VLSI III SP24 Verification Mini Project

Designed by Paul Gao, Bespoke Silicon Group & TA for VLSI III SP24

Overview

Goal

The high-level goal of this project is to familiarize you with advanced concepts in verification, while also allowing you significant freedom to choose your own path and make a unique contribution.

Task

To demonstrate your mastery of this material, you will select and verify a SystemVerilog module that:

- Is from basejump_stl
- Is synthesizable (for example, not a testbench module)
- Has not been verified with functional coverages
    - To be specific, you should not choose bsg_fifo_1r1w_small_hardened, bsg_cache_non_blocking, bsg_cache_non_blocking_tl_stage, bsg_cache_non_blocking_mhu or bsg_cache_non_blocking_miss_fifo as your DUT
- Has >= 2 covergroups (usually reset is in its own covergroup)
- Has >= 6 control signals (example code has 8 control signals)
    - Signals refer to input ports and internal registers, internal wires are not signals
    - Internal registers can be instantiated within a lower-level module, if there is a path from lower-level module's registers to lower-level module's output ports, you should regard lower-level module's output ports as signals
    - One signal may have multiple bits, you may partition it into multiple signals but the partition has to be reasonable (for example, rptr_r and wptr_r in the example code cannot be partitioned)
    - Control signals refer to non-data signals (it is possible that there are a few control bits within a data signal, those control bits are also control signals)
    - Reset is a control signal. Clock is not a control signal.
    - The module should have >= 4 internal control signals, and >= 2 input control signals
    - You can choose which control signals are used for covergroup definitions and which are used for coverpoints-cross, but all possible functional modes should be taken care of.

To achieve the goal, you will

- Write a testbench in Python using the cocotb library.
- Write a coverage file in SystemVerilog using covergroups.
- Achieve 100% functional coverage

This is a solo project.

Deliverables

- <your_selected_basejump_stl_module>_testbench.py
- <your_selected_basejump_stl_module>_cov.sv
- <your_selected_basejump_stl_module>_wrapper.sv
- Makefile

Deadlines

Deadline 1 (Framework): May 15

- Submit all deliverables
- No need to achieve 100% functional coverages
- The main testbench and coverage file can be simple and basic, as long as the test is running without error
- Submit your work on Canvas (zip package)

Deadline 2 (Optimization): May 29

- Submit all deliverables
- Achieve 100% functional coverages
- The main testbench and coverage file should have been finalized to achieve 100% functional coverages
- **NEW** Two options for turn in:
  - Via pull request if you would like to contribute your code to BaseJump STL, submit the pull request URL on Canvas
  - Submit your work on Canvas (zip package)

Opportunity to become famous

If you reach 100% coverage, then we invite you to contribute your test to the basejump_stl github project! This is purely optional, but encouraged since then your code can live on in eternity! The TA will provide more details later in the term on how to submit a pull request and what the conditions are for acceptance!

Steps to create a pull request on basejump_stl:

1. On github.com, fork basejump_stl repository
2. Create a new branch and push the new code onto it

3. Create a pull request from a fork: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request-from-a-fork

Background

Philosophy of Verification

Why do we write verification tests for our hardware? Why not wait for the bug to occur in the wild and then fix it? There are several reasons:

First, especially for ASICs, many bugs are irreversible and could end up being a huge waste (many people-months and millions of dollars) of money and time.

But second, writing unit tests is a much more efficient and time saving way of dealing with bugs. There is a well-known adage in software engineering that the longer a bug goes undiscovered, it grows exponentially more expensive to fix. The Intel FDIV floating point bug from 1994 resulted in a massive $495 million recall – the first chip recall ever – and damage to Intel's reputation. Of course if the code causes an accident in a Tesla self-driving car, then no amount of money can be spent to reverse the effects. In the context of our prototyping efforts, the cost is generally less (up to the value of a $2.5M tapeout, anyways!) but we still have a great reason for using unit tests to track down bugs earlier: *The smaller the module that we have to debug, the easier it is to localize and fix the bug.*

For example, if the bug shows up "in the wild" while we are trying to boot Linux on a core, then we have a 500M cycle window where something may have gone wrong. Just running the program once will take several days in simulation, let alone trying to figure it out. Then debugging the bug requires that we understand what Linux is supposed to do, what the FPGA or ASIC is supposed to, what the processor is supposed to do, etc; the level of expertise goes ways up (literally navigating millions of lines of codes) and up for being able to figure it out. And the process ceases to become systematic and instead becomes mostly about trial-and-error guessing. We've taken a relatively easy problem and made it a month-long PhD-level challenge (not speaking from experience here, oh no, not at all, Linux not running on the UW RISC-V BlackParrot core for months?…I invoke the fifth amendment!)

Even more nasty is if the bug only shows up randomly or once every year, etc. The best way to deal with bugs is to set traps for them and catch them early!

Of course, we also want to be agile, especially as academics, so the core question is: how do we maximize our productivity by making verification relatively easy to do?

