ZeBu Server®
Functional Coverage in Emulation

Version O-2018.09-SP1, June 2019

Contents

List of Tables

List of Figures

Preface

About This Book

The **ZeBu Server - Functional Coverage for Emulation** document describes how to collect functional coverage in emulation.

Intended Audience

This manual is written for design engineers to assist them to collect functional coverage for ZeBu Server.

These engineers should have knowledge of the following Synopsys tools:

- ZeBu Server
- Verdi

Contents of This Book

This document has the following sections:

Section	Describes
Introduction to Functional Coverage	An overview of functional coverage in emulation. Also provides information on the two types of functional coverage supported, Cover Property and Covergroup.
Performing Functional Coverage in ZeBu	The steps to enable coverage during compilation and runtime. It also provides information on how to check compilation and runtime results.
Viewing Coverage Report and Analysis Using Verdi	The key Verdi features to analyze coverage data.

Related Documentation

Document Name	Description
ZeBu Server 4 Site Planning Guide	Describes planning for ZeBu Server 4 hardware installation.
ZeBu Server 3 Site Planning Guide	Describes planning for ZeBu Server 3 hardware installation.
ZeBu Server Site Administration Guide	Provides information on administration tasks for ZeBu Server 3 and ZeBu Server 4. It includes software installation.
ZeBu Server Getting Started Guide	Provides brief information on using ZeBu Server.
ZeBu Server User Guide	Provides detailed information on using ZeBu Server.
ZeBu Server Debug Guide	Provides information on tools you can use for debugging.
ZeBu Server Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Server Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu Server Functional Coverage User Guide	Describes collecting functional coverage in emulation. For VCS and Verdi, see the following: - Coverage Technology User Guide - Coverage Technology Reference Guide - Verification Planner User Guide - Verdi Coverage User Guide and Tutorial For SystemVerilog, see the following: - SystemVerilog LRM (2017)
ZeBu Server Power Estimation User Guide	Provides the power estimation flow and the tools required to estimate the power on a System on a Chip (SoC) in emulation. For SpyGlass, see the following: - SpyGlass Power Estimation and Rules Reference - SpyGlass Power Estimation Methodology Guide - SpyGlass GuideWare2018.09 - Early-Adopter User Guide
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Server LCA Features Guide	Provides a list of LCA features available with ZeBu Server.
ZeBu Server Release Notes	Provides enhancements and limitations for a specific release.

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	*<sig-name>*
Message	Active low signal name '<sig-name>' must end with _X.
Message location	`OUT` <= IN;
Example with message removed	OUT_X <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified one time or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
. . . (Horizontal ellipsis)	Other options that you can specify

1 Introduction to Functional Coverage

Coverage is a measurement of the completeness in the verification of a design. A comprehensive coverage model combines different coverage technologies and metrics to ensure all important design scenarios are exercised and validated.

This section contains the following topics:

- *Types of Coverage*
- *Defining Functional Coverage*

1.1 Types of Coverage

Coverage measures the percentage of the verification objectives met by a testbench. There are two types of coverage:

- **Design Code Based Coverage Metrics**: Extracted based on the design RTL code. This includes line coverage, condition coverage, toggle coverage and so on. For more information, see the ***Coverage Technology User Guide***.
- **Functional Coverage**: Explicitly specified by the user to check specific scenarios.

The following figure shows the types of coverage.

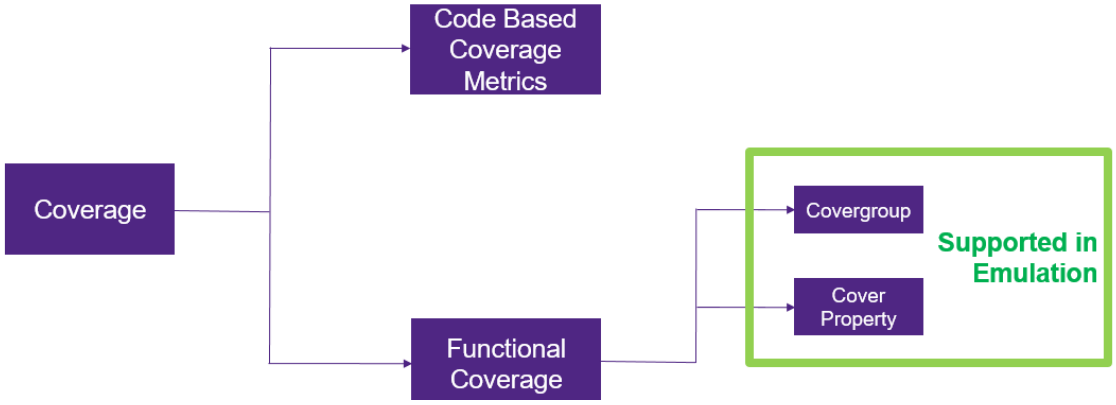


FIGURE 1. Types of coverage

1.2 Defining Functional Coverage

The following SystemVerilog constructs are used to specify functional coverage:

- *SVA cover Statement*
- *Construct: covergroup*

Note: For more information, see the ***SystemVerilog LRM***.

