Writing the bug report

This page answers: *I found it. What do I actually hand to the designer?*

You hand them a reproducer, which is a test they can run. A description, a patch or a paragraph in a chat window doesn't do the same job. The capstone is built on one idea that surprises people:

> The test is the deliverable, not the fix.

You met the reason in Lab 04, when the grade's row 2 turned out to be the deliverable, and it comes down to which of the two lasts. A patch is something one person did once, in one file, on one afternoon. A test that fails on the broken design and passes on the fixed one is what stops the bug coming back next year, in a rewrite, done by someone who has never heard of you. Of those two things, only the test will still be doing any work in 2031.

DD104 builds that idea into the grader. `make verdict` prints three rows, and the middle one is your deliverable:

```
      ok  1. the golden core          must PASS      3 tests, all passed        4.2s
  caught  2. the core you were dealt  must FAIL      3 tests, 1 FAILED          3.4s
        t01_sub_with_two_distinct_operands
            AssertionError: SUB r1, r2 with r1 = 200 and r2 = 100 left r1 = 156.
    FAIL  3. core/ as it stands now   must PASS      3 tests, 1 FAILED          3.4s
```

The core in row 2 is rebuilt fresh from the bug catalogue every time, rather than read from `core/`, so when you apply the fix, row 3 goes green and row 2 stays exactly as it was:

```
      ok  1. the golden core          must PASS      3 tests, all passed        4.2s
  caught  2. the core you were dealt  must FAIL      3 tests, 1 FAILED          3.4s
      ok  3. core/ as it stands now   must PASS      3 tests, all passed        3.5s
----------------------------------------------------------------------------
 VERDICT: PASS
   Row 2 is the part worth keeping. Commit that test.
```

Editing the file the test runs against can't turn row 2 green; only a test that catches the bug can do that. That makes "the test is the deliverable" a fact about the grader rather than an opinion.

In Lab 06, Part 7 you'll go through all of this end to end on a bug you already know, before you write the real report.

The five parts, in the order the reader needs them

`BUG-REPORT-template.md` ships in the capstone package. Next to it is a complete worked example, `BUG-REPORT-EXAMPLE.md`, filled in for `B01_sub_operands_swapped`, which we deliberately chose from outside the capstone's three bugs. Read it to see the shape of a report, then write yours. The five parts below are in the order the reader needs them.

1. The smallest program that reproduces it

Minimise the program: cut instructions until it stops failing, then put the last one back.

A ten-line repro and a three-line repro show the same bug, but they cost very different amounts of someone else's day, because every instruction you leave in is one the reader has to rule out.

The worked example is two `set_reg` calls, which come to twelve instructions of setup, and one `SUB`, and it ships complete as `labs/common/test_b01_repro.py`:

```
    sp = SimProc(dut)
    await sp.start()

    await sp.run(set_reg(1, 200) + set_reg(2, 100) + [SUB(1, 2)])
    got = sp.reg(1)
```

There is no loop, no branch, no memory access, no model, and no UART traffic beyond what the harness needs to start the core. The repro also comes with the command that runs it, from `labs/common`: `make check TB=test_b01_repro BUG=B01`.

If the reader can't run a repro, it's only a description, so give both the file and the command, and make sure both work from a clean clone, because the reader doesn't have your shell history.

2. Expected against actual, as numbers, with the arithmetic

"r1 is wrong" isn't enough. Give the number you expected, the number you got, and the sum that produces the expected one. With the sum included, the report stops being a complaint and becomes a claim that someone can check.

Put all of this in the assertion message as well as in the report, so that it appears on someone else's machine without anyone having to work it out again:

```
    assert got == 100, (
        f"SUB r1, r2 with r1 = 200 and r2 = 100 left r1 = {got}.\n"
        f"  expected  200 - 100 = 100\n"
        f"  got       {got}\n"
        f"  note      (100 - 200) mod 256 = {(100 - 200) & 0xFF}, which is what a "
        f"core computing rB - rA would give\n"
        f"  {sp.describe()}")
```

`make verdict VERBOSE=1` prints all of it:

```
SUB r1, r2 with r1 = 200 and r2 = 100 left r1 = 156.
    expected  200 - 100 = 100
    got       156
    note      (100 - 200) mod 256 = 156, which is what a core computing rB - rA would give
    PC=13  r0=100 r1=156 r2=100 r3=  0  N=1 Z=0  IDLE
assert 156 == 100
```

From that message alone you know which line of RTL to open, without rerunning anything, adding a print or opening a waveform. That is the standard to aim for, and it's cheap to meet: the last line of the message, `sp.describe()`, is a single call.