Coverage

When we design a new SystemVerilog module or modify a current module, we usually write a testbench that will stimulate that design, and then check the results to make sure that they match the expected behavior. But how do we know how good the test is? Does it cover all of the corner cases, or only some of them? In order to make sure that we have good tests, we need to be able *to measure and verify their coverage*. In short, we must verify the verification. =)

The most simple forms of coverage are *line* or *toggle coverage*, which can be used to verify that at least every line of code in the Verilog has been "executed", or that every wire has toggled. But this is far from sufficient, because much of the functionality of a module depends on a combination of wires. For example, even our most basic handshake, valid/ready; requires two signals to determine if we have adequately tested it.

Instead, to have more assurance, we would like to define a space that is the combination of multiple wires, ideally that are both independent of each other and that span the space of states for the hardware. Independence is important because dependent signals cannot be covered 100% (for example, if a = b & c; we can never cover a=1, b=0, c=0), and will require manual effort to remove that case from consideration. Spanning the space is important because it's the only way to know that our test is testing everything.

Coverage enables Random Verification (..which is awesome!!)

This may seem like a lot of work, but having support for coverage allows us to apply a new and exciting methodology: ***random verification!*** The idea is, instead of thinking about how we can setup a test to get the design into a bunch of weird corner cases to shake out bugs, we can make use of randomized input generators; and then use our coverage metrics to measure whether those randomized generators have sufficiently covered the hardware. We can tweak our tests until we get where we need to be. So, assuming that writing coverage scripts is easier than trying to write weird corner case tests (**it usually is!**)**,** using a coverage methodology actually allows us to converge faster on fully coverage hardware. (Of course, we still need to make sure that our tests correctly check the result of our tests...)

Random testing in general, is very powerful, because it allows us to essentially do ***fuzzing,*** basically throw a bunch of random stuff at the module, to see what states it can get into. One difference: in traditional fuzzing, we are trying to hack a piece of software by feeding a series of random inputs that violate the preconditions that software was designed for. In this case, we are feeding the SystemVerilog code random *valid* inputs to see if we can automatically explore all of the internal states.

The example code uses a module from BaseJump STL, bsg_fifo_1r1w_small_hardened, as a running example to show how to do functional coverage analysis. This module had a legit bug (prior to our use of the methodology in this document!) that escaped us and made it into a multi-million dollar tapeout. So it's 100% legit. =P

Example Code

Step 1: Preparations

On the Linux Lab Machine, run

echo $SHELL

You should expect the “/bin/bash” message.

- If not, you are probably not using the bash shell, please run **bash** to switch to the bash shell before moving on. Note that **for this project you will have to be on bash all the time**, for convenience you may update your shell preference (usually defined in ~/.bashrc) so that you don’t need to switch shells repeatedly.

After confirming we are on bash, run

pip3 install cocotb --user

This will install cocotb in your home directory. To verify installation, run

cocotb-config --version

You should expect the “1.8.1” message.

- If not, cocotb installation is probably unsuccessful, please run **export PATH=$PATH:/homes/<your-username>/.local/bin** to add cocotb installation directory into system path. You may also add this command to your ~/.bashrc file to make it permanent.

More cocotb installation information can be found on the cocotb website.

Step 2: Run the example code

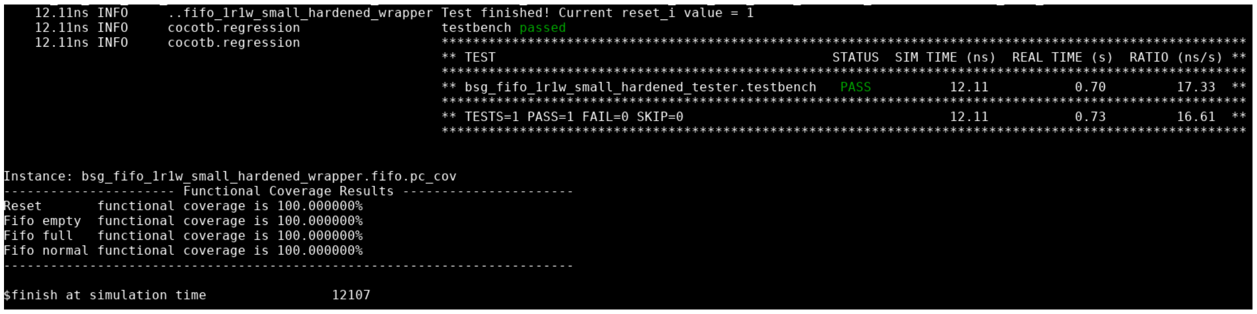
First, clone the example code repo and the basejump_stl repo in the same directory:

git clone https://github.com/bsg-external/cocotb.git
git clone https://github.com/bespoke-silicon-group/basejump_stl.git

After cloning the example code, run

cd cocotb/examples/warmup_project
make

You should expect the cocotb “testbench passed” result, and 100% functional coverage result.



```
    12.11ns INFO     ..fifo_1r1w_small_hardened_wrapper Test finished! Current reset_i value = 1
    12.11ns INFO     cocotb.regression                  testbench passed
    12.11ns INFO     cocotb.regression                  ***************************************************************************************************
                                                        ** TEST                                              STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                                        ***************************************************************************************************
                                                        ** bsg_fifo_1r1w_small_hardened_tester.testbench      PASS          12.11           0.70          17.33  **
                                                        ***************************************************************************************************
                                                        ** TESTS=1 PASS=1 FAIL=0 SKIP=0                                     12.11           0.73          16.61  **
                                                        ***************************************************************************************************


Instance: bsg_fifo_1r1w_small_hardened_wrapper.fifo.pc_cov
---------------------- Functional Coverage Results ----------------------
Reset       functional coverage is 100.000000%
Fifo empty  functional coverage is 100.000000%
Fifo full   functional coverage is 100.000000%
Fifo normal functional coverage is 100.000000%
-------------------------------------------------------------------------

$finish at simulation time                12107
```

I noticed that a few lab machines had broken GCC installations, so if you encounter the **g++: Command Not Found** error message, please try to run the example code on another lab machine.

