Coverage as a question

This page answers: *how do I know when I have tested enough?*

You can't know that for certain, and no coverage number will tell you. Everything useful about coverage follows once you accept this, rather than looking for a way around it.

It helps to treat coverage as a tool that raises questions about your stimulus, rather than as a score to push up. Each question is only worth as much as the answer you give it, and the percentage the tool attaches to them is a side effect.

The quickest way to see this is to point a real coverage tool at the two stimulus sets from The lazy testbench and watch it find the blind spot without being told anything about adders.

What the coverage run prints

The Lab 01 package has a `coverage/` add-on that runs Verilator's toggle coverage on the six hand-picked cases and on all 512 vectors. Icarus has no coverage instrumentation, so this is the one place early in the course where only the stricter simulator, Verilator, can answer the question. You'll run it in Lab 01, Part 5, and this is what it prints:

```
======================================================================
 COVERAGE   what did the stimulus actually reach?
======================================================================

----------------------------------------------------------------------
 six hand-picked cases
----------------------------------------------------------------------
cov_six.dat
  design toggle points : 48
  never toggled        : 5
  design line points   : 0
  the ones that never moved:
    ripple4.sv:3  b[2]:1->0
    ripple4.sv:4  cin:0->1
    ripple4.sv:4  cin:1->0
    ripple4.sv:8  c[0]:0->1
    ripple4.sv:8  c[0]:1->0

        // 4-bit ripple-carry adder built from four full adders.
        module ripple4 (
%000002     input  logic [3:0] a, b,
%000000     input  logic       cin,
%000002     output logic [3:0] sum,
%000001     output logic       cout
        );
%000002     logic [4:0] c;
            assign c[0] = cin;

----------------------------------------------------------------------
 all 512
----------------------------------------------------------------------
cov_all.dat
  design toggle points : 48
```

```
  never toggled        : 1
  design line points   : 0
  the ones that never moved:
    ripple4.sv:3  a[3]:1->0

=================================================================
 Coverage did not say whether the adder is right.  It said which
 parts of it your stimulus never moved.  Different question.
=================================================================
```

Both runs use `../src/ripple4.sv`, the correct adder, so there is no bug here for anything to find. Coverage isn't looking for one, and that's the first thing to understand about it.

What it found

Start with the `%000000` on the `cin` line of the annotated source.

`verilator_coverage --annotate` rewrites your source with a count in the left margin and a flag for anything it considers under-covered: `%` when a point falls below `--annotate-min` (10 by default). You don't need the full legend, only the two facts on that line: the flag is `%`, and the count is `000000`.

The list above it says the same thing more plainly. Of the 48 toggle points in the design, five never fired, and two of them are `cin:0->1` and `cin:1->0`. The carry-in never moved in either direction during the whole run, and the tool reported it on the line of source where `cin` lives.

It did that without knowing what an adder is, without a reference model, without an assertion, and without being told `cin` was interesting. It asked one mechanical question of every bit in the design, *did this go 0⟶1, and did it go 1⟶0?*, and five of the answers were no.

That's what coverage is good for. It has no idea what the correct answer is, but it knows which parts of the design your stimulus never touched, and those untouched places are where bugs survive.

The other three cold points either follow from those two or are incidental. `c[0]` is the wire that `cin` is assigned to, so it never moved either. `b[2]:1->0` is a real hole, but an incidental one: none of the six cases happens to take that bit from 1 back to 0.

Three traps in that one output

1. `design line points : 0`

Line coverage is what most people mean by "coverage", and it's what software teams track. Here it found zero line points inside `ripple4.sv` and `fulladder.sv`, which means there were no lines to measure at all, not that none of them were covered. The design is made of continuous assignments, and a continuous assignment isn't a statement that executes, so line coverage has nothing to count.

A report saying "100 % line coverage" on this design would therefore be true but would tell you nothing. It would be just as true of the six-case run as of the exhaustive run, so it can't tell the two apart.

> A coverage number means nothing unless you name the metric. "We have 92 % coverage" is not a useful sentence. "We have 92 % statement coverage and have never measured functional coverage" is.

2. The stimulus files check nothing

Open `cov_six.sv`. It has no assertions at all: it drives six values and prints them, and that's the whole file.

