ZeBu® Server Functional Coverage User Guide

Version V-2024.03-1, July 2024

SYNOPSYS®

Copyright and Proprietary Information Notice

Destination Control Statement

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

Disclaimer

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

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Feedback

Contents

Preface

This chapter has the following sections:

About This Book

The *ZeBu 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 the ZeBu system.

These engineers should have knowledge of the following Synopsys tools:

- ZeBu
- 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.

Section	Describes
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 User Guide	Provides detailed information on using ZeBu.
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu UTF Reference Guide	Describes Unified Tcl Format (UTF) commands used with ZeBu.
ZeBu Power Aware Verification User Guide	Describes how to use Power Aware verification in ZeBu environment, from the source files to runtime.
ZeBu Functional Coverage User Guide	Describes collecting functional coverage in emulation.
Simulation Acceleration User Guide	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.
ZeBu Runtime Performance Analysis With zTune User Guide	Provides information about runtime emulation performance analysis with zTune.
ZeBu Custom DPI Based Transactors User Guide	Describes ZEMI-3 that enables writing transactors for functional testing of a design.
ZeBu LCA Features Guide	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
ZeBu Synthesis Verification User Guide	Provides a description of zFmCheck.

Document Name	Description
ZeBu Transactors Compilation Application Note	Provides detailed steps to instantiate and compile a ZeBu transactor.
ZeBu zManualPartitioner Application Note	Describes the zManualPartitioner feature for ZeBu. It is a graphical interface to manually partition a design.
ZeBu Hybrid Emulation Application Note	Provides an overview of the hybrid emulation solution and its components.

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

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1

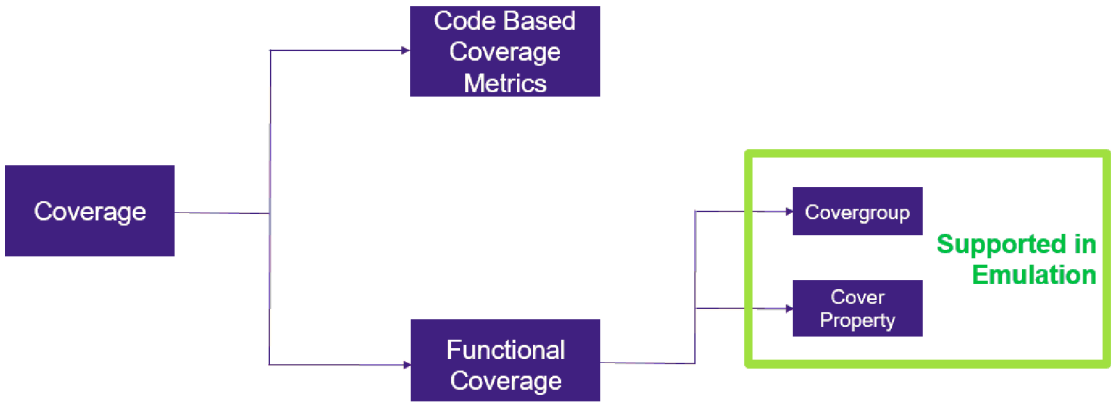
Introduction to Functional Coverage

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. 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



The following SystemVerilog constructs are used to specify functional coverage:

- SVA cover Statement
- Construct: covergroup

Note:

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

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);
```

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

Feedback

- XMRs in coverpoint expressions
- Wildcard bins
- Crosses
- Ignore bins
- Options
- Optional formal arguments
- The `sample()` method, with or without arguments. For information on coverage sampling, see Example: Covergroup Sampling.

Example: 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;
```

Example: Covergroup Sampling

```
covergroup cg_na;
…
endgroup

covergroup cg_wa (bit a, int b) with function sample (bit a, int b);
…
endgroup : cg

cg_na cg1 = new();
cg_wa cg2 = new();

always @(posedge clk) begin
   …
   cg1.sample();
```

```
    cg2.sample(arg1, arg2);
…
end
```

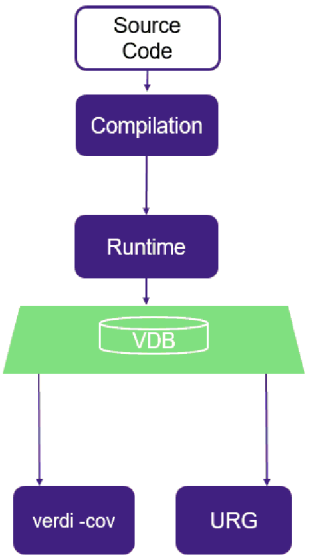
Note:

For covergroups having “with sample function” usage, the instance-based coverage report is not available.