1.2.1 SVA cover Statement

There are two categories of SVA `cover` statements, non-temporal and temporal (concurrent). ZeBu only supports concurrent cover statements. Cover property statement and cover sequence are supported.

The following SVA features for coverage are supported for emulation:

- Cover statements
  - Standalone and procedural
- All SVA sequence and property operators
- Single and multiclocked sequences/properties
- Sequence, property constructs
- `let` constructs, as applicable to all cover statements
- Default clocking
- Default disable iff

Example

The following example illustrates a concurrent coverage statement:

```
sequence S;
   strobe & ~hold ##1 ack;
endsequence
property P;
   @(posedge clk) disable iff (rst) S;
endproperty
cover property (P);
```

You can also write the preceding code as follows:

```
cover property (@(posedge clk) disable iff (rst) strobe & ~hold ##1
ack);
```

1.2.2 Construct: covergroup

Use the `covergroup` construct to specify a coverage model. The following features are supported:

- Range and transition bins
- Bin arrays
- Auto-generated bins
- XMRs in coverpoint expressions
- Wildcard bins
- Crosses
- Ignore bins
- Options
- Optional formal arguments

Example

The following is an example of a `covergroup` construct specification.

```
covergroup cg @(posedge clk);
   cp1: coverpoint expr1 {
      bins a = { [1:3], [8:17] };
      bins b[] = {[2:4]};
      wildcard bins c = { 8'b11??000? };
      ignore_bins ignore_vals = {7,8};
   }
   cp2: mod.inst1.expr2 {
      bins d = {1};
```

```
        bins e = { 2, 4 };
    };
    cp3: expr3 { bins f = (7 => 8=>9); }
    cp4: expr4;
    cr: cross cp1, cp2;
endgroup : cg
cg mycg = new;
```

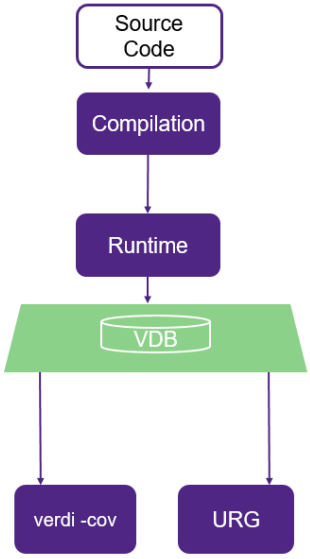
2 Performing Functional Coverage in ZeBu

This section contains the following topics pertaining to functional coverage with reference to compilation and runtime:

- *Overview*
- *Compilation and Runtime Settings for Collecting Covergroup Coverage*
- *Compilation and Runtime Settings for Using SVA*
- *Simultaneous Collection of Covergroup and SVA Coverage*
- *Limitations of Functional Coverage in Emulation*

2.1 Overview

Compilation of the design is the first step in the process. After compilation, the design is emulated and the specified coverage is collected. Thereafter, the coverage database (`.vdb`) is generated. You can then view the coverage report in Verdi or generate it with URG.



The compilation and runtime settings are different for covergroups and SVA.

2.2 Compilation and Runtime Settings for Collecting Covergroup Coverage

See the following sections for information on the following:

- *Setting Up Compilation for Collecting Covergroup Coverage*
- *Setting Up Runtime for Collecting Covergroup Coverage*

2.2.1 Setting Up Compilation for Collecting Covergroup Coverage

Before compilation, you must enable coverage by specifying commands in the UTF file. This section contains the following subsections:

- *Specifying UTF Commands to Enable Coverage for Covergroups*
- *Specifying Emulation-Specific VCS Options to Control Covergroup Coverage*

2.2.1.1 Specifying UTF Commands to Enable Coverage for Covergroups

Specify the following UTF command to enable all covergroups:

```
coverage -enable < 1 | true>
```

For more information on UTF commands, see the ***ZeBu Server User Guide***.

Example

The following specification is used to compile all covergroups:

```
coverage -enable true
```

This can also be specified as the following:

```
coverage -enable 1
```

2.2.1.2 Specifying Emulation-Specific VCS Options to Control Covergroup Coverage

For fine-grained control of covergroup coverage in compilation, use the following VCS command-line option:

`-Xfunc_hier=<hier_spec_file>`

Where, `<hier_spec_file>` is the file that contains the coverage construct inclusion/exclusion specification.

