# Functional coverage you can read

This page answers: *how do I find out what my random stimulus actually reached, on a simulator whose coverage percentage doesn't follow the standard?*

In DD104, 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. A **covergroup** is the same kind of list, written in SystemVerilog. You name the values you care about, the simulator counts how often the stimulus produces each one, and the counts are written to a file that `verilator_coverage` can merge and report on. In Lab 03, Part 2 you write one for the router's packets. `router_rand_test` passes there and catches neither of the two bugs planted in that lab's router, and your covergroup shows you two holes in the test's stimulus, which is where those bugs are.

On Verilator 5.050 the count in each bin is right, but the percentage is worked out with a different formula from the standard's, `get_coverage()` returns 0, an array of bins on a coverpoint that's in a cross becomes a single bin, and `ignore_bins` still count as bins. This page shows the form of covergroup in a class that Verilator 5.050 supports, where its counts are written, and how to read them. The example, `guide-examples/functional-coverage-you-can-read/`, samples random ALU vectors.

## Coverpoints, bins and crosses

Here is a covergroup for ALU vectors. It counts the operations, three kinds of value for `A`, and whether the two operands were equal:

```
covergroup cg_alu with function sample(int op, bit [7:0] a, bit eq);
  cp_op: coverpoint op {
    bins add = {0}; bins sub = {1}; bins or_op = {2};
    bins nand_op = {3}; bins shl = {4}; bins shr = {5};
  }
  cp_a: coverpoint a {
    bins zero  = {8'h00};
    bins ones  = {8'hFF};
    bins other = {[8'h01:8'hFE]};
  }
  cp_eq: coverpoint eq {
    bins differ = {0};
    bins equal  = {1};
  }
  x_op_eq: cross cp_op, cp_eq;
endgroup
```

Each **bin** is one question, and its count is the number of samples that answered yes. A **coverpoint** groups the bins for one value, and a **cross** makes one bin for every combination of two coverpoints' bins. `x_op_eq` therefore has 6 × 2 = 12 bins, from `add_x_differ` to `shr_x_equal`. It asks DD104's question about operands for every operation, with a `differ` bin that counts the vectors with two different operands and an `equal` bin that counts the vectors with `A == B`.

SimProc programs use equal operands on purpose, because `SUB rX, rX` is how you zero a register (nearly every program does it) and `NAND rX, rX` is how you write NOT. Equal operands also hide two of the ALU bugs from Lab 01. 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`.

The covergroup takes its values as arguments of `sample()`, and whoever calls `sample()` passes them in. Most examples of a covergroup in a class are written another way, with coverpoints that read a member of the class, as in `coverpoint v.op`, and a call to `sample()` with no arguments. Verilator 5.050 doesn't support that form. The example's `CG_MEMBER` build writes `cg_alu` that way, and Verilator prints a warning for it and stops:

```
%Warning-COVERIGN:         guide-examples/functional-coverage-you-can-read/cov_pkg.sv:48:25:
Unsupported: 'covergroup' coverpoint referencing enclosing class member; ignoring covergroup
'__vlAnonCG_cg_alu'
%Error: Exiting due to 1 warning(s)
```

Verilator ignores the whole covergroup. The course's build stops on any warning that it hasn't been told to ignore, so this build stopped after 4 s, and you find out from the build instead of from a coverage report with no bins in it. In Lab 03, Part 2 we have you write the router's covergroup this way first, so that you see the warning. Lab 04′s coverage subscriber, `router_coverage`, is written in the `with function sample` form because Verilator 5.050 ignores the other one.