2

Performing Functional Coverage in ZeBu

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.

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

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

Feedback

Compilation and Runtime Settings for Collecting Covergroup Coverage

See the following sections for information:

- Setting Up Compilation for Collecting Covergroup Coverage
- Setting Up Runtime for Collecting Covergroup Coverage
- Using Functional Coverage UCLI commands

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

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 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`

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.
- **module_node**: 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

Feedback

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 `module_node` 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
`+module_node <module_name> <covergroup_instance_name>`	`-module_node <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

+module_node test cov1
```

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`

Feedback

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();`	Defines the name of coverage database. For parameters, see Defining the Name and Format of the Coverage Database.
`Zcov::SetHierarchy();`	Selects the hierarchies to output into the coverage database. For parameters, see Setting Design Hierarchies for Coverage Collection.
`Zcov::Start();`	Starts coverage collection for covergroups. This function does not take any parameters.
`Zcov::Stop();`	Stops coverage collection for covergroups. This function does not take any parameters.
`Zcov::Flush();`	Flushes coverage information into the coverage database. This function does not take any parameters.
`Zcov::SetPeriodicDump();`	Saves the Zcov database after specified cycles of UserClock or TickClk, or after a specified time interval. Counters are reset after each database instance is saved. For parameters, see Saving Zcov Database Periodically
`Zcov::StopPeriodicDump();`	Stops saving the Zcov database periodically. This function does not take any parameters.

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
- Saving Zcov Database Periodically

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. In addition, you can specify compression options for the database.

Feedback

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.
`ZEBU_UcisXmlDumpOptions`	Controls the generation of the XML as part of the database and compresses the database when you need to keep it as a part of VDB. You can use one of the following options: • `ZEBU_UcisXmlDumpOptionsRemove` removes `ucis.xml` • `ZEBU_UcisXmlDumpOptionsCompressed` compresses `ucis.xml` • `ZEBU_UcisXmlDumpOptionsUncompressed` saves `ucis.xml` as it is

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: Using `coverageDbPath` and `format`

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");`

Example: Using `ZEBU_UcisXmlDumpOptions`

```
Zcov::SetCoverageDb("test"); //saves only VDB

Zcov::SetCoverageDb("test", ZEBU_CoverDbFormatVdb); //saves only VDB

Zcov::SetCoverageDb("test", ZEBU_CoverDbFormatVdb,
 ZEBU_UcisXmlDumpOptionsUncompressed);  //saves VDB and ucis.xml

Zcov::SetCoverageDb("test", ZEBU_CoverDbFormatVdb,
 ZEBU_UcisXmlDumpOptionsCompressed); //saves VDB and compresses ucis.xml

