Functional coverage you can read

This page answers: *how do I find out what my random stimulus reached, and how do I read the numbers cocotb-coverage gives me?*

In DD104′s Coverage as a question, functional coverage was a list of questions that you kept yourself (have I run `SUB` with two different operands? have I taken a backward branch?), and you could answer each of them with a `for` loop and a set. cocotb-coverage 2.0 keeps the same kind of list in Python. You name the values you care about, and it counts how often the stimulus produces each one. From Lab 03 on, each run in the labs writes those counts to a file, and `make cov` merges the files of all your runs and reports them. In Lab 03, Part 1 you'll see `router_rand_test` pass and catch neither of the router bugs that belong to that lab, B07 and B08. In Lab 03, Part 2 you write coverpoints for the router's packets, and their counts show you the holes in the test's stimulus where those two bugs are.

cocotb-coverage, like pyvsc, is in the course venv that Lab 03, Part 1 makes. The example for this page, `guide-examples/functional-coverage-you-can-read/`, sends random vectors to SimProc's ALU, the design you tested in Lab 01, and samples each vector.

Coverpoints, bins and crosses

Here is the coverage that the example collects for each vector it sends to the ALU. It counts the operations, three kinds of value for `A`, and whether the two operands were equal:

```
OPS = ["add", "sub", "or_op", "nand_op", "shl", "shr"]


def in_range(value, rng):
    """rel= for a bin written as (low, high)."""
    return rng[0] <= value <= rng[1]


@CoverPoint("alu.op", xf=lambda op, a, b: op, bins=list(range(6)), bins_labels=OPS)
@CoverPoint("alu.a", xf=lambda op, a, b: a, bins=[(0, 0), (1, 254), (255, 255)], rel=in_range,
            bins_labels=["zero", "other", "ones"])
@CoverPoint("alu.eq", xf=lambda op, a, b: int(a == b), bins=[0, 1], bins_labels=["differ",
"equal"])
@CoverCross("alu.x_op_eq", items=["alu.op", "alu.eq"])
def sample(op, a, b):
    pass
```

Each **bin** is one question, and its count is the number of samples that answered yes. A **coverpoint** groups the bins for one value. In cocotb-coverage a coverpoint is a decorator on a function, here `sample()`, and every call to that function is one sample. The test calls `sample(op, a, b)` once for every vector it sends. At each call, the coverpoint's `xf=` function takes the value out of the call's arguments, and the coverpoint adds one to the count of the bin that the value matches. A **cross** makes one bin for every combination of the bins of the coverpoints it crosses, so `alu.x_op_eq` has 6 × 2 = 12 bins, named by the pairs of labels they stand for, from `('add', 'differ')` to `('shr', 'equal')`. DD104 asked whether `SUB` and `NAND` had been run with two different operands, and this cross asks that of every operation. The cross has to go below the coverpoints it crosses, because at each call the decorators run from the top down, each one counting its value before it calls the one below it, and the cross reads which bin each of its coverpoints has just counted.

Every name is a dotted path, and the part before the last dot names a **group**. `alu` is the group of everything whose name starts with `alu.`, and its counts of bins and of bins hit are the sums of theirs.

SimProc programs use equal operands on purpose, because `SUB rX, rX` is how you zero a register and `NAND rX, rX` is how you write NOT. Equal operands also hide two of Lab 01's ALU bugs. With `A == B`, `SUB` gives 0 whichever way round it subtracts (A01), and a `NAND` that computes NOR (A02) still gives the right answer, because `~(A & A)` and `~(A | A)` are both `~A`.

