Where bugs hide

This page answers: *the bugs I am hunting are one line each. Why can't I just read the file?*

The short answer is that these one-line bugs aren't typos. Each one is a line that looks correct, written in the right style and in the right place, doing something the surrounding code also does, and it gives a wrong answer only on inputs you wouldn't naturally write. This page gives the evidence for that, and then a taxonomy that turns it into a way of working.

> This page doesn't tell you where any bug is. DD104 ships thirteen broken cores, and Lab 04 deals you one without saying which. Every catalogue name below, along with what each one hides behind, is something `make bugs` already prints, and no line numbers or RTL appear here. The skill we want you to build is finding a bug in a design you can't diff against a known-good copy, and a diff would let you skip it.

The 99.2 % case

`B01_sub_operands_swapped` makes the ALU compute `B - A` instead of `A - B`. Drive every legal vector through the ALU, as Lab 03 does in about twenty seconds, and it is wrong on

```
65024 of 393216 vectors have the wrong value: {'SUB': 65024}.
  First one: SUB(0x00, 0x01) = 0x01, expected 0xFF.
```

That is 65,024 of the 65,536 `SUB` vectors, or 99.2 % of them, so this processor gets subtraction wrong ninety-nine times in a hundred.

Even so, the bug passed code review, and it passed the only test the chip ever shipped with, while DD104's own six-test example suite catches it four times over, with four separate assertions.

Those facts are consistent with each other, and seeing why is the single most useful idea in verification.

Disagreement sets

It helps to think of a bug as the set of inputs on which the wrong implementation and the right one give different answers, rather than as a broken line. We'll call that its disagreement set.

- If that set is everything, you find the bug just by looking. The design never works, the first person to run it notices, and it never reaches a shuttle.
- If that set is empty, there is no bug. The two implementations are the same machine, and which one you shipped is a matter of taste. (This does happen here: `ORI` could accumulate in `r0` or in `r1`, and the two machines are exactly isomorphic, so DD104 had to pick one, because no measurement could decide it.)
- Every bug that survives to silicon lies in between, and it survives because its disagreement set leaves out the inputs people actually write.

So for `B01` the question to ask is what the 512 survivors are, rather than how nobody noticed 99.2 % of subtractions coming out wrong.

`A - B == B - A` exactly when $2(A - B) \equiv 0 \pmod{256}$, that is, when $A \equiv B$ or $A - B \equiv 128$. Each condition has 256 solutions, 512 in total, and $65,536 - 512 = 65,024$, which is the number the sweep printed.

Now look at what is in that set. SimProc has four registers and no `MOV` or `CLR`, so `SUB rX, rX` is how you zero a register, and it appears in nearly every program on this machine. That instruction is one of the 512, which means the bug is invisible on the most common way anyone writes the instruction it breaks.

This is the general pattern: a bug survives review when its disagreement set misses the idioms, because the idioms are what a reviewer's eye is trained on.

Six places a disagreement set hides

Every bug in the DD104 catalogue has one of these six shapes, and so does most of what you will find in industry. Each one comes with a habit, and the habit is the part you'll carry to other designs.

1. Aliasing

Any instruction with two register fields has a case where both fields name the same register, and in that case it doesn't matter which field the hardware reads, because there is no wrong one to pick.

Catalogue	`B01`, `B02`, `B04`, `B07`
Measured	`B02` disagrees on 65,280 of 65,536 `NAND` vectors; the 256 survivors are exactly `A == B`
Why it survives	The machine has no `NOT`, so `NAND rX, rX` is how you invert, and `SUB rX, rX` is how you clear

`B07` has the best history in the course. The first `LOAD`/`STORE` test written for this core used `LOAD r1, r1`, which loads through a pointer into the register that held the pointer. That's a real idiom, and you use it when you've finished with the address. The test passed on a core that takes the address from the wrong register field, because when both fields name the same register there is no wrong one. Nobody noticed until the bug was injected on purpose and the suite stayed green.

> The habit: make every operand distinct on purpose, and check the register that should not have moved as well as the one that should.

2. Identity elements and reset

`x | 0 == x + 0`, and `0 | imm == 0 + imm`. Operators that differ in general agree on their identity elements, and a reset sets every register in the machine to zero, which is the identity element of almost everything.