- Update: EE admin has fixed the problem, now it should be installed on all lab machines

To view the simulation waveform, run

make ee-verdi

To visualize the coverage results, run

make ee-verdi-cov
(It is normal to see error messages in red color on Verdi GUI, the error messages have no effect on this project)

To clean the simulation, run

make ee-clean

Note that “ee-clean” is different from the normal “clean”. “ee-clean” removes all the generated files while “clean” does not.

Step 3: Understand the example code

The example code are part of the testing infrastructure for the basejump_stl module bsg_fifo_1r1w_small_hardened, with data_width=16 and fifo_depth=4. The example code have two purposes:

1. Verifying the fifo is working as expected
2. Verifying the testbench triggers all possible functionalities in the fifo

There are 4 files in the example code directory:

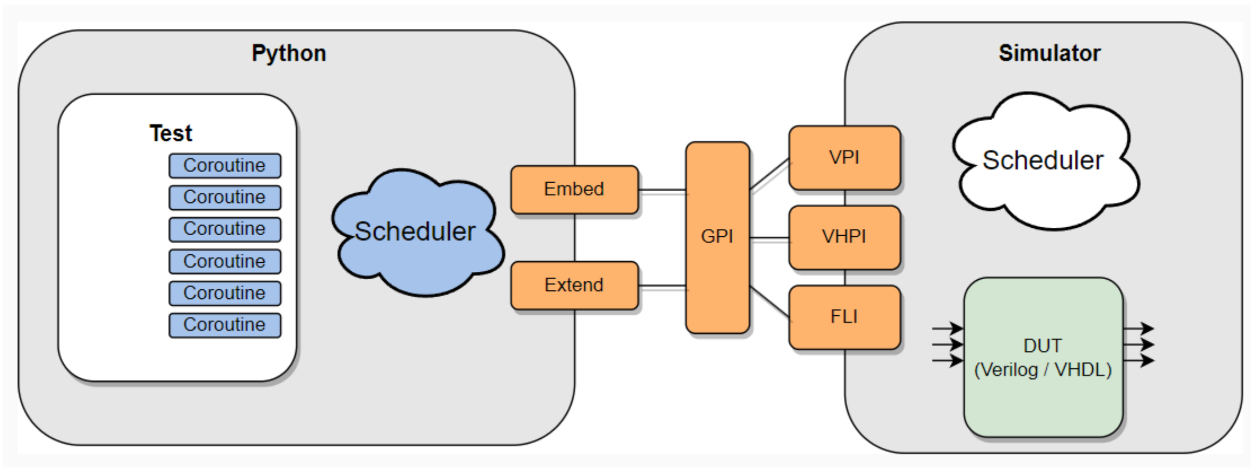
- bsg_fifo_1r1w_small_hardened_testbench.py
- bsg_fifo_1r1w_small_hardened_cov.sv
- bsg_fifo_1r1w_small_hardened_wrapper.sv
- Makefile

bsg_fifo_1r1w_small_hardened_testbench.py:

This is the main testbench that achieves the first purpose. It generates randomized data flits on the input side of the fifo, sends the data flit flow through the fifo, reads the data flit on the output side of the fifo and checks the correctness of the data flit. It also randomized the flow control on both sides of the fifo, so that the forward pressure and the back pressure of the fifo are constantly varying during the test, aiming to trigger more functionalities in the fifo.

The testbench is written in Python code, using the cocotb library. Here are the descriptions from cocotb.org:

```
Cocotb is a COroutine based COsimulation TestBench environment for verifying
VHDL and SystemVerilog RTL using Python.

A typical cocotb testbench requires no additional RTL code. The Design Under
Test (DUT) is instantiated as the toplevel in the simulator without any wrapper
code. cocotb drives stimulus onto the inputs to the DUT (or further down the
hierarchy) and monitors the outputs directly from Python. Note that cocotb can
not instantiate HDL blocks - your DUT must be complete.
```



```
A test is simply a Python function. At any given time either the simulator is
advancing time or the Python code is executing. The await keyword is used to
indicate when to pass control of execution back to the simulator. A test can
spawn multiple coroutines, allowing for independent flows of execution.
```

Writing a cocotb testbench is straightforward. Here is a tutorial for writing cocotb testbench: https://docs.cocotb.org/en/stable/writing_testbenches.html.