Ranges and rel=

cocotb-coverage puts a value in a bin when `value == bin`. That's fine for single values, but no number is equal to a tuple, so a bin written as a range such as `(1, 254)` needs a relation function, given as `rel=`, that says what "in the bin" means. `alu.a` has one, `in_range()`. The example also has a coverpoint with two single values and a range written as a tuple between them, and no `rel=`:

```
@CoverPoint("quirk.a_no_rel", xf=lambda op, a, b: a, bins=[0, (1, 254), 255],
            bins_labels=["zero", "other", "ones"])
```

After 300 random vectors, `CovPlainTest` prints its counts:

```
[COV] quirk.a_no_rel: {'zero': 1, 'other': 0, 'ones': 1}
```

Two values of `A` landed in the single-value bins, and the other 298 matched no bin at all, so `other` stayed at 0, with no warning. You find a mistake like this only by reading a bin's count against what you know the test sent. A `rel=` applies to every bin of its coverpoint, so write every bin as a range, `(0, 0)` included, as `alu.a` does. In Lab 03, Part 2 you run into the same thing with the router's level coverpoint.

coverage.xml and make cov

Each test in the example ends by writing every bin's count to `coverage.xml` in its run folder:

```
coverage_db.export_to_xml("coverage.xml")
```

`make cov` finds every `coverage.xml` under `build/runs`, merges them with cocotb-coverage's own `merge_coverage()`, and prints each group, coverpoint and cross with the bins that were never hit. Here is what it printed after one run of `CovPlainTest`, which sends 300 vectors with `A` and `B` uniformly random:

```
=========================================================================
 COVERAGE   1 run, merged by cocotb-coverage into build/coverage/merged.xml
=========================================================================
  top                                          81.48 %   22 of 27 bins
    top.alu                                     82.61 %   19 of 23 bins
      top.alu.op                                 100.00 %   6 of 6 bins
      top.alu.a                                  100.00 %   3 of 3 bins
      top.alu.eq                                 100.00 %   2 of 2 bins
      top.alu.x_op_eq                             66.67 %   8 of 12 bins
         not hit: ('sub', 'equal'), ('or_op', 'equal'), ('nand_op', 'equal'), ('shr', 'equal')
    top.quirk                                   75.00 %   3 of 4 bins
      top.quirk.a_no_rel                         66.67 %   2 of 3 bins
          not hit: other
      top.quirk.op_legal                         100.00 %   1 of 1 bins
          not hit: FAIL
=========================================================================
```

`top` comes from cocotb-coverage's `export_to_xml()`, which puts every group under one group of that name. `quirk` holds the example's other two items, `a_no_rel` from the section above and `op_legal` from the one below, in a group of their own so that they don't change `alu`'s counts.

In `alu`, every operation and every kind of `A` came up, but only `ADD` and `SHL` had a vector with equal operands, so four of the twelve cross bins are empty. Only 2 of the 300 vectors had `A == B`, which is about what uniform random operands give, because `B` matches `A` in one vector in 256. `CovBiasedTest` makes `A == B` in one vector in eight, and one run of it filled the cross:

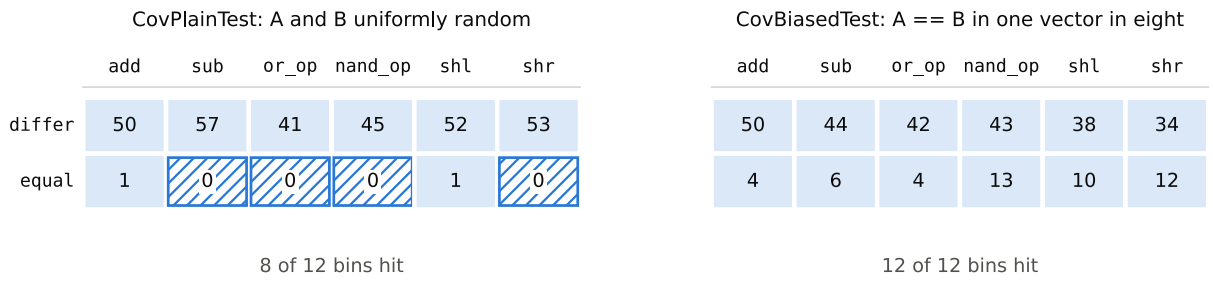


Figure 1: The cross alu.x_op_eq as a grid of counts after CovPlainTest, with four empty bins in the equal row, and after CovBiasedTest, with all twelve hit.

