.

Note: The set_com_interface command has no impact in the absence of the set_com command.

set_expr_scoring

The set_expr_scoring command:

- Enables scoring of all operators and in assignments (with the -all option).
- Specifies a scoring mode for scoring expressions (with the -sop, -control, -vector or -fcc options).
- Enables scoring of Verilog events (with the -event option).
- Enables scoring of expressions containing struct datatypes (with the -struct option)
- Disables short-circuit evaluation of VHDL AND/NAND/OR/NOR operators (with the -no_vhdl_shortcircuit option).
- Disables splitting of subexpressions that use logical ==0/!=0 (with the -vlog_remove_redundancy option).
- Enables scoring of VHDL not operator (with the -vhdl_not_as_operator option).
- Enables setting the maximum number of terms allowed in an expression marked for SOP coverage (with the -max_terms_sop option)
- Enables setting the maximum number of terms allowed in an expression marked for FCC coverage (with the -max_terms_fcc option)

The syntax of the set_expr_scoring command is:

```
set_expr_scoring [-all] [-sop | -control | -vector | -fcc] [-event]
[-struct][-no_vhdl_shortcircuit][-no_vhdl_control] [-vlog_remove_redundancy]
[-vhdl_not_as_operator][-max_terms_sop <num>][-max_terms_fcc <num>]
[<module> | *]
```

In the above syntax,

- -all enables scoring of all Verilog and VHDL operators in conditions and assignments. See Scoring of Operators 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.

- -sop enables SOP scoring of expressions.

- -control enables control scoring of expressions.

- -vector enables vector scoring of expressions.

- -fcc enables FCC scoring of expressions.

  Note: If no scoring mode is specified, by default, SOP scoring is enabled.

- -event enables scoring of Verilog events.

Note: 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.

- -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 on page 34.

- -no_vhdl_control disables control scoring for all VHDL units. You can use this command in all the mixed language designs with "set_expr_scoring -control" in the CCF file for default SOP scoring.

  Note: -no_vhdl_control is a global flag and if specified, applies to the complete design.

- -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.

- `-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.
- 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.

  Note: The `-max_terms_sop <num>` option does not apply in incremental elaboration.
- 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 no list is given, all modules are affected.

To apply vector scoring and also to score events for all applicable code, use:

```
set_expr_scoring -vector -event
```

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`.

Vector scoring is not supported for VHDL design units, which use SOP scoring instead.

set_expr_coverable_operators

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. To score coverage for all operators in condition expressions, use:

```
set_expr_coverable_operators -all
```

Note: The above command will enable scoring of all operators in condition expressions except for the VHDL not operator. To enable scoring of VHDL not operator, use the set_expr_scoring -vhdl_not_as_operator command in the coverage configuration file at elaboration.

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.

The syntax of the set_code_fine_grained_merging command is:

```
set_code_fine_grained_merging [-ignore_vunit_code_coverage]
```

where,

- -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.

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, no blocks or expressions are merged for that instance.

set_fsm_attribute

To tag an FSM state vector, use:

```
set_fsm_attribute -tag <tag> -module <module> -statereg <state_reg>
```

In the above syntax,

- -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.

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
```

Note: You cannot extract unsupported FSM styles by tagging them using the `set_fsm_attribute` command.

select_fsm

To select modules and tagged FSMs for FSM extraction, use:

```
select_fsm [ -module <modules> | -tag <tags>]
```

where

- `-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.

You can specify multiple `select_fsm` commands within a configuration file.

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 *
```

Note: 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

To ignore modules and tagged FSMs from FSM extraction, use:

```
deselect_fsm [ -module <modules> | -tag <tags>]
```

See `select_fsm` for syntax details.

Note: You can also use the deselect_coverage command to deselect FSMs. It is the preferred and easy to use command. The de`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. See select_fsm for usage examples.

You can also specify multiple `deselect_fsm` commands within a configuration file. If there are multiple `select_coverage` and `deselect_coverage` commands in a configuration file, then each command builds on the previous.

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. To enable scoring of reset states and transitions for identified FSMs, use the `set_fsm_reset_scoring` command.

set_fsm_arc_scoring

By default, ICC does not score FSM arcs. To enable scoring of arcs for identified FSMs, use:

```
set_fsm_arc_scoring [-on | -off] [ -module <modules> | -tag <tags>]
[-no_delay_check]
```

where

- `-on|-off` enables/disables scoring of arcs. If not specified, `-on` is taken as the 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.

If both `-module` and `-tag` are used, `-module` overrides `-tag`.

This 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.

Note: 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:

```
set_fsm_arc_termlimit <limit>
```

where `<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.

Note: ICC reports a warning if the term limit exceeds the default term limit or the one specified using the `set_fsm_arc_termlimit`. For such FSM transitions, 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`). To enable scoring of FSM hold transitions, use:

```
set_fsm_scoring -hold_transition
```

Note: Scoring hold transitions might impact performance.

set_toggle_strobe

Toggle coverage by default considers glitches for toggle recording. This can lead to artificially high coverage counts. To define a strobe interval for filtering glitches within the time window on nets, use:

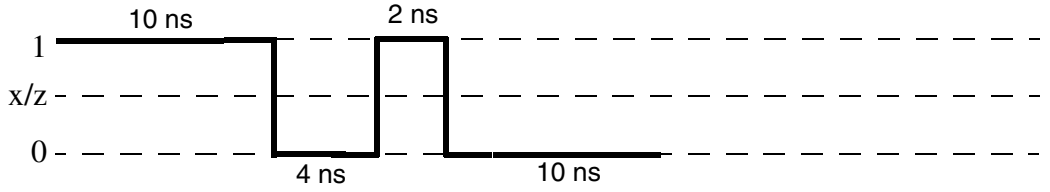
```
set_toggle_strobe <time_number> <time_unit>
```

where

- `<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`.

With glitch filtering enabled, a transition is recorded when a net transitions from one stable value to another stable value.

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 after which ncsim will stop counting. To specify a toggle limit, use:

```
set_toggle_limit <limit>
```

where `<limit>` is a numeric value in the range of 1 and 2147483647. The default limit, if not specified, is 1.

Note: 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`. To record transitions `X -> 0` and `X -> 1`, use the `set_toggle_includex` command.

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`. To record transitions `Z -> 0` and `Z -> 1`, use the `set_toggle_includez` command.

Consider transition `0 -> X -> Z -> 1 -> 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 -> 0`).

With `set_toggle_includez` command,

- The number of rise transitions is 1 (`Z -> 1`).
- The number of fall transitions is 1 (`1 -> 0`).

set_toggle_noports

To disable scoring of module ports, so that a net is scored only once within a hierarchy, use the `set_toggle_noports` command. Signals are scored only at the highest level.

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 `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

To score only module ports, when you are checking interfaces and connectivity of building blocks where little or no RTL code exists, use the `set_toggle_portsonly` command.

Consider the code shown for the `set_toggle_noports` command. With `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.

Important

The `set_toggle_noports` and `set_toggle_portsonly` commands are mutually exclusive.

set_toggle_scoring -sv_enum

By default, SystemVerilog enumerations are not scored. To enable scoring and reporting of SV enumerations, use the given ccf command:

```
set_toggle_scoring -sv_enum
```

Note:

- Reporting of toggle coverage of enumerated signals is not supported in ICCR. It is currently only supported with IMC when the `enum_toggle -set on` command is enabled before loading the test.
- Toggle scoring of enumerated objects is different from scoring of non-enumerated objects. For more information on toggle scoring of enumerated objects, refer to Chapter 4, "Toggle Coverage,".
- COM analysis is not supported for SystemVerilog enumerated signals.
- By default, 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.
  If the user initializes an enumerated variable or net to its default value at the beginning of the simulation, it will not be scored.
- Coverage of enumerated members inside SystemVerilog structures is not supported.

set_toggle_excludefile

By default, if toggle coverage is scored, all signals in the design are scored. To exclude specific signals from the design during toggle simulation, use:

```
set_toggle_excludefile [-nolog] [-bitexclude] <file>
```

where

- `-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 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
  ```

  Note: A warning is reported if `<file>` includes signals of instances/modules, which are

deselected using the `deselect_coverage` command in the coverage configuration file.

Note: The results of toggle coverage may differ in low power simulations as compared to the results in non-low power simulations. As an example, due to creation of, for example, power domain instance in low power simulations, objects inside and outside of power domain instance that may appear to be connected are actually not connected and therefore exclusion of either one object from toggle coverage does not exclude the other object.

Note: The `set_toggle_excludefile` command is supported for struct members and enums by providing additional information about these SystemVerilog constructs in the exclude file.

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 in exclude vs uncovered resolution.

Formats for specifying signals in the exclude file:

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>`

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.

Note: 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 that 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.
- 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_exclude_file 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`.

Note: 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 `o` in top-level instance.
`instance :Add:clock`	Excludes `clock` from the VHDL hierarchy :`Add`.
`instance :Add:*...`	Excludes all signals in instance :`Add` and its descendants.
`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 t1:*`	Excludes all signals from module `t1`.
`module lib3.fa(fa_str*):a`	Excludes signal `a` from all architectures starting with `fa_str` of entity `fa` in library `lib3`.
`module lib1.E1(*):*`	Excludes all signals of all architecture of entity `E1` in library `lib1`.
`module MUX:SELECT(?1)`	Excludes the signals `SELECT(11)`, `SELECT(21)`, `SELECT(31)` and so on of entity `MUX`.

`-ere instance :l1: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 `:l1:fa1:ha1` and `:l1:fa3:ha1`, the pattern should be modified as: `-ere instance :l1:fa[13]:ha1:.*`
`-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 fa+:b`	Excludes signal `b` from modules `faa`, `faaa` and so on (`a` should be present one or more times).
`-ere instance :top:dut.*:sel\([4-7]\)`	Excludes signals `sel(4)` to `sel(7)` of instances starting with `:top:dut`.

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 `ncelab` is run. This file will not be created if the `set_toggle_excludefile` command does not result in any exclusions.

Note: 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_optimize

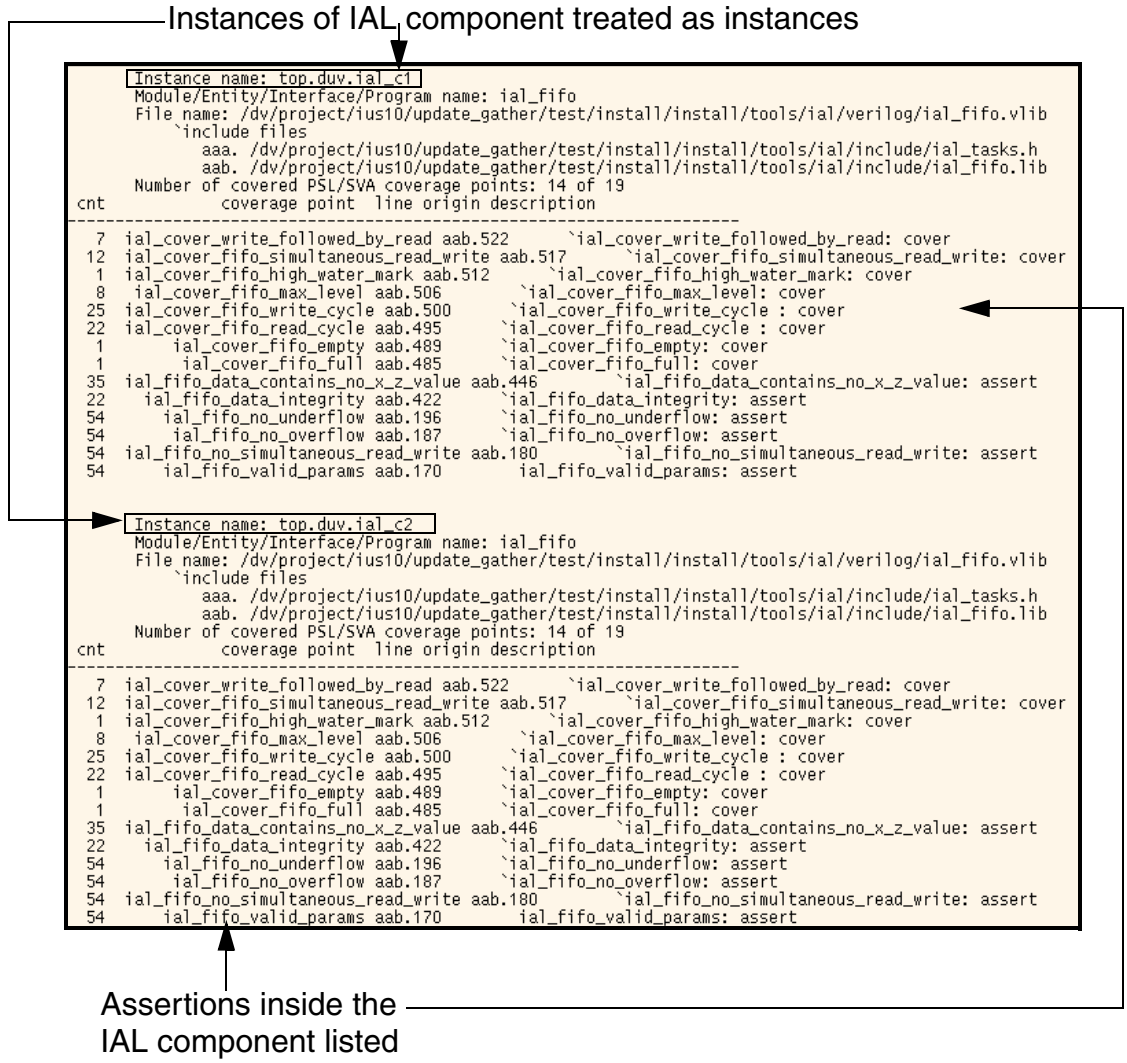
By default, instances of IAL and OVL components are treated as instances and assertions inside the IAL/OVL components are listed individually in the coverage report. If the design includes a huge number of IAL and OVL components (with a large number of assertions inside them), the coverage report becomes 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
```

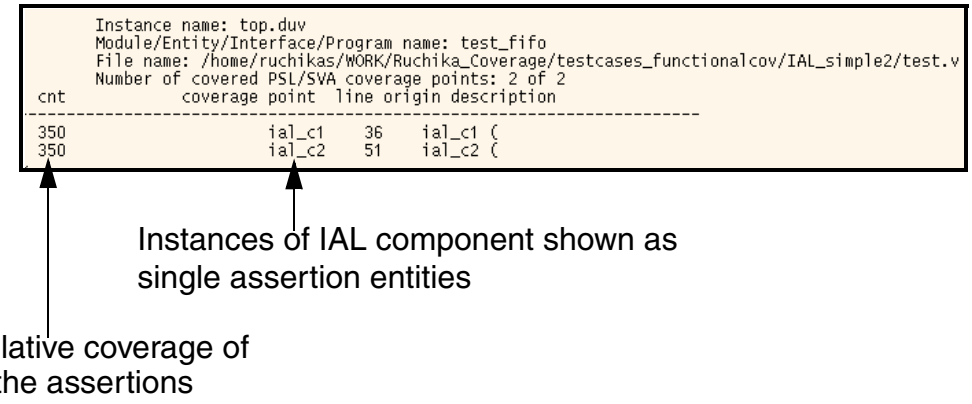
With the above command, ICC will treat instances of IAL and OVL components as individual assertions. In addition, ICCR will report rolled-up count of all of the assertions inside the component along with the instance that is treated as assertion.

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_ovl_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, as shown below:



In the above figure, instances `ial_c1` and `ial_c2` of IAL component `ial_fifo` are shown as single assertion entities and the counts are shown as the cumulative coverage (rolled-up counts) of all of the assertions inside `ial_fifo`.

select_functional

The `select_functional` command enables scoring of all compiled assertions and covergroups. If ncelab `-covdut` switch is used to provide one or more major blocks for scoring, then functional coverage is scored only within these blocks.

The `select_functional` command:

- Enables scoring of assertions inside AMS modules (with the `-ams_control` option)
- Enables reporting of immediate assertions inside a class in a package (with the `-imm_asrt_class_package` option)

select_functional -ams_control

By default, assertions inside AMS modules are not scored. To enable scoring of assertions inside AMS modules, use

```
select_functional -ams_control
```

Note: With the given 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.

select_functional -imm_asrt_class_package

By default, immediate assertions are not reported for functional coverage during elaboration. To enable reporting of immediate assertions inside a class in a package, use

```
select_functional -imm_asrt_class_package
```

Note: 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

The `set_covergroup` command:

- Reduces coverage database dumping time (with the -optimize_dump option)
- Enables reporting of uninstantiated covergroups (with the -show_uninstantiated option)
- Enables specifying the default value of the `cross_num_print_missing` covergroup option (with the -default_cross_num_print_missing option)
- Overrides the default value of the `goal` covergroup option (with the -default_goal option)
- Overrides the default value of the `goal` covergroup type option (with the -default_type_option_goal option)
- Enables saving of coverage of instances of covergroups for which the `per_instance` option is not explicitly set in the design (with the -per_instance_default_one option)
- Enables generation of a warning for any overlap in the range list or the transition list of bins for the entire design. (with the -detect_overlap option)
- Enables bin-level merging. (with the -bin_merge option)
- Treats coverpoint/cross as ungradeable, if the total number of valid bins for that particular coverpoint/cross exceeds 50 million. (with the-optimize_model option)

set_covergroup -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. To reduce the dumping time and dump all covergroup instances in the design or testbench to the coverage database, use the following command at elaboration:

```
set_covergroup -optimize_dump
```

With this command, all covergroup instances within UVM quasi-static object hierarchy are also dumped to coverage database and the coverage database performance is improved. However, the command has the following limitations:

- 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 cl ;
Covergroup cg ;
Endclass

    Cl cobj ;
Endpackage

Module top ;
    Import pkg::* ;
    Cl cobj1 ;
  Intial begin
    Cobj = new ;
    Cobj.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 -show_uninstantiated

By default, type-based coverage for uninstantiated covergroups is not reported. Covergroups instantiated within conditional statements that are not executed at simulation time are also considered uninstantiated. To report coverage for uninstantiated covergroups, use the `set_covergroup -show_uninstantiated` command. For more details on type-based coverage, see Chapter 6, “Functional Coverage”.

set_covergroup -default_cross_num_print_missing

The `set_covergroup -default_cross_num_print_missing` command 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>`. The `<value>` set using this command 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.

set_covergroup -default_goal

By default, the value of `option.goal` covergroup option is set to `100`. To override the default value of this option, use the following command in the coverage configuration file at elaboration:

```
set_covergroup -default_goal <value>
```

where, `value` must be a non-negative integral value.

The above command overrides the default value `100` with `<value>`.

The `goal` set using the above command 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 following:

- coverpoints `cp1` and `cp2`
- 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 the `goal` is set as `80` using the following command in the coverage configuration file:

```
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`.

Note: 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.

set_covergroup -default_type_option_goal

By default, the value of the covergroup type option `goal` is set to `100`. To override the default value of this option, use the following command in the coverage configuration file at elaboration:

```
set_covergroup -default_type_option_goal <value>
```

where, `value` must be a non-negative integral value.

The above command overrides the default value `100` with `<value>`.

The `goal` set using the above command 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 following command in the coverage configuration file:

```
set_covergroup -default_type_option_goal 85
```

then `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.

set_covergroup -per_instance_default_one

By default, coverage of covergroup instances is not saved to the coverage database and therefore is not shown in coverage results. You can enable saving of coverage of covergroup instances by setting the `per_instance` option to `1` in the covergroup declaration.

The `per_instance` option can be set in the covergroup declaration only and therefore, 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, use the following command in the coverage configuration file at elaboration:

```
set_covergroup -per_instance_default_one
```

With the above command, coverage of all instances of all of the covergroups in the design is saved and reported.

Note: To enable saving of coverage of covergroup instances of specific covergroups in the design, set the `per_instance` option in the covergroup declaration.

For more details on the `per_instance` option, see Example: Using per_instance Option.

set_covergroup -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 or covergroup. You can generate a warning for a coverpoint or covergroup by setting the `detect_overlap` option to `1`.

The `detect_overlap` option applies only to the coverpoint or covergroup in which it is set. If you want to enable the `detect_overlap` option for the entire design, use the following command in the coverage configuration file at elaboration:

```
set_covergroup -detect_overlap
```

With the above 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. As a result, there might be performance overhead if the design contains a large number of bins that are present in a single coverpoint.

set_covergroup -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 IMC User Guide.

Note: The impact of this command will not be visible in ICCR.

For a successful bin-level merge, both primary and secondary runs must be dumped with the `set_covergroup -bin_merge` command at elaboration.

Note: Bin-level merging is supported only for SystemVerilog covergroups and only with standard mode of merge. It is not supported for real coverpoint.

set_covergroup -optimize_model

By default, coverpoints/cross with more than 1 million valid bins are treated as ungradeable. You can use the following command in the coverage configuration file at elaboration to increase this limit to 50 million:

```
set covergroup -optimize model
```

Using this command, coverpoints/crosses with more than 50 million valid bins are treated as ungradeable. This leads to the optimized coverage model with reduced size and significant performance gains. The performance gains are expected in the following areas:

- Simulation run time memory

- Simulation dumping time of coverage model file
- Merging
- Significant improvements for union mode
- Incremental improvements for default mode
- Reporting memory

Note: 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, if the primary database is generated without this option, the secondary database is generated with this option do not get merged.
- In union merge mode, coverage databases that are generated with and without this option cannot be merged.

set_libcell_scoring

By default, coverage for Verilog library cells and VHDL VITAL cells is not scored. To enable scoring of Verilog library cells (modules defined with the `` `celldefine `` compiler directive or modules compiled with `-v`/`-y` options) and VHDL vital cells, use the `set_libcell_scoring` command.

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.

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: worklib1.entity_object(arch_object) File name: /servers/scratch03/meenal/ignoreLibraryPath/testcases/vhdl/test1/design1.vhd Number of covered blocks: 0 of 1 Number of uncovered blocks: 1 of 1 Number of excluded blocks: 0 Count Block Line Kind Origin Source Code --- 0 1 13 code block 11 report "In design1 process";	Type name: worklib2.entity_object(arch_object) File name: /servers/scratch03/meenal/ignoreLibraryPath/testcases/vhdl/test1/design2.vhd Number of covered blocks: 1 of 1 Number of uncovered blocks: 0 of 1 Number of excluded blocks: 0 Count Block Line Kind Origin Source Code --- 1 1 13 code block 11 report "In design2 process";	worklib1.entity_object(arch_object) File name: /servers/scratch03/meenal/ignoreLibraryPath/testcases/vhdl/test1/design1.vhd Number of covered blocks: 1 of 1 Number of uncovered blocks: 0 of 1 Number of excluded blocks: 0 Count Block Line Kind Origin Source Code --- 1 1 13 code block 11 report "In design1 process"

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.

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/scratch03/meenal/ignoreLibraryPath/testcases/vhdl/test1/design1.vhd	File name: /servers/scratch03/meenal/ignoreLibraryPath/testcases/vhdl/test1/design2.vhd	File name: /servers/scratch03/meenal/ignoreLibraryPath/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";

Important

The `set_ignore_library_name` command has been made redundant. This command was earlier used to merge and load data generated with differences in library name across runs.

Note: Due to the change in the default behavior of merge, coverage databases that were generated in the default mode before 12.2 cannot be merged with the databases generated in the default mode with 12.2.

set_parameterized_module_coverage

By default, for parameterized modules, module-based coverage is the representative of the first instance of the module. If coverage objects for other instances are different from the first instance, then the information for differing coverage objects is neither dumped nor reported for the module.

With the `set_parameterized_module_coverage` command different modules are dumped and reported corresponding to different parameter values combinations used to instantiate the module.

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

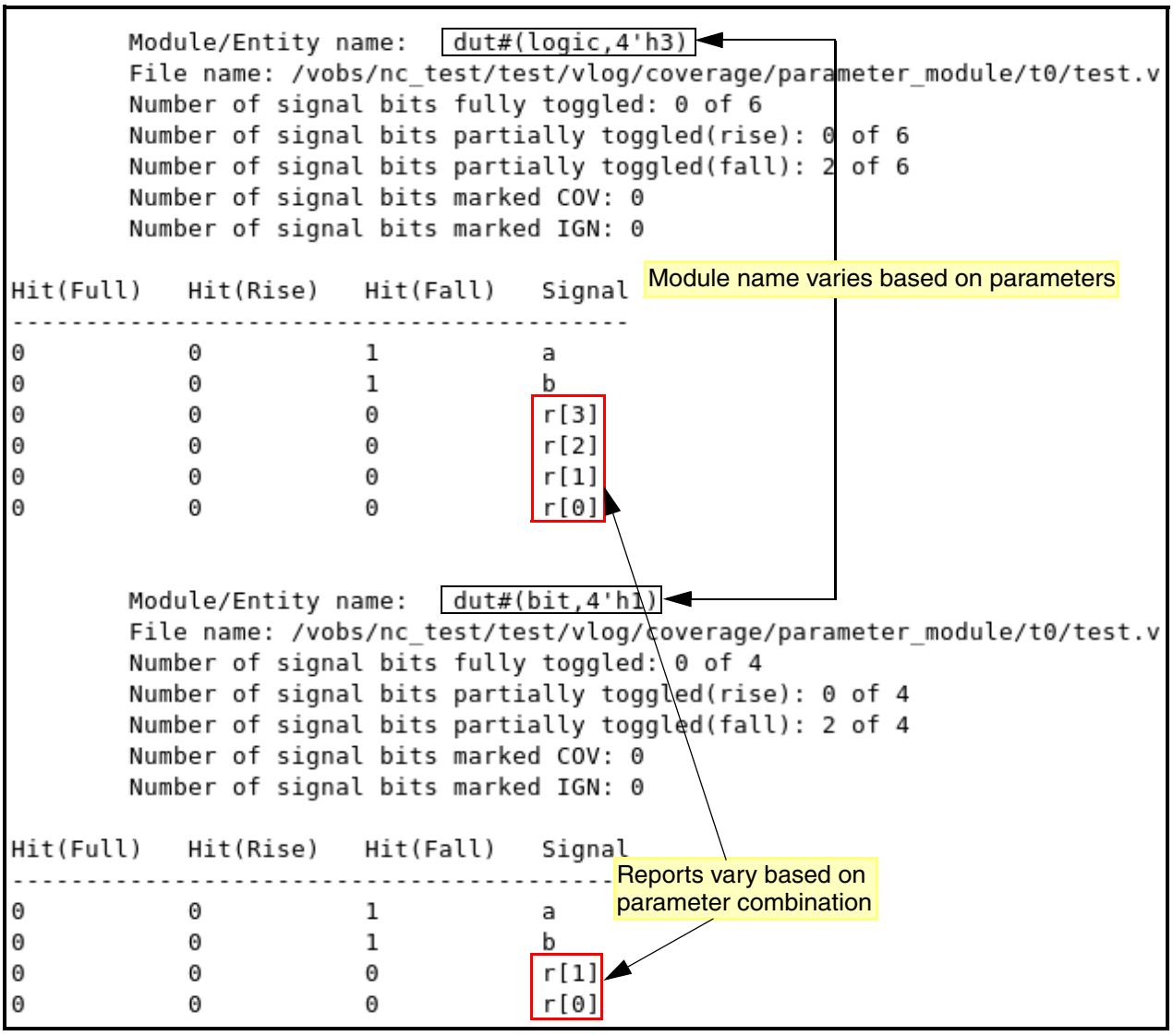
initial
begin
#10 a=1; b=1;
#10 a=0; b=0;
end

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.



Notes

- This feature is applicable only to Verilog modules.
- A database dumped with parameterization support cannot be merged with a database dumped without parameterization support.
- Toggle coverage is not supported for type-parameterized data type.

Using CCF commands across simulations

The following CCF commands impact design checksum of the coverage database:

- select_coverage
- deselect_coverage
- set_implicit_block_scoring
- set_assign_scoring
- set_branch_scoring
- set_statement_scoring
- set_expr_scoring
- set_code_fine_grained_merging
- select_fsm
- deselect_fsm
- set_fsm_arc_scoring
- set_toggle_noports
- set_toggle_portsonly
- set_toggle_scoring -sv_enum
- set_optimize
- select_functional
- set_libcell_scoring
- set_fsm_scoring
- set_covergroup
- set_merge_with_libname
- set_parameterized_module_coverage

These commands should be used in a consistent manner across simulations to allow merging of results, loading of tests, and re-using of marks file.

Selecting Coverage Using -coverage and -covdut and -covfile options

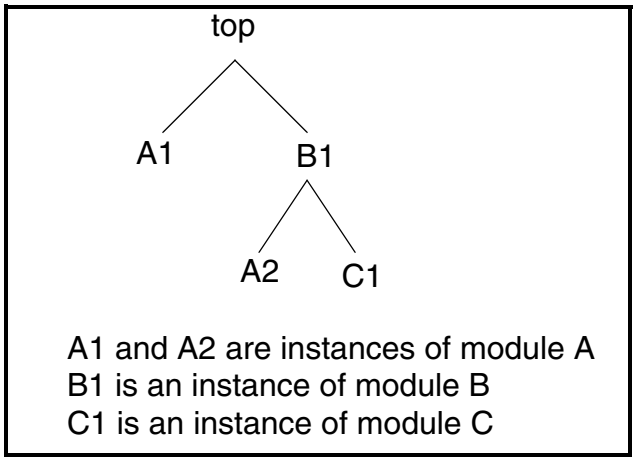
If you select coverage using both `-coverage` and `-covdut` and `-covfile` options, then take special care to understand how coverages are added as the command-line switches are processed and then selection is modified using commands in the CCF. Typically, with a coverage configuration file, you start 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:

```
% ncelab -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.

