What a passing test proves

This page answers: *I ran my design and it did the thing. Why isn't that enough?*

By the end of DD103 you had a `.gds` file, a chip layout that was DRC clean and LVS clean, produced by a flow that ran eighty stages and printed a green tick after each one. None of those stages checked whether your design was right. Each tool answers its own question and leaves the others to someone else. That's a division of labour rather than a flaw in the tools, and the gap it leaves is what this course is about.

Tool	The question it answers	The question it does not answer
Yosys	Can this be built out of gates?	Are they the right gates?
Verilator lint	Is this legal SystemVerilog?	Does it compute what you meant?
OpenSTA	Will the signals arrive in time?	Are they the right signals?
LibreLane / DRC / LVS	Can this be manufactured, and does the layout match the netlist?	Does the netlist do anything useful?

Every one of those checks can pass on a processor whose `SUB` instruction computes `B - A`, and what you get at the end is a clean chip that subtracts backwards.

Three meanings of "works"

When you say a design works, you usually mean one of these and hope it implies the others:

1. It compiled, which means the tool accepted it.
2. It ran, without hanging, crashing or emitting `x`.
3. It produced the answer you expected, on the inputs you thought of.

Only the third is about correctness, and it carries a clause people rarely say out loud: *on the inputs you thought of.* Those inputs were picked by the same person who wrote the bug, so they tend to share the bug's blind spots, and being careful doesn't fix that: a case that didn't occur to you while you wrote the design is unlikely to occur to you while you test it.

> If you came from software, tests may feel like the chore that comes after the interesting part, because if you skip them you find out in staging and push a fix. That last step doesn't carry over to hardware: you can't push a fix to a chip that has already been manufactured.
>
> If you came from circuits, you may expect broken things to look broken, the way a floating node reads as X or a shorted output smokes. An RTL bug doesn't damage anything. What you built is a different, well-formed circuit that computes the wrong function at full speed, with clean edges on every signal, so there's nothing to see that would tell you it's wrong. This is DD103's `unique case` lesson again, in a worse form.

The test SimProc shipped with

From Lab 02 on, the design under test is SimProc: an 8-bit processor of about 1,200 lines of SystemVerilog, written by students on this team and wrapped for a real chip. It has registers, an ALU, a memory and a UART you can talk to.

It shipped with exactly one test. Here is `test/test.py` from the SimProc repository, with the boilerplate stripped out:

```
dut.ui_in.value = 1
dut.uio_in.value = 0
```

```
await ClockCycles(dut.clk, 100)
assert dut.uo_out.value == 3
```

It sets one input pin high, waits a hundred clocks and checks that the output byte equals 3. To see what that means, decode `3` against the chip's pin map, where `uo_out[0]` is the UART transmit line, `uo_out[1]` is `halt` and `uo_out[2]` is `done`:

```
uo_out = 0b00000011
           |      └── uart_tx = 1   the serial line is idle
           |     └─── halt    = 1   the core is sitting in its IDLE state
           |    └──── done    = 0   no instruction has completed
```

So what the test asserts is that the processor is not running. It sends no command, loads no program and executes no instruction, and it never looks at a register, a flag or a byte of memory. This is the real file, it passes, and for a long time it was the only evidence anyone had that SimProc worked.

The file even admits as much. In the original, the two lines just above the assertion are this comment:

```
    # The following assersion is just an example of how to check the output values.
    # Change it to match the actual expected output of your module:
    assert dut.uo_out.value == 3
```

That comment comes from the project template, typo included. Nobody overrode a warning or ignored a failure here. The placeholder assertion was simply never replaced, and because it kept passing, the tick stayed green and nothing ever looked wrong. That's why this failure mode is so common: almost every first tapeout looks like this.

Thirteen broken processors

DD104 ships thirteen deliberately broken versions of SimProc. Each one differs from the real core by a one-line edit: `SUB` computes `B - A`, `NAND` is wired as `NOR`, the zero flag looks only at the bottom four bits, branches zero-extend their displacement instead of sign-extending it, `LOAD` reads its address from the destination register instead of the pointer, and so on.

