Flags as architectural state

This page answers: *I checked the answer and it was right. What else is there to check?*

The answer is the flags. SimProc has two: `N` is bit 7 of the last result, and `Z` is 1 when that result was zero. `ADD`, `SUB`, `NAND`, `SHIFT` and `ORI` write them, and `LOAD`, `STORE` and every branch leave them alone. Each flag is a single bit, but the flags are **architectural state**: they are as much a result of an instruction as the byte that lands in a register, and on this machine they are the only thing a branch can see.

> It's natural to assume that if the numbers are right, the machine is right. On this processor you can get every number right, in every register and every byte of memory, on every instruction, and still ship a chip that takes the wrong branch.

A one-character bug

`B03_zero_flag_low_nibble` changes one expression:

```
good:    assign Z = ~(|ALUout);
broken:  assign Z = ~(|ALUout[3:0]);
```

The zero detector now looks at the bottom four bits of the result instead of all eight, so the broken core says "that was zero" for 16, 32, 64, 128 and 240, and for every other non-zero byte whose low nibble happens to be zero.

Every value the broken core computes is still correct: `ALUout` is untouched, the register and memory get the right byte, the program counter is right, and so is `N`. The only thing wrong is one bit, and only for some results.

The exhaustive sweep

The ALU is 34 lines long, and its whole input space is 393,216 vectors, few enough that Icarus walks through all of them in about twenty seconds. That walk is what you do in Lab 03. With a sweep that checks values and flags in separate buckets, `B03` produces this (you'll get there in Lab 03, Part 6):

```
      ok  the golden ALU                    3 tests, all passed            19.4s
  caught  B03_zero_flag_low_nibble          3 tests, 1 FAILED              18.6s
        t02_every_vector_the_alu_can_be_given
              AssertionError: Every value is correct, and 25726 of 393216 vectors have the
wrong FLAGS
```

The assertion message says what this page is about: every value is correct, and 25,726 of the 393,216 vectors have the wrong flags.

`make check` truncates the message to fit its table, but the run itself prints all of it:

```
393216 vectors checked -- 0 value mismatches , 25726 flag mismatches
    {'ADD': 3840, 'SUB': 3840, 'OR': 255, 'NAND': 255, 'SHL': 14400, 'SHR': 3136}
  First one: ADD(0x00, 0x10) = 0x10 with N=0 Z=1, expected N=0 Z=0.
```

Before you open any RTL, you can read three things off this output.

First, the value count is zero for all six operations, so the adder adds, the subtractor subtracts and the shifter shifts.

Second, the flag count is non-zero for all six. A fault that affects every operation but no value can't be inside any one operation. It has to be in something all six share and none of them computes, and this module has exactly two such things: `N` and `Z`.

Third, the counts factor, and the factors describe the bug. Each of the six counts is exactly the number of vectors whose result is non-zero but has a zero low nibble, which is the only prediction the hypothesis "`Z` is wired to `ALUout[3:0]`" makes:

	Count	=	Why
ADD, SUB	3,840	15 × 256	15 results `0x10`…`0xF0`, each reachable from 256 operand pairs
OR	255	16 × 16 − 1	`a|b` ends in four zeros only if both operands do, minus the one case where the result really is zero
NAND	255	16 × 16 − 1	Mirror image: both operands must end in `0xF`
SHL	14,400	960 + 1,920 + 3,840 + 7,680	Shifts of 0, 1, 2 and 3; each larger shift creates more low zero bits
SHR	3,136	960 + 896 + 768 + 512	Shifting right removes low zeros instead, so the count shrinks
	25,726		

The column adds up to exactly 25,726. When a failure count factors into a sentence about the design, you have a hypothesis you can act on; when it doesn't, you have a number you still have to explain. The `SHL`/`SHR` asymmetry alone rules out every explanation except "something is looking at the bottom of the byte", because nothing else would care which way the bits moved.

The last line, `First one: ADD(0x00, 0x10) = 0x10 with N=0 Z=1`, gives you a three-line reproducer.

So from one number and its factorisation you've narrowed the bug down to one of two wires, without opening the RTL. That is what we mean by "the count is the diagnosis", and it's the difference between a bug report a designer can act on and one that says "the ALU is broken".

A value-only testbench

`labs/common/test_values_only.py` is a reasonable testbench, not a strawman. It has three tests: `ADD` and `SUB` over twelve operand pairs each (zero, one, the halves of the byte, the wrap); `NAND` over the same twelve plus twenty shifts; and a sixteen-instruction program checked for both its answer and its exact cycle count.

```
@cocotb.test()
async def v01_arithmetic_over_a_spread_of_operands(dut):
    """ADD and SUB, twelve operand pairs each, every result checked."""
    sp = SimProc(dut)
    await sp.start()

    for a, b in PAIRS:
        await sp.run(set_reg(1, a) + set_reg(2, b) + [ADD(1, 2)])
        assert sp.reg(1) == (a + b) & 0xFF, (
```

```
            f"{a} + {b}: r1 is {sp.reg(1)}, expected {(a + b) & 0xFF}.\n"
            f"  {sp.describe()}")
```

It runs for 1,889,970 ns of simulated time (3.7× as long as the example testbench) and takes twenty-two seconds of wall clock, and it's more thorough than most first attempts. Against `B03` it gives:

```
      ok  the golden core                    3 tests, all passed         21.3s
  MISSED  B03_zero_flag_low_nibble           3 tests, all passed         20.1s
------------------------------------------------------------------------------
 VERDICT: FAIL — your testbench passes B03_zero_flag_low_nibble too.
   A test that passes on a broken core is not testing that core.
   That bug hides behind: every test that checks results and not flags
```

The testbench reports `MISSED`, and more luck or more vectors wouldn't have helped, because it cannot see the bug at all. Every value this processor computes is visible in a register or in memory, but `N` and `Z` show up in neither. A test that only reads registers and memory is testing a smaller machine than the one on the die, so you could run it for a week and it would still miss `B03`. You'll run it yourself in Lab 03, Part 5.