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.

Important

Coverage can be enabled at the time of creating primary and incremental snapshots. However, specifying a coverage configuration file is allowed only at the time of creating the primary snapshot. You cannot specify a coverage configuration file at the time of creating the incremental snapshot.

Coverage Configuration 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:

- select_coverage
- deselect_coverage
- 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_toggle_strobe
- set_toggle_limit
- set_toggle_includex
- set_toggle_includez
- set_toggle_noports

- set_toggle_portsonly
- set_toggle_scoring -sv_enum
- set_toggle_excludefile
- select_functional (Except the `-ams_control` and `-imm_asrt_class_package` option)
- set_covergroup
- set_optimize (Except the `-ial_ovl_inst_asrt` option)
- set_libcell_scoring
- set_merge_with_libname

Coverage Configuration Commands Unsupported in the MSIE Flow

The following coverage configuration commands are unsupported in the MSIE flow and an error is generated if these commads are included in the CCF during elaboration:

- 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_parameterized_module_coverage

When enabling coverage in the MSIE flow, remember that:

- Coverage configuration file cannot be specified at the time of creating the incremental snapshot. 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.

- 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.

  Note: Other types of coverage (block, expression, toggle, FSM) can be enabled when elaborating either the primary or 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:

```
ncsim <design_snapshot> [-covoverwrite] [-covworkdir <workdir>]
                        [-covscope][-covtest <test>|-covbaserun <test>]
[-covnomodeldump][-covmodeldir <modeldir>][-covfirstbinmatch][-cov_debuglog]
[-cov_gen_merge]
```

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/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 on page 251.
- `-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 `ncsim` 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, `ncsim` creates it, and if `ncsim` is unable to create it, an error is reported and coverage model is not saved. Further, if the directory exists but `ncsim` is unable to reuse or save coverage model, an error is reported.

  Note: If `ncsim` 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
                                                                              //  was the first significant bin
                0(1)      b2    17   bins b2 = binsof (A)  intersect {1} ; // 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 enables printing the coverage-related information for different phases of simulation. By default, logs are not printed for this information.

- -cov_gen_merge generates sop unification of an instance, in which only the sops that are under elaborated generates for that instance along with the remaining sops of the instance are considered. 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.

Interactive mode

You use the ncsim `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>]
                [-test <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>] [-test <test>] [-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.
- `-test <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 on page 251.
- `-dut <dut_instance>` defines the hierarchy reference for reporting. This option tells NC-Sim 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:

  ```
  ncsim> coverage -setup -dut tb_sys.sys.mic_0
  ```

  Simulation 2:

  ```
  ncsim> 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 ncelab 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:

  ```
  ncsim> 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:

  ```
  ncsim> coverage -functional -select top.* -depth all
  ```

  To include all of the scopes in the module `top`, use:

  ```
  ncsim> 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:
`ncsim> coverage -functional -deselect <assertion_path>`

For example, to deselect all assertions inside a package "`pkg`" use:
`ncsim> 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: ICC scores coverage 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 *Incisive 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 *Incisive 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 *Incisive Metrics Center User Guide*.

Note: Prior to this release, `coverage -vplan_analyze` command was available to launch the Enterprise Manager vPlan window to analyze coverage data. With this release, the `coverage -vplan_analyze` command is not supported. However, for backward compatibility (that is to launch Enterprise Manager as the coverage analysis tool), you can set the `CDN_COVERAGE_VPLAN` environment variable, and then use the `coverage -analyze` command. For more details on coverage analysis with Enterprise Manager, see the *Enterprise Manager Analyzing Metrics* guide available in the Enterprise Manager (EMGR) installation.

If the `CDN_COVERAGE_VPLAN` variable is set, then:

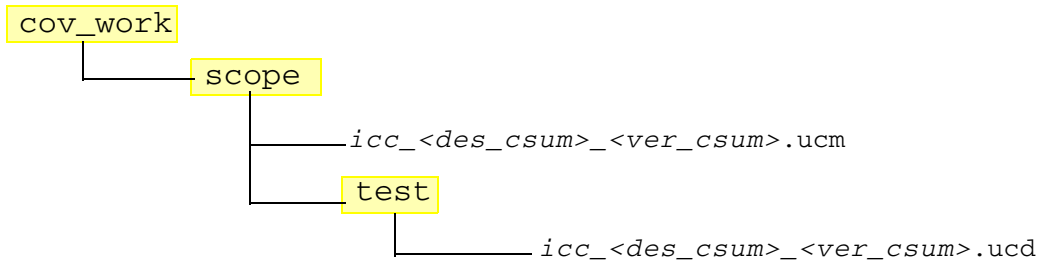
- The `coverage -analyze` command launches the vPlan window of Enterprise Manager.
- If the `coverage -analyze` command is executed multiple times in an `ncsim` session, then the vPlan window is launched multiple times.
- If the simulation advances, then the new results are visible only in the vPlan window launched after simulation advancement. The previously launched windows will not show new results and there is no mechanism to refresh the results in previously launched windows.

If the `CDN_COVERAGE_VPLAN` variable is not set, then:

- The `coverage -analyze` command by default, launches IMC as the coverage analysis tool.
- If the `coverage -analyze` command is executed multiple times in an `ncsim` session, then IMC is launched multiple times.
- If the simulation advances, then the new results are visible in the IMC window launched after simulation advancement. The previously launched windows will not show new results. To refresh the results in previously launched windows, select *Reload* from the *File* menu.

Coverage Data Storage

By default, coverage data in ICC 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 ncsim `-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 irun command is used:

  ```
  irun -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 irun command does not contain the `-ovmtest` option:

  ```
  irun -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
	`select_coverage -block -instance dut`
	Run 2 - CCF
	`select_coverage -block -instance *...`

Different DUT name is specified to the ncelab command-line argument `-covdut`.	Run 1
	`ncelab -covdut B -covfile cov.ccf worklib.top:v`
	Run 2
	`ncelab -covdut A -covfile cov.ccf worklib.top:v`

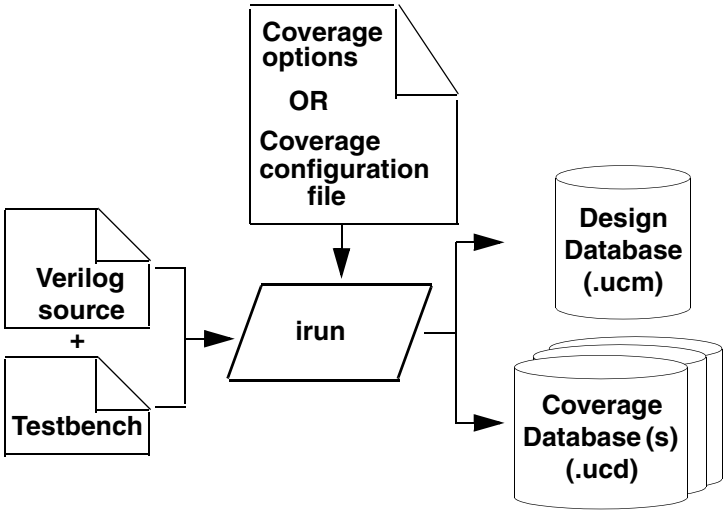
Different coverage metrics are instrumented in different runs at elaboration.	Run 1
	`ncelab -coverage b worklib.top:v`
	Run 2
	`ncelab -coverage b:e worklib.top:v`

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.

8.2 Generating Coverage Data using Single-Step Simulation

The following diagram illustrates the single-step process of generating the coverage data using IRUN:



To generate coverage data using single-step simulation, you use *irun*. The syntax for coverage specific options in `irun` is:

```
irun  [-coverage <coverage_types> |
       -covfile <coverage_configuration_file> ]
       -covdut <DUT_module>[<DUT_module>]
       -covworkdir <workdir>
       -covscope <scope>
       -covtest <test>
       -covmodeldir <modeldir>
       -covoverwrite
       -tcl <tclfile>
       <other_irun_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 on page 251.
- `-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 `ncsim` 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, `ncsim` creates it, and if `ncsim` is unable to create it, an error is reported and coverage model is not saved. Further, if the directory exists but `ncsim` is unable to reuse or save coverage model, an error is reported.

  Note: If `ncsim` 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:

```
% irun design_top.vhd testbench.v -coverage b
```

8.3 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:

```
ncsc_run -ncelab_args, [+nccovfile+<coverage_configuration_file>
|+nccoverage+<coverage_types> ] <other_ncsc_run_options>
```

In the above syntax,

- `-ncelab_args` specifies coverage options to `ncelab`.
- `+nccovfile` option passes a configuration file to `ncelab` for instrumentation.
- `+nccoverage` option enables coverage data generation for different coverage types.
- `<other_ncsc_run_options>` specifies options related to simulating the design. See Simulating SystemC Models in *NC-SC Simulator User Guide* for more details.

To generate coverage data for block and expression coverage from a mixed HDL-SystemC design, use:

```
ncsc_run -noscv -dynamic -INPUT run_intg.tcl -ACCESS +RWC -ncelab_args,
+nccoverage+b:e model.cpp sctop.cpp DUT.v model.v vchild.v
```

8.4 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 on page 67.

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 -sv enum` 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.
- Expression coverage for enums for most of the use cases is supported.

9

Launching the Reporting Tool (ICCR)

ICCR is a coverage reporting tool to merge coverage data, display textual and graphical reports for coverage data, mark coverage items, and analyze coverage data.

Note: Another alternate tool, Incisive Metrics Center (IMC), is available for coverage data analysis. For more details on IMC, see the *Incisive Metrics Center User Guide*.

You can launch ICCR in the following modes:

- GUI mode

  To launch ICCR in GUI mode, type `iccr -gui` at the command prompt and press `Enter`. This launches the ICCR Analyzer, which is a graphical user interface to observe coverage results, compare simulations, and mark coverage items. See *ICC Analysis User Guide* for more details.

- Interactive mode

  To launch ICCR in interactive mode, type `iccr` at the command prompt and press `Enter`. This brings up the `iccr` command prompt and you can issue ICCR commands. In this mode, control returns to you after execution of each command.

- Batch mode

  To launch ICCR in the batch mode, type `iccr <command_file>` at the command prompt and press `Enter`. In this mode, all commands are placed in a `<command_file>` which is passed to iccr for execution. If a command in this file fails, ICCR terminates without executing the remaining commands.

9.0.1 Command Syntax

The syntax for the `iccr` command is:

```
% iccr [options] [<command_file>]
```

where

- `[<command_file>]` launches ICCR in batch mode.

- `[options]` can be:

Options	Description
`[-help]`	Displays a list of the iccr command options with a brief description about each option.
`[-version]`	Prints the ICCR version number.
`[-64bit]`	Launches the 64-bit version of the ICCR executable.
`[-logfile <filename>]`	Uses *<filename>* as the log file instead of the default `iccr.log`.
`[-test <test>]`	Loads coverage from *[[<workdir>/]<scope>]<test>* while launching ICCR.
`[-GUI]`	Launches ICCR in GUI mode.
`[-STATUS]`	Prints coverage model statistics, i.e., number of coverage items, such as blocks and expressions, only when the model is successfully dumped. **Note:** This information is not displayed if the coverage model is not dumped.
`[-LICQUEUE]`	Queues the request for a license, if not currently available, and runs the tool when a license becomes available.
`[-KEYFILE <filename>]`	Uses *<filename>* as the key file instead of the default `iccr.key`. Note: Key files are useful when you want to reproduce an ICCR session. A key file contains all the commands that you run in an ICCR session, including misspelled commands and the exit command. You might edit the key file before you pass it to ICCR as a command file.
`[-keywords+<key1>+<key2>]`	Turns on keywords *<key1>*, *<key2>* defined in the *<command_file>*. This option works only if *<command_file>* is specified.

To load coverage test `test1` while launching ICCR and also to start the GUI, use:

```
iccr -test test1 -GUI
```

To load coverage data with all tests starting with `test_` stored in `/mywork/mydesign/` while launching ICCR, use:

```
iccr -test 'mywork/mydesign/test_*'
```

Note: When specifying multiple tests using wildcards, enclose the test specification within quotes.

Consider the following `iccr.cmd` file which includes keywords `summary` and `detail`.

```
load_test test1
<summary>
report_summary -module -be * > summary.rpt
</summary>
<detail>
report_detail -module -both -be * > detail.rpt
</detail>
```

Note: Use of > redirects the output of the command to the file name specified after the > sign. Use of >> appends the output to the command to the file name specified after the >> sign.

To launch ICCR in batch mode and execute the code within the `summary` keyword, use:

```
iccr -KEYWORDS+summary iccr.cmd
```

10

Merging Coverage Data

After simulation, the coverage database files are available for report generation and analysis. Coverage analysis generally is performed on aggregated data from multiple tests run on one or more design configurations. Coverage data must be merged to provide a comprehensive view of what is covered. This chapter discusses how to merge coverage data using ICCR.

Note: Another tool, *Incisive Metrics Center* (IMC), is available for merging metrics data, displaying textual and graphical reports, and analyzing metrics data. Soon IMC will replace ICCR. For more details on IMC, see the *Incisive Metrics Center User Guide*.

The merge operation merges data from multiple coverage databases from potentially different design hierarchies to a new database. As merge is a repetitive task, generally it is performed in scripts that run at the end of a regression. Merging coverage data involves:

- Setting the DUT Modules
- Specifying the Merge Behavior
- Using the merge Command

Note: The merge operation does not require loading of coverage tests.

10.1 Setting the DUT Modules

Before merging the coverage tests, you can specify a set of DUT modules for which to merge coverage. If DUT modules are not set before the merge operation, then the top DUT(s) present in the primary test is considered as the DUT to be merged.

To specify a set of DUT modules for which to merge coverage, use:

```
set_dut_modules <dut_modules>
```

where `<dut_modules>` is the set of modules to be merged. The modules specified as `<dut_modules>` automatically merge the sub-hierarchy. As a result, there is no need to mention modules present in the sub-hierarchy to the `set_dut_modules` command.

The <dut_modules> can contain wildcards. For example, to set the dut modules as tdsp_core_glue and tdsp_core_mach, use:

```
set_dut_modules tdsp_core_*
```

instead of

```
set_dut_modules tdsp_core_glue tdsp_core_mach
```

The set_dut_modules command is not additive, and the modules listed in the last command are considered.

10.2 Specifying the Merge Behavior

After specifying the DUT modules for merge, you can:

- Set the Mode of the Merge Operation
  - Standard Merge
    - Fine-grained Merging
  - Union Merge
- Enable Signal-Level Merging

10.2.1 Set the Mode of the Merge Operation

The mode of merge can be *standard* or *union*. The behavior of the merge operation depends on the mode set for the merge. The default mode is *standard*. To set the mode of merge as *union*, use:

```
set_merge -union
```

To switch back to the *standard* mode, use:

```
set_merge -nounion
```

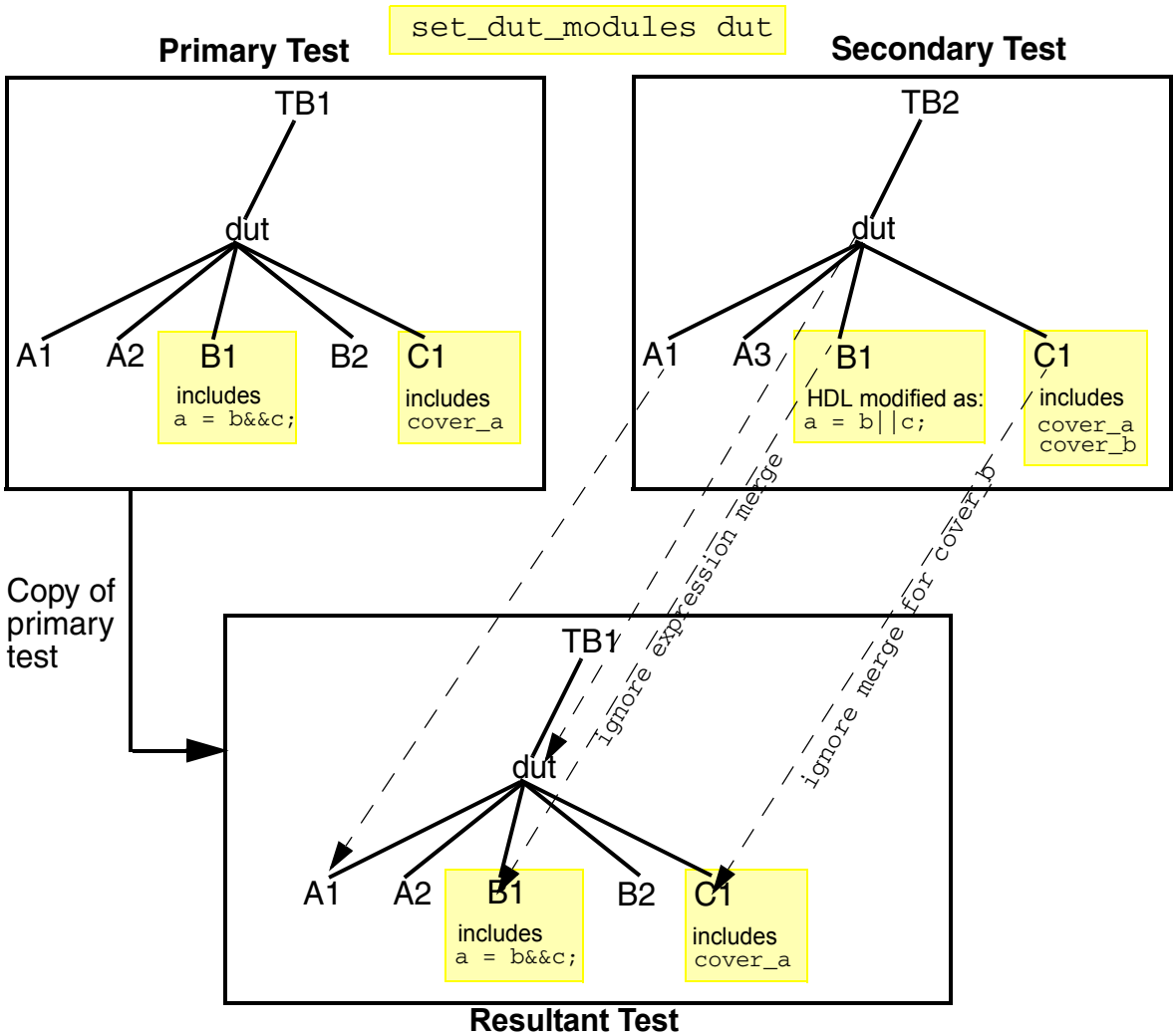
10.2.1.1 Standard Merge

In a standard merge, a copy of the primary test is created, into which one or more secondary tests are merged. The merge operation merges coverage for items in the secondary test that exist with the same definition as in the primary test. In addition, only sampled (covered) coverage points from the secondary test are considered for merging with the primary test. And if, and only if, a coverage point is considered for merge, it is either merged or a merge conflict is reported for it. Note that in a standard merge, model (.ucm) of the primary test is used as a reference for merging.

Code coverage types (block, expression, and toggle) and FSM coverage are merged independent of each other. If the instance checksum for a coverage type matches between the primary and secondary tests then coverage of that type is merged.

Functional coverage merge resolution happens below the instance or module level. If the checksum for a functional coverage item matches between the primary and secondary tests then coverage for that item is merged.

In a standard merge, all instances of specified DUT modules present in secondary tests are merged into the first instance of that DUT module in the primary test, as shown here.



For the above scenario, the `set_dut_modules` is specified as `dut`. The merge operation creates a copy of the primary test, which is the resultant test, and then merges:

- TB2.dut --> TB1.dut
- TB2.dut.A1 --> TB1.dut.A1
- All coverages except for expression coverage for TB2.dut.B1 to TB1.dut.B1

  Note: Expression coverage merge for instance B1 is ignored because HDL description differs (expression is modified in secondary test).

- TB2.dut.C1 --> TB1.dut.C1 (Ignoring merge of cover_b)

Both in standard merge and union merge, two tests can be merged if they have modules with the same name but may have been compiled in different libraries. Consider the given example, in which the primary test with the given design hierarchy, in which l1 is an instance of the module lib1.A and l2 is an instance of lib2.A:

```
lib5.dut
   |__I1 : lib1.A
   |__I2 : lib2.A
```

is merged with the secondary test with the following design hierarchy, in which l1, l2, and l3 are instances of lib3.A, lib2.B, and lib1.A, respectively:

```
lib6.dut
   |__I1 : lib3.A
   |__I2 : lib2.B
   |__I3 : lib1.A
```

In the given example, while instance-level merging shall take place based on the hierarchical name of the instance, in case of module-level merging:

- The module lib6.dut from secondary test is merged to the module lib5.dut in the primary test because there is one module with the same name even though the library names are different in the primary test.
- The module lib3.A from the secondary test is not merged to any of the modules in the primary test, because there is more than one module with the same name, lib1.A and lib2.A, in the primary test. In this case, a warning is reported.
- The module lib2.B from the secondary test is not merged because there is no module with the same name in the primary test. In this case, a warning is reported.
- The module lib1.A from the secondary test is merged to lib1.A in the primary test because there is a module with the same name, lib1.A in the primary test.

Fine-grained Merging

Fine-grained merging provides merge resilience at concurrent block level instead of default module/instance level. By default, if there is any change in the HDL code that affects the block or expression, no blocks or expressions are merged for that instance.

Consider an example where some modifications are made to the HDL across runs.

Run 1

```
18  initial begin
19          #200 $finish();
20  end
21
22  initial begin
23          count = 0;
24          input1 = 1'b0;
25          input2 = 1'b0;
26          input3 = 1'b0;
27  end
28
29  always@(input1 or input2 or input3 or clock)
30  begin
31        expr;
32  end
33
34  task expr;
35  begin
36
37        out[0] = ~input1;
38        out[1] = &input2;
39        out[2] = |input3;
40        out[3] = ^(input1 + input2);
41        out[4] = ~&(input2 + input3);
42
43  end
44  endtask
```

Run 2

```
18  initial begin
19          #200 $finish();
20  end
21
22  initial begin
23          count = 0;
24          input1 = 1'b0;
25  #10     input2 = 1'b1;
26          input3 = 1'b0;
27  end
28
29  always@(input1 or input2 or input3)
30  begin
31        expr;
32  end
33
34  task expr;
35  begin
36
37        out[0] = ~input1;
38        out[1] = &input2;
39        out[2] = |input3;
40        out[3] = ^(input1 + input2);
41        out[4] = ~&(input2 + input3);
42
43  end
44  endtask
```

HDL modified on Line 25 and Line 29

The following figure displays coverage results from Run 1, Run 2, and the merged output.

Coverage from Run 1 (Primary)

```
cnt  block line number            line origin description
-------------------------------------------------------------
  1      18                          18   initial begin
  0      19                          19   #200 $finish();
  1      22                          22   initial begin
 18      30                          30   begin
 18      35                          35   begin

  0      48                          48   f1 = in3?(in1===
  7      51                          51   assign out[5] =
```

Coverage from Run 2 (Secondary)

```
 cnt  block line number            line origin description
-------------------------------------------------------------
   1      18                          18   initial begin
   0      19                          19   #200 $finish();
   1      22                          22   initial begin
   1      25                          25   #10        input2
  11      30                          30   begin
  11      35                          35   begin
   0      48                          48   f1 = in3?(in1===i
   9      51                          51   assign out[5] = ~
```

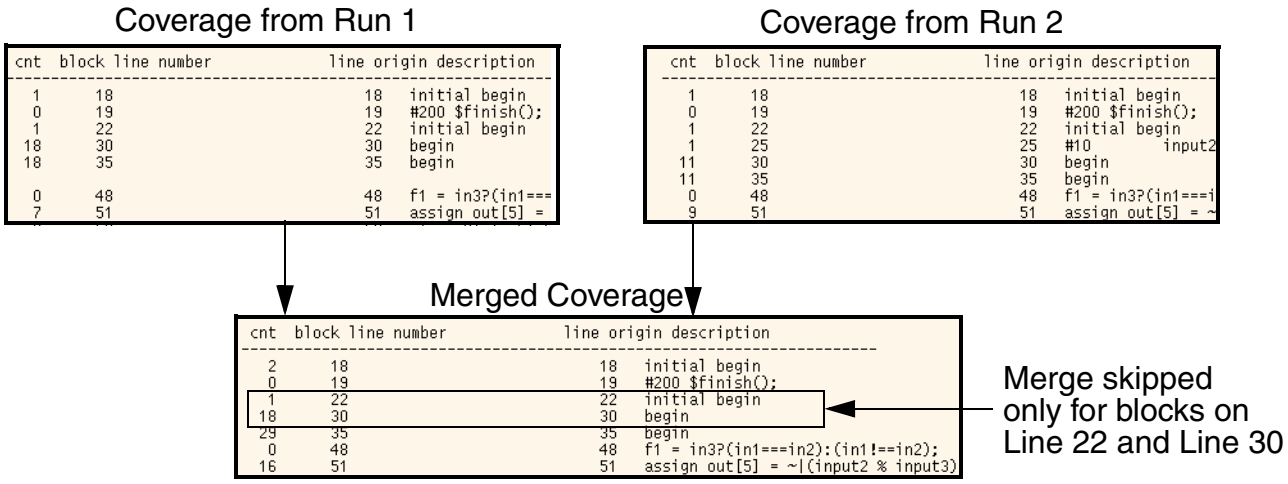
Merged Coverage (same as Run 1 -- merge ignored as HDL differs)

```
 cnt  block line number            line origin description
-------------------------------------------------------------
   1      18                          18   initial begin
   0      19                          19   #200 $finish();
   1      22                          22   initial begin
  18      30                          30   begin
  18      35                          35   begin
   0      48                          48   f1 = in3?(in1===in2):(in1!==in2);
   7      51                          51   assign out[5] = ~|(input2 % input3
```

In this example, merge is ignored for the instance because the HDL code differs across runs. With fine-grained merging, merge resilience is added at concurrent block level by which

merge is skipped only for the modified blocks within an instance. To enable fine-grained merging, use the set_code_fine_grained_merging command in the coverage configuration file during elaboration.