3. The shape

This part says how many things are wrong, out of how many, and what that count tells you before anyone opens the RTL.

A count becomes a diagnosis when it factors, that is, when you can explain exactly where the number comes from. Here is the ALU driven exhaustively, 393,216 vectors in about nineteen seconds:

```
ALU ADD  value_mismatch=     0 flag_mismatch=     0
ALU SUB  value_mismatch= 65024 flag_mismatch= 65024
ALU OR   value_mismatch=     0 flag_mismatch=     0
ALU NAND value_mismatch=     0 flag_mismatch=     0
ALU SHL  value_mismatch=     0 flag_mismatch=     0
ALU SHR  value_mismatch=     0 flag_mismatch=     0
```

65,024 of the 65,536 `SUB` pairs are wrong, and every other operation is perfect. The next step is to account for the 512 pairs that survive, because that is where the diagnosis is. `A - B == B - A (mod 256)` exactly when `2(A - B) == 0 (mod 256)`, which happens when `A - B` is `0` or `128`. That gives 256 pairs with `A == B` plus 256 with `|A - B| == 128`, and `65,536 − 512 = 65,024` exactly. Only "the operands are the wrong way round" produces exactly 512 survivors, so a single number has already narrowed the bug down to one explanation.

Some shapes are worth learning to recognise:

The count says	It means
`0` value mismatches, `25,726` flag mismatches	The arithmetic is fine; look at the flag logic
Exactly half	The fault tracks a signal that is high half the time
A count that factors into a sentence	You have a hypothesis; go and confirm it
A count that does not factor	You have a number, and you still owe an explanation for it

The sweep output above also gives you a finding for free: `65024` appears in both columns. `N` and `Z` are computed from `ALUout`, so they are wrong exactly when it is. That clears the flag logic, and you didn't need a separate test to find that out.

4. Where it is not

People tend to skip this part, but it's what makes a report credible, and it also narrows down where the fix has to go.

- Which instructions are unaffected? (`ADD`, `OR`, `NAND`, `SHL`, `SHR`: zero of 65,536 each.)
- Which registers, addresses and flags are unaffected?
- Does it survive a reset? (This bug is combinational, so reset makes no difference, and saying so rules out a whole class of explanations.)
- Does it depend on the operands being different? (Yes. `SUB rX, rX` is the usual way to clear a register, which is why this bug gets through almost every program in this course unnoticed.)
- Which of your other tests pass on this core, and what does that rule out?

The worked example ships two tests that pass on the broken core on purpose, `t02_the_idiom_that_hides_it` and `t03_the_512_survivors`. If a report contains only failing tests, nobody has checked which parts of the design still work.

> In software testing, a passing test in a bug report is noise. Here it is evidence, because each one marks a region of the design the reader can stop suspecting, and narrowing the search is most of debugging.

5. The habit the suite gained

Write one line giving the rule a person can carry to the next design, and name the test that enforces it here.

> Reflex: *never test a binary operation with `op(x, x)`. Aliased operands make a swapped, mirrored or wrong-source implementation indistinguishable from a correct one, and the aliases you write by habit (`SUB rX, rX` for clear, `NAND rX, rX` for not, `LOAD rX, rX` for a pointer dereference in place) are exactly the ones you will never suspect.*
>
> Enforced by: `t01_sub_with_two_distinct_operands` in `labs/common/test_b01_repro.py`, which fails on this core and passes on the golden one.

A demonstration with a one-line rule attached becomes a habit, while a demonstration on its own stays an anecdote. The rule has to be short enough that you still remember it six months later, and it has to name a test, because otherwise nothing enforces it.

Four ways a bug report fails

The first is "It doesn't work." A report like that has no numbers in it, so nobody can tell whether they have fixed the bug.

The second is a patch with no test, which is the quickest way to end up with the same bug twice. If you did fix the bug as well, that's good: put the diff at the bottom, under the reproducer.

The third is a repro that only runs on your machine, because it uses a file path that isn't in the repo, a testbench that imports something you wrote at 1 a.m., or a `make` invocation with four variables set. Everything in your report must run from a clean clone with the command you printed.

The fourth is a failing test with no passing one, and that is what the first row of `make verdict` is there for. If a test fails on the broken core and also fails on the golden one, it has found a problem in your testbench rather than your bug, so row 1 must be `ok` before row 2 means anything.

Check yourself

1. Your repro needs a loop and a branch to fail. What do you try before you write the report?
2. Why does the worked example include two tests that pass on the broken core?

Next: Lab 06: Capstone.