The following subsections describe how to create `<hier_spec_file>`:

- *Creating a <hier_spec_file> File*
- *Including or Excluding All Covergroup Instances*
- *Including or Excluding Specific Covergroup Instances*

For information on other VCS command-line options pertaining to coverage, see the **Coverage Technology User Guide**.

Creating a <hier_spec_file> File

In the `<hier_spec_file>` file, you can specify the following commands:

- **`module`**: Specify to exclude or include all `covergroup` instances in a module. To include, precede this command with a plus (+) sign. To exclude, precede this command, with a minus (-) sign.
- **`modulenode`**: Specify to exclude or include specific `covergroup` instances in a module. To include, precede this command with a plus (+) sign. To exclude, precede this command, with a minus (-) sign.

The command specifications are applied sequentially as they appear in the file.

If the first command is `+module`, it is assumed that initially no modules were selected for coverage collection. For example, if the file begins with `+module mod1`, then after processing this command, only the `mod1` module is included in the coverage collection; nothing else. All subsequent commands in the `<hier_spec_file>` file either add or remove modules from this module set.

If the first command is `-module`, it is assumed that initially all modules have been selected from coverage collection. For example, if the file begins with `-module mod1`, all modules except for the `mod1` module are included in the coverage collection. All subsequent commands in the `<hier_spec_file>` file

either add or remove modules from this module set.

Note *You can use wildcards to specify the module name.*

Including or Excluding All Covergroup Instances

To include or exclude all `covergroup` instances declared in a specific module, specify the `module` option in the `<hier_spec_file>` file, as shown in the following table:

TABLE 1 Include/Exclude All Covergroup Instances

Include all covergroup instances	Exclude all covergroup instances
`+module <module_name>`	`-module <module_name>`
Where, `<module_name>` is the name of the module to include or exclude from functional coverage evaluation.	

Example

In the following specification, all functional coverage instances that are declared in module test are excluded, if not enabled explicitly by other commands specified later in the `<hier_spec_file>` file.

```
-module test
```

Including or Excluding Specific Covergroup Instances

To include or exclude specific `covergroup` instances from all instances of a specific module, specify the `modulenode` option in the `<hier_spec_file>` file, as shown in the following table:

TABLE 2 Include/Exclude Specific Covergroup Instances

Include specific covergroup instances	Exclude specific covergroup instances
`+modulenode module_name.covergroup_instance_name`	`-modulenode module_name.covergroup_instance_name`
Where: `<module_name.coverage_instance_name>` is the name of the `covergroup` instance in the `<module_name>` module to include or exclude from functional coverage evaluation.	

Example

In this example, even though the entire module `test` is excluded, the `cov1` `covergroup` instance is included in the evaluation of the functional coverage flow.

```
-module test
+modulenode test.cov1
```

2.2.2 Setting Up Runtime for Collecting Covergroup Coverage

During runtime, you can control functional coverage collection by using APIs.

The Zcov class provides an interface that you can use to control `covergroup` coverage collection at runtime. This section explains the Zcov functions and parameters.

This section contains the following subsections:

- *Using Zcov API Functions*
- *Setting Parameters for Zcov Functions*
- *Zcov Examples: Controlling Functional Coverage at Runtime*

For a complete list of APIs, see the following file:

`$ZEBU_ROOT/include/libZebu.hh`

2.2.2.1 Using Zcov API Functions

The Zcov class has several defined functions for coverage collection. The following table describes these functions. These functions belong to the namespace ZEBU.

TABLE 3 Functions for Coverage Collection at Runtime

Function	Description
`Zcov::SetCoverageDb();`	Use to define the name of coverage database. For parameters, see *Defining the Name and Format of the Coverage Database*.
`Zcov::SetHierarchy();`	Use to select the hierarchies to output into the coverage database. For parameters, see *Setting Design Hierarchies for Coverage Collection*.
`Zcov::Start();`	Use to start coverage collection for covergroups. This function does not take any parameters.
`Zcov::Stop();`	Use to stop coverage collection for covergroups. This function does not take any parameters.
`Zcov::Flush();`	Use to flush coverage information into the coverage database. This function does not take any parameters.

2.2.2.2 Setting Parameters for Zcov Functions

The following subsections describe the parameters that you can use in the functions of the Zcov classes:

- *Defining the Name and Format of the Coverage Database*
- *Setting Design Hierarchies for Coverage Collection*

Defining the Name and Format of the Coverage Database

To define the name and format of the coverage database, use the `Zcov::SetCoverageDb()` function. The formal parameters of the `Zcov::SetCoverageDb()` function are listed in the following table. By setting these parameters, you can set the name and format of the generated coverage database.