If coverage data for both the runs (primary and secondary) is generated with the `set_code_fine_grained_merging` command, then coverage is merged as:



With fine-grained merging, merge is skipped only for blocks on Line 22 and Line 30. Coverage for all the remaining blocks is merged. This is because with the `set_code_fine_grained_merging` command, merge resilience is enabled at the concurrent block level. The following table lists the supported concurrent blocks in Verilog and VHDL, which are scored in ICC.

Concurrent Blocks in Verilog	Concurrent Blocks in VHDL
initial block	process block
always block	conditional signal assignment
continuous block	selected signal assignment
task/function	

Important

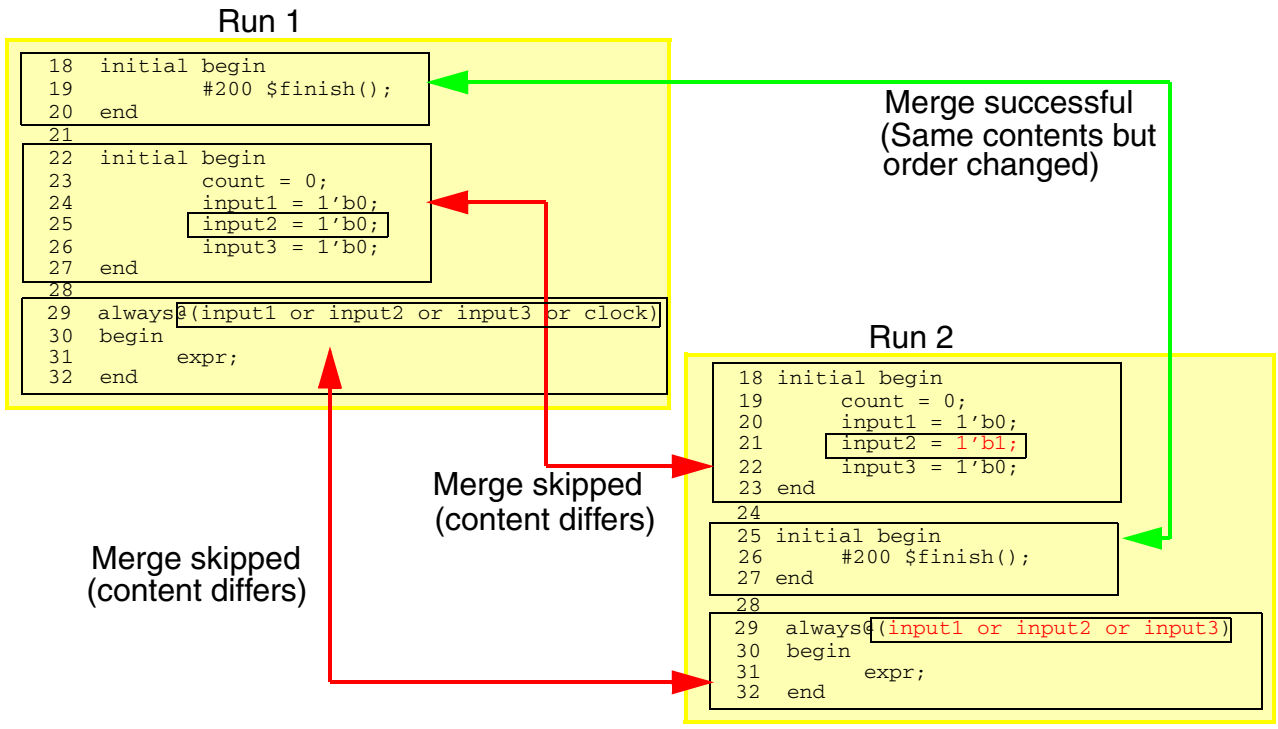
The `set_code_fine_grained_merging` command traverses each concurrent block to compute the checksum. As a result, there might be a performance overhead if the design includes a large number of concurrent blocks.

Note: For a successful fine-grained merging, it is important that both primary and secondary tests are dumped with the `set_code_fine_grained_merging` command during elaboration.

The following table lists the merge behavior in different scenarios with fine-grained merging:

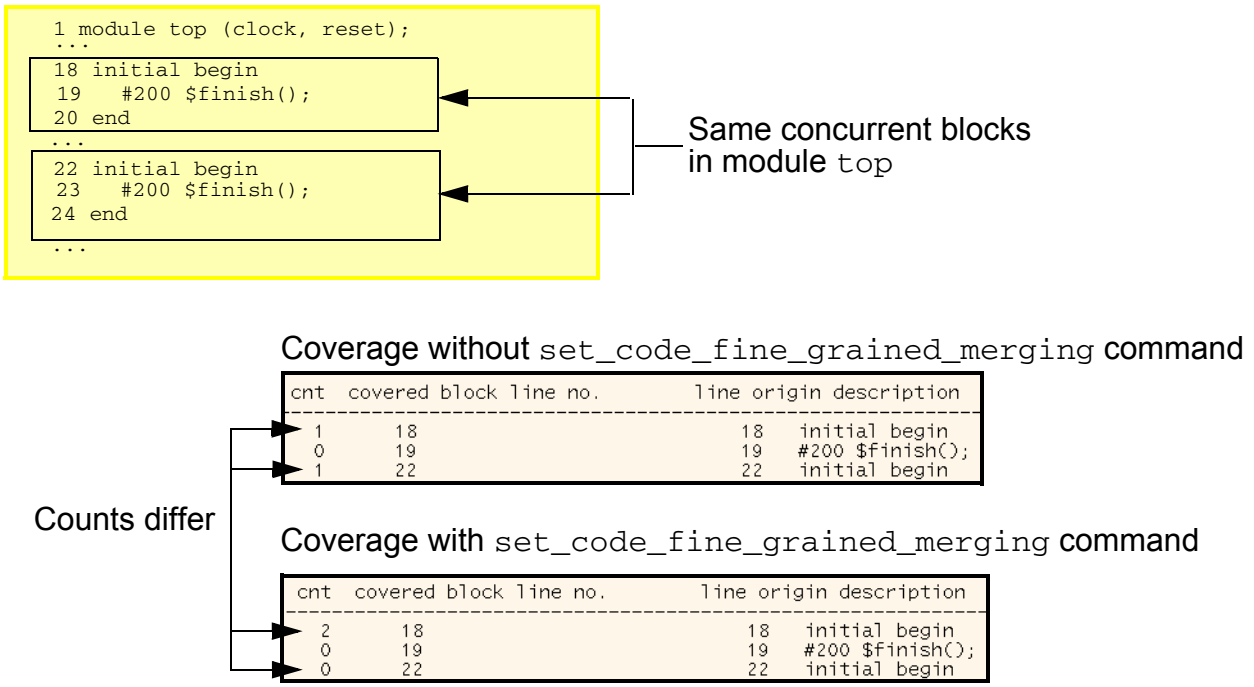
Scenario	Merge Behavior
Contents of two concurrent blocks same across runs	Merge successful for the block
Contents of two concurrent blocks same across runs (order of blocks changed)	Merge successful for the block
Contents of two concurrent blocks differ across runs	Merge skipped for the block

The following figure displays the merge behavior in these scenarios:



If the contents of two concurrent blocks are exactly same in a particular scope, then you might see differences in coverage counts when coverage data is generated with and without the `set_code_fine_grained_merging` command at elaboration.

The following figure demonstrates this.



In the above example, the same concurrent block is repeated in module `top`. In the above reports, notice the difference in counts for blocks on Line 18 and Line 22 when coverage is generated with and without the `set_code_fine_grained_merging` command at elaboration.

Note: It is recommended to use Union merge with fine-grained merging for Verilog `` `ifdef `` configured HDL. When two databases with different configurations are merged, the source/line information is correctly captured in the merged database. However, if the sources of these databases to be merged are different, then the resultant merged database may have confusing source information. Consider the given example:

```
`ifdef PCIEXG3TLDLP_BUSWIDTH_64
  always @(posedge PCLK) P1_Size_P0          <=  1'd0;
`endif

`ifdef PCIEXG3TLDLP_BUSWIDTH_128
  always @(posedge PCLK) P1_Size_P0          <=  2'd0;
`endif

`ifdef PCIEXG3TLDLP_BUSWIDTH_256
  always @(posedge PCLK) P1_Size_P0          <=  3'd0;
`endif
```

In the given example, union-merged database will contain coverage objects from all of the tests. The merged output test will contain blocks and expressions from all the three always blocks:

```
P1_Size_P0          <=  1'd0;
P1_Size_P0          <=  2'd0;
P1_Size_P0          <=  3'd0;
```

Note: Consider an example in which two databases, DB1 and DB2, from two different files, F1 and F2, are merged. The initial blocks for F1 is not labeled:

```
initial
#20 $finish;
```

And, the initial block for F2 is labeled:

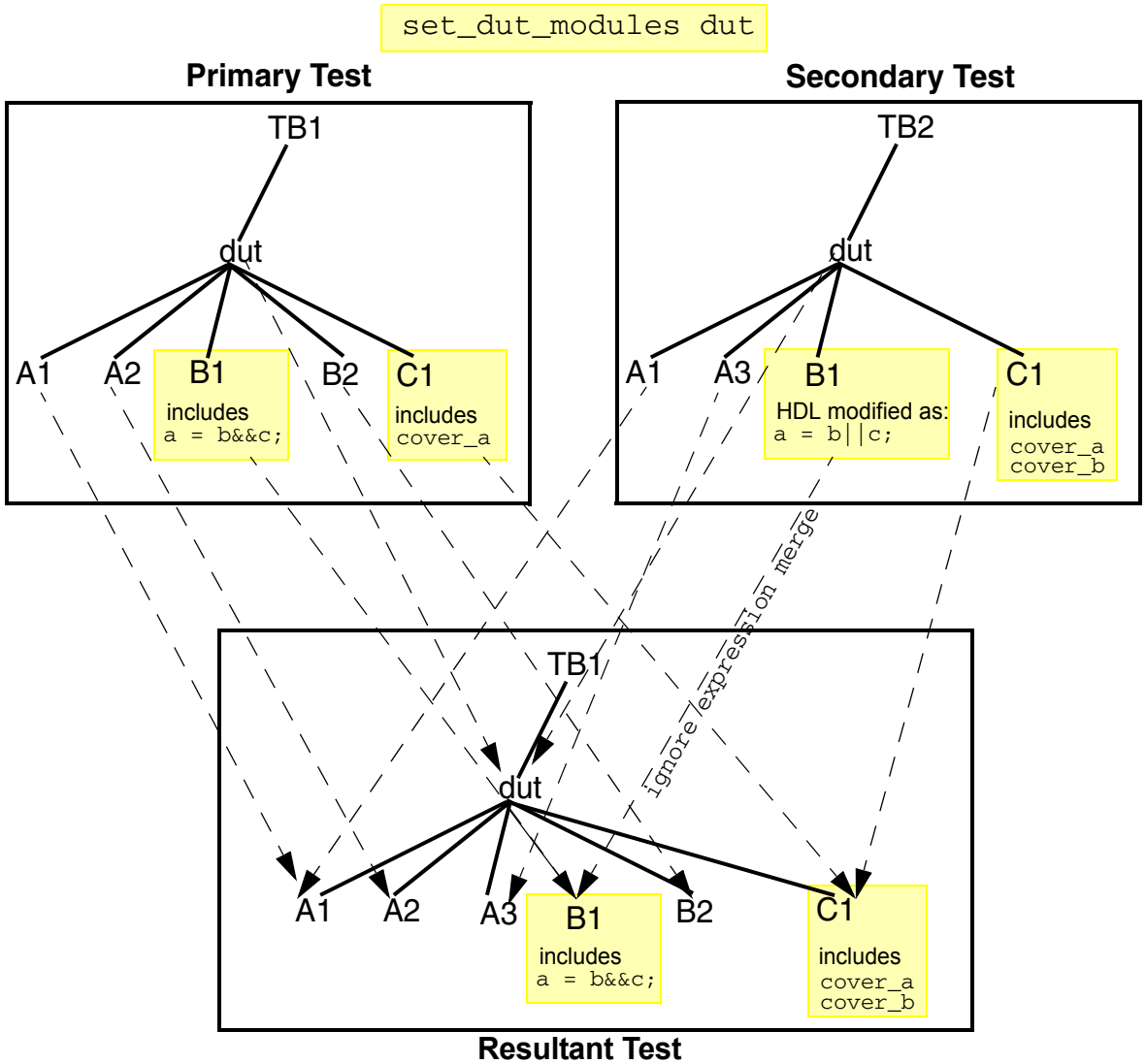
```
initial : STOP
#20 $finish;
```

In this case, the merge will be unsuccessful even when the coverage is generated with the `set_code_fine_grained_merging` command at elaboration.

10.2.1.2 Union Merge

In a union merge, a copy of primary model (`.ucm`) is created in the merged output directory to create the resultant model. Then, models of all secondary tests are unioned to the resultant model. The resultant model includes coverage types, non-conflicting instances, assertions, coverpoints, crosses, and covergroups that are not available in the primary model.

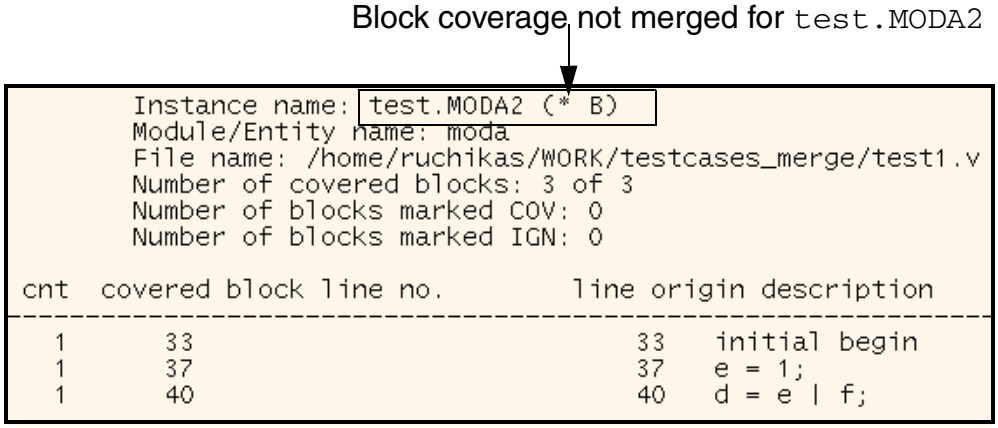
Finally, coverage from all tests for items that exist and match with the resultant model gets merged. In addition, all instances of specified DUT modules present in secondary tests are merged into the first instance of that DUT module in the primary test, as shown here.



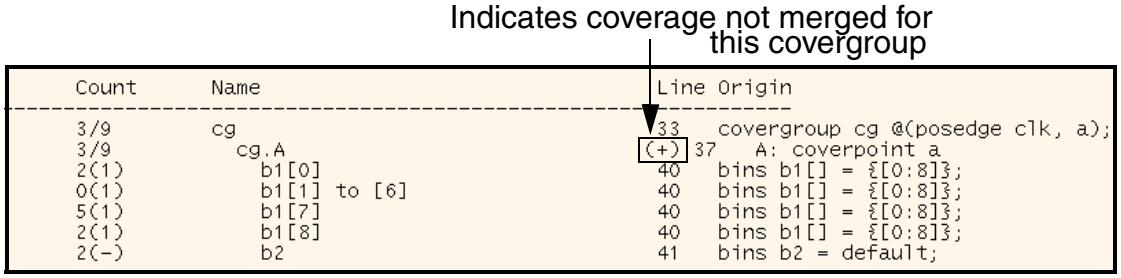
Data projection/merge to resultant model is done as:

- TB1.dut and TB2.dut to TB1.dut of resultant model
- TB1.dut.A1 and TB2.dut.A1 to TB1.dut.A1 of resultant model
- TB1.dut.A2 to TB1.dut.A2 of resultant model
- TB2.dut.A3 to TB1.dut.A3 of resultant model
- TB1.dut.B2 to TB1.dut.B2 of resultant model
- TB1.dut.C1 and TB2.dut.C1 to TB1.dut.C1 of resultant model
- All coverages except for expression coverage for TB1.dut.B1 and TB2.dut.B1 to TB1.dut.B1 of resultant model

  Note: Expression coverage merge for instance B1 is ignored because HDL description differs (expression is modified in secondary test). In case of block, expression, toggle, and FSM coverage, such instances/modules are marked with * followed by the coverage type in the coverage report, as shown below.



  In case of assertion or covergroup coverage, such assertions/covergroups are marked with the + sign, as shown below.



  Note: The * or the + sign indicates that there was a conflict during the merge operation and the specified coverage type was not merged for that instance, assertion, or

covergroup.

Some Recommendations on Selecting the Mode of Merge

The following recommendations might help you determine the mode of merge suitable for your environment:

- The union merge is beneficial when there is a single design configuration that has not changed between tests, but the functional coverage differs between the tests.
- Use of union merge might complicate subsequent analysis in following scenarios:
  - When you merge between different configurations of chips, systems, and blocks. This is because the union merge will build these in parallel as independent roots.
  - When you have an evolving design where the HDL code differs between older and newer tests. This is because the union merge will also merge instances that were removed.
- Use of standard merge is recommended in such scenarios.

10.2.2 Enable Signal-Level Merging

By default, the merge operation does not merge any signals in an instance if any of the following applies for that instance:

- Change in width for port and local signals
- Change in type for port and local signals
- Signals added or deleted

To enable merging of unchanged signals for the above scenarios, use the `-toggle_relax` option with the `set_merge` command, as shown below:

`set_merge -union -toggle_relax`	Enables signal level merging in union mode
`set_merge -nounion -toggle_relax`	Enables signal level merging in standard mode.

To disable signal-level merging, use:

```
set_merge -no_relax
```

The merge behavior when the `-toggle_relax` option is used in different scenarios is discussed below.

Scenario	Resultant Test - Standard Merge	Resultant Test - Union Merge
Change in bit width	- Bits from primary test - Counts from primary test	- Bits from all tests - Counts added for common signals
Signals added or deleted	- Signals from primary test - Counts added	- Signals from all tests - Counts added
Change in signal type	- Signals from primary test - Counts added	- Signals from all tests - Counts added

Note: It is recommended to use Union merge with `-toggle_relax` for Verilog `` `ifdef `` configured HDL. When two databases with different configurations are merged, the source/line information is correctly captured in the merged database. However, if the sources of these databases to be merged are different, then the resultant merged database may have confusing source information. Consider the given example:

```
`ifdef PCIEXG3TLDLP_BUSWIDTH_64
wire      P1_Size_P2;
reg       P1_Size_P0;
`endif

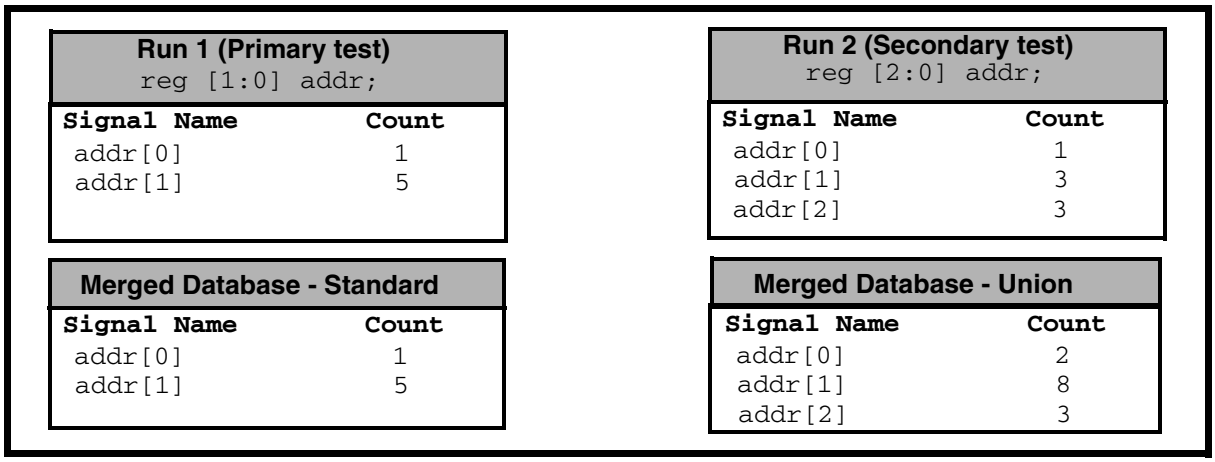
`ifdef PCIEXG3TLDLP_BUSWIDTH_128
wire  [  1:  0]      P1_Size_P2;
reg   [  1:  0]      P1_Size_P0;
`endif

`ifdef PCIEXG3TLDLP_BUSWIDTH_256
wire  [ 2:  0]      P1_Size_P2;
reg   [ 2:  0]      P1_Size_P0;
`endif
```

In the given example, the same signal with different widths is reported across different runs.

Example: Change in bit width

In the following example, the bit width for `addr` differs across `Run 1` and `Run 2`. Data merge for both standard and union mode is shown below.



Run 1 (Primary test)
`reg [1:0] addr;`

Signal Name	Count
addr[0]	1
addr[1]	5

Run 2 (Secondary test)
`reg [2:0] addr;`

Signal Name	Count
addr[0]	1
addr[1]	3
addr[2]	3

Merged Database - Standard

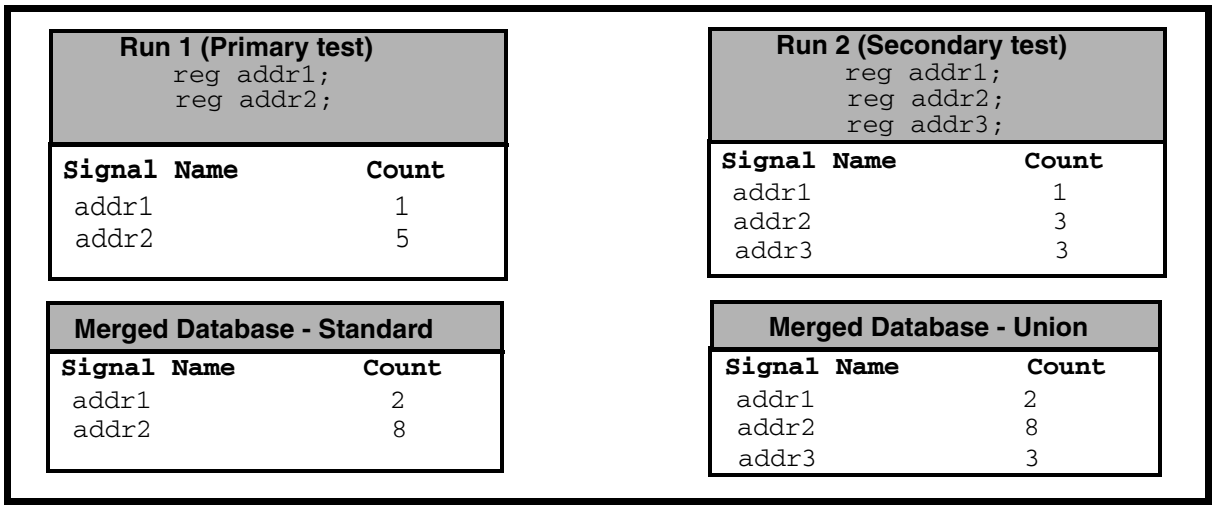
Signal Name	Count
addr[0]	1
addr[1]	5

Merged Database - Union

Signal Name	Count
addr[0]	2
addr[1]	8
addr[2]	3

Example: Number of signals differ

In the following example, `Run 2` includes an additional signal `addr3`. Data merge for both standard and union mode is shown below.



Run 1 (Primary test)
```
reg addr1;
reg addr2;
```

Signal Name	Count
addr1	1
addr2	5

Run 2 (Secondary test)
```
reg addr1;
reg addr2;
reg addr3;
```

Signal Name	Count
addr1	1
addr2	3
addr3	3

Merged Database - Standard

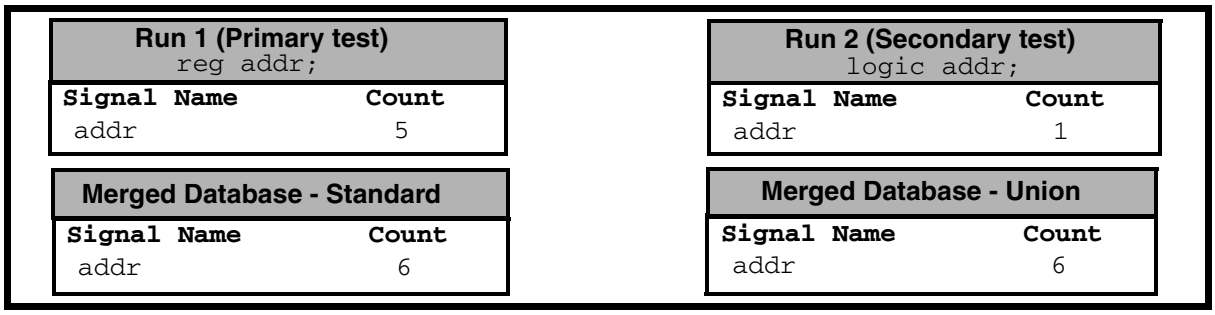
Signal Name	Count
addr1	2
addr2	8

Merged Database - Union

Signal Name	Count
addr1	2
addr2	8
addr3	3

Example: Change in signal type

In the following example, signal type for `addr` differs across `Run 1` and `Run 2`. Data merge for both standard and union mode is shown below.



10.3 Using the merge Command

The `merge` command merges coverage data of two or more tests and saves merged data as a new test. To merge coverage data, use:

```
merge [coverages] {<test_list> | -testfile <filename>}
      -output <output_test> [-message]
[coverages]  ::= [-code] [-toggle] [-fsm] [-functional]
<test_list> ::= <primary_test> {<one_or_more_secondary_tests>} | *
```

where

- `coverages` specifies the type of coverage to merge. If not specified, all coverage is merged.
- `<test_list>` specifies the tests for merge. The first test specified in the `<test_list>` is the primary test. The remaining tests that follow are the secondary tests. If `*` is used for `<test_list>`, ICC considers the first test in the resolved list as the primary test, and remaining tests as the secondary tests.

  The tests specified in the `<test_list>` can include just the test name or the complete path, as:

  ```
  [[<workdir>/]<scope>]<test>
  ```

  where

  - `<workdir>` specifies the root location where all coverage data is stored. It should contain valid filename characters and path separators but cannot contain wildcards. `<workdir>` can use both absolute paths (for example, `/home/me/chip/verification/todaysrev`) and relative paths (for example, `verification/todaysrev`).

- <scope> specifies a particular configuration of hardware (DUT). It should contain valid filename characters and cannot contain path separators or wildcards.
- <test> specifies the test name for a single simulation. It should contain valid filename characters and can contain wildcards.

- -testfile <filename> specifies the file that includes a list of tests to be merged. The tests in <filename> should be listed one per line. You can include comments in <filename> using #.
- -output <output_test> specifies where to store merged data. If the <output_test> argument includes just the test name, the merged output is stored in directory where the primary test is located. If the <output_test> includes the complete path, the merged data is stored at the specified location.

  Consider following tests:

  ```
  ./cov_work/my_scope1/test1/
  ./cov_work/my_scope2/test1/
  ```

  If <output_test> is specified as:

  ```
  merge cov_work/my_scope1/test1 cov_work/my_scope2/test1 -output all
  ```

  then the merged data is stored at ./cov_work/my_scope1/all.

  If <output_test> is specified as:

  ```
  merge cov_work/my_scope1/test1 cov_work/my_scope2/test1 -output /tmp/usr/all
  ```

  then the merged data is stored at /tmp/usr/all. In this case, the .ucm file is also stored at /tmp/usr/all. For more details on .ucm file, see Coverage Data Storage.