Here is what the shipped test prints when it runs against the good core and all thirteen broken ones, which you'll do yourself in Lab 02, Part 1:

```
------------------------------------------------------------------------
    PASS  the golden core                              5.7s
    PASS  B01_sub_operands_swapped                     2.1s
    PASS  B02_nand_is_nor                              2.1s
    ...
    PASS  B12_uart_rx_bit_period                       2.0s
------------------------------------------------------------------------
 13 of 13 deliberately broken processors pass the
 only test this chip has.
========================================================================
```

You could put one missed bug down to bad luck, but not thirteen out of thirteen. The test spends its hundred clock cycles watching a processor sit still, and every one of the thirteen bugs is somewhere the processor only goes when it is running, so the stimulus never reaches the thing the test is supposed to check.

What a bug costs

Where the bug is found	What it costs
In simulation	An edit and a rerun, about eight seconds
In FPGA bring-up	A re-synthesis and an afternoon in the lab
In silicon	A new mask set: months and a lot of money
In silicon, in a product	A recall, or a customer working around your bug forever

Each row costs roughly a hundred times as much as the row above it, and that ratio is why pre-silicon verification is a job title, why verification engineers outnumber design engineers on most commercial chip teams, and why the field has its own tools and its own idea of evidence.

In this field, a passing test doesn't count as evidence until you know what it would have caught. A green tick claims only as much as the stimulus behind it exercised, so for `test/test.py` the claim is "this chip powers up idle".

What to ask of a passing test

You'll see this question on every page of the course:

> When a test passes, ask what it would have caught. If you cannot name a specific bug that would turn it red, the green has told you nothing.

The mechanical way to answer it is to run your test against a design you know is broken. Every lab package does this with `make check`, and it won't give you a passing verdict until your testbench has both passed the good design and failed a broken one, because a result needs both of those sides.

Two kinds of red

You should be just as suspicious of a failing test, and forgetting that has already given a DD104 student a false PASS. A test that didn't pass hasn't necessarily checked anything, because there are two unrelated ways for a test to come back red:

	What happened	What it tells you
The check fired	You drove the design, compared what came out with what should have, and they differed	Something about the design
The test blew up	`NameError`, `TypeError`, a failed import, a variable spelled two ways	Something about the testbench

Only the first is a caught bug. The second never compared anything, so it would say the same thing about a perfectly correct design.

This kind of failure tends to hide in code that only runs when something is wrong, such as the line that reports a failed check:

```
if bad:
    raise AssertionError(f"{len(bad)} of {checked} wrong: {summary}")   # `summary` undefined
```

That line doesn't run until the first time the design gives a wrong answer. So the testbench passes the good design, crashes on the broken one, and from the outside looks exactly as if it had caught the bug. cocotb records both outcomes in `results.xml` as a `<failure>` element, and only the `error_type` attribute tells them apart:

```
<failure error_type="AssertionError" error_msg="256 of 512 vectors are wrong" />
<failure error_type="NameError"      error_msg="name 'summary' is not defined" />
```

DD104's graders once read those two lines the same way, and told a student `3 of 3 ALU bugs caught` about a testbench that had crashed all three times. Every checker in the course now goes through `labs/common/verdict.py`, which is the one place that makes the distinction, and it marks a run like that `BROKEN`, never `caught` or `ok`:

```
  BROKEN  B01_sub_operands_swapped           3 tests, 1 BLEW UP
--------------------------------------------------------------------------
 VERDICT: FAIL -- 1 of your tests did not run to the end.
```

So the question has a second half: when a test fails, ask whether it really failed or just fell over.

Check yourself

1. In one sentence, write the strongest true claim the shipped test above supports. (You'll compare your answer with the course's in Lab 02, Part 1.)
2. A testbench passes on the good core and raises `TypeError` on a broken one. Has it caught the bug?

Next: What a testbench is.