TABLE 4 Runtime Parameters for Name and Format of the Coverage Database

Parameter	Description
`coverageDbPath`	Specifies the path to output coverage database
`format`	Specifies the coverage database format. You can either set it to VDB (`ZEBU_CoverDbFormatVdb`) or UCIS (`ZEBU_CoverDbFormatUcis`). Default is VDB.

VDB is the default Synopsys coverage database format used by VCS, Verdi, and other Synopsys tools. Use VDB if you want to visualize coverage with Verdi and merge coverage data between runs.

UCIS is an XML format.

Example

This example generates the coverage database in the VDB format. In this case, the coverage database is outputted at `./myzebu.vdb`.

```
Zcov::SetCoverageDb("myzebu.vdb", ZEBU_CoverDbFormatVdb);
```

The same can be written as the following:

```
Zcov::SetCoverageDb("myzebu.vdb");
```

Setting Design Hierarchies for Coverage Collection

To set the design hierarchies for coverage collection, use the `Zcov::SetHierarchy()` function. The parameters associated with the `Zcov::SetHierarchy()` function are listed in the following table. By setting these parameters, you can set the design hierarchies for coverage collection.

TABLE 5 Runtime Parameters for Design Hierarchy of Coverage Collection

Parameter	Description
`regularExpression`	Use to specify a regular expression of the hierarchical names of module instances where the `covergroup` coverage collection is to be enabled. Default is an empty string or NULL; this means that the scope of coverage is the entire hierarchy under hardware top.
`invert`	Use to invert the sense of the regular expression. Default is false.
`ignoreCase`	Use to ignore case distinctions in the hierarchical names. Default is false.
`hierarchicalSeparator`	Use to define the hierarchical separator character. Default is ".”

Note *If you do not invoke the Zcov::SetHierarchy () function, by default the entire hierarchy is selected.*

Example - Using Regular Expressions

Consider the following invocation:

```
Zcov::SetHierarchy("A2$");
```

This regular expression matches for example the following hierarchy:

```
A1
L0.A2
A2.L0
L0.L1.A2
L0.L1.L2.A2
L0.L1.L2.A3
L0.L1.L2.A4
```

2.2.2.3 Zcov Examples: Controlling Functional Coverage at Runtime

The examples in the section show the sequence of commands you need to specify to control functional coverage at runtime.

- *Example - Basic Usage*
- *Example - Coverage Collection Across Multiple Hierarchies*

Example - Basic Usage

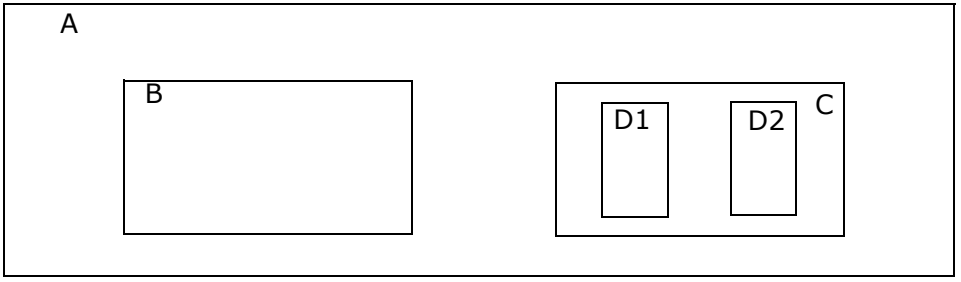
This example shows the commands you need to enable coverage at runtime for the entire hierarchy.

The sequence of commands is as follows:

```
using namespace ZEBU;
// User code to start emulation
// ...
// Specify the coverage database location; myzebu.vdb in the current
// directory. Use VDB format.
Zcov::SetCoverageDb("myzebu.vdb");
// ...
// Call the following function where you want to start coverage
// collection. Typically, this is done at the beginning of emulation.
Zcov::Start();
// ...
// Call the following function whenever you want to stop the overage
// collection.
Zcov::Stop();
// Write the collected coverage data into the database.
Zcov::Flush();
// ...
```

Example - Coverage Collection Across Multiple Hierarchies

Consider the following design hierarchy:



The following specification would start the coverage collection in A.B and later also in C.D1 hierarchies:

```
using namespace ZEBU;
...
Zcov::SetCoverageDb("myzebu.vdb");
...
Zcov::SetHierarchy("A.B");
Zcov::Start();  // Coverage collection starts at A.B
...
Zcov::SetHierarchy("C.D1");
Zcov::Start(); // Now coverage is collected at A.B and C.D1
...
Zcov::Stop();
Zcov::Flush();
//...
```

2.3 Compilation and Runtime Settings for Using SVA

For more information on UTF commands and SVA API functions, see the following sections:

- *Setting Up Compilation for Using SVA*
- *Setting Up Runtime to Enable Coverage for Using SVA*

2.3.1 Setting Up Compilation for Using SVA

Before compilation, you must enable coverage by specifying commands in the UTF file. This section contains the following subsections:

- *Specifying SVA-Specific UTF Commands to Enable Coverage*
- *Example: Setting UTF Commands*

2.3.1.1 Specifying SVA-Specific UTF Commands to Enable Coverage

The following table shows the commands you need to specify in the UTF file to enable coverage.