  Note: The <output_test> argument cannot include wildcards.

- -message adds diagnostic output to help locate configuration differences that limit merging. By default, only progress messages are output.

Consider the following tests:

```
./cov_work/scope/test1/
./cov_work/scope/test2/
./cov_work/scope/test3/
```

To merge coverage for above tests (test1 being the primary test) to all, use:

```
merge test1 test2 test3 -output all
```

OR

```
merge test1 * -output all
```

After the merge operation, merged results are stored in ./cov_work/scope/all.

Consider another set of tests:

```
./cov_work/my_scope1/test1/
./cov_work/my_scope2/test1/
```

To merge coverages from these tests, use:

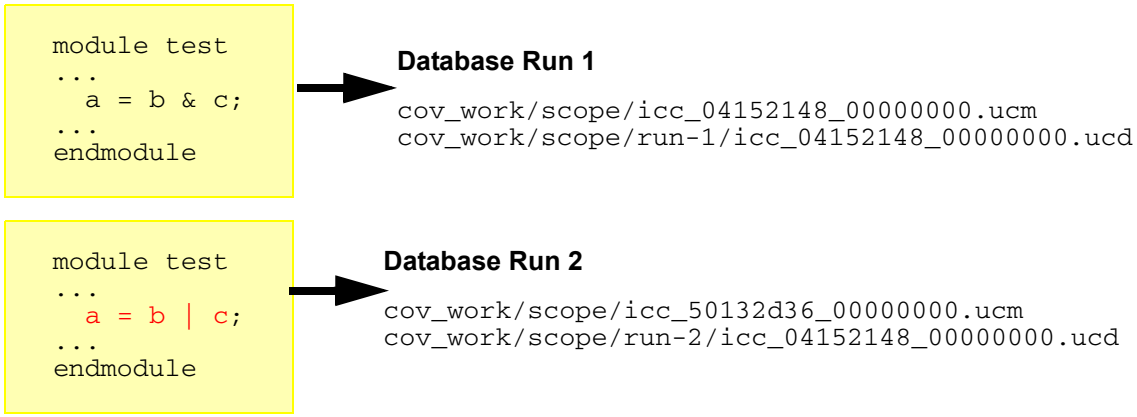
```
merge cov_work/my_scope1/test1 cov_work/my_scope2/test1 -output all
```

OR

```
merge my_scope1/test1 my_scope2/test1 -output all
```

After the merge operation, merged results are stored in `./cov_work/my_scope1/all`.

Consider another example where the design is modified and union merge is performed. If the design is modified, a separate model file (`.ucm`) is created for that test, as shown below.



To union merge coverage for above tests, use:

```
set_merge -union
set_dut_modules test
merge  -message run-1 run-2 -output merged_result
```

The above command saves the resultant model and the data file in the `merged_result` directory, as:

```
cov_work/scope/merged_result/icc_04152148_00000000.ucm
cov_work/scope/merged_result/icc_04152148_00000000.ucd
```

Notice that in the case of union merge, a new model file is created with the same name as the model of primary test (Run 1). This is because in a union merge, a copy of primary test (`.ucm`) is created in the merged output directory, and models of all secondary tests are unioned to the resultant model. Even though the name of the model file is same; however, the one in the merged output directory includes coverage types, non-conflicting instances, assertions, coverpoints, crosses, and covergroups that are not available in the primary model.

10.4 Causes of Merge Failure

The following table lists possible causes of merge failure from different simulations and some suggested actions:

Cause of Merge Failure	Suggested Action
DUT(s) that vary in multiple runs are included in the coverage scored in simulations.	■ Check the DUT(s) reported at the end of simulation. ■ Report on complete hierarchy and identify what is scored. ■ Set the DUT module for merge to the actual DUT.
HDL code differs between simulations. The code may be identical but still differ due to conditional compiles using `+define`/ `` `ifdef ``. (Checksum error is reported for this issue.)	■ Merge different configurations separately.
HDL code differs across simulations. For example, some extraneous code might get included for some runs because of a vunit file added through the `-propfile` option during compilation. (Checksum error is reported for this issue.)	■ Ensure that all the runs are instrumented with the `set_code_fine_grained_merging` command in the coverage configuration file during elaboration.
Design is instrumented with different coverage metrics for different tests. (Checksum error is reported for this issue.) **Note:** Applies to standard merge only.	■ Always instrument with the same coverage metrics. ■ Ensure that primary database has all coverage types to be merged from secondary tests.

Cause of Merge Failure	Suggested Action
Different coverage configuration options were used in elaboration for the tests, which then generated databases with different coverage points. Some of the examples are `set_assign_scoring`, `set_branch_scoring`, or `set_expr_scoring`. (Checksum error is reported for this issue. For a complete list of coverage configuration options that can lead checksum error, see Using CCF commands across simulations.)	■ Score with a fixed set of options.
Different work libraries across runs. This happens when one of the simulations is compiled using the `-v` or `-y` option. (Checksum error is reported for this issue.)	■ Ensure that all the tests to be merged are elaborated using the set_merge_with_libname command in the coverage configuration file.

10.5 Precedence while Merging

Each coverage point can be one of the following:

- COVERED (Hit count > 0)
- UNCOVERED (Hit count = 0)
- IGN (Marked IGNORED through COM or iccr `mark` command)
- CON (Determined impossible during instrumentation due to literal constants)
- COV (Marked COVERED using iccr `mark` command)

Note: Partially covered toggle signals (Rise Only/Fall Only hit) are treated as UNCOVERED for merge precedence.

When merging coverage using `merge` or `union` command, the result is the strongest of the values, as defined here:

```
COV > COVERED > IGN > CON > UNCOVERED
```

Note: The same precedence rule applies to module-based reporting.

If you want `IGN` to be considered stronger than real coverage, use:

```
iccr> set ign_stronger_than_real_coverage 1
```

With the above command, the precedence is changed as:

```
COV > IGN > CON > COVERED > UNCOVERED
```

Note: IGN and CON are treated as the same. The changed precedence applies to subsequent ICCR commands in that ICCR session.

11

Analyzing Coverage Data

After merging coverage data, the coverage tests are available for report generation and analysis. In the analysis phase, you evaluate the quality of the test stimuli and detect areas that require more testing. For this, you generate reports and finally order tests based on coverage efficiency. This chapter discusses the following topics:

- Loading Coverage Data
- Identifying Scored Coverage
- Generating Coverage Reports
- Analyzing Coverage in GUI
- Ranking and Ordering Tests
- Operating on Test Databases
- Other ICCR Commands and Functionality

Note: Starting this release, another tool Incisive Metrics Center (IMC) is available for coverage data analysis. For more details on IMC, see the *Incisive Metrics Center User Guide*.

11.1 Loading Coverage Data

To load coverage data for one or more tests and their associated model file, use:

```
load_test {<test_specification> | -testfile <filename>}
```

where

- `<test_specification>` specifies the coverage tests to be loaded. It can include just the test name (that can contain wildcard characters `*` and `?`), or the complete path, as:

  ```
  [[<workdir>/]<scope>/]<test>
  ```

  where

- <workdir> specifies the root location where all of the coverage data is stored. It must contain valid filename characters and path separators. <workdir> can use both absolute paths (for example, /home/me/chip/verification/todaysrev) and relative paths (for example, verification/todaysrev).
- <scope> specifies a particular configuration of hardware (DUT). It must contain valid filename characters.
- <test> specifies the test name for a single simulation. It should contain valid filename characters and can contain wildcards.

  Note: If you specify the complete path for a test, the current working directory and design directory is set as the one specified in the complete path. See set_workdir and set_design for more details.

- -testfile <filename> specifies the file that includes a list of tests to be loaded. The tests in <filename> should be listed one per line. You can include comments in <filename> using #.

Multiple tests that do not share the same model file cannot be loaded into a single session of ICCR.

To load all tests from the default cov_work/scope directory, use:

```
load_test *
```

To load tests TB1, TB2, and TB3 from the default cov_work/scope directory, use:

```
load_test TB1 TB2 TB3
```

To load coverage from mywork/myscope/test1 and mywork/myscope/test2, use:

```
load_test ./mywork/myscope/test1 ./mywork/myscope/test2
```

OR

```
load_test ./myw*/myd*/test*
```

If the number of path characters (/) is less than two, then the missing path is assumed to be the default path, which is cov_work. For example, if <test> is specified as:

```
load_test myscope/test1
```

then the coverage data is loaded from cov_work/myscope/test1.

If coverage data is stored at <current_working_directory>/myscope/test1 instead of cov_work/myscope/test1 then to load test data, use:

```
load_test ./myscope/test1
```

11.1.1 Listing the Loaded Tests

After loading the required tests, you can view the loaded tests using:

```
list_coverage_files
```

The output of the above command is:

```
Coverage model file
-------------------
./cov_work/scope/icc_671c3397_0cbded8d.ucm

Coverage data files
-------------------
D1       /home/ruchikas/cov_demo_dtmf/design/cov_work/scope/test1/icc_671c3397_0cbded8d.ucd
D2       /home/ruchikas/cov_demo_dtmf/design/cov_work/scope/test2/icc_671c3397_0cbded8d.ucd
```

11.1.2 Resetting Coverage

If you find the loaded tests are not the ones that are required, you can unload the tests using:

```
reset_coverage
```

This command unloads all of the loaded tests (data files and the associated model file). After `reset_coverage`, you can load the required tests using `load_test`.

Note: Any selections or marks must be saved before `reset_coverage` and then reapplied if desired after reloading a new test. For information on marks, see Chapter 12, “Manual Marking of Coverage”.

11.2 Identifying Scored Coverage

After loading the required tests, you might want to identify the modules and instances on which coverage is scored. Based on this information, you can generate various reports.

list_coverage

To list modules/design entities or instances in a design with enabled coverage types, use:

```
list_coverage [-instance | -module | -entity] <list>
```

where `<list>` is a list of instance names or module/entity names depending on whether `-instance` or `-module` /`-entity` is specified. It also can use wildcard characters.

Wildcard	Description	Example
*	matches any text from current location until next delimiter or end of string	`list_coverage -module test_mod*` matches all names beginning with `test_mod` such as `test_mod1`, `test_module`, and so on
?	matches any single character	`list_coverage -module test_mod?` matches all names beginning with `test_mod` followed by a single character such as `test_mod1` and `test_mod2`

...	matches that module/instance and all its descendents	`list_coverage -module mod...`
		matches all instances of `mod` and all its descendents
		`list_coverage -instance test.mod...`
		matches instance `test.mod` and all its descendents

Note: Instance names in any of the ICCR commands must be hierarchical paths that begin with the module name of the design top rather than the module name of the testbench.

To list all instances in the design with the coverage enabled for that instance, use:

```
list_coverage -instance *...
```

The output of the above command is:

```
Coverages            Instance
--------------       --------------------
b e t s a f d        dtmf_recvr_core
b e t s a f d        dtmf_recvr_core.TEST_CONTROL_INST
b e t s a f d        dtmf_recvr_core.ROM_512x16_INST
b e t s a f d        dtmf_recvr_core.RAM_128x16_TEST_INST
b e t s a f d        dtmf_recvr_core.RAM_128x16_TEST_INST.RAM_128x16_INST
b e t s a f d        dtmf_recvr_core.RAM_256x16_TEST_INST
b e t s a f d        dtmf_recvr_core.RAM_256x16_TEST_INST.RAM_256x16_INST
b e t s a f d        dtmf_recvr_core.TDSP_DS_CS_INST
```

List of enabled coverages

In the above output, `betsafd` indicates the type of coverage.

list_modules

Displays the list of modules within a design.

list_design_entities

Displays the list of all modules/design entities in a design.

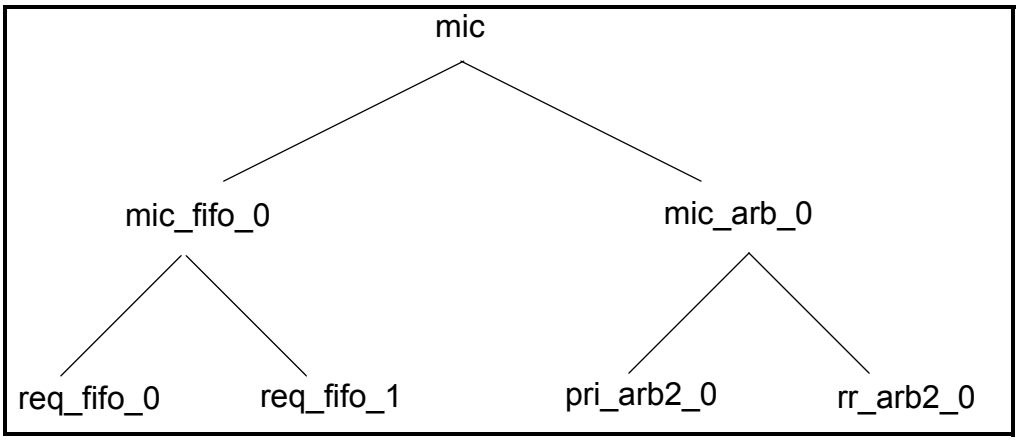
list_instances

Lists all instances of specified `[instance]`.

```
list_instances [instance]
```

If no `[instance]` is specified, all instances at the top-level in the design are displayed.

Consider the following hierarchy.



To list the instances within the instance `mic`, use:

```
list_instances mic
```

To list the instances within the instance `mic_fifo_0`, use:

```
list_instances mic.mic_fifo_0
```

In the above command, notice the use of (`.`) as the hierarchy delimiter. Use a (`.`) in a pure Verilog design. In a mixed design, use (`:`) as the hierarchy delimiter.

list_fsms

Lists all FSMs in the design.

```
list_fsms [-instance | -module | -entity]
```

- `-instance` lists instances in the design and state register for each instance.
- `-module` or `-entity` lists modules/entities in the design and state register for that module/entity.

11.3 Generating Coverage Reports

After identifying the available modules and instances in the design, you generate different reports for data analysis. By default, coverage reports are written to standard output unless redirected to a file. ICC provides commands to generate the following reports:

- Summary Report
- Detailed Report
- Tabular Report
- Block-Annotated Report
- HTML Report

11.3.1 Summary Report

Summary report lists coverage statistics as the number of hit items versus the total number of items. It gives a quick snapshot view of the coverage numbers. To generate a summary report, use:

```
report_summary [-instance | -module | -entity] [-hidezero] [-nocgopt]
     [-betsafd] <list>
```

In the above syntax,

- `-instance | -module | -entity` specifies if report should be generated based on instance, module, or entity. If not specified, `-module` is assumed.
- `-hidezero` hides items with no self coverage data (0/0). By default, such items are printed, which then shows the cumulative coverage.
- `-nocgopt` disables reporting of covergroup options. By default, covergroup options are reported. It applies to data-oriented functional coverage only. See Specifying Coverage Options for more details.
- `-betsafd` specifies the type of coverage to report:
  `b` indicates block coverage

`e` indicates expression coverage
`t` indicates toggle coverage
`s` indicates state coverage
`a` indicates arc and transition coverage
`f` indicates PSL/SVA based functional coverage
`d` indicates SystemVerilog based functional coverage

If not specified, all coverage types are reported.

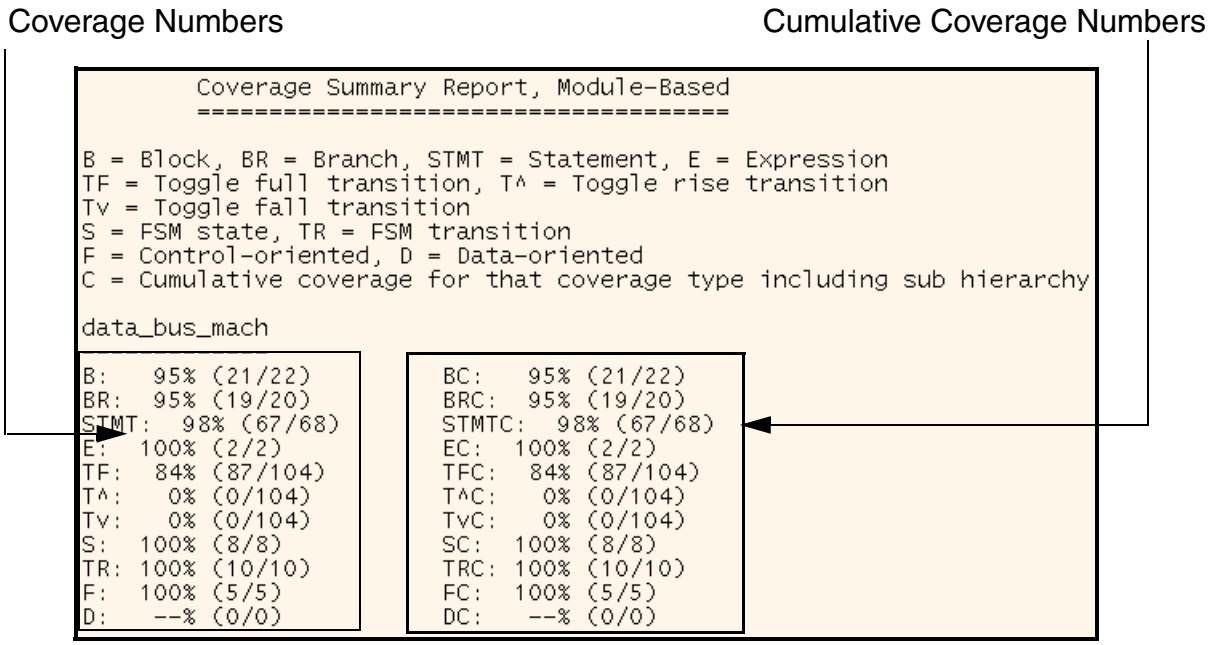
Note: The nomenclature `betsafd` applies to all ICCR commands and output.

- `<list>` specifies the names of modules or design entities, or instances for which the coverage report will be generated. It can include wildcard characters `*` and `?`. Use `...` to specify all descendants of an instance. Instance names must be hierarchical paths that begin with the module name of the design top rather than the module name of the testbench.

Example: All Coverage Types

To print a module-based summary report for all coverage types for module `data_bus_mach`, as shown below, use:

```
report_summary -module data_bus_mach
```



The summary report shows absolute coverage numbers for a coverage type as (Covered items / Total items /Marked items). If there are no marked items, the report shows the coverage numbers as (Covered items / Total items).

Self coverage shows the number of covered items in the current instance or module divided by the total number of items in that instance or module.

Cumulative coverage of a module is the sum of coverage of all instantiations of that module, and each module in its sub-hierarchy. Each module contributes only once to the total number of covered items regardless of the number of instantiations, but its coverage is merged for every instance. For example, module `mod2` is instantiated in `topA` and `topB`. Coverage for module `mod2` is `(6/10)` in `topA` and `(3/10)` in `topB`. The cumulative coverage calculated for `modA` is `(9/10)`.

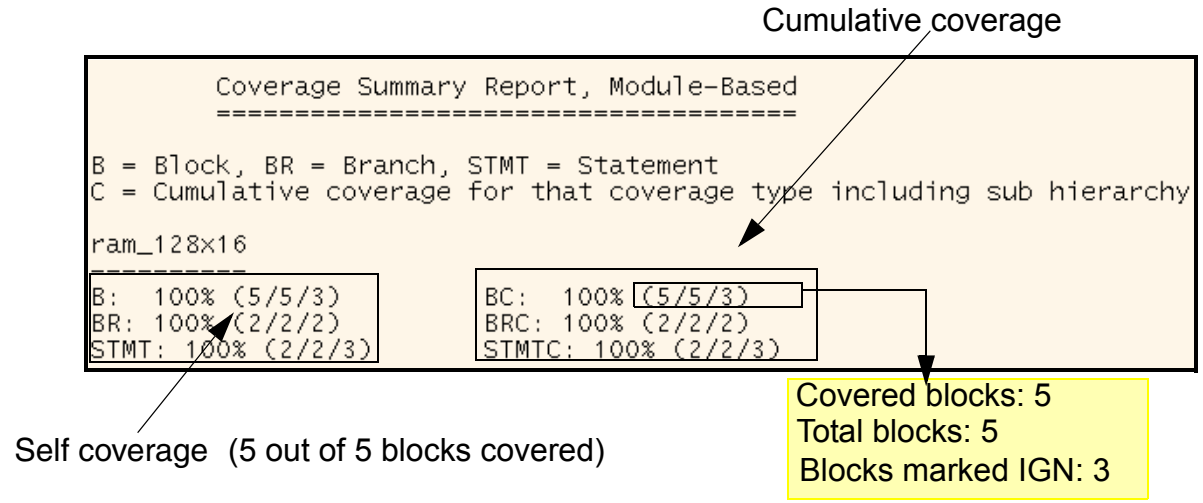
Cumulative coverage of an instance is the sum of coverage in that instance and each instance present in its sub-hierarchy.

Note: In calculating coverage and percentages, an object is considered covered if its hit count is at least 1. For toggle coverage, an object is considered covered if the full toggle hit count is at least 1.

Example: Block Coverage

To print a summary report for block coverage for module `ram_256x16_test`, as shown below, use:

```
report_summary -module -b ram_128x16
```



The above report is generated after enabling branch and statement coverage during elaboration.

Note: If more than one test is loaded and the instance(s) or module(s) for which the report is generated exist in more than one, the coverage numbers for those instance(s) or module(s) in each test are reported, as shown below:

```
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
<> COVERAGE FILE: /home/ruchikas/cov_demo_dtmf/design/cov_work/design/test1/icc.ucd <>
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

        Coverage Summary Report, Module-Based
        =====================================

B = Block, BR = Branch, STMT = Statement
C = Cumulative coverage for that coverage type including sub hierarchy

ram_128x16
----------
B:  100% (5/5/3)         BC:  100% (5/5/3)
BR: 100% (2/2/2)         BRC: 100% (2/2/2)
STMT: 100% (2/2/3)       STMTC: 100% (2/2/3)
```

Coverage from test1

```
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
<> COVERAGE FILE: /home/ruchikas/cov_demo_dtmf/design/cov_work/design/test2/icc.ucd <>
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

        Coverage Summary Report, Module-Based
        =====================================

B = Block, BR = Branch, STMT = Statement
C = Cumulative coverage for that coverage type including sub hierarchy

ram_128x16
----------
B:  100% (5/5/3)         BC:  100% (5/5/3)
BR: 100% (2/2/2)         BRC: 100% (2/2/2)
STMT: 100% (2/2/3)       STMTC: 100% (2/2/3)
```

Coverage from test2

Example: Expression Coverage

To print an expression coverage summary report for instance `TEST_CONTROL_INST` in module `dtmf_recvr_core`, use:

```
report_summary -instance -e dtmf_recvr_core.TEST_CONTROL_INST
```

```
        Coverage Summary Report, Instance-Based
        =======================================

E = Expression
C = Cumulative coverage for that coverage type including sub hierarchy

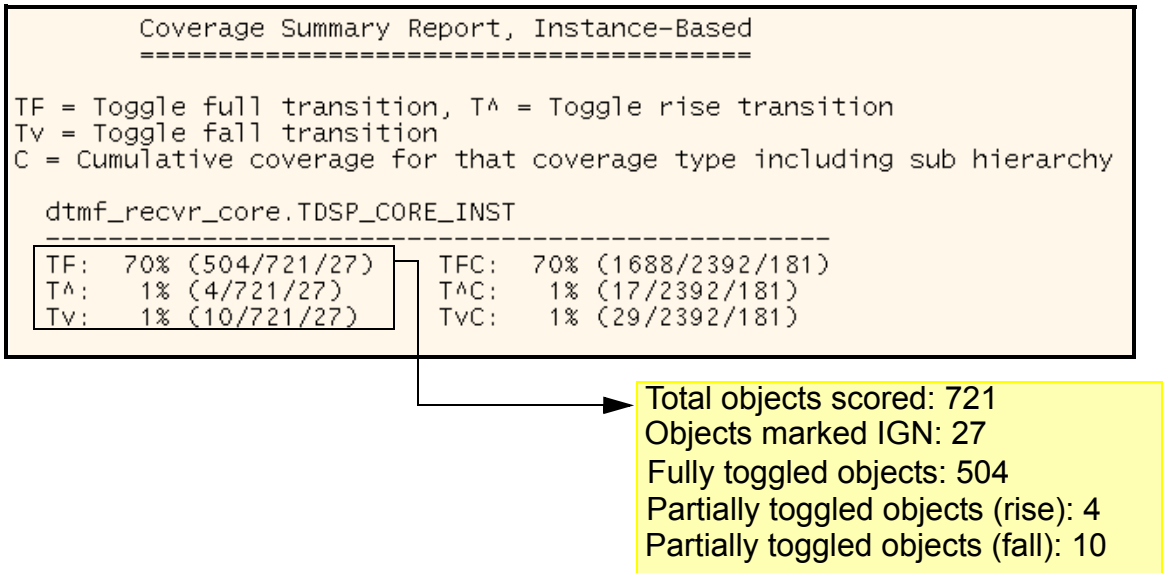
  dtmf_recvr_core.TEST_CONTROL_INST
  --------------------------------------------------
  E:    40% (8/20)          EC:    40% (8/20)
```

8 out of 20 expression terms are covered

Example: Toggle Coverage

To print a toggle summary report for instance `dtmf_recvr_core.TDSP_CORE_INST`, use:

```
report_summary -t -instance dtmf_recvr_core.TDSP_CORE_INST
```



Note: Similarly, you can generate summary reports for FSM and functional coverage data.

11.3.2 Detailed Report

The detailed report lists details for all or selected module/design entity or instances in the design. To generate a detailed report, use:

```
report_detail [-instance | -module | -entity] [-all | -covered | -uncovered | -both
     | -marked] [-nocgopt] [-nosource] [-nocompact] [-betsafd] <list>
```

In the above syntax:

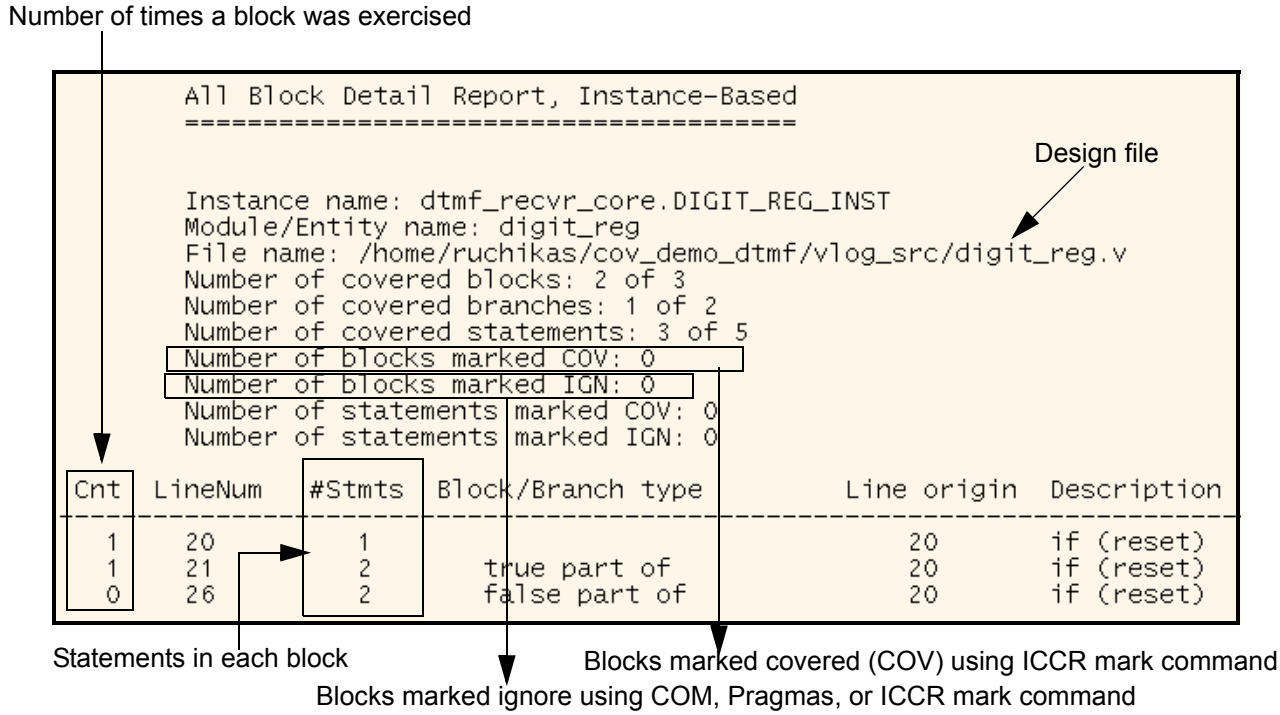
- `-all | -covered | -uncovered | -both | -marked` specifies whether to generate reports for all, covered, uncovered, both covered and uncovered, or marked coverage items.
- `-nosource` disables reporting of source text and line numbers from the source file. By default, this information is reported.
- `-nocompact` enables listing of consecutive covergroup bins in separate rows. By default, such covergroup bins are listed as a range in a single row. It applies to data-oriented functional coverage only.