Zcov::SetCoverageDb("test", ZEBU_CoverDbFormatVdb,
 ZEBU_UcisXmlDumpOptionsRemove); //saves only 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
```

Saving Zcov Database Periodically

To automatically create a database of functional coverage data after every `n` cycles of clock specified by a user, use the `Zcov::SetPeriodicDump()` function. To stop saving the Zcov database, use the `Zcov::StopPeriodicDump()` function. The parameters associated with the `Zcov::SetPeriodicDump()` function are described in . The `Zcov::StopPeriodicDump()` function does not take any parameters.

Feedback

Table 6 Parameters for Creating Functional Coverage Database

Parameter	Description
`NumCycles`	Saves the Zcov database after specified cycles, `NumCycles`, of UserClock or TickClk. The databases is saved as `coverageDbpath_<NumCycles*1>_tickClk.vdb`, `coverageDbpath_<NumCycles*2>_tickClk.vdb` and so on. Here, the `coverageDbpath` is set by the `Zcov::SetCoverageDb()` function. **Note:** The database can be saved either after a specified number of cycles of user clock or after a specified time interval, not both. So, specify one of the parameters, either `NumCycles` or `Time`.
`Time`	Saves the Zcov database after specified interval of time. If no time is specified, the database is saved after a default unit of time. The database is saved as `coverageDbpath_<Time*1>`, `coverageDbpath_<Time*2>` and so on. **Note:** You cannot specify the `Time` parameter with `UserClockPath`.
`UserClockPath`	Saves the Zcov database after every NumCycles of the user clock. The database is saved as `coverageDbpath_<NumCycles*1>_<UserClockPath>.vdb`, `coverageDbpath_<NumCycles*2>_<UserClockPath>.vdb` and so on. Here, `coverageDbpath` is set by `Zcov::SetCoverageDb()` function. `UserClockPath` cannot be used in the following cases: • With the `Time` parameter. • With the `NumCycles` parameter if it is a CDP or RTL clock • As blank in case of zcie clocks.

Warning:

`Zcov::SetCoverageDb(coverageDbpath)` must be called before `Zcov::SetPeriodicDump`; otherwise an error message is displayed and the program terminates abruptly. If, however, `Zcov::SetCoverageDb` is called after `Zcov::SetPeriodicDump` but before `Zcov::StopPeriodicDump` then, a warning message is displayed and the call to `SetCoverageDb` is discarded

Limitations

Overflow Detection is turned off when `SetPeriodicDump()` starts. For example, if you have two clocks, `fast_clk` and `slow_clk`, and the `slow_clk` is 20 times slower than the `fast_clk`. If you set the `SetPeriodicDump()` function on `slow_clk`, counters running on `fast_clk` might overflow and not give correct value

Example 1: Using <NumCycles> and <UserClkPath> in case of Zcie clocks

```
Zcov::SetCoverageDb("testbench")
Zcov::SetPeriodicDump("1000", "top.clk") //top.clk is a zcie clock
```

```
Zcov::Start()
//Run 2000 User Clock Cycles
Zcov::StopPeriodicDump()
Zcov::SetCoverageDb(“testbench2”)
Zcov::SetPeriodicDump(“1000”,”top.clk”) //top.clk is a zcie clock
//Run 2000 User Clock Cycles
Zcov::StopPeriodicDump()
```

In this example, the following steps happen at runtime:

Run 1000 cycles of `top.clk`.

Save `testbench_1000_top.clk.vdb`.

Reset coverage counters.

Run 1000 cycles of `top.clk`.

Save `testbench_2000_top.clk.vdb`.

Reset coverage counters.

Run 1000 cycles of `top.clk`.

Save `testbench2_1000_top.clk.vdb`.

Run 1000 cycles of `top.clk`.

Save `testbench2_2000_top.clk.vdb`.

Example 2: Using `<Time>`

```
Zcov::SetCoverageDb("testbench1");
Zcov::SetPeriodicDump(“1000”);
Zcov::Start();
//Run 2000 Cycles
Zcov::StopPeriodicDump();
Zcov::SetCoverageDb(“testbench2”);
Zcov::SetPeriodicDump(“1000”);
//Run 2000 Cycles
Zcov::StopPeriodicDump();
```

In this example, the database is automatically saved after every 1000 simulation time units. The databases are saved as `testbench1_1000`, `testbench1_2000`. Then, the databases are saved as `testbench2_1000`, `testbench2_2000`. The last database is saved for the remaining time by the `Board::close()` function.

Example 3: `<Time>` provided by the user

```
Zcov::SetCoverageDb("testbench1");
Zcov::SetPeriodicDump(“1000ns”);
Zcov::Start();
//Run 2000ns time
Zcov::StopPeriodicDump();
```