Below is a cocotb quick reference list:

- **async def**: Python coroutine definition, handle multiple DUT signals and/or run multiple sub-tests in parallel
- **@cocotb.test()**: Define testbench main function
- **cocotb.start_soon**: launch coroutines and run them in parallel
- **await**: wait for a coroutine to finish
- **Timer(n, units="ps")**: create a coroutine that delays for n ps
- **Clock(dut.clk_i, CLK_PERIOD, units="ps"):** Create a 10ps period clock on DUT port clk_i
- **await RisingEdge(dut.clk_i); await Timer(1, units="ps")**: Wait for the "rising edge triggered" coroutine to finish (wait for the next rising edge), then wait for 1ps before read/write DUT signals to eliminate ambiguity in simulation.
- **dut.signal_i.value = xxx**: write value xxx to DUT input port signal_i. Writes are not applied immediately, but delayed until the next write cycle. Similar to **<=** in Verilog.
- **dut.signal_i.setimmediatevalue(xxx)**: assign value xxx to DUT input port signal_i immediately, within the same cycle.
- **xxx = dut.signal_o.value**: read the value xxx on DUT output port signal_o.
- **xxx = dut.signal_i.value**: read the last-cycle-assigned value xxx on DUT input port signal_i. Note that the newly same-cycle-written value will not be read correctly.
- **dut._log.info**: print log
- **assert**: useful for correctness checking

bsg_fifo_1r1w_small_hardened_cov.sv:

This is the functional coverage definition file that achieves the second purpose. We define all the functionalities (legal states) we would like to test in the fifo, and also define all the impossible functionalities (illegal states) that will never happen in the fifo.

When running the tests, the simulator will monitor all the states inside DUT. It will throw an error if any illegal state is encountered. At the end of the simulation, it will check if all the legal states have been covered in the test. The goal is to achieve 100% coverage, which means all the functionalities have been triggered in the test.

It is the designer's responsibility to ensure both the main testbench and the coverage definition file are implemented correctly:

- If the main testbench has a bug, the fifo might have sent wrong data flits but the testbench could not catch them, even though the test passed.
- If the coverage file has a bug, the fifo might have legal states that have not been encountered, even though the coverage result shows 100%.

You could build a testbench_testbench to verify the correctness of the main testbench, but this seems like a recursive problem.. It is hard to prove our DUT is bug free. Writing a verification file is like implementing the same DUT in another way / language. 100% functional coverage does not mean our DUT is 100% correct, it actually means there is a higher chance that our DUT is going to work correctly.

Although cocotb-coverage also supports simple coverage functions, however, in SystemVerilog **covergroup**, **coverpoint** and **cross** are unique language constructs, and there is no straightforward equivalence between these constructs and cocotb-coverage objects. **CoverPoint** and **CoverCross** are rough substitutes for **coverpoint** and **cross**, but **covergroup** has to be defined implicitly. In the example code I wrote the coverage file in SystemVerilog for simplicity. Later parts of this doc will introduce how to write a coverage file in SystemVerilog from scratch.

bsg_fifo_1r1w_small_hardened_wrapper.sv:

This is a wrapper module that instantiates the DUT module with certain parameters. It also binds the coverage module to the DUT module. FSDB waveform file is dumped to support Verdi.

The best practice is to run the test multiple times with different sets of reasonable DUT parameters. The example code only tests one set of parameters.

Make sure to choose an interesting set(s) of parameters that are worth verifying (for example, it is not interesting to choose els_p=1 for a fifo).

Makefile:

Filelist and include paths are defined in the Makefile. If you would like to verify another module, you will need to update the VERILOG_SOURCES and EXTRA_ARGS, as well as modifying the TOPLEVEL and MODULE accordingly.

Project Begins

Step 1: Design a coverage module to measure coverage

Surprisingly, the first step in verifying the hardware is not to write the test. It is to write the code that measures our coverage. =) With that code in place, it is easier to think about how to efficiently write a test that attains the desired coverage.

First, we must decide what signals we will cover. If we think of hardware as a function, its outputs and next states are a function of the inputs and the current internal state (i.e. all registers and SRAM inside the module). **So if we can cover all of the important combinations of the inputs and the current state, then we have attained very good**

coverage of the whole system. Moreover, generally, but not always, these will be relatively independent of each other, reducing the amount of work that we have to do in order to prove that it's okay that our test did not reach particular states. Later in this section, we will show you how to handle cases where certain combinations of inputs are invalid; or where certain internal states are impossible to reach.

A ***covergroup*** is a SystemVerilog user-defined type that encapsulates the specification of a coverage model. Our goal is to, with the minimum amount of effort, define a set of covergroups that cover the valid state and input space of the hardware. It might be hard to specify everything in a single covergroup, especially for complicated modules that have hundreds of lines. Instead we could partition it into multiple smaller covergroups based on *operating states* (our term) of the module, which correspond to our own internal reasoning about the module.

Since `fifo_1r1w_small_hardened` is a fifo, we could imagine that there would be 4 operating states: `fifo_reset, fifo_empty, fifo_full` and `fifo_normal`. In this way, it is easier to decompose the behavior of the module into cases and determine which cases are legal.

In each covergroup, **`coverpoint`** s are specified to record the occurrence of valid values (bins) for all input signals & internal registers. Then we do a **`cross`** (cross product) on bins of coverpoints to cover all possible cases.