See Summary Report for details on syntax description of other options.

Example: Block Coverage

To print a block coverage report for the instance `DIGIT_REG_INST`, as shown below, use:

```
report_detail -instance -both -b dtmf_recvr_core.DIGIT_REG_INST
```



The above report is generated after enabling branch and statement coverage.

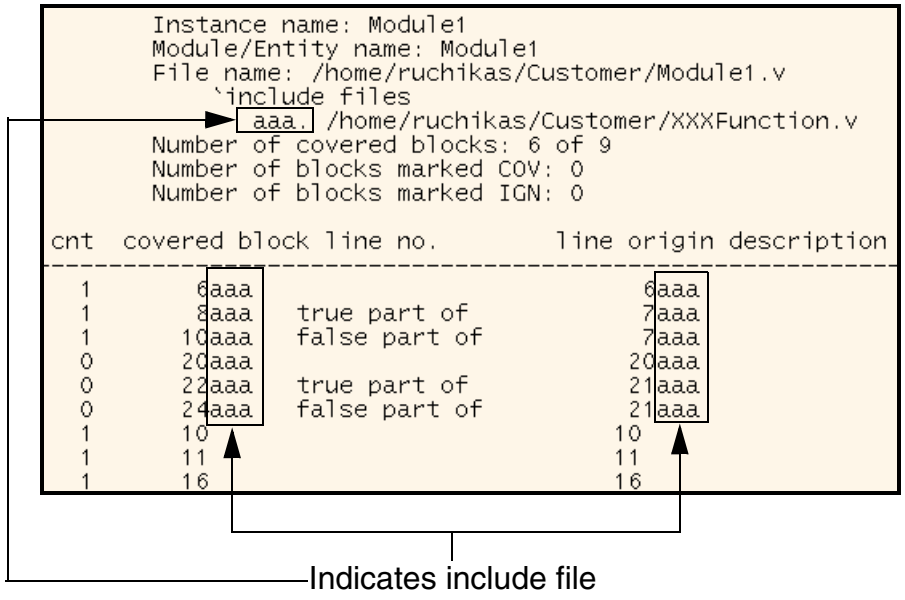
Note: In a block detailed report, the branches are identified by descriptions next to them as ternary, true part of, false part of, a case item of, or implicit else. The above report has branches on Line `21` and Line `26` (which are also scored as blocks).

If the code is coming from `include` files, you will observe:

- Additional lines printed in the report header to indicate the name and location of the `include` file.
- An additional string (for example, `aaa`) prior to printing the location of the include file in the report header.
- An additional string (for example, `aaa`) appended to the line number in data rows.

Note: String `aaa` indicates the first include file. In case of multiple include files, appended string is named as `aab`, `aac`, and so on.

The following block coverage report shows code being printed from include file `XXXFunction.v`.

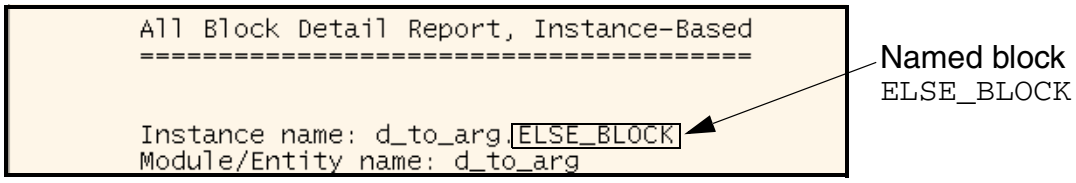


Note: Similar information is printed in the case of expression coverage reports.

If named blocks are defined in a `generate` block, as shown below.

```
generate
   if(I==0)
      begin : IF_BLOCK
         ...
      end
   else
      begin : ELSE_BLOCK
        ...
      end
endgenerate
```

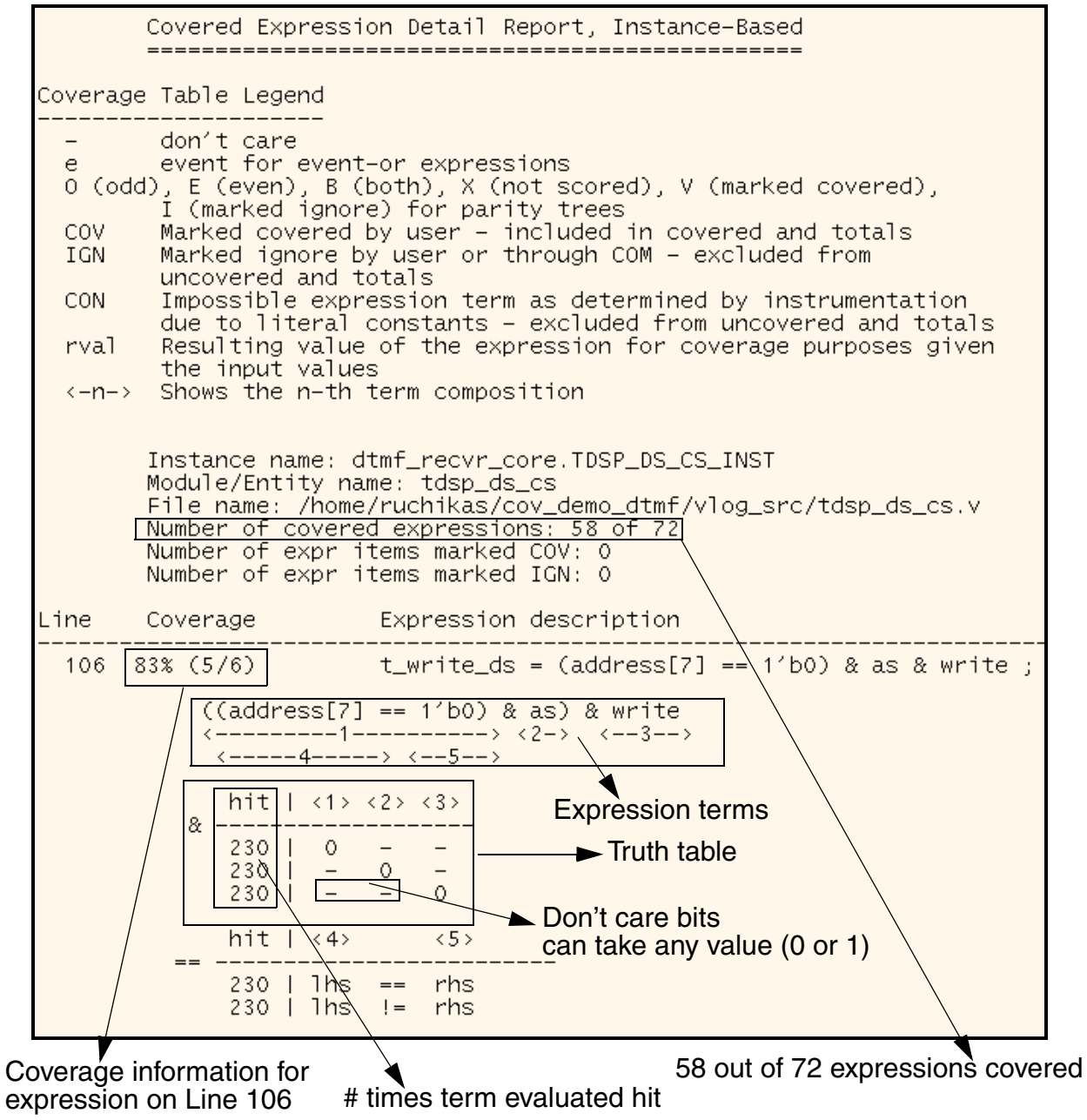
then the instance name in the report, includes the name of the named block as:



Example: Expression Coverage

To print a detail report displaying expression coverage data for the instance `TDSP_DS_CS_INST`, as shown below, use:

```
report_detail -instance -covered -e dtmf_recvr_core.TDSP_DS_CS_INST
```



In the expression coverage detailed report:

- `-` indicates don't care bits
- `e` indicates event for event-or expressions
- `O` represents odd bit in a parity tree
- `E` represents even bit in a parity tree
- `B` represents both even and odd bits in a parity tree
- `X` indicates that the bit is not scored in a parity tree
- `V` indicates that the bit is marked covered in a parity tree
- `I` indicates that the bit is marked ignored in a parity tree
- `COV` represents the items marked covered by the user
- `IGN` represents items marked ignored by the user or through COM
- `CON` represents items that are determined impossible during instrumentation due to literal constants
- `rval` displays the resulting value of expression
- `<-n->` shows the n^{th} term composition

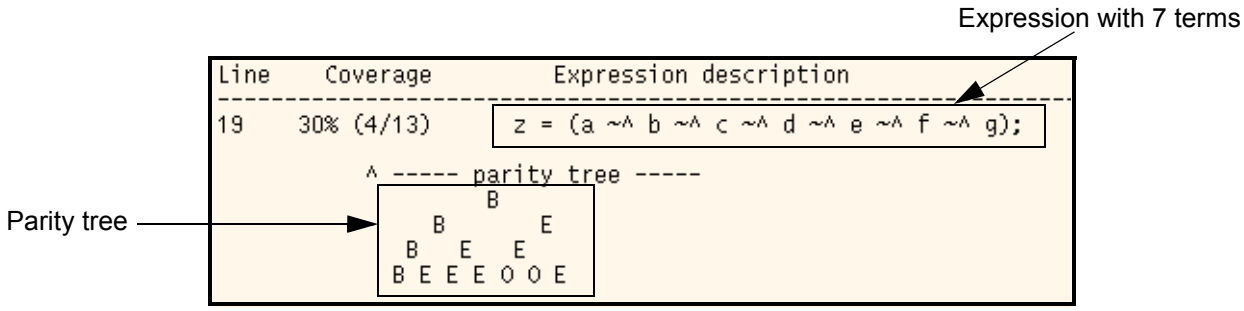
Example: ^ and ~^ Operators (Exclusive OR and Equivalence, Greater than 4 terms)

Exclusive OR and bit-wise equivalence operations of more than four terms are scored using a parity tree. Instead of a truth table, the report contains a parity tree. In a parity tree, you typically find the following terms:

- E - Represents even
- O - Represents odd
- B - Represents both even and odd
- X - Represents no scoring
- V - Represents marked covered (using the `mark` command)
- I - Represents marked ignored (using the `mark` command)

Scoring is 100% when each node of the tree has both (B). Any x or z input results in no scoring. The lowest level in a parity tree represents each term in the expression individually. For example, five values at the lowest level indicate five terms in the expression. A parity tree is formed by performing an exclusive OR between two expression terms (making a pair) at

the lowest level. The resultant output is the node at the next level. If the number of terms at the lowest level is odd, the last term for which a pair could not be created is ignored for the time being and a pair is formed later when a similar situation is encountered at any of the above levels. Consider the following output.



Notice how the parity tree is created in this case.

- Lowest level - `B E E E O O E` for each term in the expression
- Second level - `B E E` (Exclusive OR between `B` - `E`, `E` - `E`, and `O` - `O`) last `E` is ignored for the time being
- Third level - `B E` (Exclusive OR between `B` - `E` from the second level and `E` (from the second level - `E` from the lowest level)
- Top level - `B` (Exclusive OR between `B` - `E` from the third level)

In the above report, 4/13 indicates that there were 13 nodes in the parity tree out of which four nodes had a value B, which represents the four expressions that are considered covered.

For more information on interpreting results using optional control and vector scoring, see Chapter 3, “Expression Coverage.”

Example: Toggle Coverage

To print a detail report displaying toggle coverage data for the instance `ARB_INST`, as shown below, use:

```
report_detail -t -both -inst dtmf_recvr_core.ARB_INST
```

```
        All Toggle Detail Report, Instance-Based
        ========================================


        Instance name: dtmf_recvr_core.ARB_INST
        Module/Entity name: arb
        File name: /home/ruchikas/cov_demo_dtmf/vlog_src/arb.v
        Number of signal bits fully toggled: 12 of 13
        Number of signal bits partially toggled(rise): 0 of 13
        Number of signal bits partially toggled(fall): 0 of 13
        Number of signal bits marked COV: 0
        Number of signal bits marked IGN: 2

Hit(Full)   Hit(Rise)   Hit(Fall)   Signal
-----------------------------------------
1           1           1           reset
50          50          50          clk
17          17          17          dma_breq
50          50          50          tdsp_breq
17          17          17          dma_grant
17          18          17          tdsp_grant
0           0           0           scan_en
17          17          17          next_state[2]
50          50          50          next_state[1]
50          50          50          next_state[0]
17          17          17          present_state[2]
50          50          50          present_state[1]
50          50          50          present_state[0]
```

Signal toggle information

Signal rise 18 times
Signal fall 17 times

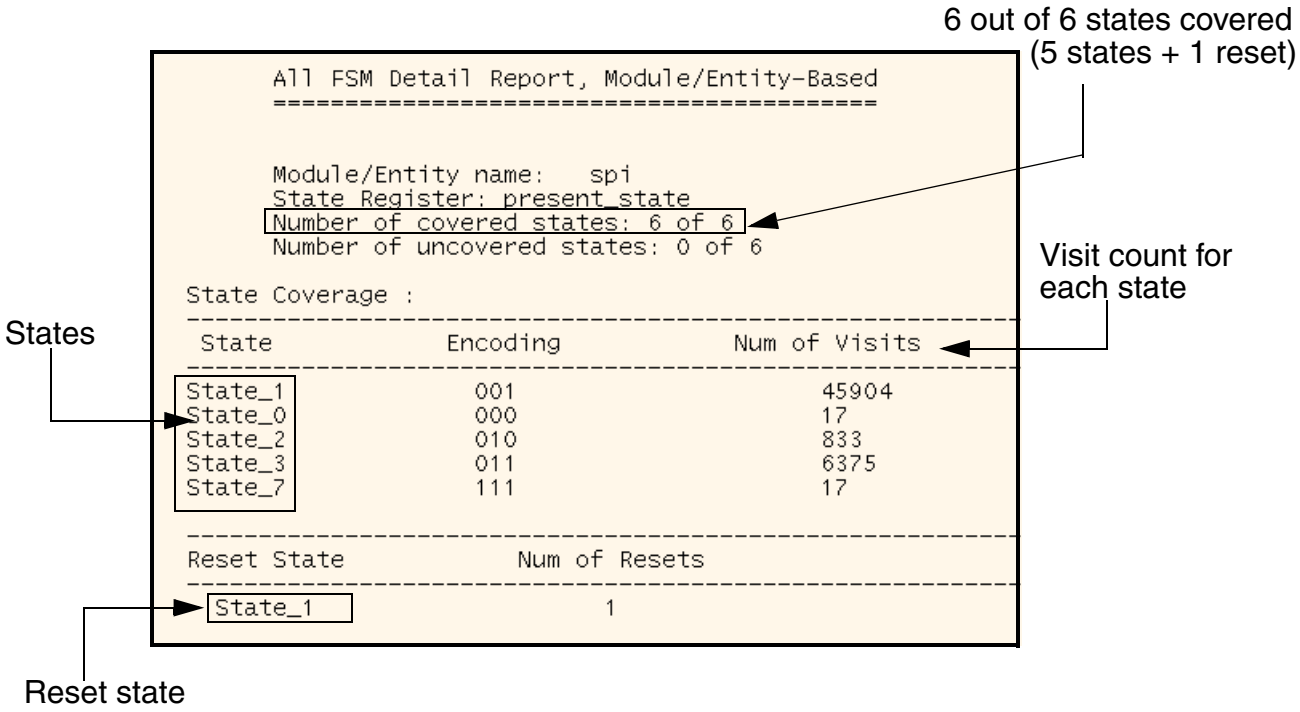
In the above report,

- `Signal bits fully toggled` have toggled through rise as well as fall transitions at least once.
- `Signal bits partially toggled (rise)` have toggled through rise transition only.
- `Signal bits partially toggled (fall)` have toggled through fall transition only.
- `Hit(Full)` is calculated as a minimum of `Hit(Rise)` and `Hit(fall)` counts.

Examples: FSM State and Transition Coverage

To report detailed state coverage data for the module `spi`, as shown below, use:

```
report_detail -both -module -s spi
```

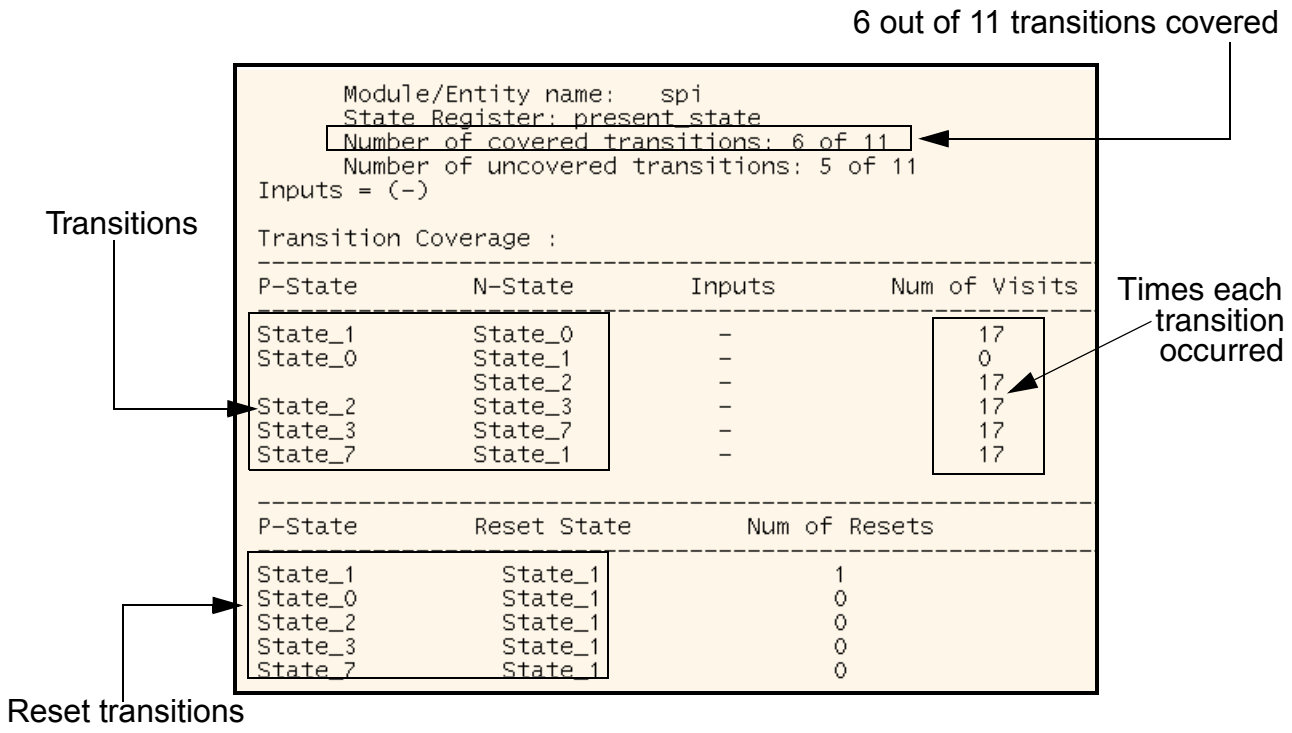


By default, reset states are not considered in calculating the number of states. To consider reset states in coverage numbers, use the `set_fsm_reset_scoring` command in the CCF and regenerate coverage data. The above report is generated with `set_fsm_reset_scoring` enabled in the CCF.

In the absence of this command, the number of states would be 5 instead of 6, and number of visits for state `State_1` would be 45905.

To report detailed transition coverage data for the module `spi`, as shown below, use:

```
report_detail -both -module -a spi
```



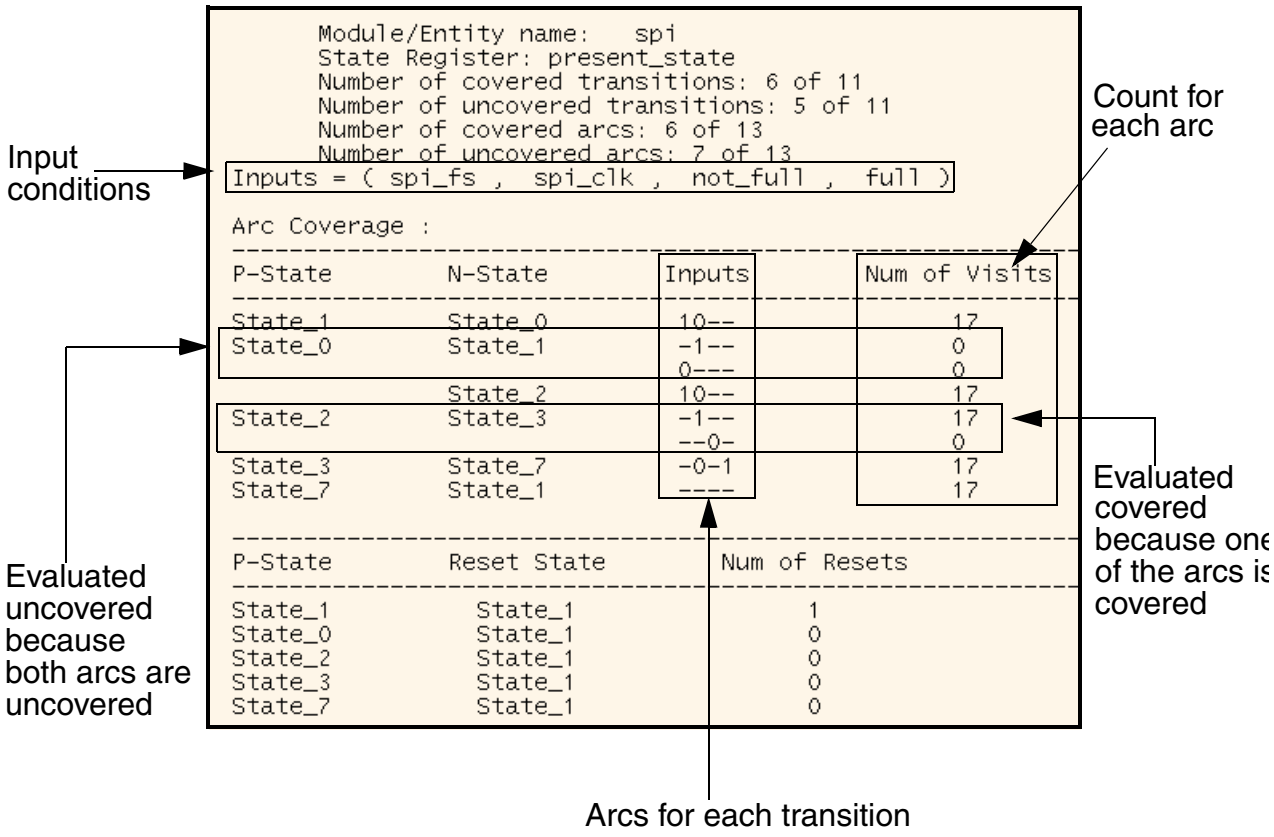
By default, reset transitions are not considered in calculating the number of transitions. To consider reset transitions in coverage numbers, use the `set_fsm_reset_scoring` command in the CCF and regenerate coverage data. The above report is generated with `set_fsm_reset_scoring` enabled in the CCF.

In the absence of this command, total number of transitions would be 6 instead of 11 (excluding 5 reset transitions).

Note: By default, ICC does not score FSM hold transitions (`State_0 -> State_0`). To enable scoring of FSM hold transitions, use the `set_fsm_scoring -hold_transition` in the CCF, and regenerate coverage data.

In the report, the `Input` column displays a (`-`) because arc coverage scoring is OFF. To report arcs, first enable arc scoring for selected FSMs, using the set_fsm_arc_scoring command in the CCF and regenerate coverage data. After enabling arc scoring, each state

transition with all possible input conditions under which the transition takes place is reported, as shown below.



The Input column lists all possible arcs for each transition. In the above report, transition `State_2` **->** `State_3` happens under two input conditions, `-1--` and `--0-`, respectively.

A transition is evaluated covered if any of the arcs for that transition are covered. A transition is evaluated uncovered, if all of the arcs for that transition are uncovered.

Note: The number of visits for a state may vary when arc coverage is enabled versus when arc coverage is disabled. If any of the inputs is `X`, it will match the corresponding column in the SOP row only if it is `-`. It will not match `1/0`. If the input values do not match any SOP row because of any input being `X`, ICC will not increment state and transition counts for that transition. If arc scoring is disabled, ICC does not match any SOP rows and counts the transition.