```
Zcov::SetCoverageDb(“testbench2”)
Zcov::SetPeriodicDump(“1000ns”)
//Run 2000ns time
Zcov::StopPeriodicDump()
```

In this example, the database is automatically saved after every 1000ns time interval as `testbench1_1000ns`, `testbench1_2000ns`, `testbench2_1000ns`, `testbench2_2000ns` and so on. Database for the remaining time is saved by the `Board::close()` function.

Example 4: `Zcov::Start` and `Zcov::Stop` called after `Zcov::SetPeriodicDump`

```
Zcov::SetCoverageDb("testbench1");
Zcov::SetPeriodicDump(“1000”, “top.clk”);
Zcov::Start();
//Run Cycles
Zcov::Stop();
//Run Cycles
Zcov::Start();
```

Example 5

```
Zcov::SetCoverageDb("testbench1");
Zcov::SetPeriodicDump(“1000”, “top.clk”); //Dump counters on every 20000
 cycles of “top.clk”
Zcov::Start();
//Run Cycles
std::string currentDbPath = Zcov::GetCoverageDb();
Zcov::Flush()
//Run Cycles
```

In this example, you can retrieve the current database path using `Zcov::GetCoverageDb()` and save the data to check coverage counters by using the `Zcov::flush()` function. However, the database is overwritten when next multiple of `<NumCycles>` or `<Time>` is reached or when `Board::Close()` is called.

Example 6: Using `<tickClk>`

```
Zcov::SetCoverageDb("testbench1");
Zcov::SetPeriodicDump(“1000”, “tickClk”);
Zcov::Start();
//Run 2000 Cycles
Zcov::StopPeriodicDump();
Zcov::SetCoverageDb(“testbench2”);
Zcov::SetPeriodicDump(“1000”, “tickClk”);
//Run 2000 Cycles
Zcov::StopPeriodicDump();
```

In this example, the database is automatically saved after every 1000 cycles of `tickClk` as `testbench1_1000`, `testbench1_2000`. Then, databases are saved as `testbench2_1000`, `testbench2_2000`. The last database is saved for the remaining cycles until when the `Board::close()` is called.

Example 7

```
Zcov::SetPeriodicDump("1000", top.clk)
Zcov::SetCoverageDb("testbench3")
```

In this example, a call to the `Zcov::SetCoverageDb` function is made after `Zcov::SetPeriodicDump` but before `Zcov:StopPeriodicDump`. In this case, the following warning is displayed:

```
"Zcov SetPeriodicDump is active, hence call to SetCoverageDb is
 discarded"
```

Example 8

```
Zcov::SetPeriodicDump("1000ns", "top.clk");
```

In this example, `<UserClockPath>` is specified with `<Time>` that leads to the following error:

```
"UserClockPath cannot be specified with Time"
```

Example 9

```
Zcov::SetPeriodicDump("1000", "top.clk");
```

In this example, `top.clk` is a CDP or RTL clock, therefore, `Zcov::SetPeriodicDump` displays the following error:

```
"top.clk cannot be specified with NumCycles since it is an CDP/RTL clock"
```

Example 10

```
Zcov::SetPeriodicDump("1000");
```

In this example, `UserclkPath` for the ZCEI clock is not specified, therefore, `Zcov::SetPeriodicDump` displays the following error:

```
"UserclkPath cannot be empty in case of zcie clks"
```

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

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

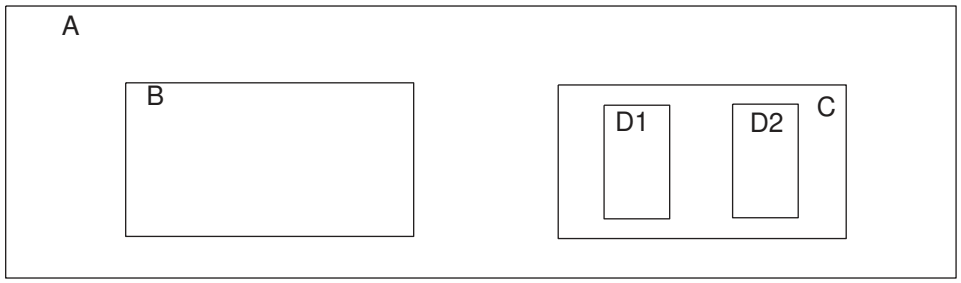
The sequence of commands is as follows:

```
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();
//...
```

Feedback

Using Functional Coverage UCLI commands

Apart from using Zcov API functions, you can use `zRci` for functional coverage. The following table describes the Functional Coverage UCLI commands that can be used with `zRci`.

Table 7 Functional Coverage UCLI Commands for zRci

Command	Description
`coverage -db <path> [-vdb|-ucis]`	Use to define the name of coverage database. The following options can be used: `[-db <path>]`: Set coverage database path. `[-vdb]`: Select VDB database format (default). `[-ucis]`: Select UCIS database format.
`coverage -hierarchy <reg_exp> [-i] [-v] [-h <character>]`	Specify design hierarchies for coverage collection. The following options can be used: `-hierarchy <reg_exp>`: Specify the value that should be evaluated as a regular expression. `[-i]`: Use to ignore case distinctions in the hierarchical names. Default is false. `[-v]`: Use to invert the sense of the regular expression. Default is false. `[-h <character>]`: Use to define the hierarchical separator character. Default is '.'.
`coverage -functional on|off`	Retrieve status of functional coverage collection.
`coverage -flush`	Flush pending coverage data actions in database.

For more information, see the following `coverage` UCLI command examples:

- Example: Specify coverage database and enable coverage collection
- Example: Specify design hierarchy for coverage collection
- Example: Specify coverage database in UCIS format

For UCLI commands pertaining to other activities, including Debug and Runtime, see the *ZeBu UCLI User Guide*.

Example: Specify coverage database and enable coverage collection

```
# ...
# Set coverage database testbench.vdb in the current directory.
# Use VDB format by default
```

Feedback

```
coverage -db testbench.vdb
# ...
# Start coverage collection.
coverage -functional on
# ...
# Write the collected coverage data into the database.
coverage -flush
```

Example: Specify design hierarchy for coverage collection

```
# ...
# Set coverage database testbench in the current directory.
# Use VDB format by default
coverage -db testbench.vdb
#Coverage collection starts at A.B
coverage -hierarchy A.B
#...
# Start coverage collection.
coverage -functional on
# ...
# Write the collected coverage data into the database.
coverage -flush
```

Example: Specify coverage database in UCIS format

```
coverage -db testbench -ucis
```

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

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

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 8 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 User Guide*.

Example: Setting UTF Commands

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

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

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`

Feedback

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 9 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.
`SVA::Stop();`	Use to stop coverage collection for cover properties. This function takes only the `board` parameter as described in Table 10.
`Zcov::Flush();`	Use to flush coverage information into the coverage database. This function does not take any parameters.

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
- Example - Using Regular Expressions

Defining the Name and Format of the Coverage Database

For more information, see the Setting Up Compilation for Collecting Covergroup Coverage section.

Feedback

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 10 Runtime Parameters of SVA::Start()

Parameter	Description
`board`	Use to specify C++ handler on Board
`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 11 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 “.”

Feedback

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
```

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.

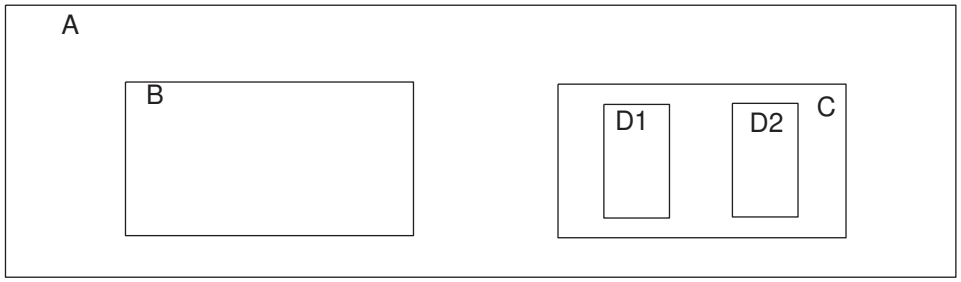
The sequence of commands is as follows:

```
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();
// ...
```

Feedback

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();
//...
```

Example - 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
```