The `[COV] x_op_eq` lines of the two runs, as grids. A hatched bin is empty.

`make cov` names only the empty bins. The merged file, `build/coverage/merged.xml`, has every bin with its count, and Lab 03, Part 2 has you read the length bins' counts from it with `grep`.

The percentage

cocotb-coverage's `82.61 %` for `alu` is the number of bins hit divided by the number of bins, 19 of 23, with every bin of every coverpoint and cross under the group counted the same. The SystemVerilog standard works it out another way, and gives a group the average of its coverpoints' and crosses' coverage. For `alu` that's the average of 100, 100, 100 and 66.67%, which is 91.67%. The two differ because the cross has 12 of the group's 23 bins, so it makes up more than half of cocotb-coverage's number, where in the standard's average it is one item of four. `top` comes to 81.48%, 22 of 27, because it counts `quirk`'s four bins too.

Lab 03, Part 2 does the same sums for the router's 41 bins, which come to 46.34% by cocotb-coverage's formula and 61% by the standard's. Twelve of those bins can't be hit by any test in that lab, and they count in both percentages, so in Lab 03 neither percentage can reach 100%, however good your stimulus is. The list of bins that were never hit is the same whichever formula you use, so read that list to find the holes, as you did above for the four `equal` bins.

Ignore bins and illegal values

In cocotb-coverage a coverpoint has no ignore bins. A value that matches none of its bins isn't counted anywhere, so the way to ignore a value is to leave it out of the bins. SimProc's ALU has no operations 6 and 7, and `alu.op` has no bins for them. A cross can ignore combinations of bins. `CoverCross` takes `ign_bins`, a list of label pairs such as `("full", "empty")`, and leaves those bins out of the cross and out of its count. In the course's reference testbench for the router, `labs/ref`, one cross has the seven combinations that the router can't produce in its `ign_bins`, so that cross has 13 bins, and the reference testbench's regression hits all 13. Lab 03 leaves its impossible bins in, and has you work out from the specification which ones they are.

A value that should never happen at all goes in a `CoverCheck`. At each sample it counts a failure if its `f_fail` function returns true, and otherwise a pass if its `f_pass` function does. The example's check fails on an operation the ALU doesn't have, and its `f_pass` passes every other sample:

```
@CoverCheck("quirk.op_legal", f_fail=lambda op, a, b: op > 5, f_pass=lambda op, a, b: True)
```

`CovIllegalTest` sends `CovPlainTest`'s kind of vectors and then one vector with `ALUop` 6. Among its `[COV]` lines are:

```
[COV] quirk: 2 of 4 bins, 50.00%
[COV] quirk.a_no_rel: 2 of 3 bins, 66.67%
[COV] quirk.op_legal: 0 of 1 bins, 0.00%
```

and the check's counts, `{'PASS': 300, 'FAIL': 1}`. The run still ended `ok`, because a `CoverCheck` raises no error. The check is one of `quirk`'s four bins, and a check that has failed counts as an empty bin, so `quirk` is at 2 of 4 here, where `CovPlainTest`'s run had 3 of 4. If the check were in `alu`, its failure would take a bin off `alu`'s count in the same way. Then `make cov` merged all three runs:

```
    top.quirk                                  100.00 %   4 of 4 bins
      top.quirk.a_no_rel                         66.67 %   2 of 3 bins
          not hit: other
      top.quirk.op_legal                        200.00 %   2 of 1 bins
```

The merge counted the check's `PASS` and `FAIL` as two bins hit, out of a size of 1, so a check that failed came out at 200%. That lifted `quirk` to 100%, with `other` still empty below it. So read a check by its `FAIL` count. If an illegal value should fail the test, give the scoreboard a check of its own for it, rather than leaving it to the coverage. (In the merged report of the passing runs, `make cov` lists the check's `FAIL` bin as "not hit", which for a check is the result you want.)