Examples: Functional Coverage

To generate a detailed report for control-oriented functional coverage items for instance `DMA_INST` in module `dtmf_recvr_core`, use:

```
report_detail -f -both -instance dtmf_recvr_core.DMA_INST
```

```
        All Control-oriented Functional Coverage Detail Report, Instance-Based
        ======================================================================

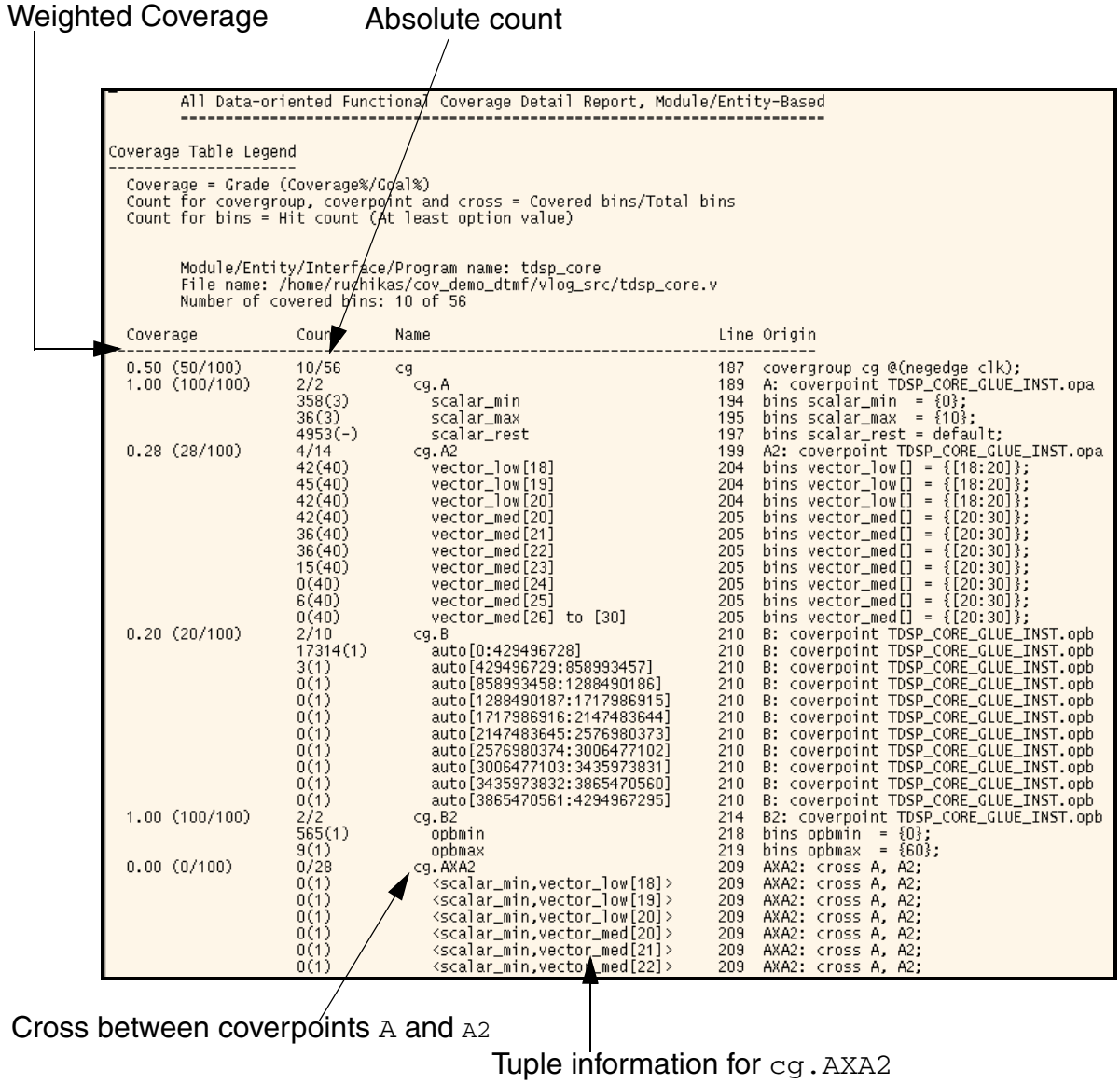

        Instance name: dtmf_recvr_core.DMA_INST
        Module/Entity/Interface/Program name: dma
        File name: /home/ruchikas/cov_demo_dtmf/vlog_src/dtmf_recvr_core.v
        Number of covered PSL/SVA coverage points: 3 of 7
 cnt             coverage point  line origin description
-------------------------------------------------------------------------------
   0    dma_req_when_data_ready   165 // psl assert dma_req_when_data_ready;
  17          dma_wait_for_grant   172 // psl assert dma_wait_for_grant;
   0          dma_read_spi_pulse   187 // psl assert dma_read_spi_pulse;
   0   dma_address_strobe_pulse   190 // psl assert dma_address_strobe_pulse;
  17   dma_transfer_reset_delay   199 // psl assert dma_transfer_reset_delay;
   0              dma_bufselect1   205 // psl assert dma_bufselect1;
   4              dma_bufselect0   208 // psl assert dma_bufselect0;
```

Property successfully finished 17 times

3 out of 7 coverage points covered

To display a detail report for data-oriented functional coverage points for module `tdsp_core`, use:

```
report_detail -both -d -mod tdsp_core
```



In the above report:

- *Name* column displays the names of covergroups, coverpoints, crosses, and bins.
- *Count* column shows the absolute coverage counts for each coverage item in the design. The value displayed for different coverage items is:

Covergroup, coverpoint, and cross	`Total covered bins / Total bins`

Coverpoint bins	`Hit count` (value of `at least` option)
Default bins	`Hit count` (–)

- *Coverage* column shows the weighted coverage for each coverage item in the design. The calculation for different coverage items is shown below:

Coverpoint	`Weighted coverage (Actual goal / Target goal)` where, - `Target goal` is the value set using the `goal` option. If not set, 90 is assumed. - `Actual goal` is `(Covered bins / Total bins) * 100`. The value displayed is rounded off. - `Weighted coverage` is the normalized positive number <= 1, which is calculated as `Actual goal / Target goal`

Covergroup	`Weighted coverage (Actual goal / Target goal)`
	where,
	- `Target goal` is the value set using the `goal` option. If not set, 90 is assumed.
	- `Actual goal` is calculated as:
	$\frac{\text{weight\_of\_cov\_item\_1} * (\text{hit\%\_of\_cov\_item\_1}) + \text{weight\_of\_cov\_item\_n} * (\text{hit\%\_of\_cov\_item\_n})}{\text{weight\_of\_cov\_item\_1} + \text{weight\_of\_cov\_item\_n}}$
	where,
	`cov_item` includes all the coverpoints and crosses in the covergroup.
	`hit%_of_cov_item` is `(Covered bins / Total bins) * 100`
	- `Weighted coverage` is the normalized positive number <= 1, which is calculated as `Actual goal / Target goal`
	Note: The weight can be set either using the `type_option` keyword or using the `option` keyword. See Specifying Coverage Options for more details.

Data-oriented coverage	`Cumulative coverage (Weighted total coverage / Weighted total goal)`
	where,
	- `Weighted total goal` is `sum_of_weight_of_all_covergroups * 100`.
	- `Weighted total coverage` is calculated as:
	`(weight_of_covergrp_1 * actual_goal_of_covergrp_1)`
	`+(weight_of_covergrp_n * actual_goal_of_covergrp_n)`
	- `Cumulative coverage` is the normalized positive number <= 1, `Weighted total coverage / Weighted total goal`

In the above report, tuple information for each cross is displayed. If the bin count for a cross exceeds the predefined limit of 5000, then the detail tuple information for uncovered cross bins is not stored in the coverage database. Currently, you cannot modify the predefined limit. In such situations, the tuple information is shown as a range of values in the coverage report.

Note: If any of the coverpoint of a cross is of type `enum`, then the bin for that coverpoint will be shown using enum constant in the bin tuple.

In a detailed report, by default consecutive uncovered bins are reported as a range in a single row. To report such bins as separate items, use the `-nocompact` option.

In a detailed report, a default bin, if covered (with hit count > 1) is reported. Displaying default bins helps users to keep track of erroneous values or any uncaptured values by any other bin. The covered default bins though reported are ignored while calculating the total number of bins.

Following items are not reported in a detailed report:

- Uncovered default bins
- Uncovered bins of a vector bin, if the individual bins of a vector bin exceed the predefined limit of 1 million.
- Uncovered bins if the total number of bins for a coverpoint reaches the limit of `0xfffff` with a particular bin. For example,

```
A: coverpoint a{
   bins b1[] = {5,8};
   bins b2[] = {$:4};
   bins b3[] = {7,6};
   bins b4[] = {9,11};
}
```

  For the above code, if the total number of bins reaches the limit of `0xfffff` with bin `b2`, then uncovered bins for `b2` and bins defined after `b2` (`b3` and `b4`) will not be reported.

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.

Important

The detailed report for block, expression, and functional coverage displays the line description information from the source file used to generate coverage data. Line description information is not displayed if the location or content of the source file changes. If the location changes, use the map_source_path command to map paths so that ICCR can locate the source files. If the content changes, a warning is reported.

11.3.3 Tabular Report

A tabular report presents coverage summary data sorted in a tabular format. The syntax for generating a tabular report is:

```
report_tabular_summary [-betsafd] [-sort <betsafd>] [-raw] [-hidezero] [-nocgopt]
[-instance | -module | -entity] <list>
```

By default, results are sorted on the coverage types specified in the report and using percentages. To specify different sorting options, use:

- `-sort <betsafd>` to specify the coverage type to use for sorting coverage results.
- `-raw` to sort the coverage results by the number of uncovered items instead of the coverage percentages. It helps you identify instances or module/entities with most uncovered items.

See Summary Report for details on syntax description of other options.

To print a tabular summary report for block and expression coverage for all of the instances in the design and to sort the report based on uncovered items, as shown below, use:

```
report_tabular_summary -be -raw -instance *...
```

Block coverage statistics

Expression coverage statistics

```
        Coverage Summary Report, Instance-Based
        =======================================

B = Block. BR = Branch. E = Expression
ToU =  Total uncovered items
Coverage data sorted by:  Block Expression
Coverage data sorted using uncovered items (-raw)

ToU     B(%)                E(%)                Name
-------------------------------------------------------
392      43 (211/485)        36 (69/187)        dtmf_recvr_core.TDSP_CORE_INST.EXECUTE_INST
108       9 (9/99)           10 (2/20)          dtmf_recvr_core.RESULTS_CONV_INST
38       69 (16/23)          29 (13/44)         dtmf_recvr_core.TDSP_CORE_INST.ALU_32_INST
30       58 (41/70)          83 (5/6)           dtmf_recvr_core.TDSP_CORE_INST.DECODE_INST
23       50 (7/14)           50 (16/32)         dtmf_recvr_core.TDSP_CORE_INST.MULT_32_INST
19       64 (9/14)           80 (58/72)         dtmf_recvr_core.TDSP_DS_CS_INST
17       50 (5/10)           40 (8/20)          dtmf_recvr_core.TEST_CONTROL_INST
12       38 (7/18)           50 (1/2)           dtmf_recvr_core.TDSP_CORE_INST.PORT_BUS_MACH_INST
12       89 (26/29)          83 (44/53)         dtmf_recvr_core.TDSP_CORE_INST.TDSP_CORE_GLUE_INST
7        92 (26/28)          81 (22/27)         dtmf_recvr_core.SPI_INST
5        72 (13/18)         100 (2/2)           dtmf_recvr_core.TDSP_CORE_INST.PROG_BUS_MACH_INST
5        50 (1/2)            0 (0/4)           dtmf_recvr_core.RAM_256x16_TEST_INST
5        -- (0/0)           77 (17/22)         dtmf_recvr_core.TDSP_CORE_INST.ACCUM_STAT_INST
4        87 (27/31)         100 (9/9)           dtmf_recvr_core.ARB_INST
3        94 (17/18)          77 (7/9)           dtmf_recvr_core.TDSP_CORE_INST.TDSP_CORE_MACH_INST
3        50 (1/2)            50 (2/4)           dtmf_recvr_core.RAM_128x16_TEST_INST
2       100 (2/2)            66 (4/6)           dtmf_recvr_core.ULAW_LIN_CONV_INST
1        95 (21/22)         100 (2/2)           dtmf_recvr_core.TDSP_CORE_INST.DATA_BUS_MACH_INST
1        66 (2/3)            -- (0/0)           dtmf_recvr_core.DIGIT_REG_INST
1        94 (17/18)          -- (0/0)           dtmf_recvr_core.DMA_INST
0       100 (5/5)           100 (3/3)           dtmf_recvr_core.RAM_128x16_TEST_INST.RAM_128x16_INST
0       100 (3/3)           -- (0/0)            dtmf_recvr_core.RAM_256x16_TEST_INST.RAM_256x16_INST
0       100 (9/9)           -- (0/0)            dtmf_recvr_core.DATA_SAMPLE_MUX_INST
```

Total number of uncovered blocks and expressions for `dtmf_recvr_core.TDSP_CORE_INST.EXECUTE_INST`

The above report is sorted based on uncovered items and hence the instances with 100% coverage items are reported last. In the absence of the `-raw` option, the report is generated and sorted based on covered items.

11.3.4 Block-Annotated Report

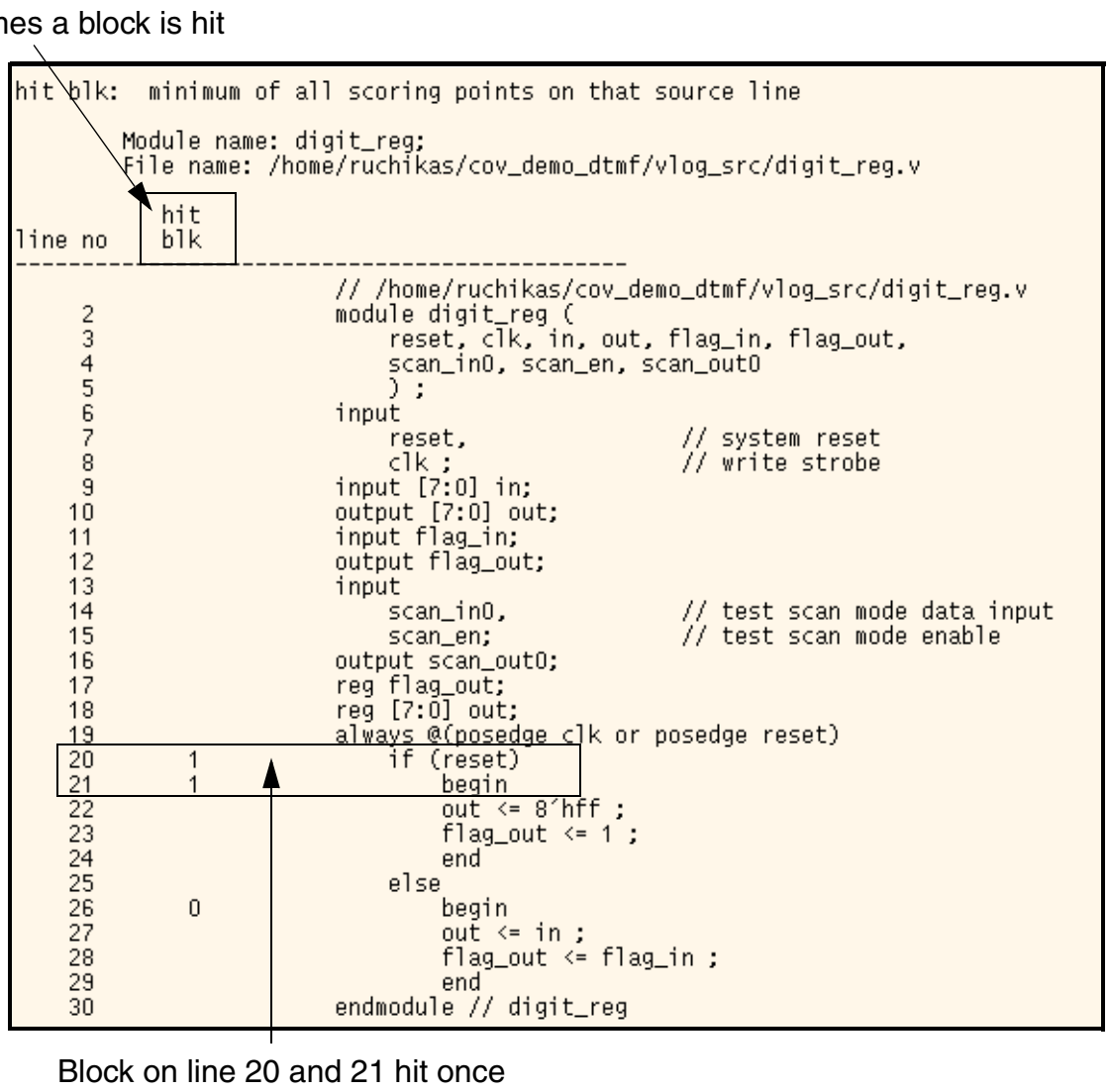
Block-annotated source reports display the source annotated with block coverage data. Each block in the report is accompanied by its hit count. To generate a block-annotated report, use:

```
report_block_annotated_source [-instance | -module | -entity] <list>
```

See Summary Report for details on syntax description.

To print a block-annotated report for module `digit_reg`, as shown below, use:

```
report_block_annotated_source -module digit_reg
```



Note: In a block-annotated report, if a line has more than one statement, then the `hit blk` column displays the minimum hit value of the statements. For example, if the line has two statements and one is uncovered, then the whole line will be considered uncovered.

11.3.5 HTML Report

Coverage data can also be analyzed in an HTML browser. To do so, generate an HTML report using the `report_html` command. The HTML report includes hyperlinks to navigate the design hierarchy and covergroup items, view cumulative and self coverage numbers, view detailed reports for specific coverage types, and view relevant source code.

11.3.5.1 Generating HTML Reports

An HTML report can be generated by scripts as part of a nightly regression run and merged so that the data can be viewed in the morning. To generate HTML reports, use:

```
report_html [-instance | -module | -entity] [-output <rptname>]
            [-all [betsafd]] [-covered [betsafd]] [-uncovered [betsafd]]
            [-both [betsafd]] [-marked [bet]]
            [-nosource] [-nocompact] [-noabsolute] [-betsafd] <list>
```

In the above syntax,

- `-output <rptname>` redirects the HTML report files to an alternate location instead of the default `html_<timestamp>` directory. In the absence of the `-output` option, a directory named `html_<timestamp>` is created in the current working directory, and coverage HTML reports along with the top-level summary page (`index.html`) are stored in the `html_<timestamp>` directory.

  Note: A top-level summary page `index.html` is used to navigate through the HTML reports available in the `html_<timestamp>` directory.

  With the `-output` option, the command redirects the HTML output to `<rptname>`. If `<rptname>` includes just the name of the output directory, then the command creates the output directory named `<rptname>` in the current working directory, and stores the HTML report files along with the top-level summary page in `<rptname>`. For example, if `<rptname>` is specified as:

  ```
  report_html -bet -all -module * -output day1
  ```

  then directory named `day1` is created in the current working directory to store HTML report files.

  Note: If `<rptname>` already exists, the output directory is not overwritten.

  If `<rptname>` includes the complete path, then the command creates a directory by the name mentioned at the lowest level in the path, and stores the HTML report files at that location. For example, if `<rptname>` is specified as:

  ```
  report_html -bet -all -module * -output data/myreports/day1
  ```

  then directory named `day1` is created under `data/myreports` to store HTML report files. In this case, the path (`data/myreports`) must exist. The directory `day1` must not exist, as it will be created and not overwritten.

- `-noabsolute` disables printing of absolute numbers (Covered items / Total items / Marked items) in the summary tables. By default, the summary table also shows absolute coverage numbers for that coverage type.

- `-all [betsafd]` generates reports for all (covered, uncovered, and marked) coverage items for specified coverage types.

- `-covered [betsafd]` generates reports for covered coverage items for specified coverage types.
- `-uncovered [betsafd]` generates reports for uncovered coverage items for specified coverage types.
- `-both [betsafd]` generates reports for both covered and uncovered coverage items for specified coverage types.
- `-marked [bet]` generates reports for marked coverage items for specified coverage types.

  Note: A coverage type (`b`, `e`, `t`, `s`, `a`, `f`, or `d`) should be specified with only one of the `-all`, `-covered`, `-uncovered`, `-both`, and `-marked` options. If a coverage type is specified with multiple options, then the coverage report is generated with following precedence:

  ```
  all > marked > both > uncovered > covered
  ```

  For example, the following command generates block coverage report for marked items.

  ```
  report_html -covered be -marked b -mod *
  ```

  In addition, if a coverage type is specified with any of the `-all`, `-covered`, `-uncovered`, `-both`, and `-marked` options, then specifying it with `-betsafd` becomes redundant. For example, in the following command, `-be` becomes redundant as it is already specified with `-covered` and `-marked` options, respectively.

  ```
  report_html -covered b -marked e -bet -mod *
  ```

See Detailed Report for details on syntax description of other options.

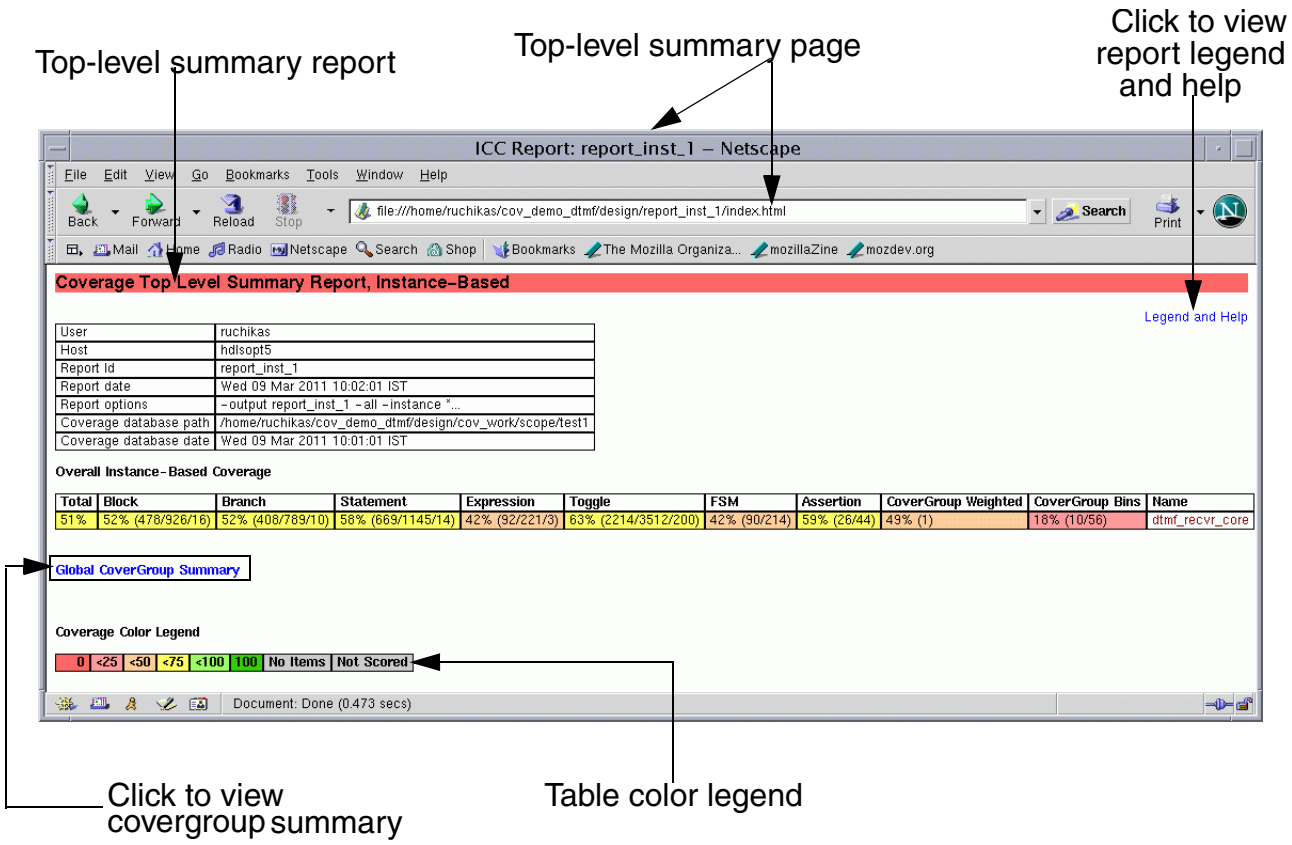
To generate HTML reports for all coverage types and items for all of the instances in the design, use:

```
report_html -output report_inst_1 -all -instance *...
```

11.3.5.2 Viewing HTML Reports

The reports can be viewed in any standard web browser, such as Internet Explorer, Firefox, and Mozilla. To view HTML reports, open `report_inst_1/index.html` in a standard web browser.

The top-level summary page `index.html` is displayed in the browser as:



The top-level summary page:

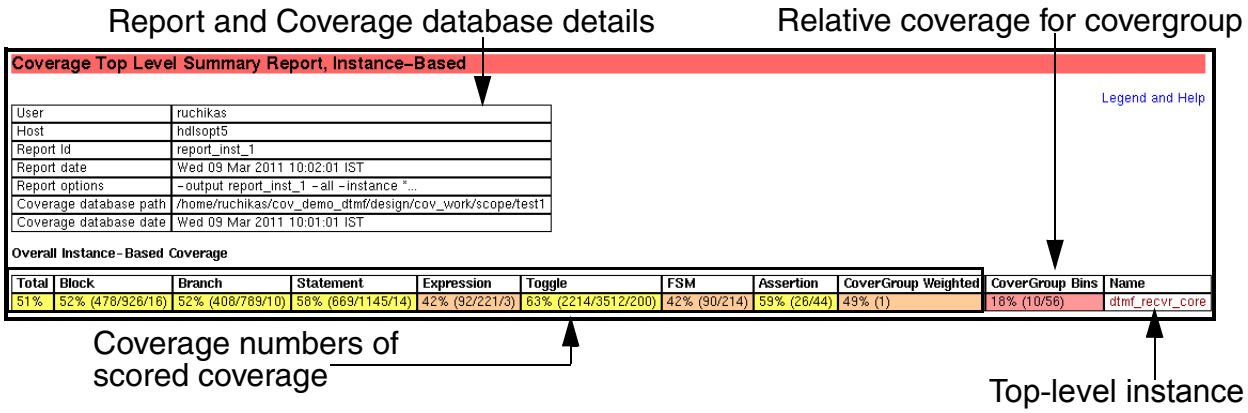
- Displays the Coverage Top-Level Summary Report and
- Provides a link to access the Global Covergroup Summary

Note: The HTML report formatting and color schemes are controlled through a cascading style sheet file `icc.css`, which can be customized based on your requirements. See Customizing the icc.css file for more details.

Coverage Top-Level Summary Report

The *Coverage Top-level Summary Report* section displays information, such as who generated the report, when the report was generated, the options used to generate the report, coverage database location, and time when the coverage database was generated. The report also includes coverage information about scored coverages.

The following figure displays the *Coverage Top-level Summary Report* section.

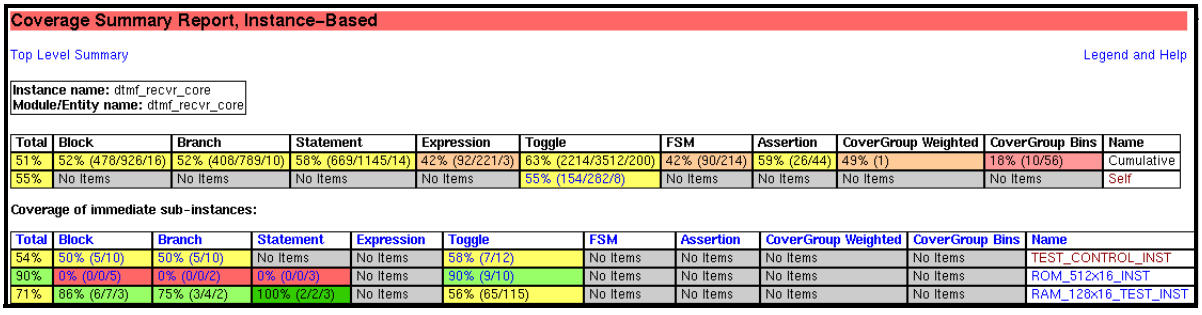


The report, by default, includes absolute coverage numbers that are shown as:
-- (Covered items / Total items / Marked items) for block, expression, and toggle coverage
-- (Covered items/Total items) for FSM and assertion coverage
-- (Cumulative weight of all covergroups) for covergroup weighted coverage in the *Covergroup Weighted* column
-- (Covered bins/Total bins) for covergroup bins coverage in the *Covergroup Bins* column

The printing of absolute coverage numbers can be disabled using the -noabsolute option. In the top-level summary report, the FSM column includes coverage numbers for both state and transition coverage.