For each coverpoints and the cross-product, some bins are impossible to hit during the test due to the functionality of the module. Those bins should be specified as "illegal_bins" so that they are excluded from functional coverage analysis. Bins that are possible to hit but have zero impact on the functionality of the module could be specified as "ignore_bins". Generally, assertions that check invalid conditions in the code will have corresponding illegal_bins in the covergroup.

Both ignore_bins and illegal_bins can be used to ignore impossible status, the only difference is that when a possible state is encountered in simulation but ignored in the coverage file, illegal_bins will assert an error while ignore_bin will not. Generally, illegal_bins is safer because it will assert error when the coverage file designer accidentally ignores a possible state (no legal state should be ignored).

Every illegal_bin or ignore_bin must have a comment that explains why these states do not need to be covered. The comment should be enough for an external reviewer to easily understand. We do not mark something illegal or ignore simply because we are unable to get to the state with our random test. =)

At the end of the coverage module, we instantiate new instances for all covergroups with the syntax "`<inst_name> <covergroup_name> = new;`". If we don't create new instances, the tool will not sample the bins of covergroups.

Coverage results are printed out at the end of the simulation. By default the tool reports overall coverages of each covergroup, which is usually too optimistic because coverpoint results are also taken into account. It is better to report cross-product bins coverage only to better reveal the quality of functional coverages.

Check out the example code bsg_fifo_1r1w_small_hardened_cov.sv, and see how it is written.

Typically, we will bind this module to the DUT in our testbench. The input ports of this module should have the same port / wire / reg names as inside the DUT; but when you instantiate the module, you can use hierarchical naming (i.e. foo.x.bar) to reach into internal modules to access state.

On hierarchy and scalability

Should we x-ray all the way into multiple levels of sub-modules to access all internal registers? Perhaps. Random verification is a great time saver, but as the number of states starts to grow past 2^32, we are getting past the seconds mark for a small test to run (okay); and after 2^48, we are days. Similarly, there may be interactions between the state spaces of modules that are very vague to define and exclude from the target covered space.

At some point, we will have to rely on hierarchy for time reasons. Sub-modules are verified independently (not hard to do, if we have a methodology where we verify as we add modules to BaseJump STL) and upper-level modules can use interface signals (both inputs and outputs) for functional coverage analysis. Since we generally try to use latency-insensitive design, covering handshakes with preverified instantiated modules instead of all internal state, should be a reasonable proxy for correctness.

Step 2: Write a good testbench

An ideal testbench should do the following:

1. **Generate inputs** that cover all legal bins of coverpoints and cross-product of coverpoints.

2. **Verify the correctness of outputs** for the module, which is the basic requirement of a testbench. Use randomness if it simplifies the process.

3. **Check internal invariants, if they would not be exposed as an output** (for example, performance invariants (this always takes less than N cycles), or state invariants -- e.g., “the processor never modifies supervisor state if we execute only user mode instructions”, the internal register contains the max of all numbers so far) Ideally your tests can be structured so that you can verify the internal invariants just by looking at

outputs (i.e. add an instruction to read the internal register to the test bench), but it's not always possible.

Then we can run the testbench with our coverage metrics, and see how good coverage we attain.

Continuing our running example

To see what these coverage metrics look like, we will use the original poor-coverage testbench that we came up with before we used the BSG Coverage-Based Verification Methodology. Take a look at the test. It is actually quite complicated, despite not attaining good coverage!

(single clock speed)

```
---------------------- Functional Coverage Results ----------------------
Reset       functional coverage is 100.000000%
Fifo empty  functional coverage is 37.500000%
Fifo full   functional coverage is 0.000000%
Fifo normal functional coverage is 7.812500%
-------------------------------------------------------------------------
```

(multiple combinations of clock speeds)

```
---------------------- Functional Coverage Results ----------------------
Reset       functional coverage is 100.000000%
Fifo empty  functional coverage is 100.000000%
Fifo full   functional coverage is 75.000000%
Fifo normal functional coverage is 81.250000%
-------------------------------------------------------------------------
```

It does check the correctness of outputs (Property #2 of a good testbench!) perfectly. However, it does not cover all possible bins, even with multiple clock regions and different combinations of clock speeds! In fact, it did not cover fifo_full **at all** !

Non-100% coverage means some functionalities of the module are not tested at all, and such testbench can hide bugs and result in severe chip failure. Our goal is not to get 90% coverage and call it a day; our goal is to get to a 100% of what is coverable. Only then do we know that something is not lurking there.

Use ~~the force~~ randomness

Most of the time it is hard to think of how to write a testbench that covers all possible cases, especially for more complicated modules. Patching an existing testbench Whack-a-Mole style may eventually solve the problem, but the testbench could become super complicated and impossible to be modified.

Instead, we could generate random input values to the modules. Theoretically the functional coverage will converge to 100% with sufficient number of clock cycles. A randomized testbench can automatically cover all possible cases, including both legal and illegal ones. Developers should filter out illegal bins in the testbench.