Catalogue	`B10`, `B11`
Measured	`ORI 5 ; ORI 3` gives 8 on a core whose `ORI` adds; it should give 7. `ORI 8 ; ORI 3` gives 11 on both
Why it survives	The first `ORI` after reset is `0 | imm`, which is the right value under both bugs. Building a constant from disjoint bit groups, the natural way to assemble a value from a 5-bit immediate, never overlaps, so it never diverges

`B11` is just as hard to think of, but it is the easier of the two to catch. It writes the right value to the wrong register: after a reset `r0` and `r1` both hold zero, so the first `ORI` computes `0 | imm` correctly and puts it in `r1`, and the machine holds the right number under the wrong name. A test that checks `r0` after a single `ORI` finds zero there and fails, which is how Lab 02's `t03` catches it, and seven of the eleven reference tests fail on `B11` against two on `B10`. A test that only looks for the value somewhere in the registers misses it, but only until the second `ORI`, which reads an `r0` that the first one never wrote.

> The habit: never test an operation only from a reset state, and never test it only once. Once you do two of anything (two `ORI`s, two branches off one compare, two writes to the same register), the state the second one sees can no longer be right by coincidence.

3. Parameters left at their default

A parameter that is only ever driven at its default has been tested at one point of a range, and the silicon will be used across the whole of that range.

Catalogue	`B12`, and `B06` in spirit
Measured	`B12` scores 10/10 on the full systematic suite at `clk_per_bit = 32`, indistinguishable from a correct core, and 0/10 at `clk_per_bit = 8`
Why it survives	`clk_per_bit` is set by eight external pins, every lab uses one value, and at that value the receiver's drift still lands inside the bit

There is a second trap on top of the first. At `clk_per_bit = 16`, `PING` still answers `0xAA` over a link that can't carry data. `PING` sends `0xF0`, whose top bit is already 1, so a receiver that samples one bit-time late reads the stop bit in its place, which is also 1, and the command decodes perfectly. In other words, a link test can pass over a broken link.

`B06` has the same shape one level down. The shift-amount field is two bits wide, amounts 0 and 1 are correct, and 1 is the amount everybody writes.

> The habit: sweep the field instead of sampling it. Four shift amounts, sixteen branch displacements and five baud rates are all small enough to walk through, and testing every one of them is simply finishing the job.

4. One half of a signed range

Catalogue	`B05`
Measured	All eight forward displacements, `+0` to `+7`, land exactly right; every backward one lands 16 too far forward
Why it survives	"Skip the next instruction" is the first branch anybody writes, and it works. What breaks first is the first loop, because a loop is the only reason to branch backwards

Half the number line is correct, and it's the half you test first, so the bug survives every test that only branches forwards.

> The habit: when a field is signed, the sweep must cross zero.

5. Outputs that are not values

In this shape every value the design computes is correct, and what's wrong is some output that isn't a value.

Catalogue	`B03` (flags), `B08` (`done`), `B13` (`halt` and one cycle)
Measured	The exhaustive ALU sweep on `B03` reports 0 value mismatches and 25,726 flag mismatches out of 393,216
Why it survives	A testbench that reads registers and memory is testing a smaller machine than the one on the die

`B03` is the one to think about. A testbench that checks only values can't see it at all, however many vectors you add, so adding stimulus won't help; what it needs is a different kind of check. Flags as architectural state covers that check.

`B08` does the opposite of what you expect a bug to do. If your scoreboard triggers on `done` and one instruction stops pulsing it, the scoreboard doesn't report an error, because it was never woken up for that instruction. It compares everything it was asked to compare, finds no mismatches, and reports that truthfully, having made one comparison fewer than there were instructions. So a missing trigger doesn't make your test fail; it quietly stops your test from testing, which is worse, because a failure at least shows up.

> The habit: list the design's outputs. Every one you never assert on is unverified, and the list is shorter than you think. For SimProc it is `done`, `halt`, `uart_tx` and the clock.

6. Using state twice

State that is correct when you use it once can be destroyed before you use it twice.

Catalogue	`B09`, and `B11` again
Measured	One branch after a compare works every time. A second branch off the same compare reads garbage
Why it survives	Most programs compare once and branch once, and this machine's flags are read combinationally in the same cycle the branch decides, so the first branch always sees the right value

In a machine whose `SUB` doubles as its compare, it matters how long the flags stay valid, and reading the source won't tell you. "The comparison is still valid" is an assumption, and an assumption needs a test.