Three kinds of empty bin

A bin can be empty for one of three reasons. Some bins can't be hit by any test, because the specification rules them out. In Lab 03, seven of the router's cross bins are like that. Three of them, such as `('full', 'empty')`, pair a FULL drop with a level of 8 or less, and the router drops a packet as FULL only when the output its route names holds more than 8 beats. The other four pair a forwarded packet with a level of 9 or more, and the router admits a packet only at a level of 8 or less. Some bins can be hit, but not by the testbench you have, because of how it was built. Lab 03's tracker takes output 1's level as 0, for example, so `('out1', 'low')` stays empty until Lab 04 gives output 1 its real level, and no test in Lab 03 asks for a FULL drop at all. The rest are holes in the stimulus, like the `equal` bins above, and those are the ones worth your time, because a bug can sit in them. Lab 03, Part 2 asks you to sort the router's 41 bins into these three groups, and both of the lab's planted bugs are in bins of the third.

Check yourself

1. A coverpoint for an output's level has `bins=[0, (1, 7), 8, 9, (10, 16)]` and no `rel=`. A run's tracker says the levels went up to 12. Which bins can be hit, and what do you change?

   Answer: A1, in Answers at the end.

2. After a regression, `make cov` says a group is at 100%, but one bin under it is listed as not hit. How can that be?

   Answer: A2, in Answers at the end.

3. After `CovPlainTest`, `make cov` reports `19 of 23 bins` for `alu`. Which of the four empty bins would more seeds of the same test fill?

   Answer: A3, in Answers at the end.

Further reading

For cocotb-coverage's API, go to its own documentation. ChipVerify's coverage pages explain the same ideas in SystemVerilog's covergroup syntax. Where they give a percentage, it's the standard's average, not cocotb-coverage's share of bins hit.

- cocotb-coverage's Tutorials, for `rel=` and `ign_bins`, under "Translating SystemVerilog Constructs to cocotb-coverage". Correction: in "Using CoverCheck as Assertions", the callback that should fail the test is a `lambda` with a `raise` statement in it, which isn't valid Python. Write it as a function with `def`.
- cocotb-coverage's Reference, for `CoverPoint`, `CoverCross`, `CoverCheck` and `merge_coverage()`. It says that a failed `CoverCheck` is set to 0 "permanently". That holds within one run; after `merge_coverage()`, the example's failed check came out at 200%.
- ChipVerify's Covergroup and Coverpoint, for what bins are for.
- ChipVerify's Cross Coverage. On that page, a value that falls in `illegal_bins` raises an error at run time. cocotb-coverage's nearest thing, a `CoverCheck`, raises none, as `CovIllegalTest` shows.

Next: Lab 03: Constraints and coverage.

Answers

A1. Check yourself

Only the three written as numbers, 0, 8 and 9. No level is equal to a tuple, so the two range bins stay empty, with no warning. Write every bin as a range, `(0, 0)`, `(1, 7)`, `(8, 8)`, `(9, 9)` and `(10, 16)`, and give the coverpoint a `rel=` that tests `low <= value <= high`.

A2. Check yourself

Something under the group counts more than its size. In the example, a `CoverCheck` that failed in one run of three came out as `2 of 1` after the merge, and that extra bin made up for the empty `other`. Read each coverpoint, cross and check on its own, and read a check by its `FAIL` count rather than by the group's percentage.

A3. Check yourself

All four, but slowly. They are `equal` bins, and only about one vector in 256 has equal operands, so each operation gets one about once in every 1,500 vectors. A test that makes `A == B` on purpose, as `CovBiasedTest` does, fills them in one run.