The example code shows the testbench coded using a randomized input. In this testbench, input signals have half chance to be 0 and half chance to be 1. Illegal case “v_o==1 & yumi_i==0” is filtered out in the testbench. Using the same coverage program as before, we can run it and see what kind of coverage we get. Typically, you will periodically print out the coverage (say every N iterations), but eventually, you will just dial in the actual number of inputs that you need to get to coverage.

Turns out, with 4-elements, it does not take much randomness to get to 100% coverage: 350 insertions into the FIFO!

```
$make run
---------------------- Functional Coverage Results ----------------------
Reset       functional coverage is 100.000000%
Fifo empty  functional coverage is 100.000000%
Fifo full   functional coverage is 100.000000%
Fifo normal functional coverage is 100.000000%
-------------------------------------------------------------------------
```

Results above are achieved with `els_p==4`. Other configurations like `els_p==8` and `els_p==16` are also tested and converged with less than 50,000 packets sent.

Step 3: Achieve 100% coverage

Assume we have randomized testbench ready:

1. Set the number of iterations / clock cycles to a relatively large number, so that it has a larger chance to cover all possible bins.

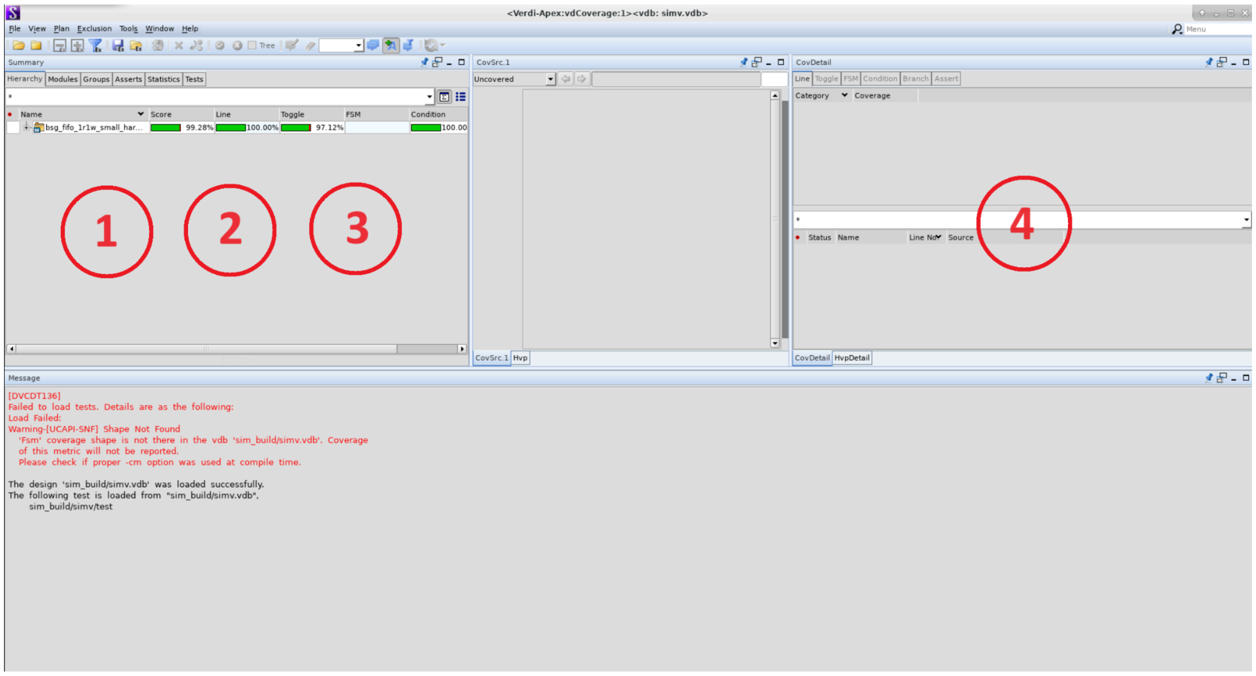
2. Filter out illegal bins from the testbench, until all tests pass without error.

3. When coverage is less than 100%, there are three possible causes:
    a. Runtime could be too short for the randomized testbench to cover all cases. Increase (double) the number of iterations until coverage number converges.
    b. (Most common reason) lack of illegal_bins / ignore_bins specifications for coverpoints / cross-products. Open up the GUI and check which bins are missed, then determine whether the bins are legal or not.

c. Testbench is not generalized enough and does not cover all cases. Open up the GUI and check which bins are missed, then add missing cases to the testbench.