Even so, it gets exactly the score that a version checking every result would get. Coverage instruments the design, so it can only measure what your stimulus reached; it has no way of knowing whether you looked at the answer. A testbench that drives every input, toggles every bit and asserts nothing would be 100 % covered and still useless. This happens in practice, too: it's what a large, old, unmaintained regression suite tends to drift into.

	Asserts nothing	Asserts properly
Poor coverage	Useless, and it looks it	Useless, and it looks fine
Good coverage	Useless, and it looks excellent	The only useful combination

The dangerous cell is the bottom-left one, because it's the one with a good number on it.

3. Even the exhaustive run has a hole

In the second block, all 512 vectors ran and one point still never moved: `a[3]:1->0`.

That doesn't mean anything went untested, because every input combination was applied. The sweep ends with `a = 15` and stops there, so the top bit of `a` goes up once and never comes back down. Toggle coverage counts transitions rather than values, which makes the order of your loops part of your stimulus.

The same package has a sharper version of this. `cov_all.sv` puts `cin` in the innermost loop on purpose. If you wrote the same sweep with `cin` outermost, 256 vectors at `cin = 0` and then 256 at `cin = 1`, you would visit every value of `cin` while toggling it only once, so `cin:1->0` would stay cold in a test that really is exhaustive.

> Treat a coverage hole as a question to answer. Uncovered does not mean untested, and covered does not mean correct, so find out which one you are looking at.

Some holes are real gaps, like `cin` in the six-case run, and the fix is to improve your stimulus. Some are artefacts of how the metric is defined, like `a[3]:1->0`, and you can note them and move on. Some are unreachable by construction, and those are the most valuable, because a point your design cannot reach is either dead logic to delete or a sign that part of your understanding is wrong. Chasing a number would have you paper over all three kinds the same way.

Functional coverage on a processor

Toggle coverage on a nine-line adder is a good teacher, because you can hold the whole design in your head. On SimProc it's nearly useless, because almost any program toggles nearly every bit, and you could sit at 99 % with `SUB` subtracting its operands the wrong way round.

What you need there is **functional coverage**: coverage of things you named yourself because they matter to the design's specification, as opposed to its wiring. You write that list yourself. For SimProc it starts like this:

The question	Why it is on the list
Have I executed every opcode?	Eleven exist; three encodings are unimplemented on purpose
Have I executed `SUB` with two different operands?	`SUB rX, rX` is symmetric and hides an operand swap
Have I executed `NAND` with two different operands?	`NAND rX, rX` is how you write NOT, the one case where NAND and NOR agree
Have I used all four shift amounts, in both directions?	Shifting by 1 is what everybody writes
Have I taken a backward branch?	One sign-extension bug still gets forward displacements right

The question	Why it is on the list
Have I taken and not taken every branch?	A branch never taken has never been tested
Have I run two branches off one compare?	Flags are supposed to survive a branch
Have I checked for one `done` pulse per retired instruction?	One bug disturbs nothing else
Have I run the UART at more than one baud rate?	One bug is invisible at the default rate

You could answer each of those today with a `for` loop and a set, without any coverage tool, because functional coverage is really a list you maintain. Writing the list is the engineering part; the counting afterwards is clerical work.

Read the table again and you'll see it is organised around the ways an operation can be symmetric, defaulted or one-sided, rather than around the design's structure. Those are the places where a wrong implementation and a right one give the same answer, and that is where the thirteen SimProc bugs live. Where bugs hide has the full taxonomy, and every entry in it is a hole that this kind of list is designed to notice.

> Never quote a coverage number without naming the metric and the assertions behind it. When you find a hole, ask a question about it instead of just closing it. Asking "Why did nothing ever set `cin`?" finds the bug, and asking "How do I get this to 100 %?" hides it.

Before Lab 01

You now have the four ideas that the rest of DD104 builds on:

1. A green tick is a claim exactly as large as the stimulus behind it.
2. A testbench drives, observes and decides, and it is only a testbench once you have seen it fail.
3. If you can walk the input space, walk it. When you can't, the shape of a failure still tells you where to look.
4. Coverage tells you where your stimulus has not been, but never whether the design was right in the places it has been.

Check yourself

1. A colleague reports "100 % line coverage" on a design made entirely of `assign` statements. What has been measured?
2. Why can a testbench with no assertions score full toggle coverage?

Next: Lab 01: The adder you already trust.