TABLE 6 UTF Commands for Compilation to Enable Coverage Using SVA

Command	Description
`assertion_synthesis -enable COVER`	Use to enable SVA cover statements only.
`assertion_synthesis -enable ALL`	Use to enable all SVA statements (assertions, assumptions, and cover).
`assertion_synthesis -cover_max_states <int>`	Use to specify a threshold to prevent expensive cover properties (SVA) from being synthesized. Specify an integer <int> for the maximum number of states allowed for a cover property.

For more information on assertions, see the ***ZeBu Server User Guide***.

2.3.1.2 Example: Setting UTF Commands

The following specification is used to compile both `covergroups` and SVA cover statements:

```
coverage -enable true
assertion_synthesis -enable COVER
```

2.3.2 Setting Up Runtime to Enable Coverage for Using SVA

During runtime, you can control functional coverage collection by using APIs.

This section contains the following subsections:

- *Using SVA API Functions*
- *Setting Parameters for SVA Functions*
- *SVA Examples: Controlling Functional Coverage at Runtime*

For a complete list of APIs, see the following file:

`$ZEBU_ROOT/include/libZebu.hh`

2.3.2.1 Using SVA API Functions

The SVA class has several defined functions for coverage collection. The following table describes these functions. These functions belong to the namespace ZEBU.

TABLE 7 SVA Functions for Coverage Collection at Runtime

Function	Description
`Zcov::SetCoverageDb();`	Use to define the name of coverage database. For parameters, see *Defining the Name and Format of the Coverage Database*.
`SVA::Start();`	Use to start coverage collection for cover properties. For parameters, see *Starting SVA Processing*.
`SVA::Set();`	Use to set enable values for collection of cover properties. For parameters, see *Setting Design Hierarchies for Coverage Collection*.

TABLE 7 SVA Functions for Coverage Collection at Runtime

Function	Description
`SVA::Stop();`	Use to stop coverage collection for cover properties. This function takes only the *board* parameter.
`Zcov::Flush();`	Use to flush coverage information into the coverage database. This function does not take any parameters.

2.3.2.2 Setting Parameters for SVA Functions

The following subsections describe the parameters that you can use in the functions of the SVA class:

- *Defining the Name and Format of the Coverage Database*
- *Starting SVA Processing*
- *Setting Design Hierarchies for Coverage Collection*

Defining the Name and Format of the Coverage Database

For more information, see the *Defining the Name and Format of the Coverage Database* section.

Starting SVA Processing

To start the SVA processing the static method `SVA::Start()` has to be called. The parameters associated with the `SVA::Start()` function are listed in the following table. By setting these parameters, you can set the design hierarchies for coverage collection.

TABLE 8 Runtime Parameters of SVA::Start()

Parameter	Description
`board`	Use to specify C++ handler on Board

TABLE 8 Runtime Parameters of SVA::Start()

Parameter	Description
`clockName`	Use to specify the name of the reference reporting clock. It must be in the sampling clock group.
`enableType`	Use to enable or disable SVA coverage collection. The possible values are as follows: - SVA::ENABLE_REPORT: Enables SVA coverage collection - SVA::DISABLE: Disables SVA coverage collection

Setting Design Hierarchies for Coverage Collection

To set the design hierarchies for coverage collection, use the `SVA::Set()` function. The parameters associated with the `SVA::Set()` function are listed in the following table. By setting these parameters, you can set the design hierarchies for coverage collection.

TABLE 9 Runtime Parameters for SVA::Set()

Parameter	Description
`types`	Use to initialize the value of an SVA or a group of SVAs. The possible values are as follows: - SVA::ENABLE_REPORT: Enables SVA coverage collection - SVA::DISABLE: Disables SVA coverage collection
`regularExpression`	Use to specify a regular expression of the hierarchical names of module instances where the coverage collection is to be enabled. Default is an empty string or NULL; this means that the scope of coverage is the entire hierarchy under hardware top.
`invert`	Use to invert the sense of the regular expression. Default is false.
`ignoreCase`	Use to ignore case distinctions in the hierarchical names. Default is false.
`hierarchicalSeparator`	Use to define the hierarchical separator character. Default is “.”