So the rule is: for every instruction that writes a flag, assert the flag, and for every instruction that must not write one, assert that it did not move. Treat it as a per-instruction rule, the same way you already check the result of every instruction.

Four lines that catch it

Here are the four lines that make the difference, from the fourth test in `test_example.py`:

```
    # r1 = 16, and make that the LAST flag-setting result.
    await sp.run(set_reg(1, 16))
    assert sp.reg(1) == 16, f"r1 is {sp.reg(1)}, expected 16"
    n, z = sp.flags()
    assert (n, z) == (0, 0), (
        f"after r1 = 16 the flags are N={n} Z={z}, expected N=0 Z=0.\n"
        f"  16 is 0b00010000: not negative, and not zero. If Z is 1 here, "
        f"something is looking at only part of the byte.")
```

Against `B03`:

```
  caught  B03_zero_flag_low_nibble           6 tests, 2 FAILED           2.7s
        t04_flags_are_state
            AssertionError: after r1 = 16 the flags are N=0 Z=1, expected N=0 Z=0.
```

What makes this test work is the value 16. Its low four bits are zero but the byte as a whole isn't, so a full-byte zero detector and a nibble-wide one give different answers for it. 32, 64, 128 and 240 would work too, but 5, 255 and 0 would not, because both detectors agree on them. If you pick the values for a flag test without thinking about the flag, the bug tends to agree with the correct design on every one of them.

Branching on a wrong flag

So far the bug is a wrong bit in a status register that nobody reads. What makes it a chip you can't ship is a branch that reads it, as in the last four lines of `R07` in `test_reference.py`:

```
    # And the consequence, which is what actually bites: a branch on a bad flag.
    prog = set_reg(1, 16) + [SUB(0, 0), ADD(1, 3), BZ(1), ORI(1), SUB(3, 3)]
    await sp.run(prog)
    assert sp.reg(0) == 1, (
        f"r0 is {sp.reg(0)}, expected 1: the BZ after r1 = 16 was TAKEN, so the "
        f"core thinks 16 is zero.")
```

`r1` holds 16. `ADD r1, r3` adds zero, so `r1` is still 16 and the flags now describe 16. Then `BZ +1` asks "was that zero?" and skips the next instruction if it was.

On the good core `Z` is 0, the branch is not taken, `ORI 1` runs, and `r0` is 1. On the broken core `Z` is 1, the branch is taken, `ORI 1` is skipped, and `r0` is 0.

Two instructions later the machine is running different code, even though no register and no byte of memory was ever wrong. The program went the other way at the branch, so everything after it is wrong, and none of that looks like a flag bug any more.

> A flag set late or set wrong is invisible until something branches on it. The gap between the defect and the symptom is however many instructions run between the compare and the branch, and in real firmware that can be a hundred.

A flag that changes when it should not

The mirror-image bug is `B09_branch_clobbers_flags`: one added line that makes a branch write the flags with the result of its own PC arithmetic.

SimProc's branches don't set flags, and that's deliberate. The ISA note gives the reason in one sentence: *"you can compare once and branch on it twice."* That is a real idiom (`BZ here; BNZ there`), and it's the only stimulus that can see this bug, because with one branch per compare everything still works.

This is the reference suite against `B09`:

```
  caught  B09_branch_clobbers_flags          11 tests, 2 FAILED             18.1s
        R05_flags_survive_a_branch
            AssertionError: r0 is 8, expected 1.
        R07_zero_and_negative_flags
            AssertionError: after r1 = 128 the N flag is 0; N is bit 7 of the result
```

The program in `R05` is eight lines and reads like a proof:

```
        SUB   r1, r1        # r1 = 0  ->  Z = 1, N = 0
        BZ    taken         # taken
        ORI   16            # skipped
taken:  BNZ   wrong         # must NOT be taken: Z is still 1
        ORI   1             # r0 = 1, the proof we got here
        JMP   done
wrong:  ORI   8             # r0 = 8 if the flags were clobbered
done:   SUB   r3, r3
```

`r0 == 1` means the flags survived the first branch, and `r0 == 8` means they did not, so a single register with two possible values tells you which of the two things happened. Write your own assertions the same way, so that a wrong answer reads as a sentence and not just as a number.

What to put in your own suite

Here are four rules, each of which costs about one line of code per instruction:

1. After every flag-setting instruction, assert `N` and `Z`. `sp.flags()` returns the pair.
2. After every instruction that must leave the flags alone (`LOAD`, `STORE`, all four branches), assert they did not move: read them before and after, and compare.
3. Pick values that tell the two implementations apart. For `Z`: something non-zero whose low nibble is zero (16, 32, 64, 128, 240). For `N`: something ≥ 128, and something just below it. A value like `5` tells you nothing about either flag.
4. Then branch on the flag and check where you landed. A flag assertion catches the defect and a branch catches its consequence, and the consequence is what turns up in silicon.

`set_reg()` clobbers `N` and `Z`, because it is built from `SUB`, `ORI` and `SHIFT`, all of which set flags. So set your registers up before the compare, never between the compare and the branch. The harness docstring warns about this too, and it's the most common way for a flag test to end up testing `set_reg` instead of the design.

Check yourself

1. Which of these values would catch `B03`'s nibble-wide zero detector: 0, 8, 48, 255?
2. A suite reads every register after every instruction. Why can it still miss `B09`?
3. You set up `r1` with `set_reg` between a `SUB` and a `BZ`. What is the `BZ` now testing?

Next: Lab 03: Three blocks before the processor.