Feedback

```
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();
```

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 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);`

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:

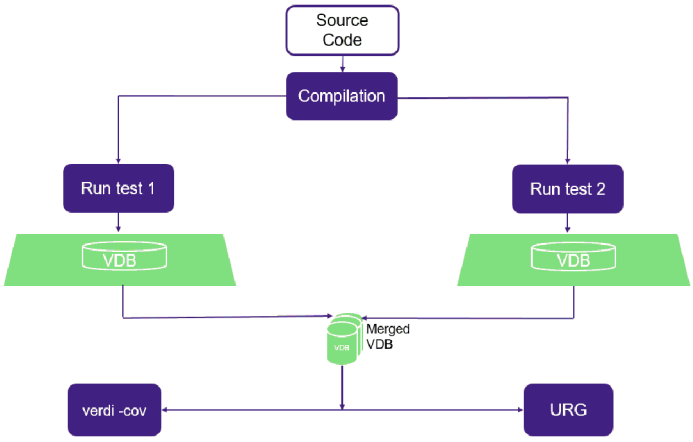
Generating Coverage Reports

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 2 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
  ```

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

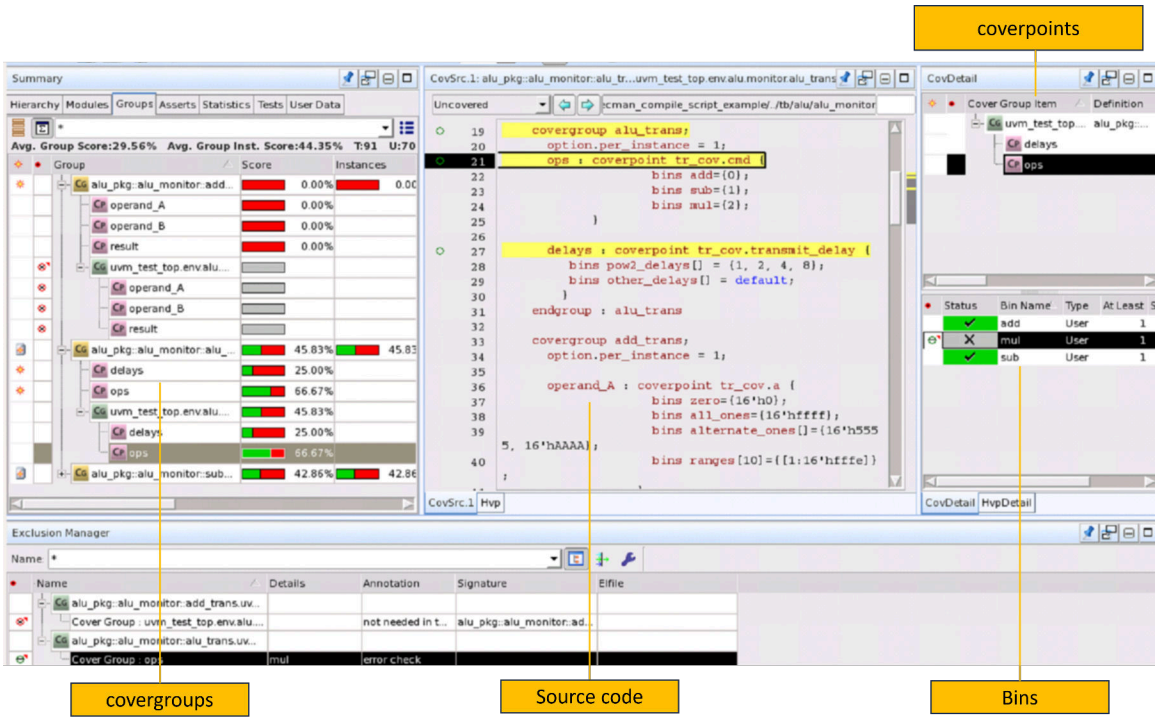
Verdi also supports all features of URG, such as generating HTML views.

Feedback

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 3 Single view of covergroups, coverpoints, bins, and source code



View Native HTML Reports

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

Feedback

Figure 4 Coverage report in HTML