Example - Using Regular Expressions

Consider the following invocation:

```
SVA::Set("A2$");
```

This regular expression matches for example the following hierarchy:

```
A1
L0.A2
A2.L0
L0.L1.A2
L0.L1.L2.A2
L0.L1.L2.A3
L0.L1.L2.A4
```

2.3.2.3 SVA Examples: Controlling Functional Coverage at Runtime

The examples in the section show the sequence of commands you need to specify to control functional coverage at runtime.

- *Example - Basic Usage*
- *Example - Coverage Collection Across Multiple Hierarchies*

Example - Basic Usage

This example shows the commands you need to enable coverage at runtime for the entire hierarchy.

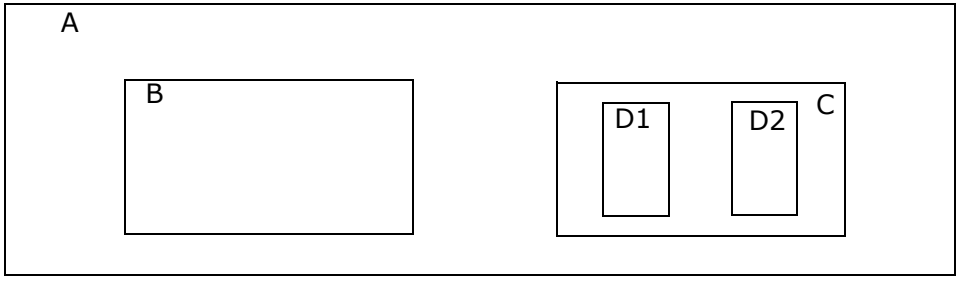
The sequence of commands is as follows:

```
using namespace ZEBU;
// User code to start emulation
// ...
Board *zebu = Board::open(ZEBUWORK)
// Specify the coverage database location; myzebu.vdb in the current
// directory. Use VDB format.
```

```
Zcov::SetCoverageDb("myzebu.vdb");
// ...
// Call the following function where you want to start coverage
// collection. Typically, this is done at the beginning of emulation.
SVA::Start(zebu,"clk", SVA::ENABLE_REPORT);
// ...
// Call the following function whenever you want to stop the overage
// collection.
SVA::Stop(zebu);
...
// Write the collected coverage data into the database.
Zcov::Flush();
// ...
```

Example - Coverage Collection Across Multiple Hierarchies

Consider the following design hierarchy:



The following specification would start the coverage collection in A and later disable coverage collection in A.C and then A.B hierarchies:

```
using namespace ZEBU;
...
Board *zebu = Board::open(ZEBUWORK)
Zcov::SetCoverageDb("myzebu.vdb");
```

```
...
SVA::Start(zebu,"clk", SVA::ENABLE_REPORT); // Coverage collection
// starts for the whole design (A)
SVA::Set(zebu, SVA::DISABLE, "A.C");   // Coverage collection is
// disabled for A.C hierarchy
...
SVA::Set(zebu, SVA::DISABLE, "A.B");   // Coverage collection is
// disabled for A.B // hierarchy
...
SVA::Stop(zebu);
//...
Zcov::Flush();
//...
```

2.4 Simultaneous Collection of Covergroup and SVA Coverage

The following example shows the simultaneous collection of covergroups and SVA coverage.

```
using namespace ZEBU;
// User code to start emulation
// ...
Board *zebu = Board::open(ZEBUWORK)
// Specify the coverage database location; myzebu.vdb in the current
// directory. Use VDB format.
Zcov::SetCoverageDb("myzebu.vdb");
// ...
// Start covergroup coverage collection
Zcov::Start();
// Start SVA coverage collection and run emulation testbench
SVA::Start(zebu,"top.clk", SVA::ENABLE_REPORT);
// ...
// Stop covergroup coverage collection
Zcov::Stop();
// Stop SVA coverage collection
SVA::Stop(zebu);
// Store collected data in coverage database
Zcov::Flush();
```

2.5 Limitations of Functional Coverage in Emulation

The following limitations currently exist:

- Functional coverage collection is supported in DUT only, but not in transactors.
- No support for embedded covergroups (covergroups that are members of a class). For example, the following specification is not supported:

```
class myclass;
...
covergroup ccg @clk; // embedded covergroup
...
endgroup
...
endclass
```

- No support for explicit covergroup sampling. For example, the following specification is not supported:

```
cg: covergroup;
...
endgroup : cg
...
cg mycg = new;
...
mycg.sample; // Not supported
```

- No support for local variables and covergroup sampling from cover properties. For example, the following specification is not supported:

```
property p1;
int x;
@(posedge clk)(a, x = b) ##1 (c, cg1.sample(a, x));
endproperty : p1
```

- No support for non-constant expressions in `bin` specifications. For example, the following specification is not supported:

```
int size;
bit [7:0] val;
...
cg: covergroup (int width);
cp: coverpoint val {
   bins b[width] = { [0:9] };
}
endgroup : cg
cg mycg1 = new(size); // Not supported
cg mycg2 = new(5); // Supported
```