In the case of an instance-based HTML report, the top-level instance name is displayed in the *Name* column, which you can use to navigate through the design hierarchy. To navigate through the design hierarchy:

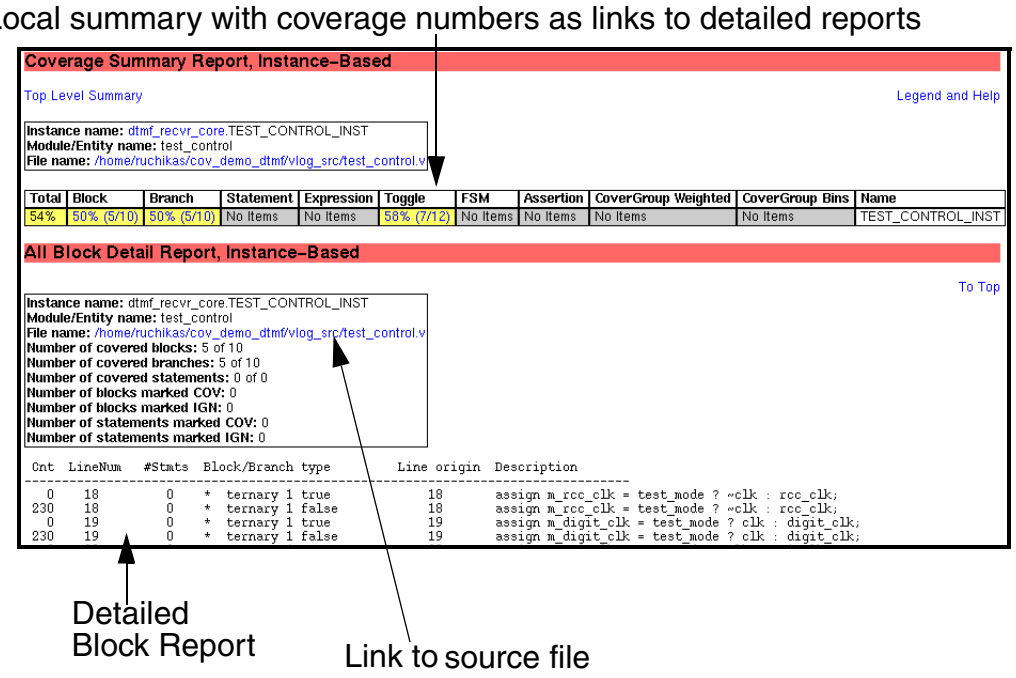
1. Click the `dtmf_recvr_core` link.



The page displays the coverage for instances under `dtmf_recvr_core`. It also includes a link named `Self`, using which you can view the detailed report for the `dtmf_recvr_core` instance. You can view the detailed report of any of the instances within `dtmf_recvr_core` by clicking its hyperlink in the *name* column.

Note: You can also sort coverage data in different tables by clicking a column header.

2. To view detailed coverage for `TEST_CONTROL_INST`, click its hyperlink in the *Name* column.



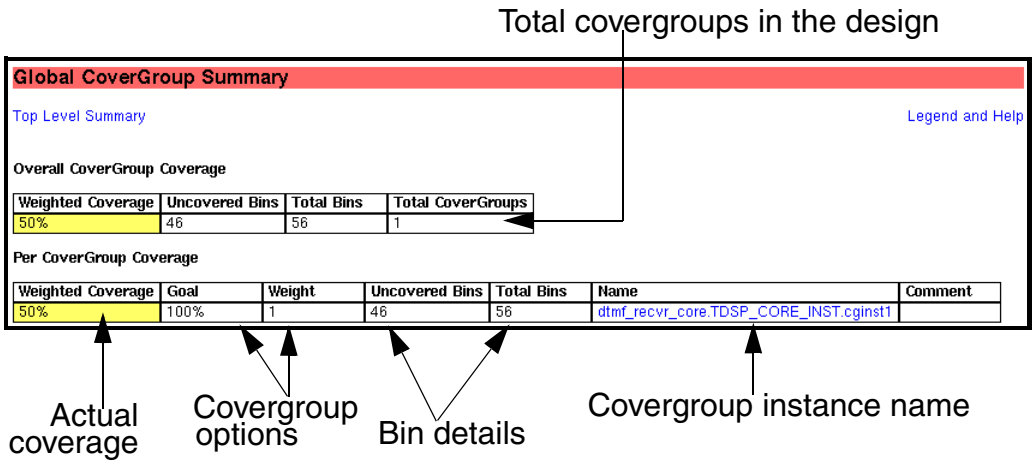
The above report displays the block coverage report of `TEST_CONTROL_INST`. You can view the detailed report of other metrics types of this instance by clicking the hyperlinks in the report header. Similarly, you can navigate through other instances of the design.

Important

You can move the HTML reports to another location for analysis. When you move HTML files, the links to source files break if the absolute source path is not available on the system where the HTML report is viewed. If the absolute source path is available, links to source files work fine. All other navigation links work fine even if the absolute source path is not available.

Global Covergroup Summary

The following figure displays the *Global Covergroup Summary* section that is displayed when you click the *Global Covergroup Summary* link on the `index.html` page.



The *Global Covergroup Summary* section provides a global view of all covergoups regardless of location. This section has two tables:

- Overall Covergroup Coverage, which summarizes all covergroup coverage in the design.
- Per Covergroup Coverage, which lists coverage for each covergroup type (if module-based report is generated) or covergroup instance (if instance-based report is generated).

Overall Covergroup Coverage

The *Overall Covergroup Coverage* table displays the overall coverage of all of the covergroups, uncovered bins in all of the covergroups, total bins in all of the covergroups, and total covergroups in the design. The overall coverage is calculated as:

```
Coverage of all the covergroups / Total covergroups in the design
```

Per Covergroup Coverage

The *Per Covergroup Coverage* table displays information about actual coverage, covergroup options, and bin details for different covergroups in the design.

If an instance-based report is generated, the Weight and Goal columns display the weight and goal set using the `option` keyword.

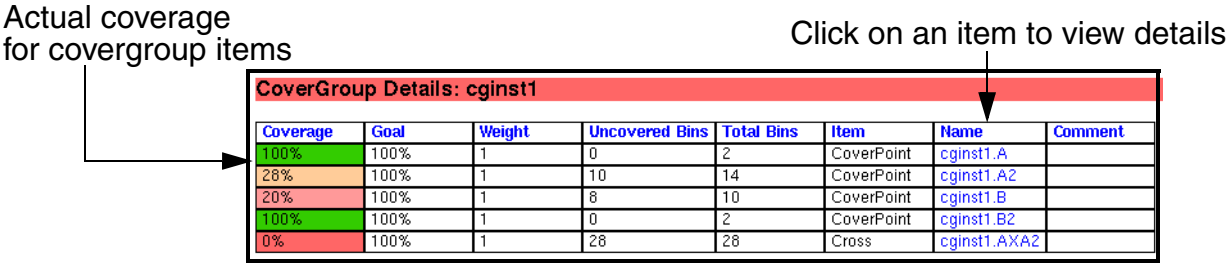
If a module-based report is generated, the Weight and Goal columns display the weight and goal set using the `type_option` keyword.

Note: The covergroup options control the behavior of the covergroup, coverpoint, and cross. For details on covergroup options, see Specifying Coverage Options. For information on how coverage is calculated, see the Detailed Covergroup report section.

The *Per Covergroup Coverage* table also allows you to navigate through the covergroup items, such as coverpoints and crosses within the covergroup type/ covergroup instance.

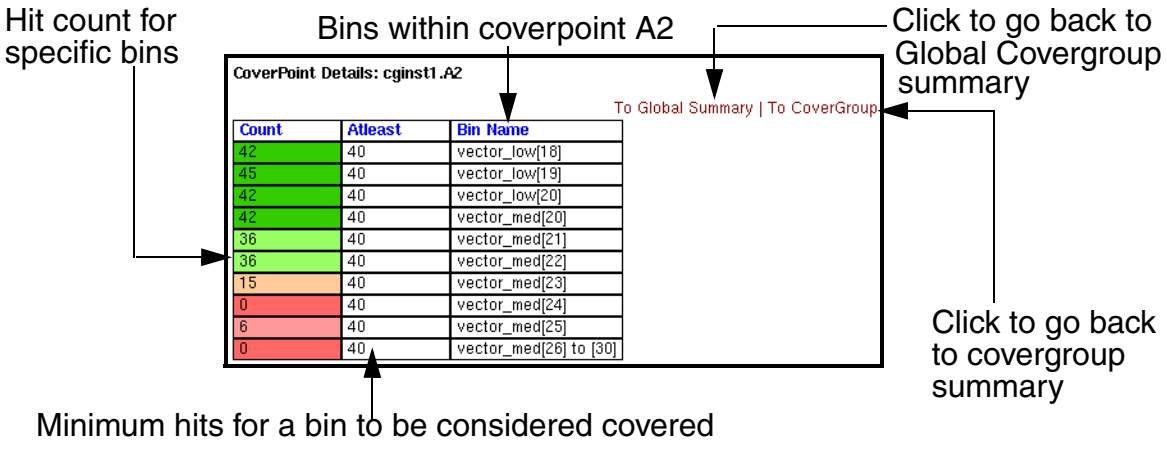
To navigate through the coverage items of covergroup instance `cginst1`:

1. Click the `dtmf_recvr_core.TDSP_CORE_INST.cginst1` link. The following figure displays the details of covergroup instance `cginst1`.



   The page displays the coverage information about coverpoints and crosses within the covergroup instance `cginst1`. The *Coverage* column shows the actual coverage for each coverage item. See the Detailed Covergroup report section for details on coverage calculation for a coverage item. For details on setting covergroup options (`goal` and `weight`), see Specifying Coverage Options.

2. To display the bins within a coverpoint, click its hyperlink in the *Name* column. Click the `cginst1.A2` coverage point.



Similarly, you can navigate through other covergroup items.

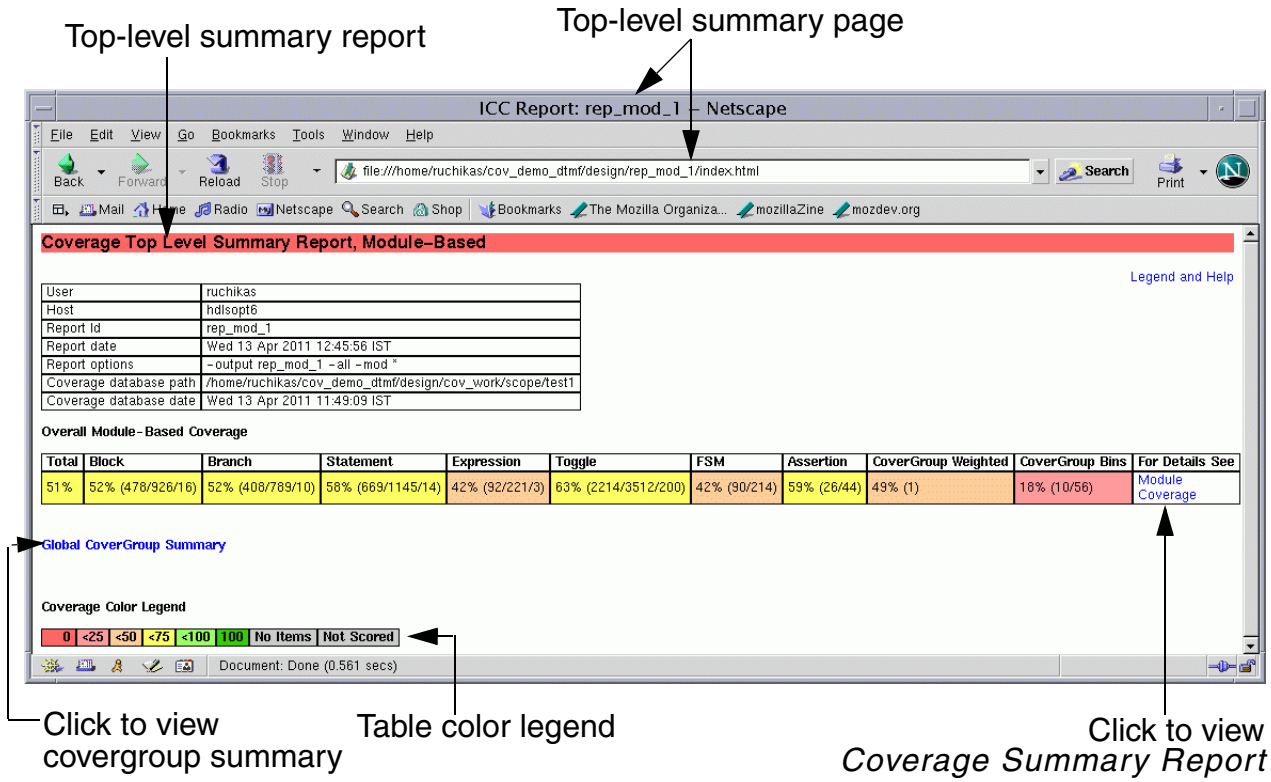
Coverage Top-Level Summary Report -- Module-Based

Consider an example of a module-based HTML report generated using:

```
report_html -output rep_mod_1 -mod *
```

To view HTML reports generated with the above command, open `rep_mod_1/index.html` in a standard web browser.

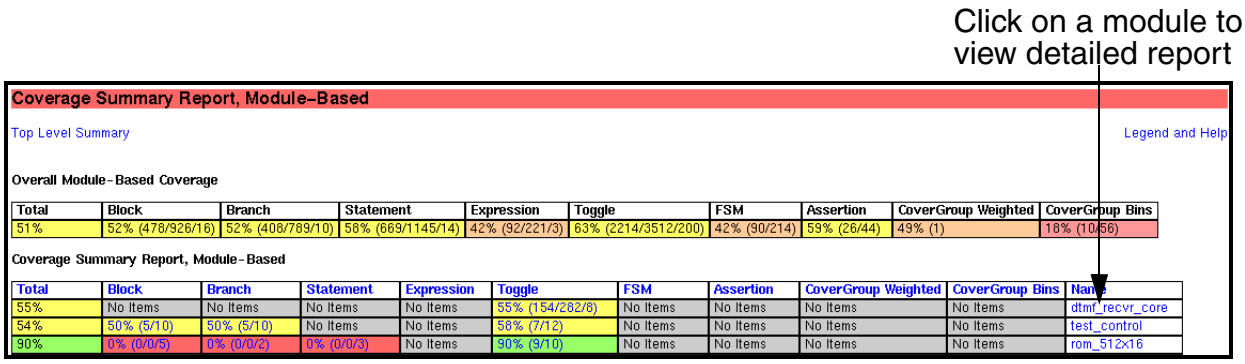
The following figure displays the top-level summary page `index.html` in the browser.



Similar to instance-based report, it displays information such as who generated the report, when the report was generated, the options used to generate the report, coverage database location, and time when the coverage database was generated. For more details on *Global Covergroup Summary*, see Global Covergroup Summary on page 314.

To view the module-based *Coverage Summary Report*, click `Module Coverage`.

In the case of a module-based HTML report, the *Coverage Summary Report* lists all of the modules in the *Name* column, as shown below:



To view the detailed report for a module, click its hyperlink in the *Name* column.

11.3.5.3 Customizing the icc.css file

The `icc.css` file contains specifications for table, text, font, and color styles used by the browser to display HTML reports. By default, when you generate an HTML report, the `icc.css` file is copied from the installation location (`<installation_dir>/tools/iccr/files/icc.css`) to the HTML report directory. The browser refers to the `icc.css` file in the HTML report directory to display HTML reports. The browser by default uses the settings (fonts/color styles) specified in the `icc.css` file. However, you can modify the settings as per your requirements and create a new version of the `icc.css` file.

Important

You cannot modify the `icc.css` file available at the installation location. Make a copy of it at the required location, and then modify it.

To enable the browser to use the customized `<cssfile>` for subsequent HTML reports, use the following command:

```
set_report_html_css_path <cssfile>
```

where `<cssfile>` specifies the new customized cascading style sheet file. The `<cssfile>` argument can use both absolute and relative paths.

The `set_report_html_css_path` command sets the cascading style sheet file to `<cssfile>` instead of the default `<installation_dir>/tools/iccr/files/icc.css`.

After the `<cssfile>` is set, the subsequent `report_html` commands copy the new `<cssfile>` to the HTML report directory, and it will be used when viewing that report.

11.4 Analyzing Coverage in GUI

The ICC Analyzer is a graphical user interface that enables you to:

- Observe the percentage of coverage that a simulation run achieved for each coverage type.
- Mark blocks, and expressions, and re-analyze for ignored, covered, and uncovered items.

Before starting the GUI from the command line, you can:

- Load tests using the `load_test` command.
- Load marks using the `load_icf` command.
- Filter coverage in modules and instances using the `de`(`select_coverage`) command.

Next, launch the GUI from the `iccr` prompt using the `view_graphics` command.

view_graphics

Opens the graphical user interface for analyzing coverage results and waits until the GUI window is closed, before it returns to read the next command at the `iccr` command prompt. This command is the equivalent of `iccr -gui`.

Note: The GUI window can be closed using the *File -> Exit* menu or *File -> Close* menu. See *ICC Analysis User Guide* for more details.

If coverage data is already loaded before the `view_graphics` command is run, it opens the GUI with coverage tests loaded.

When GUI is opened from the batch mode, marks and selections applied using the `mark`, `undo_mark`, and `(de)select_coverage` commands are automatically transferred to the GUI.

Any marks applied in GUI do not automatically transfer back to interactive batch mode when the GUI is exited. Information must be saved in GUI through *File -> Save ICF*, and reloaded in batch mode using the `load_icf` command.

11.5 Ranking and Ordering Tests

During coverage analysis, you can rank the loaded tests and identify the effectiveness of each test. The `test_order` command ranks the loaded tests according to coverage contribution, and displays them in decreasing order of their effectiveness. Tests that do not contribute anything to coverage are listed as redundant tests. To rank tests, use:

```
test_order [-module <module> | -instance <instance>]
           [-effort high] |[-effort medium [-inmemory]] | [-effort low [-inmemory]]
           [-b [<wt>]] [-e [<wt>]] [-t [<wt>]] [-s [<wt>]] [-a [<wt>]]
           [-f [<wt>]] [-d [<wt>]] [-cpu [<wt>]]
           [-max [<val>]] [-target <cumulative_coverage_goal>]
           [-output <test_list_name>]
```

where

- `-module <module>` ranks tests based on all instances of specified module/entity. Multiple modules cannot be specified.
- `-instance <instance>` ranks tests based on specified instance and all of its descendants. Multiple instances cannot be specified.
- `-effort` specifies the effort level to rank tests. Effort level can be:
  - `high` — an optimized algorithm to accurately provide the least number of tests to achieve coverage goal.
  - `medium` — a less accurate algorithm provided for backward compatibility.
  - `low` — a less accurate algorithm provided for backward compatibility.

  Note: By default, effort level `high` is applied to rank tests.
- `-inmemory` option is for backward compatibility with effort levels `medium` and `low`. It reduces the time required to rank tests because it first loads all of the tests in memory, and then performs test ordering.
- `<wt>` increases or decreases the weight (wt) of a given coverage type or simulation CPU time. The default weight, if not specified, is 1. A value of 1 indicates that all available coverage types and simulation CPU time are considered while ranking and have the same weight. If the weight is 0, then coverage for that item is not considered while ranking tests. The higher the weight, the more impact it will have on the result. In ranking tests, `<wt>` can be specified with:

  `-b` Block coverage
  `-e` Expression coverage
  `-t` Toggle coverage
  `-s` State coverage

`-a` Transition coverage
`-f` Control-oriented functional coverage
`-d` Data-oriented functional coverage
`-cpu` Simulation CPU time

Note: Simulation CPU time affects ranking inversely. The longer the simulation time, the lower the rank.

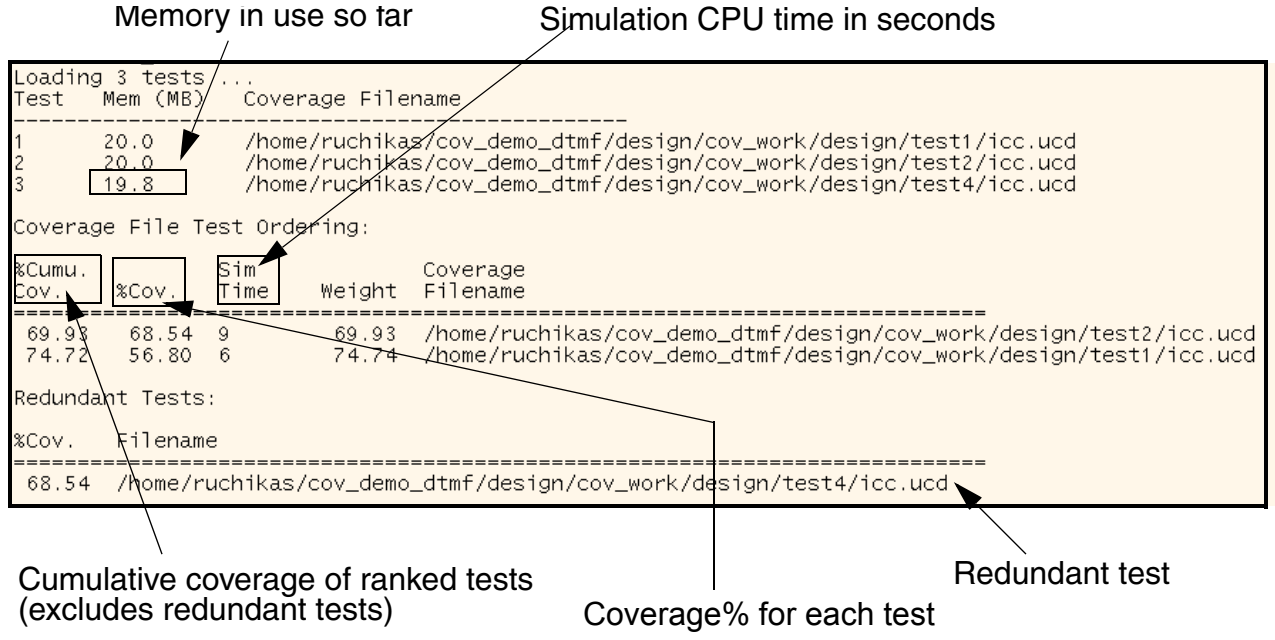
- `-max <val>` specifies the number of tests on which test ranking is performed. If `<val>` is not specified, test ranking is performed on all loaded tests.
- `-target <cumulative_coverage_goal>` stops the test ordering once the targeted cumulative coverage goal is achieved.
- `-output <test_list_name>` generates a list of nonredundant tests (in the decreasing order of their effectiveness) and writes that list to `<test_list_name>`. By default, the output file (`<test_list_name>`) is created in the current working directory. With this option, you do not need to wait for the test order operation to complete to check the results. You can open the output file, and view the test order results in progression.

  Note: If the `-max` option is used with the `-output` option, then the number of tests included in the output file will be less than or equal to the value of `<val>` argument specified with the `-max` option.

To rank all loaded tests, use:

```
test_order
```

Consider the output of a `test_order` command.

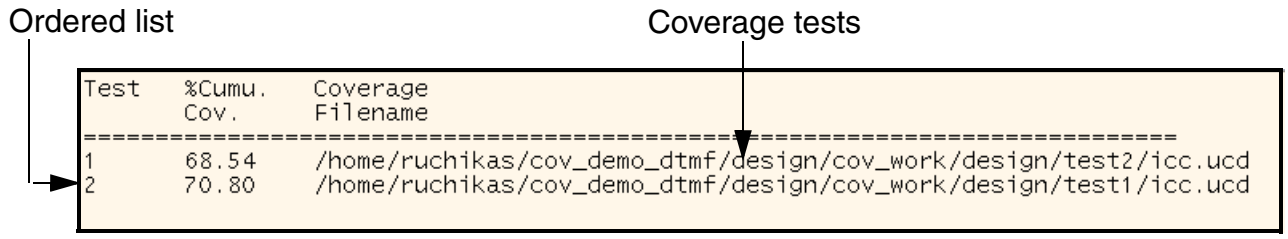


The above results indicate that `test4` is redundant and `test2` contributes that maximum to overall coverage results.

To output the list of nonredundant tests to a file named `mylist`, use:

```
test_order -output mylist
```

A sample output file (`mylist`) is shown below:



To rank all loaded tests based on module `m1`, use:

```
test_order -module m1
```

To emphasize simulation time and ignore expression coverage while ranking tests, use:

```
test_order -cpu 2 -e 0
```

To rank tests to achieve a cumulative coverage goal of 90%, use:

```
test_order -target 90
```

To ignore block and expression coverage while ranking tests, use:

```
test_order -b 0 -e 0
```

To emphasize block coverage while ranking tests, use:

```
test_order -b 2
```

11.6 Operating on Test Databases

To compare tests, you can perform operations like difference, intersection, and union, as discussed in this section. These commands work only on tests that use the same model file and therefore can be loaded together.

- Identifying Recent Test Enhancements (difference)
- Identifying Duplicates across Tests (intersection)
- Generating Cumulative Numbers from Tests (union)

11.6.1 Identifying Recent Test Enhancements (difference)

The `difference` command helps you validate recent test enhancements. It compares coverage numbers from all of the loaded tests and reports the values that are unique to the first loaded test. To perform a difference operation, use:

```
difference <output_test>
```

where `<output_test>` is the test where the result of difference operation is saved. The resultant test includes only those elements that got covered in the first loaded test. The count for the coverage items in the resultant test is the difference of counts from loaded tests.

If the difference operation is run on more than two coverage tests, all coverage tests are compared with the first one, in sequential order.

11.6.2 Identifying Duplicates across Tests (intersection)

The `intersection` command compares coverage numbers from all of the loaded tests and identifies the coverage items covered in all tests. To perform an intersection operation, use:

```
intersection <output_test>
```

where `<output_test>` is the test where the results of an intersection operation are saved. The count for the coverage items in the resultant test is the least coverage count in all of the loaded tests.

If the intersection operation is run on more than two coverage tests, all coverage tests are compared with the first one, in sequential order.

11.6.3 Generating Cumulative Numbers from Tests (union)

The `union` command generates aggregated coverage numbers from all the loaded tests. To perform a union operation, use:

```
union <output_test>
```

where `<output_test>` is the test where the results of a union operation are saved.

When you perform a union operation, the count from all of the loaded tests are added and the results are stored in the resultant test.

For example, to union results from `test1` and `test2` to a test named `all`, use:

```
load_test test1 test2
union all
```

When performing a union operation, if loading multiple tests fails, then use the `merge` command (It works on the unchanged pieces of a hierarchy).

Important

After a union, intersection, or difference operation, ICC automatically loads the resultant test and unloads all other tests. As a result, any operation (marking, selection, deselection, and so on) after the union, intersection, or difference operation applies to the resultant test. To work with individual tests, `reset_coverage` and load the required tests.

Database operations and markings in resultant database

The coverage results from runs might include objects marked as IGN, COV, or CON. The database operation commands (`union`, `intersection`, and `difference`) apply the following priority to mark data in the resultant database.

```
COV > COVERED > IGN > CON > UNCOVERED
```

11.7 Other ICCR Commands and Functionality

The following are additional ICCR commands:

alias	unalias	define
version	source	sh
history	set_history_size	quit / exit
map_source_path	deselect_coverage	select_coverage
set_instance_context	end_instance_context	list_context
set_workdir	set_design	set_logfile

To view a list of commands that you can execute in iccr, type `help` at the `iccr` prompt and press the `Enter` key. You can also view detailed help on specific commands by specifying the `help <command>` command.

alias

Sets an alias `<alias_name>` for command name `<command_name>`.

```
alias <alias_name> <command_name>
```

To set the alias `li` for `list_instances`, use:

```
alias li list_instances
```

unalias

Removes alias `<alias_name>` set earlier using the `alias` command.

```
unalias <alias_name>
```

define

Defines a variable and assigns it a value.

```
define [<variable> <value>]
```

The `<variable>` must contain only valid character set for TCL variables. The valid character set values include, a-zA-Z0-9_.

Without any arguments, the command lists all of the variables defined.

The following commands demonstrate the use of define command.

Command	Description
`define imode "-instance"`	Defines a variable `imode` and assigns `-instance` to it.
`define ilist "*..."`	Defines a variable `ilist` and assigns `*...` to it
`list_coverage $imode $ilist`	Uses both the defined variables. On execution, the command is interpreted as: `list_coverage -instance *...`

version

Displays the current ICCR version.

source

Executes commands from `<filename>` and returns to the ICCR prompt.

```
source <filename>
```

sh

Executes the specified <shell_command>.

```
sh <shell_command>
```

To get a directory listing of the current directory, use:

```
sh ls
```

history

Displays a list of previously executed commands.