## coverage.dat and make cov

By default, the course's `uvm.mk` builds every simulator with Verilator's `--coverage-user` option, and every run then writes the count of every bin to `coverage.dat` in its own run folder. `make cov` merges every `coverage.dat` under `build/runs` with `verilator_coverage`. It prints how many bins of each coverpoint and cross were hit, then lists by name every bin that was never hit, and it writes every bin with its count to `build/coverage/bins.txt`. Here is what it printed after one run of `cov_plain_test`, which samples 300 vectors with `A` and `B` uniformly random:

```
 verilator_coverage: covergroup : 82.9% (34/41)
-------------------------------------------------------------------------
 41 covergroup bins, 34 hit at least once:
  __vlAnonCG_cg_alu.cp_a                          3 of 3    bins hit
  __vlAnonCG_cg_alu.cp_eq                         2 of 2    bins hit
  __vlAnonCG_cg_alu.cp_op                         6 of 6    bins hit
  __vlAnonCG_cg_alu.x_op_eq                       8 of 12   bins hit
  __vlAnonCG_cg_arr.cp_alone                      4 of 4    bins hit
  __vlAnonCG_cg_arr.cp_crossed                    1 of 1    bins hit
  __vlAnonCG_cg_arr.cp_half                       2 of 2    bins hit
  __vlAnonCG_cg_arr.x_w_half                      2 of 2    bins hit
  __vlAnonCG_cg_ign.cp_op                         6 of 9    bins hit
-------------------------------------------------------------------------
 7 bins never hit:
    __vlAnonCG_cg_alu.x_op_eq.add_x_equal
    __vlAnonCG_cg_alu.x_op_eq.or_op_x_equal
    __vlAnonCG_cg_alu.x_op_eq.shl_x_equal
    __vlAnonCG_cg_alu.x_op_eq.shr_x_equal
    __vlAnonCG_cg_ign.cp_op.op[6]
    __vlAnonCG_cg_ign.cp_op.op[7]
    __vlAnonCG_cg_ign.cp_op.unused
```

Above the `covergroup` line, the summary also lists line, toggle and four other kinds of coverage, all at `0.0% ( 0/ 0)`, because the course's builds collect only covergroups. `__vlAnonCG_cg_alu` is the name Verilator gives the covergroup `cg_alu`.

The other two covergroups in the list, `cg_arr` and `cg_ign`, are in the example to show how Verilator counts bins, and the next section comes back to them. In `cg_alu`, every operation and every kind of `A` came up, but no `ADD`, `OR`, `SHL` or `SHR` vector had 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. `cov_biased_test` makes `A == B` in one vector in eight, and that fills the cross:

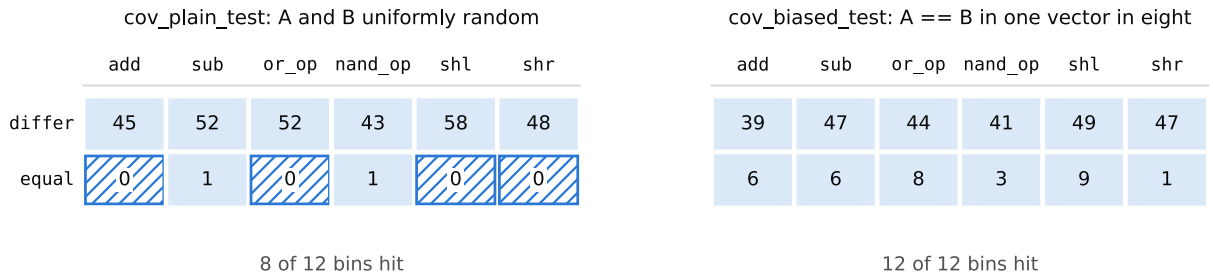



Figure 1: The cross x_op_eq as a grid of counts after cov_plain_test, with four empty bins in the equal row, and after cov_biased_test, with all twelve hit.

*The 12 bins of `x_op_eq` after one run of each test. With uniform operands, the `equal` row has a count of 1 for `sub` and `nand_op` and 0 for the other four operations. With `A == B` in one vector in eight, every bin is hit.*

## The percentage

Verilator's `82.9% (34/41)` is the number of bins hit divided by the number of bins, with every bin of every covergroup counted the same, crosses included. The standard works it out another way, and gives a covergroup the average of the coverage of its coverpoints and crosses. For `cg_alu` on its own, the test's report phase prints what the covergroup's two methods return:

```
[COV] cg_alu: get_inst_coverage() = 82.61, get_coverage() = 0.00
```

`get_inst_coverage()` uses Verilator's formula again, because 82.61% is 19 bins hit out of `cg_alu`'s 23. The standard's formula would average `cp_op`, `cp_a` and `cp_eq` (100% each) with `x_op_eq` (8 of 12, 67%), which gives 92%. `get_coverage()`, which the standard defines as the coverage of the covergroup type as a whole, returned 0.