- No support for non-constant expressions in options of covergroups, coverpoint and so on. These options cannot be assigned externally.
- No support for covergroups in encrypted modules.
- `Covergroup` instantiation is supported only at the `covergroup` variable declaration or in initial or always procedures. For example, the following specification is supported:

```
covergroup cov;
...
endgroup

cov cginst = new();
```

  The following specification is not supported:

```
function automatic func1 (bit  [2:0] in1);
       cginst = new();
endfunction
```

- No support for non-temporal cover statements. For example, the following specifications are not supported:

  Immediate:

  ```
  cover (cond);
  ```

  Deferred Observed:

  ```
  cover #0 (cond);
  ```

  Deferred Final:

  ```
  cover final (cond);
  ```

- No support for constrained transition array bins. For example, the following specification is not supported:

  ```
  enum{one,two,three,four,five,six,seven,eight,nine}i;
    covergroup cg @(i);
        cp3 : coverpoint i{ wildcard bins ss0 [2] = (2'b1?[*3]=>2);}
    endgroup
  ```

3 Viewing Coverage Report and Analysis Using Verdi

After the coverage database is generated, you can view the coverage reports to analyze the coverage data. If the coverage data is less, you can regenerate the coverage database by increasing coverage. This process is iterative.

This section contains the following topics:

- *Generating Coverage Reports*
- *Using Verdi for Functional Coverage Analysis*

3.1 Generating Coverage Reports

To generate coverage reports, you can use the following methods:

- **Verdi (recommended)**: Specify the following at the command prompt to launch Verdi with the coverage database:

  ```
  %verdi -cov -covdir ./<coverage>.vdb
  ```

  You can also merge coverage data from different runs and from different engines, as shown in the following illustration.

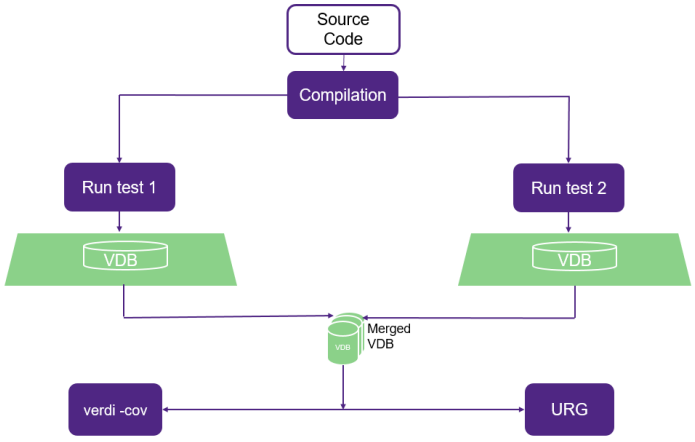


FIGURE 1. Merge coverage data

To merge coverage data from different runs and from different engines, specify the following:

```
%verdi -cov -covdir ./<coverage1>.vdb -covdir ./<coverage2>.vdb...
```

- **Legacy URG flow**: Specify the following at the command prompt to see the urgReport:

```
%urg -dir <coverage>.vdb
```

3.2 Using Verdi for Functional Coverage Analysis

Verdi is a coverage analysis and report generation tool, which is used to:

- Visualize coverage results
- Debug coverage results
- Perform coverage gap analysis
- Compare two coverage databases (.vdb)

For details on Verdi features, see the **Verdi Coverage User Guide and Tutorial** document.

The following sections show key Verdi features that enable efficient functional coverage analysis:

- *See Coverage Information in a Single View*
- *View Native HTML Reports*

Verdi also supports all features of URG, such as generating HTML views.

3.2.1 See Coverage Information in a Single View

Verdi presents coverage information, such as covergroups and coverpoints, in a single view. The following figure shows the single view provided by Verdi:

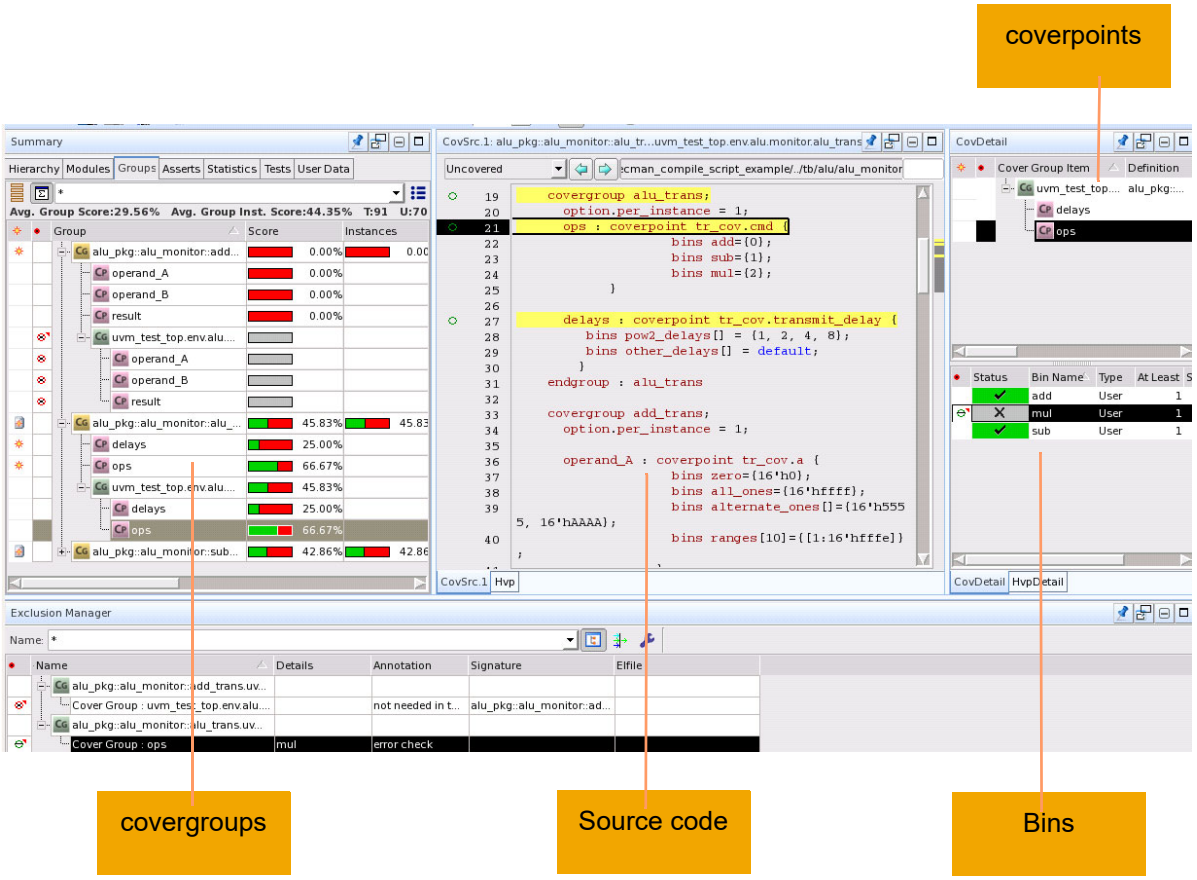


FIGURE 2. Single view of covergroups, coverpoints, bins, and source code

3.2.2 View Native HTML Reports

The following figure shows the native HTML reports that are generated. The groups coverage summary is displayed.

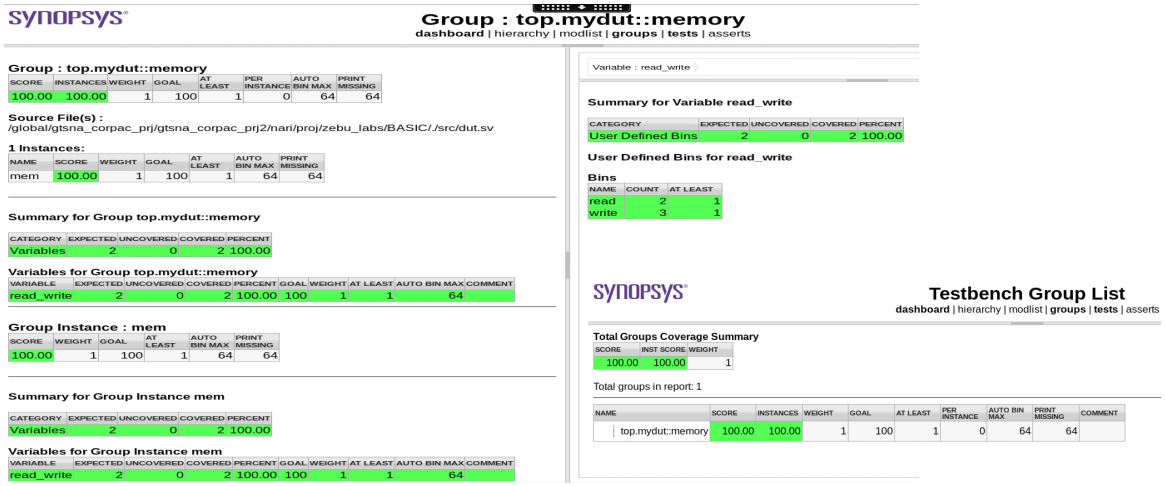


FIGURE 3. Coverage report in HTML