> The habit: use each piece of state twice, and assert that the second use sees what the first did.

The evidence

All thirteen broken cores pass the only test the chip has: you saw `13 of 13` in Lab 02, Part 1.

A real testbench isn't necessarily enough against a real bug either. `test_example.py` has six worked tests: arithmetic on distinct operands, flags, a loop with a hand-checked cycle count, and a full golden-model comparison over 29 instructions. Here is what it reports against `B10`:

```
      ok  the golden core                   6 tests, all passed          6.6s
  MISSED  B10_ori_adds_instead_of_ors       6 tests, all passed          5.7s
------------------------------------------------------------------------
 VERDICT: FAIL — your testbench passes B10_ori_adds_instead_of_ors too.
   A test that passes on a broken core is not testing that core.
   That bug hides behind: ORing in bits that were not already set — ADD agrees
   Write a test whose stimulus does not do that.
```

That's six good tests, with twenty-nine instructions checked register by register against a reference model, and the bug is still `MISSED`, because none of those instructions happened to OR in a bit that was already set. You'll run this yourself in Lab 06, Part 2.

The test that catches it is two lines long, and it works because of its stimulus:

```
await sp.run([ORI(5), ORI(3), ORI(24)])
assert sp.reg(0) == 31          # 5 | 3 = 7, 7 | 24 = 31.  Added: 5 + 3 + 24 = 32.
```

Verification is mostly a matter of choosing inputs well, rather than of machinery, assertion count or framework. The reference suite that catches all thirteen bugs is eleven ordinary tests, each a short program, a run and a comparison, and they work because their stimulus steps outside the idioms: subtracting a number from something other than itself, shifting by more than one, branching backwards as well as forwards, and ORing in bits that were already set.

Reading the file

Reading the source tells you where to look, but it can't tell you that you're finished. On a design you didn't write it often can't even tell you which of two plausible lines is wrong, because the wrong line usually looks like a correct line from four clauses further down.

There are three reasons the "read it carefully" instinct fails on this class of bug:

1. You don't have the good version to diff against. That's normal in industry, and it's what Lab 04 is built around. With a diff you would find all thirteen in four seconds and learn nothing.
2. A one-line bug is written in the file's own style. One of the thirteen adds a line that looks like it belongs, in the right style and the right place, which shows that an injected bug doesn't have to be a typo.
3. The effort of reading grows with the size of the file, while the effort of testing grows with the size of the machine. The file is 1,178 lines, and the machine is 554 bits of state and eleven instructions. On your next project one of those numbers will grow and the other won't.

The question list

This is the same taxonomy, turned into a list you can work down while a core is in front of you. Each question either narrows the search or rules out a region of the design.

	Question	What a "yes" rules in	Shape
1	Which instruction classes are wrong: arithmetic, shift, load/store, branch?	The decode path for that class	
2	Is it wrong for all operands, or only some?	A data-dependent path: the ALU, or a width	
3	Does it depend on rA and rB being different?	A register-select field, read or write	1
4	Which register ends up wrong: the destination, a source, a bystander?	The write-back address	1
5	Does it survive a reset? Is the first one right and the second not?	Somewhere reset makes two things equal	2
6	Are the values right and something else wrong?	Flags, `done`, `halt`, the cycle count	5

	Question	What a "yes" rules in	Shape
7	Does a field you have only driven at one setting change it?	A shift amount, a displacement, a baud divider	3, 4

Work through them in order. Each answer takes one small program, and what separates a five-minute debug from a two-hour one is rarely insight; it's whether you spent the first ten minutes narrowing the search or reading the file.

> Before you write a test, name the input class it won't reach: which operand pattern, which field value, which half of which range, which output pin. Write the answer in the docstring. It's the most valuable line in most testbenches, and almost nobody writes it.

There is a corollary, and `make check` automates it: a test you have never seen fail is a claim you have never checked. Break the thing your test is supposed to watch, and see whether the test notices. That is how `B07` was found in this repository, by the person who had written the test that missed it.

Check yourself

1. A broken `ADD` gives the right answer whenever one operand is zero. Which of the six shapes is that?
2. A suite tests every branch with displacement `+1`. Which bug in the catalogue can it never see?
3. For `NAND`, how many of the 65,536 operand pairs survive a NAND-is-NOR bug, and which ones?

Next: Directed and random tests.