The number of commands maintained in the history is set using the set_history_size command.

set_history_size

Sets the number of commands maintained in the history list.

```
set_history_size <size>
```

where <size> is a positive integer that defines the number of commands stored in the history list. The default history size is 20.

quit / exit

Either of these commands ends the execution of the iccr program.

map_source_path

ICC by default, looks for the design source files in the directory from which ICCR is launched. If the design instrumentation location is different from the one from which ICCR is launched, map the source paths so that ICCR can locate the design source files. This command is useful if files are moved, or are referenced differently on different systems.

```
map_source_path [[<from>] <to>] | [-d <from> | #]
```

where

- <from> indicates the path of the source files at the time of coverage data generation. The <from> path must start at the root level.
- <to> indicates the new location of source files. The <to> path must start at the root level.

- `-d` allows deletion of an already mapped path.
- `#` is the numeric ID assigned to each mapped path.

Note: Without any parameters, lists currently mapped paths along with the numeric identifier assigned to that path.

To map the directory `/home1` to the directory `/mnt/emily/home1`, use:

```
map_source_path /home1 /mnt/emily/home1
```

To map all paths to `/home2/sourcefiles`, use:

```
map_source_path /home2/sourcefiles
```

To list all mapped paths, use:

```
map_source_path
```

For example, the following paths are mapped:

```
<1> /home1 /mnt/sally/home1
<2> /home2 /mnt/sally/home2
<3> /home3 /mnt/emily/home3
```

To delete the path with numeric ID 2, use:

```
map_source_path -d 2
```

OR

```
map_source_path -d /home2
```

deselect_coverage

Deselects the given coverage types for the specified instance or module/design entity in currently loaded tests.

```
deselect_coverage [-betsa] [-instance | -module | -entity] <list>
```

See Summary Report for details on syntax description of other options.

To deselect block coverage from module `data_bus_mach`, use:

```
deselect_coverage -b -module data_bus_mach
```

This deselection will affect subsequent reports.

Important

If a module is deselected, then all of its instances are deselected. However, if an instance is deselected, then the deselection information is not propagated to the module. As a result, if an instance is deselected, then the deselections are reflected only in the instance-based reports and not in the module-based reports.

Note: You can apply selection/deselection on generated instances. Any use of `[]` or `()` to define `for-generate` paths must escape each such character with a `\`. For example,

```
deselect_coverage -instance -be test.genblk1\[0\].ctlfor
```

select_coverage

Selects the given coverage types for the specified instance or module/design entity (previously deselected using the deselect_coverage command) in the currently loaded tests.

```
select_coverage [-betsa] [-instance | -module | -entity] <list>
```

Note: You can apply selection/deselection on generated instances. Any use of `[]` or `()` to define `for-generate` paths must escape each such character with a `\`. For example,

```
select_coverage -instance -be test.genblk1\[0\].ctlfor
```

See Summary Report for details on syntax description of other options.

Note: You cannot select coverage for modules that have not been enabled during instrumentation. The `select_coverage` command enables only the items deselected using the `deselect_coverage` command.

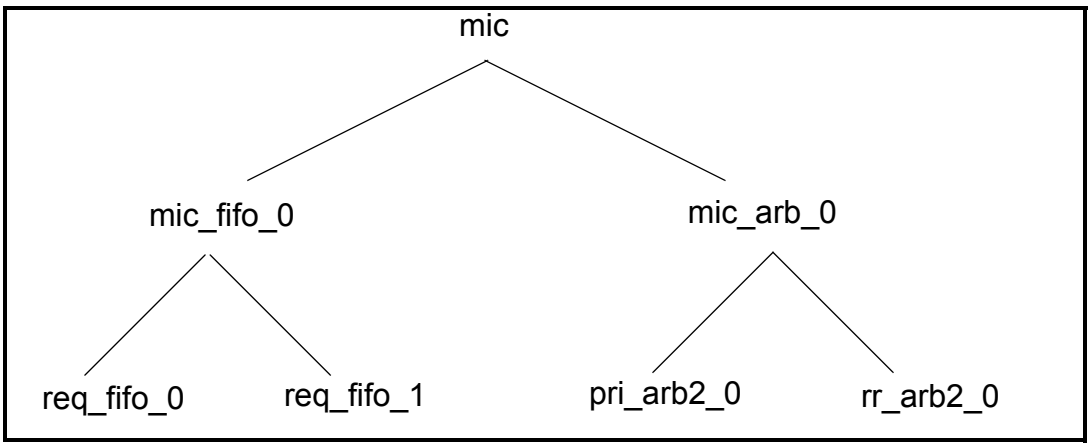
set_instance_context

Sets the current instance context. All commands in which an instance may be specified will use this context as a starting point.

```
set_instance_context [-all] <instance>
```

- `<instance>` specifies the name of the instance to be set. Instance names must be hierarchical paths that begin with the module name of the design top rather than the module name of the testbench.
- `-all` causes the instance context to be reset, and to be set to `<instance>`.

Consider the following hierarchy:



To change the current instance context to `mic.mic_arb_0`, use:

```
set_instance_context mic.mic_arb_0
```

end_instance_context

Returns to the instance context set by a previous `set_instance_context` command.

```
end_instance_context [-all]
```

The `-all` option sets the instance context to the top-level instance.

list_context

Lists the current instance context.

set_workdir

By default, when you launch `iccr`, the working directory is relative to the directory in which `iccr` was started. To set an alternate working directory, use:

```
set_workdir <workdir>
```

For subsequent ICCR commands, `<workdir>` is considered the working directory.

set_design

Sets the design directory for subsequent ICCR commands.

```
set_design <design>
```

Tests are loaded or merged from the set <design>. In the case of multiple set_design commands, the most recent command is used for loading or merging coverage data. It cannot contain wildcards.

set_logfile

Sets an alternate log file <filename> instead of the default iccr.log.

```
set_logfile <logfile>
```

The log file is opened and updated the first time any command has a correct output. If the log file is not set by this time, the default log file iccr.log is used. If you attempt to set the log file after the log file is opened, an error is generated.

12

Manual Marking of Coverage

This chapter discusses manual marking of uncovered items that should not be considered in the overall analysis of coverage. Marks can be applied in ICCR's command-line mode and by adding pragmas to the HDL code, both of which are described here. Marks can also be applied in the ICCR GUI. See the *ICC Analysis User Guide* for more details.

Note: Starting this release, another tool Incisive Metrics Center (IMC) is available for applying marks, merging data, and analyzing coverage data. Soon IMC will replace ICCR. For more details on IMC, see the *Incisive Metrics Center User Guide*.

12.1 ICCR Mark Command

During coverage analysis, you can manually mark coverage items as ignore (IGN) or covered (COV) so that these items can be excluded from coverage calculations. ICCR marking applies only to block and expression coverage. Marking of toggle signals is not allowed. However, you can exclude specific signals from toggle recording using the set_toggle_excludefile command. Marking items in ICCR's command-line mode includes:

- Generating Detailed Reports with Indices
- Marking Desired Items using Indices
- Saving and Re-Using Marks
- Reporting on Marks to Confirm Validity

12.1.1 Generating Detailed Reports with Indices

By default, *iccr* reports do not number the report items, as shown below.

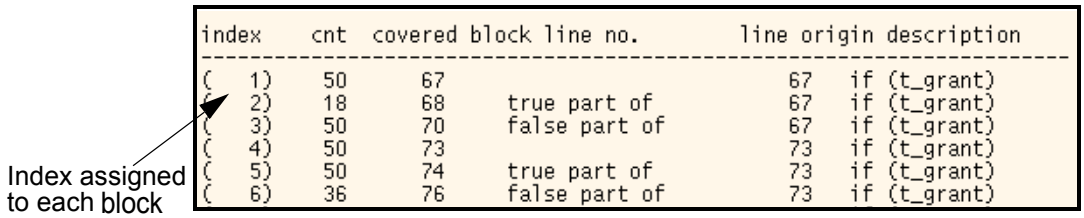
```
cnt  covered block line no.        line origin description
-------------------------------------------------------------------
 50      67                          67   if (t_grant)
 18      68     true part of         67   if (t_grant)
 50      70     false part of        67   if (t_grant)
 50      73                          73   if (t_grant)
 50      74     true part of         73   if (t_grant)
 36      76     false part of        73   if (t_grant)
```

To mark coverage items, you first number the report items using numeric identifiers called “indices” as:

```
indices [-on | -off]
```

If no argument is specified, `-on` is assumed.

After indexing is enabled, sequential numbers are assigned to each block or expression in the detail report, as shown below.




```
index    cnt  covered block line no.        line origin description
-------------------------------------------------------------------
(   1)    50      67                          67   if (t_grant)
(   2)    18      68     true part of         67   if (t_grant)
(   3)    50      70     false part of        67   if (t_grant)
(   4)    50      73                          73   if (t_grant)
(   5)    50      74     true part of         73   if (t_grant)
(   6)    36      76     false part of        73   if (t_grant)
```

12.1.2 Marking Desired Items using Indices

To mark a particular block or expression as covered or ignored, use:

```
mark <-ignore | -covered> <-instance | -module | -entity>
     { -b <name> <indices> | -e <name> <indices> <indices> }
<indices> ::= { <n>[,] | <n1>-<n2>[,] | * }
```

where,

- `-ignore` marks a particular coverage item as IGN. The IGN items are removed from both uncovered and total counts.
- `-covered` marks a particular coverage item as COV. The COV items are considered to calculate coverage counts.
- `-module | -instance | -entity` specifies if marking should be applied to all instances of a module or to a specific instance.

- `-b` indicates marking of blocks.
- `-e` indicates marking of expressions.
- `name` specifies the instance or a module name to which marking applies. Instance names must be hierarchical paths that begin with the module name of the design top.
- `<indices>` are integer values to identify report items to be marked. It can be:
  - `<n>` to indicate single item
  - `<n1>-<n2>` to specify a range of items
  - `*` to indicate all items

Note: Multiple expressions that are part of different rows cannot be marked using commas. For example, the following code to mark expression 1of row 2 and expression 2 of row 1 will result in an error:

```
mark -ign -module -e counter 1 2,2 1  //leads to an error
```

The correct usage to mark multiple rows is:

```
mark -ign -module -e counter 1,2
mark -ign -module -e counter 2,1
```

Note: Markings applied using `mark` and `unmark` commands apply only to that ICCR session. To save the effort of manually marking in each ICCR run, you can save these marks to a file and later, in another iccr session, load the saved marks. The marks will disappear once you quit ICCR. See Saving and Re-Using Marks for more details.

12.1.2.1 Marking Block Coverage Items

Consider the following block coverage output of an instance named `dtmf_recvr_core.TDSP_CORE_INST.ALU_32_INST`.

```
        Number of covered blocks: 6 of 9
        Number of blocks turned off by user: 0
        Number of blocks marked COV: 0
        Number of blocks marked IGN: 0

index    cnt  covered block line no.        line origin description
-------------------------------------------------------------------------
(   1)    50      78                              78   begin : alu_function
(   2)    50      80     a case item of           79   case (cmd)
(   3)    50      83     a case item of           79   case (cmd)
(   4)     0      86     a case item of           79   case (cmd)
(   5)     0      89     a case item of           79   case (cmd)
(   6)     0      92     a case item of           79   case (cmd)
(   7)    50      95     a case item of           79   case (cmd)
```

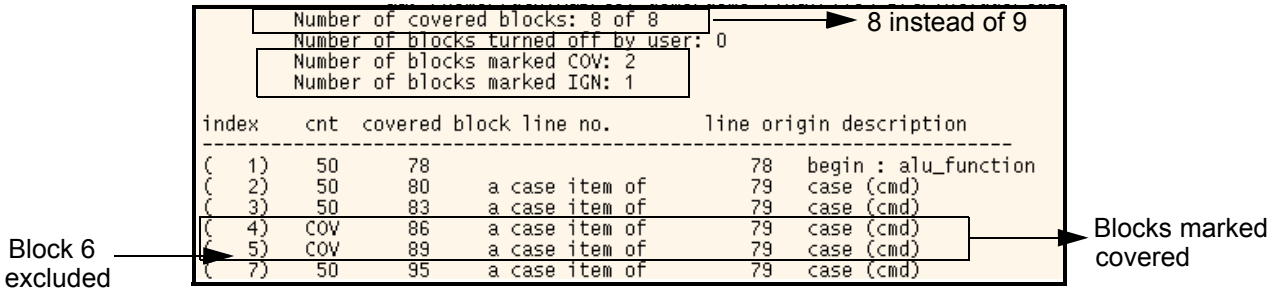
To mark blocks 4 and 5 as covered, use:

```
mark -cov -instance -b dtmf_recvr_core.TDSP_CORE_INST.ALU_32_INST 4-5
```

To mark block 6 as ignore, use:

```
mark -ign -instance -b dtmf_recvr_core.TDSP_CORE_INST.ALU_32_INST 6
```

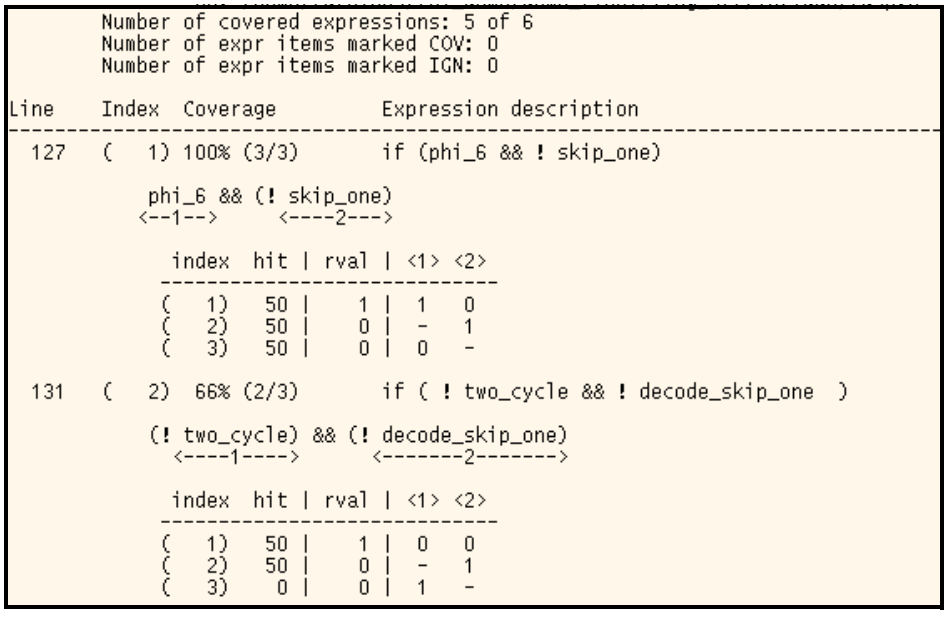
After executing the above `mark` commands, the report displays the following coverage counts.



12.1.2.2 Marking Expression Coverage Items

For marking expressions, you specify indices as {A B} where (A) refers to the index of expression which will be marked, and (B) refers to the exact row of the specified expression's coverage table, as it appears in the detail report.

Consider the following expression coverage output of an instance named `dtmf_recvr_core.TDSP_CORE_INST.DECODE_INST`.



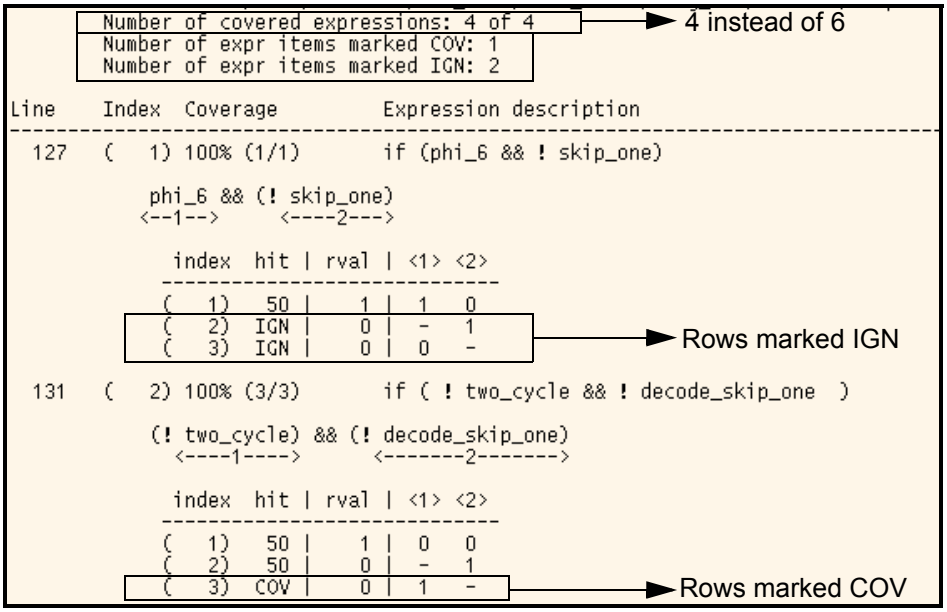
To mark row 3 of expression 2 as covered, use:

```
mark -cov -instance -e dtmf_recvr_core.TDSP_CORE_INST.DECODE_INST 2 3
```

To mark rows 2 and 3 of expression 1 as ignored, use:

```
mark -ignore -instance -e dtmf_recvr_core.TDSP_CORE_INST.DECODE_INST 1 2-3
```

After executing the above `mark` commands, the report displays the following coverage counts.



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

12.1.2.3 Unmarking Coverage Items

To unmark marked items, use:

```
undo_mark [-all] | [<-ignore | -covered> <-instance | -module | -entity>
     { -b <name> <indices> |  -e <name> <indices> <indices> }
<indices> ::= { <n>[,] | <n1>-<n2>[,] | * }]
```

This command undoes the effect of a previous `mark` command. If the `-all` option is used, *all* marking is undone.

12.1.3 Saving and Re-Using Marks

While analyzing coverage data, you first merge all tests for a set of regression runs and then analyze the resulting coverage. You then may mark coverage items as Covered (COV) or Ignore (IGN). To save the effort of manually marking in each ICCR run, you can save these

marks to an ICF (ICCR Configuration File) using the save_icf command and later, in another iccr session, load the saved marks (using the load_icf command) and apply them to the new set of regressions.

Save Marks

To save marks for reuse with a new set of tests, use:

```
save_icf <file>
```

This command creates a <file> in the current directory and saves mark, undo_marks, and (de)select_coverage commands run in an iccr session in this file.

Consider the following set of commands that you run in an iccr session.

```
load_test test1
indices
report_detail -module -covered -b fsm_8state
mark -ignore -module -b fsm_8state 3
mark -covered -module -b fsm_8state 7-9
save_icf my_marks
```

The save_icf command creates file my_marks in the current directory and saves information about marks and selection commands in this file. The file also includes a header, which includes the information about when was the file generated, who generated the file, the host information, and the test using which the file was generated. Starting IES10.2-s13 release, the header also includes information about the ICCR release version using which the file was saved. By default, this version is checked at the time of loading the marks file, which can be ignored using the -no_version_check option of the load_icf command.

Note: If the file exists at the time of saving, then the old file is saved as <file>.bak.

Load marks

To reuse the saved marks with a new set of coverage data on the same design configuration, use:

```
load_icf [-no_version_check] <file>
```

where

- <file> is the one that was saved earlier using the save_icf command. The load_icf command executes the commands saved in the specified file.
- -no_version_check is used to ignore the checking of version number (available in the header of the <file>) at the time of loading the <file>.

  Note: By default, the version of the <file> is checked at the time of loading the ICF to

check if the ICF is compatible with the current version of ICCR.

To reuse the earlier saved marks in the next ICCR session for another test, use:

```
load_test test2
indices
load_icf my_marks
```

To reuse marks, it is important that the source code is not changed.

When using the `load_icf` command, remember that:

- The `load_icf` command allows at the most one level of nesting, as shown below:

```
# 1.icf
mark ...
undo_mark ...
load_icf 2.icf --> OK - first level of nesting
# EOF 1.icf


# 2.icf
mark ...
load_icf 3.icf --> Error - more than one level of nesting
# EOF 2.icf
```

- The `load_icf` command does not allow recursive loading, as shown below:

- ```
  # 1.icf
  mark ...
  undo_mark ...
  load_icf 1.icf --> Error - recursive loading not allowed.
  # EOF 1.icf
  ```

Loading an ICF saved with IES9.2 release

An ICF saved with IES9.2 release version is not compatible with the IES10.2 release version due to change in expression indexing. To make the ICF compatible with the current version of ICCR, use the translate_icf command.

If the ICF includes only block coverage marks, then `translate_icf` is not required and you can do any of the following:

- Add the following header string at the top of the ICF.

  ```
  # Tool: iccr 10.20-s013
  ```

- Use the `-no_version_check` option with the `load_icf` command to ignore checking of version number at the time of loading the file.

Note: If the ICF includes expression coverage marks, then the use of `-no_version_check` will ignore checking the version. However, the expression marks might not be applied
  ```

correctly. You must use the `translate_icf` command to ensure that the expression marks apply correctly.

Loading an ICF saved with IES10.2 release (but prior to 10.2-s13 release)

An ICF saved prior to IES10.2-s13 release did not include the ICCR release version in the header. To load an ICF saved with IES10.2 release but with a version prior to IES10.2-s13 release, use any of the following:

- Add the following header string at the top of the ICF file and then load the file.

  ```
  # Tool: iccr 10.20-s013
  ```

- Use the `-no_version_check` option with the `load_icf` command to ignore checking of version number at the time of loading the file.

Note: An ICF saved with IES10.2-s13 or a later version works fine and does not require any conversion or version checking at the time of loading.

Loading a Manually Written ICF

A marks file can also be written manually instead of being saved using the `save_icf` command. In such cases:

- To load an ICF written in IES9.2 release, see Loading an ICF saved with IES9.2 release
- To load an ICF written in IES10.2 release, see Loading an ICF saved with IES10.2 release (but prior to 10.2-s13 release)

Translate ICF

An ICF saved with IES9.2 release version is not compatible with the IES10.2 release version. This is because there is a change in the expression coverage indices in the IES10.2 release version.

To make the ICF saved with IES9.2 version compatible with IES10.2 version, use:

```
translate_icf <9.2_input_filename> <10.2_version_filename>
```

where

- `<9.2_input_filename>` is the name of the file that was saved with IES9.2 version.
- `<10.2_version_filename>` is the name of the file that will be generated after translation.

Note: Coverage data must be loaded before executing the `translate_icf` command.

For example, to make `expr92.icf` (saved with IES9.2 release) compatible with IES10.2 release, use:

```
translate_icf expr92.icf expr10_2.icf
```

With the above command, `expr92.icf` is translated and saved as `expr10_2.icf`. You can now use the `expr10_2.icf` file with the `load_icf` command to apply marks.

Important

If an ICF includes only block coverage marks, then `translate_icf` is not required and you can do any of the following:

- Add the following header string at the top of the ICF.

  ```
  # Tool: iccr 10.20-s013
  ```

- Use the `-no_version_check` option with the `load_icf` command to ignore checking of version number at the time of loading the file.

For more details, see Loading an ICF saved with IES9.2 release.

12.1.4 Reporting on Marks to Confirm Validity

To ensure that the source code has not been modified in a way that invalidates a previously-saved (and now used) `mark` command, it is recommended that before verification sign-off, all marks are reported and checked again. To report all marks, use:

```
report_detail -marked -instance *... >all.marks
```

An invalid `mark` command might apply the mark to an incorrect item. For example, a different block than what was originally intended.

12.2 Coverage Pragmas

Coverage pragmas are another way of marking coverage items manually. 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.

Unlike the ICCR `mark` command, 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.

12.2.1 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
```

Important

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.

Note: Pragmas within an inactive `ifdef are ignored.

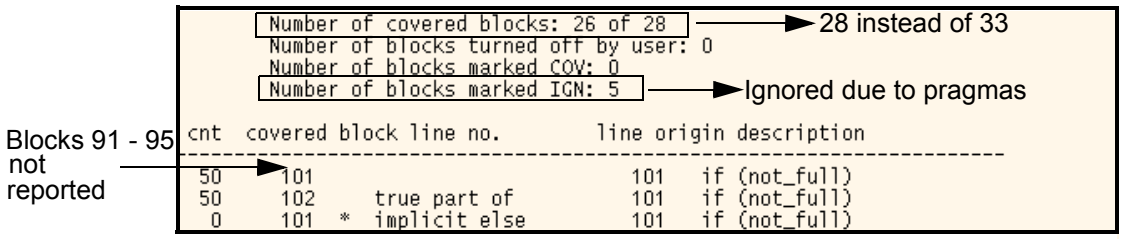
12.2.1.1 Pragmas and Coverage Calculation

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:

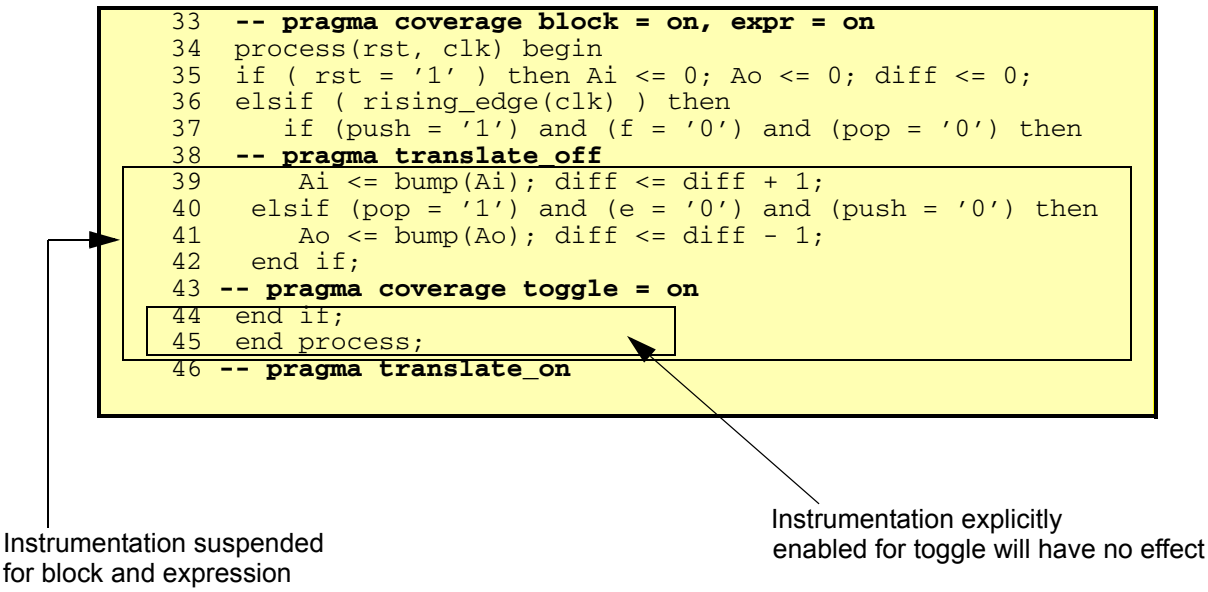


You can view items marked, using the `-marked` option of the `report_detail` command.

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

12.2.2 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.



Note: For information on how pragmas are processed in a run, see *NC-Verilog Simulator Help* guide.

12.2.3 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
```

12.2.4 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
```

12.2.5 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
```

A

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.
- 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:

A.1 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.

A.2 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

A.3 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.
- Does not score in Verilog/VHDL generate blocks or named blocks.
- Is not scored for `supply0` and `supply1` nets.

- Scores toggle objects inside system verilog interfaces. In addition, it scores System Verilog 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
```

- 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.

A.4 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.

A.5 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.

Index

U

V

W