The example's other two covergroups show two more differences, and these change the bins themselves:

```
covergroup cg_arr with function sample(int v, int w);
  cp_alone:   coverpoint v { bins v[] = {[0:3]}; }
  cp_crossed: coverpoint w { bins w[] = {[0:3]}; }
  cp_half:    coverpoint v { bins lo = {[0:1]}; bins hi = {[2:3]}; }
  x_w_half:   cross cp_crossed, cp_half;
endgroup

covergroup cg_ign with function sample(int op);
  cp_op: coverpoint op { bins op[] = {[0:7]}; ignore_bins unused = {6, 7}; }
endgroup
```

- An array of bins on a crossed coverpoint becomes one bin. `bins v[] = {[0:3]}` gave four bins on `cp_alone`, but the same array on `cp_crossed` gave a single bin, `w`, which counted all 300 samples, and so the cross `x_w_half` has 2 bins where the standard gives it 8. In a coverpoint that's part of a cross, write every bin out by name, as Lab 03 asks you to.

- `ignore_bins` stay in the count. The standard removes 6 and 7 from `cp_op`, which leaves 6 bins, all of them hit, so `cp_op` would be at 100%. Verilator keeps `op[6]` and `op[7]` as bins and lists the ignored set as a bin of its own, `unused`, so `verilator_coverage` counts 6 hit of 9. `get_inst_coverage()` gave a third answer, 75.00, which is 6 of 8.

In Lab 03, Part 2 you work the same sums for the router's covergroup, and get 46.3% by Verilator's formula and 61% by the standard's. Twelve of that covergroup's 41 bins can't be hit by any test in the lab, and they count in both percentages, so neither can reach 100% however good your stimulus is. The counts in `coverage.dat` don't depend on either formula, so to find the holes, read the list of bins that were never hit at the end of `make cov`'s output, as you did above for the four `equal` bins.

## 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. `op[6]` and `op[7]` above are two of them, since SimProc's ALU has only the six operations 0 to 5. The router's `full_x_low` in Lab 03 is another, since the router drops a packet as FULL only when the output its route names holds more than 8 beats. 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_x_low` stays empty until you fix that in Lab 04. 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 `cp_kind` has `bins k[] = {[0:3]}` and is crossed with a coverpoint that has two bins. How many bins does the cross have on Verilator 5.050, and what do you change?

   *Answer: A1, in Answers at the end.*

2. A run's report prints `get_coverage() = 0.00` for its covergroup. Was nothing covered?

   *Answer: A2, in Answers at the end.*

3. After `cov_plain_test`, `make cov` reports `82.9% (34/41)`. Which of the 7 empty bins would more seeds of the same test fill, and which can no test fill?

   *Answer: A3, in Answers at the end.*

## Further reading

ChipVerify's coverage pages explain the language well, but the numbers on them come from commercial simulators.

- Covergroup and Coverpoint. Correction: it says a 32-bit coverpoint with default bins would make billions of bins; the standard caps automatic bins at `auto_bin_max`, 64 by default, so it makes 64 range bins.
- Coverpoint Bins, for `bins x[]`, `bins x[N]` and `default`. It doesn't cover `ignore_bins` or `illegal_bins` on a coverpoint, or transition bins.
- Cross Coverage, for `binsof` and `intersect`. On Verilator 5.050 the percentage is computed differently (bins hit over all bins), and an array of bins on a crossed coverpoint becomes one bin.
- Functional Coverage. Correction: its `sample()` example constructs the covergroup with `CovGrp = new;`, which assigns to the type name; it needs an instance variable. It also repeats the claim that default bins make "one bin per value".
- UVM Functional Coverage, for a coverage subscriber. Its covergroup reads a member of the class, which Verilator 5.050 ignores (the COVERIGN warning above), so write yours `with function sample`, as Lab 04′s `router_coverage` is written.

Next: Lab 03: Constraints and coverage.

## Answers

### A1. Check yourself

Two, because `k[]` becomes a single bin on a crossed coverpoint. Write the four bins out by name (`bins k0 = {0};` `bins k1 = {1};` and so on), and the cross then has 4 × 2 = 8.

### A2. Check yourself

Not necessarily. In the example's `cov_biased_test` run, all 23 of `cg_alu`'s bins were hit, and `get_coverage()` still returned 0. Read the counts from `make cov` instead.

### A3. Check yourself

More seeds would fill the four `equal` bins, but slowly, because only about one vector in 256 has equal operands and those few are spread over six operations. `op[6]` and `op[7]` can't be hit, because SimProc's ALU has no such operations, and `unused` is the ignored set, which no sample is meant to hit.