We use Verdi (DVE substitute) to view the functional coverage results. Verdi is commonly used for opening .fsdb files to visualize simulation waveforms. Here we add -cov flag to open the automatically-generated .vdb file to show the coverages. Image below shows the GUI for coverage analysis:

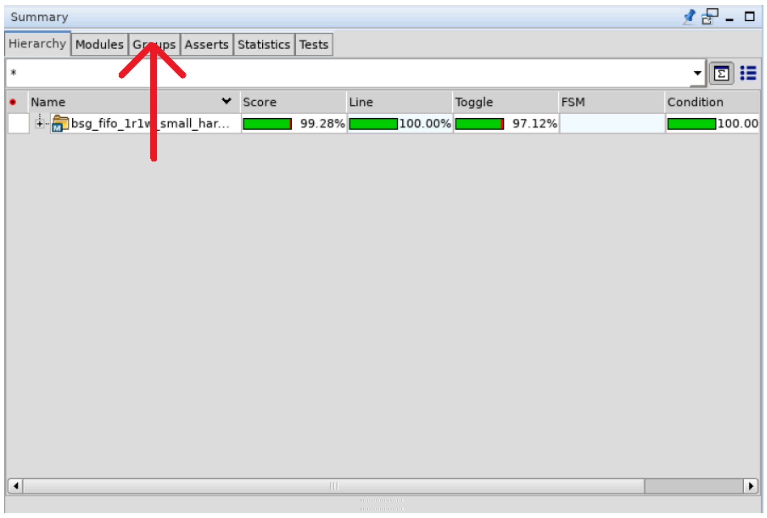
```
make ee-verdi-cov
```



<Verdi-Apex:vdCoverage:1><vdb: simv.vdb>
Summary
Hierarchy Modules Groups Asserts Statistics Tests
CovSrc.1
Uncovered
CovDetail
Line Toggle FSM Condition Branch Assert
Category Coverage
Status Name Line No Source
1
2
3
4
Message

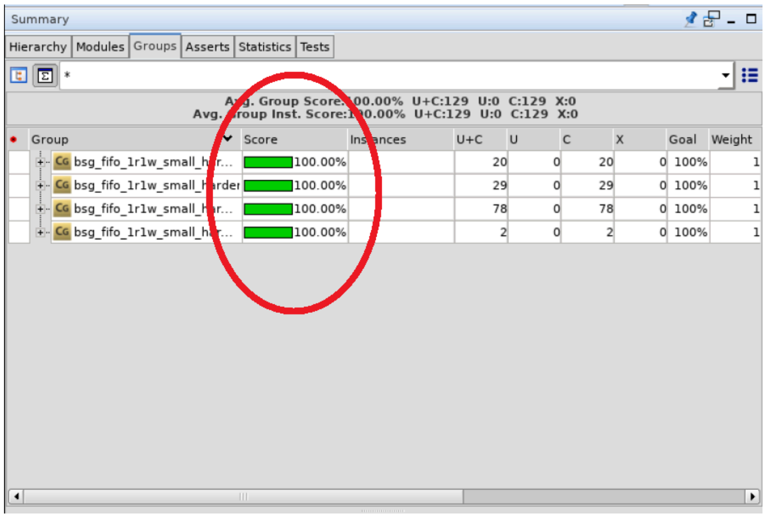
```
[DVCDT136]
Failed to load tests. Details are as the following:
Load Failed:
Warning-[UCAPI-SNF] Shape Not Found
  'Fsm' coverage shape is not there in the vdb 'sim_build/simv.vdb'. Coverage
  of this metric will not be reported.
  Please check if proper -cm option was used at compile time.

The design 'sim_build/simv.vdb' was loaded successfully.
The following test is loaded from "sim_build/simv.vdb",
    sim_build/simv/test
```

1. Click on the Groups tab to view covergroups



Name	Score	Line	Toggle	FSM	Condition
bsg_fifo_1r1w_small_har...	99.28%	100.00%	97.12%		100.00

2. **All covergroups should have achieved 100% coverage.**
    a. Double click each covergroup to view detailed results



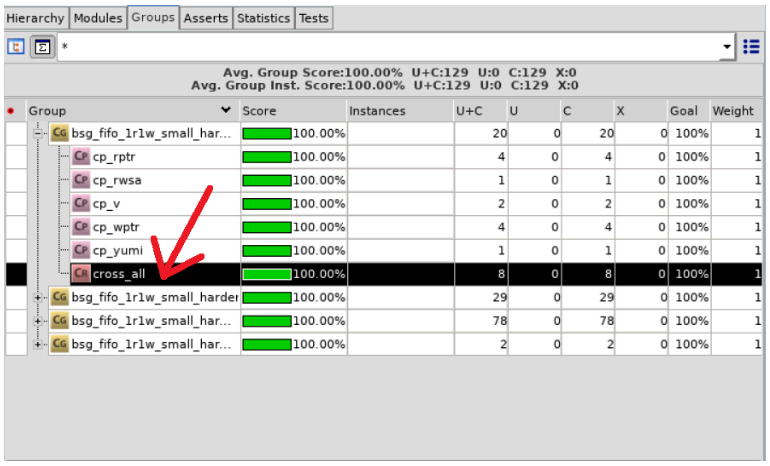
Summary

Hierarchy | Modules | Groups | Asserts | Statistics | Tests

Avg. Group Score:100.00% U+C:129 U:0 C:129 X:0
Avg. Group Inst. Score:100.00% U+C:129 U:0 C:129 X:0

Group	Score	Instances	U+C	U	C	X	Goal	Weight
bsg_fifo_1r1w_small_har…	100.00%		20	0	20	0	100%	1
bsg_fifo_1r1w_small_harder	100.00%		29	0	29	0	100%	1
bsg_fifo_1r1w_small_har…	100.00%		78	0	78	0	100%	1
bsg_fifo_1r1w_small_har…	100.00%		2	0	2	0	100%	1

