Driving and checking a design

This page answers: *my testbench passes. Is that good?*

On its own, a pass tells you almost nothing. SimProc's one shipped test asserts `uo_out == 3`, which decodes to "the processor is not running", so it passes on the real core and on all thirteen broken ones, including the one whose `SUB` computes `B - A`. When a testbench passes, all you have learned is that it accepted the design, which is a claim about the testbench rather than about the design. The claim you want to be able to make about a testbench has two sides:

> It passes on the design that is right, and it fails on the design that is wrong.

Running `make` in `labs/cocotb-intro` checks both sides, which is why the package ships a broken counter alongside the real one. This is the verdict it prints:

```
----------------------------------------------------------------------
 VERDICT
----------------------------------------------------------------------
   test_03_checking.py PASSED on rtl/counter.sv
   test_03_checking.py FAILED on rtl/counter_broken.sv

   Both halves matter. A testbench that only passes has not been
   shown to be able to fail, and a testbench that cannot fail is
   not evidence of anything.
----------------------------------------------------------------------
```

You'll run it in cocotb on a counter, Part 4.

The two counters

`rtl/counter_broken.sv` differs from `rtl/counter.sv` by one term:

```
good:    assign tc = en & (q == 8'hFF);
broken:  assign tc =      (q == 8'hFF);
```

With the `en &` missing, every value of `q` is still correct, on every cycle and from every starting point, so the count itself is perfect. Only `tc` is wrong, and only while `en` is low.

Most real bugs look like this: the design as a whole still works, and one output is wrong in one situation that nobody tests.

The six tests

`test_03_checking.py` has six tests, and their names say what they are for:

Test	What it is for	Catches the `tc` bug?
`load_then_count`	The ordinary path: load a value, count eight times	No
`enable_holds_the_count`	The case where the design must do nothing	No

Test	What it is for	Catches the `tc` bug?
`load_wins_over_enable`	Priority between two controls asserted together	No
`terminal_count_needs_enable`	`tc` at `q = 0xFF` with `en` low	Yes, every time
`the_wrap_is_where_the_bugs_live`	The exact cycle 255 → 0, as `(q, tc)` pairs	No
`against_a_reference_model`	600 random clocks against three lines of Python	No

Five of the six pass on the broken counter, and one of those five is the 600-clock random test.

Three test shapes worth copying

The first is a test of the case where nothing should happen. `enable_holds_the_count` drives `en = 0`, clocks twenty times, and checks that the count did not move. Half of any specification is about when the design must hold still, and few people write a test for it.

The second tests priority explicitly. `load_wins_over_enable` raises `load` and `en` together, so one of them has to win. Which one wins is decided by the order of two `else if` branches, and someone tidying up the code in an editor can swap those without any tool warning them.

The third tests the output nobody looks at, which here is `tc`. `terminal_count_needs_enable` is that test, and it's the only one in the file that catches the bug. Flags, status bits, `done` pulses and `valid` lines are architectural state just as much as a data value is, and a testbench that checks only data has no way to see a bug in a flag.

Assertion messages

Here is the assertion most people write:

```
assert dut.tc.value == 0
```

and here is the one in the lab:

```
assert tc == 0, (
    f"q is 0xFF and en is 0, so the count is NOT about to wrap, but tc is {tc}.\n"
    f"  tc should be `en & (q == 8'hFF)`. A tc of 1 here means the `en &`\n"
    f"  is missing and tc is really just `q == 8'hFF`.\n"
    f"  Everything about the COUNT is still correct -- only the flag is wrong,\n"
    f"  and only while en is low.\n  {state(dut)}")
```

`make broken` prints it:

```
   404.00ns WARNING  ..able.terminal_count_needs_enable q is 0xFF and en is 0, so the count
is NOT about to wrap, but tc is 1.
                                                          tc should be `en & (q == 8'hFF)`.
A tc of 1 here means the `en &`
                                                           is missing and tc is really just
`q == 8'hFF`.
                                                              Everything about the COUNT is
still correct -- only the flag is wrong,
```

```
                                                          and only while en is low.
                                                           q=0xFF tc=1 en=0 load=0 rst_n=1
at t=404 ns
                                                    assert Logic('1') == 0
```

When you read that, you know which line of RTL to open, without rerunning anything, adding a print or opening a waveform. That's the standard to aim for: write the message you would want to see if this test failed at 2 a.m. six months from now, on someone else's machine. Three habits get you there.

