The lazy testbench

This page answers: *how careful do I actually have to be? My tests look reasonable.*

They probably do look reasonable, and that's the problem, because a test can look reasonable and still miss a bug. The cheapest place to see this happen is in a circuit you already understand completely.

You wrote a 4-bit ripple-carry adder in DD102 and measured it in DD103. It's nine lines long and you know exactly what it does, which makes it the last design under test in this course that you'll be able to say that about. It ships as `labs/lab-01-the-adder-you-already-trust/src/ripple4.sv`:

```
module ripple4 (
    input  logic [3:0] a, b,
    input  logic       cin,
    output logic [3:0] sum,
    output logic       cout
);
    logic [4:0] c;
    assign c[0] = cin;
    genvar i;
    generate
        for (i = 0; i < 4; i++) begin : slice
            fulladder fa (.a(a[i]), .b(b[i]), .cin(c[i]),
                          .sum(sum[i]), .cout(c[i+1]));
        end
    endgenerate
    assign cout = c[4];
endmodule
```

The Lab 01 package also ships `broken/ripple4.sv`, which is that file with one line changed:

```
    assign c[0] = 1'b0;             // <-- was: assign c[0] = cin;
```

With that change, the carry-in never reaches the bottom slice. From Lab 04 on we'll ask you not to read the broken designs, but you're allowed to read this one (the file says so at the top), because Lab 01 isn't a bug hunt. You already know where the bug is; the question here is what kind of testbench notices it.

It's a real kind of bug, too. A port that was tied off during bring-up and never reconnected is one of the most common defects in industrial RTL, and it survives review because the line looks deliberate.

The shipped testbench

The testbench, `test_adder.py`, ships with two tests. Whoever wrote the first one, `t01_the_obvious_six`, was thinking about addition:

```
    cases = [
        # a,  b,  cin
        (0,  0,  0),
        (1,  1,  0),
        (2,  3,  0),
        (7,  8,  0),
        (15, 0,  0),
```

```
        (15, 1,  0),
    ]
```

The list covers zero, one, a middling pair, a carry that walks between slices, the top of the range and the wrap. That's a defensible choice, and it's what most people come up with when asked for "a few good test cases".

The second test is the improvement you'd probably reach for next:

```
@cocotb.test()
async def t02_every_a_and_every_b(dut):
    """Every value of a against every value of b.  256 vectors."""
```

It runs every value of `a` against every value of `b`, which is the complete cross product, sixteen by sixteen. If someone asked you whether that testbench was thorough, you'd say yes, and you'd probably call it exhaustive. That answer would be wrong, and the reason it's wrong is worth remembering.

(These are excerpts only. The complete file is `labs/lab-01-the-adder-you-already-trust/test_adder.py`, and it ships in working order.)

What the two tests report

Run against the good adder, both tests pass, and the log reports how much of the input space they covered:

```
   262.00ns INFO     cocotb.ripple4                     checked 256 of the 512 possible
input vectors
```

Run against the adder in `broken/`, they pass as well. `make check` runs the testbench on both adders, and it passes only if the testbench passes the good one (`ok`) and fails the broken one (`caught`). A testbench that passes both gets `MISSED` and `VERDICT: FAIL`. This is what it prints for the shipped testbench, which you'll see for yourself in Lab 01, Part 2:

```
********************************************************************************
 CHECK   testbench: test_adder      simulator: icarus
********************************************************************************
      ok  src/  the good adder           2 tests, all passed              1.9s
  MISSED  broken/  carry-in dropped      2 tests, all passed              1.9s
--------------------------------------------------------------------------------
 VERDICT: FAIL -- your testbench passes the broken adder too.
   broken/ripple4.sv drops the carry-in on the way into slice 0, so
   every vector with cin = 1 comes out one too small. Every vector
   with cin = 0 is perfectly correct.
   Look at the third column of your stimulus. If cin is 0 in every
   row, this bug is not reachable by anything you drove -- and you
   drove 262 of the 512 vectors this module has.
********************************************************************************
```

`MISSED` means that every vector the testbench drove came out green on an adder that's broken. The verdict counts 262 vectors driven, 6 from `t01` and 256 from `t02`, but the six from `t01` are repeated inside the 256.

What “exhaustive” covers

Look again at the line the run printed: `checked 256 of the 512 possible input vectors`. The testbench is telling you itself that it covered only half the inputs. Four bits of `a`, four bits of `b` and one bit of `cin` make $4 + 4 + 1 = 9$ bits of input, and $2^9 = 512$. The cross product of `a` and `b` is complete over eight of those nine bits, which is why it reaches only 256 of the 512 vectors.