3. **View cross-product coverages**
    a. Double click cross_all to view detailed coverages

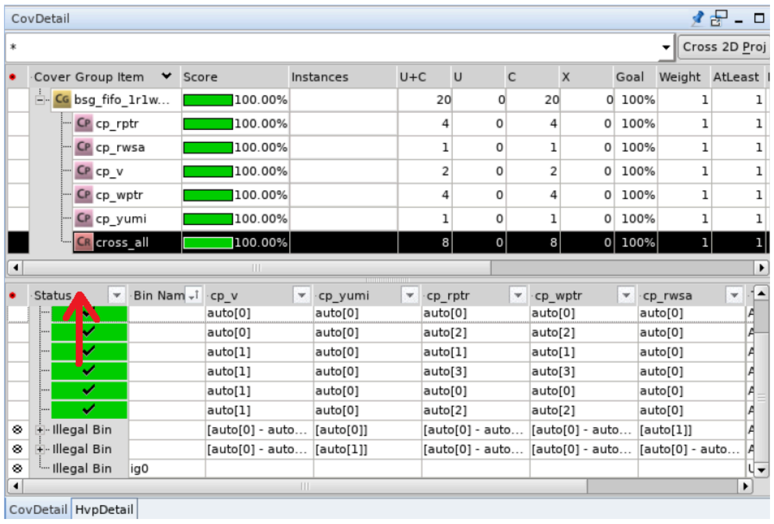


Hierarchy | Modules | Groups | Asserts | Statistics | Tests

Avg. Group Score:100.00% U+C:129 U:0 C:129 X:0
Avg. Group Inst. Score:100.00% U+C:129 U:0 C:129 X:0

Group	Score	Instances	U+C	U	C	X	Goal	Weight
bsg_fifo_1r1w_small_har…	100.00%		20	0	20	0	100%	1
cp_rptr	100.00%		4	0	4	0	100%	1
cp_rwsa	100.00%		1	0	1	0	100%	1
cp_v	100.00%		2	0	2	0	100%	1
cp_wptr	100.00%		4	0	4	0	100%	1
cp_yumi	100.00%		1	0	1	0	100%	1
cross_all	100.00%		8	0	8	0	100%	1
bsg_fifo_1r1w_small_harder	100.00%		29	0	29	0	100%	1
bsg_fifo_1r1w_small_har…	100.00%		78	0	78	0	100%	1
bsg_fifo_1r1w_small_har…	100.00%		2	0	2	0	100%	1

4. **Detailed coverage analysis**
    a. Status block turns red when some legal bins are missing in the test (hit count == 0). To achieve 100% coverage all blocks should be green (except for illegal bins and ignore bins).



CovDetail

Cover Group Item	Score	Instances	U+C	U	C	X	Goal	Weight	AtLeast
bsg_fifo_1r1w...	100.00%		20	0	20	0	100%	1	1
cp_rptr	100.00%		4	0	4	0	100%	1	1
cp_rwsa	100.00%		1	0	1	0	100%	1	1
cp_v	100.00%		2	0	2	0	100%	1	1
cp_wptr	100.00%		4	0	4	0	100%	1	1
cp_yumi	100.00%		1	0	1	0	100%	1	1
cross_all	100.00%		8	0	8	0	100%	1	1

Status	Bin Nam	cp_v	cp_yumi	cp_rptr	cp_wptr	cp_rwsa
✓		auto[0]	auto[0]	auto[0]	auto[0]	auto[0]
✓		auto[0]	auto[0]	auto[2]	auto[2]	auto[0]
✓		auto[1]	auto[0]	auto[1]	auto[1]	auto[0]
✓		auto[1]	auto[0]	auto[3]	auto[3]	auto[0]
✓		auto[1]	auto[0]	auto[0]	auto[0]	auto[0]
✓		auto[1]	auto[0]	auto[2]	auto[2]	auto[0]
Illegal Bin		[auto[0] - auto...	[auto[0]]	[auto[0] - auto...	[auto[0] - auto...	[auto[1]]
Illegal Bin		[auto[0] - auto...	[auto[1]]	[auto[0] - auto...	[auto[0] - auto...	[auto[0] - auto...
Illegal Bin	ig0					

Of course, 100% coverage does not mean bug-free! The tests also need to be self-checking and pass without error for all inputs. =)

Step 4 (Optional for this warm up project): Adjust configurations (parameters) of target module

So far we achieved 100% coverage and passed all tests, but it does not mean the target module is fully verified. Indeed, we only verified the module **with one set of parameters**. For example, a fifo can have larger/smaller data width, or can hold more/less elements. Other configurations should also be tested.

Usually developers may only test the specific set of parameters that they are going to use immediately. This saves time for now but may cause bigger trouble in the future when others try to instantiate the module with different parameters. Frankly speaking it could be hard to enumerate all combinations of parameters, but it is feasible to list all corner cases and test them all. One possible way is to enumerate parameters in the Makefile and pass them down to the testbench (add flag `-pvalue+<param_name>=<value>`), so that all cases can run in parallel.

Note that if clock domain crossing (CDC) happens inside the target module, different combinations of clock speeds are needed in the testbench.