The first is a `state()` helper whose output you paste into every message:

```
def state(dut):
    return (f"q=0x{int(dut.q.value):02X} tc={dut.tc.value} "
            f"en={dut.en.value} load={dut.load.value} "
            f"rst_n={dut.rst_n.value} at t={get_sim_time('ns'):.0f} ns")
```

It prints one line with every input, every output and the simulation time, and you only have to write it once per design.

The second is to say in the message what each possible wrong answer would mean:

```
assert got == 0x11, (
    f"with load AND en both high, q became 0x{got:02X}.\n"
    f"  0x11 means load won (correct for this design).\n"
    f"  0x12 means load won and en also applied.\n"
    f"  anything else means en won.\n  {state(dut)}")
```

With messages like that, a failure hands you a diagnosis that you wrote before you needed it.

The third is to report the difference as well as the two values. `expected 0x35, got 0x36 — a difference of +1` names the bug, whereas `expected 0x35, got 0x36` leaves the reader to do the subtraction at 2 a.m.

Whole-sequence and random checks

Comparing a whole sequence tells you more than comparing one value. `the_wrap_is_where_the_bugs_live` collects five `(q, tc)` pairs and compares the list in a single assertion, so when it fails you see the whole trajectory:

```
   486.00ns INFO  cocotb.counter  q, tc across the wrap: (254,0) (255,1) (  0,0) (  1,0) (  2,0)
```

Random stimulus finds bugs only by accident. `against_a_reference_model` runs 600 clocks against a six-line Python model of the counter and checks `q` and `tc` on every one. It is the most thorough test in the file, yet it passes on the broken counter, and its own log shows why:

```
  6507.01ns INFO  cocotb.counter  600 random clocks, q and tc agreed with the model on every
one (final q = 0x7C)
  6507.01ns INFO  cocotb.counter  of those 600 clocks, 0 had q = 0xFF with en low -- the only
ones where a broken tc would have shown
```

In 600 random clocks, the one situation that exposes the bug occurred zero times, because the counter only reaches `0xFF` while it is running, and while it is running the broken `tc` and the correct `tc` agree.

The seed is pinned (`random.Random(104)`), so that zero means the same thing on your machine as it does here. If you unpin it, you get a test that passes on Tuesday and fails on Wednesday, and you have no way to tell whether you changed the design or just drew a different hand.

When a random or constrained-random test passes, ask how many of its vectors reached the situation you care about, and count them, which takes a counter in the loop and one extra log line. If the answer is zero, the test told you nothing about that situation, however many vectors it ran. Running a lot of vectors is not the same as covering the cases that matter.

Reference models

The model in `against_a_reference_model` is three lines:

```
if load:      model_q = d
elif en:      model_q = (model_q + 1) & 0xFF
model_tc = 1 if (en and model_q == 0xFF) else 0
```

It is written from the specification, in a different language, by someone who is allowed to be lazy about how. It's worth having because it checks things you did not think to check: a directed test looks at one signal at a time, while a model looks at all of them on every cycle.

It also has a limit that matters just as much: the model can only be wrong in the same ways you are. This one can't see `tc` depending on `en` in any situation the stimulus never reached, and it has no idea that `done` or `halt` exist. In SimProc that gap is a whole class of bug, because a core that computes every answer correctly and gets its handshake wrong passes a model comparison, and that's why a suite needs more than one kind of test.

Check yourself

1. Why does `against_a_reference_model` pass on the broken counter even though it checks `tc` on every clock?
2. Write the assertion message you would want for a counter that holds when it should count. What values and signals does it name?

Next: Waveforms.