Careful people make this mistake too, because it happens whenever you list the inputs you happen to be thinking about. The author of `t02` was thinking “an adder takes two numbers”, which is what an adder is in every context except testing it. `cin` is plumbing rather than one of the numbers: it’s the port you use when you chain adders, and you don’t notice it unless you’re chaining them.

> Looping over your inputs makes you thorough only over the input list you wrote down, and that list came from the same mental model as the design, so it has the same blind spots. When you call a testbench “exhaustive”, you’re making a claim about an input space rather than about the testbench, and the claim is only as good as the space you chose.

If you add `cin` to the sweep and collect the failures instead of stopping at the first one, the test reports how many vectors are wrong. (The reference solution also renames `t02` to `t02_exhaustive`.) Now `t02` fails on the broken adder, so the testbench can tell the two adders apart, and the verdict is PASS:

```
  caught  broken/  carry-in dropped      2 tests, 1 FAILED               1.9s
        t02_exhaustive
              AssertionError: 256 of 512 vectors are wrong. First five: 0+0+1=0 (want 1),
0+1+1=1 (want
----------------------------------------------------------------------
 VERDICT: PASS
   512 vectors on the good adder, all correct.
   512 vectors on the broken one, and the failure is a count, not
   an anecdote:
      t02_exhaustive: AssertionError: 256 of 512 vectors are wrong. First five: 0+0+1=0 (want
1), 0+1+
   That is a test worth committing: it will still catch this bug in
   a year, on a rewrite, done by somebody who has never met you.
```

Making that change is what you do in Lab 01, and it takes four lines.

Reading the count

The fixed test doesn’t stop at the first disagreement. It walks all 512 vectors, counts every wrong one and reports the total, which here is exactly half.

Exactly half is an unlikely count for a random mistake; a scrambled carry chain or a bad XOR would give you some awkward fraction instead. A count of exactly 256 out of 512 tells you that the failures track a signal that’s high exactly half the time. In a space of $16 \times 16 \times 2$, `cin` has that shape, but so does each bit of `a` and `b`, so the count narrows the search, and the wrong vectors themselves do the rest.

The reference solution’s failure message lists the first few wrong vectors and then states the pattern outright, because it was written to:

```
Every wrong vector above has cin=1 and every one is short by exactly 1.
```

No vector is wrong with `cin = 0`, every vector is wrong with `cin = 1`, and each wrong one is short by one. That's enough to diagnose the bug, and you have it before you've read a line of RTL.

Shapes worth recognising

The count	What it usually means
Exactly half	The failure tracks one input that is high half the time
Exactly one quarter	Two such inputs, both high together
Exactly one	An edge: the maximum, the wrap, a single decoded value
All but a handful	The operation is fundamentally wrong; the survivors are cases where right and wrong happen to agree
An odd, un-round number	Probably a real functional bug rather than a wiring mistake; start reading

You'll meet the fourth row soon. One of the SimProc bugs swaps the operands of `SUB`, and of all 65,536 `(A, B)` pairs it gets 65,024 wrong. The 512 pairs that survive aren't random: `A - B` and `B - A` agree exactly when `A == B` (256 pairs) or when the difference is 128 (256 more). `SUB rX, rX`, the usual way to clear a register, is always among those 512, so the most common use of the instruction is one of the few that hide the bug.

When you can't walk everything

Nine input bits give 512 vectors, which is 512 ns of simulated time and about a hundredth of a second of real time. When you're in that situation, take advantage of it.

> If the input space is small enough to walk, walk it. Don't hand-pick cases for a combinational block with fewer than about 2^{20} input combinations, because you won't choose better than a `for` loop that tries everything.

SimProc's ALU is still small enough to walk, at 6 operations × 256 × 256 = 393,216 vectors. You'll do that in Lab 03, where brute force turns up three bugs.

The processor around the ALU isn't. Its four registers, 64 bytes of memory, program counter, two flags and UART make a state space far too big to walk, and because its behaviour depends on sequences, enumerating states wouldn't be enough even if you could. From Lab 04 on, you need the two things that replace exhaustive testing:

- a reference model to compare the design against, instruction by instruction: The golden model;
- a way to ask what your stimulus actually reached, which is not the same as counting how many tests you wrote: Coverage as a question, next.

Two habits

> Sweep the input space instead of sampling it, and count the space first. Add up the input bits, raise two to that power, and make your testbench print how many of them it checked. A line like `256 of the 512 possible input vectors` is enough on its own to catch this bug.

> Read the count before you read the values. A count of exactly half, exactly one or exactly 2^k points at the signal the failure tracks, and it gets you there faster than a waveform will.

Check yourself

1. A module has an 8-bit input, a 3-bit select and an enable. How many input vectors does it have?
2. A sweep of that module reports exactly half of its vectors wrong. What kind of signal do you look for first?

Next: Coverage as a question.