Lab 03: Three blocks before the processor

SimProc is 1,178 lines long and you can't test all of it, but three of its modules are only 16, 31 and 34 lines, and those you can test completely. In this lab you'll finish the tests for the program counter, find out how the register file treats `r0`, drive all 393,216 legal vectors through the ALU, and watch a sweep of the entire input space miss a bug until you start checking the flags as well as the values.

This lab takes about 2 hours.

Package: `labs/lab-03-three-blocks-before-the-processor/`

Before you start

- Finish Lab 02: you have seen `13 of 13`.
- Read The lazy testbench, Coverage as a question and Flags as architectural state.
- Keep SimProc in one page open in another tab.

You don't need the chip around a module to test it. cocotb hands the simulator a list of source files and the name of one module to elaborate as the top level, so if you give it `simproc_system.sv` and `TOPLEVEL=ALU`, Icarus compiles all 1,178 lines but builds a design that contains only the ALU. The other 1,100 lines are compiled and left unconnected, so the design has no processor, no clock and no memory, just three input ports and three output ports. That makes the number of cases you have to cover very different from one block to the next:

	Inputs	Possible vectors	Exhaustive?
`program_counter`	`rst`, `pc_wr`, `pc_in[7:0]` + one register of state	4 distinct behaviours	Yes, by cases
`register_file`	`rst`, `RFWrite`, three 2-bit selects, `dataW[7:0]` + 32 bits of state	Small	Yes, by cases
`ALU`	`ALUop[2:0]`, `A[7:0]`, `B[7:0]`, no state	$2^{19} = 524,288$, of which 393,216 are legal	Yes, by brute force
`simproc`	All of the above plus 64 bytes of RAM and a PC	Far too many	No

You can finish testing a small block, which you can't do for a processor, and once the small blocks are finished you know that any failure in the processor lies somewhere outside them. Much of verification is this kind of work: shrinking the region a bug could be hiding in.

```
mod dd104     # in a new terminal, load .designinit first (Lab 01, Part 1)
cd labs/lab-03-three-blocks-before-the-processor
```

A bare `make` runs `pc`, `rf`, `alu` and then `check` (against `B01`), which takes about a minute. You don't need to run it now, because the individual targets below are faster and they're the ones you'll use.

Part 1: The program counter

What does a complete unit test of a small block look like?

1. Run the program-counter tests, which ship complete:

   ```
   make pc
   ```

   It takes about 3 seconds. You should see:

```
** test_pc.t01_reset_makes_it_zero   PASS          21.00 **
** test_pc.t02_load_when_pc_wr       PASS        2571.00 **
** test_pc.t03_hold_when_not_pc_wr   PASS          61.00 **
** test_pc.t04_reset_beats_load      PASS          21.00 **
** TESTS=4 PASS=4 FAIL=0 SKIP=0                 2674.00 **
```

All five simulated times are exact. The same 28 Icarus `sorry:` notes you saw in Lab 02 also appear on these unit-level compiles, and Output that is not an error explains them.

2. Read `test_pc.py`. It's 120 lines including comments, and it's the model for everything you write in this lab: drive on one clock edge, sample after the other, and never let your stimulus race the DUT's clock. `program_counter` is an 8-bit register with a synchronous reset and a load enable, so there are exactly four things you can ask it to do: reset, load, hold, and reset while loading. The file tests all four.

3. `t03_hold_when_not_pc_wr` and `t04_reset_beats_load` are the two tests people usually skip. What could go wrong in the RTL that each one catches and `t02` does not?

   Answer: A1, in Answers at the end.

Check: `TESTS=4 PASS=4`, `2674.00` ns.

Part 2: The register file

Is `r0` hardwired to zero, as it is on RISC-V and MIPS?

1. Run the register-file tests, which ship with two tests and three TODOs:

   ```
   make rf
   ```

   You should see `TESTS=2 PASS=2` and `51.00` ns total, both exact.

2. Each TODO is a comment describing a test, and you write it as a new `@cocotb.test()` function, the way `t01` and `t02` are written. Do `t03` first. Most small processors hardwire register 0 to zero: RISC-V's `x0` reads as zero and discards writes, and so does MIPS's `$zero`. Write down whether you expect the same here, then check it: write `0xFF` into `r0` and read it back, then write a different value and read it again. One write that lands doesn't prove anything, because a register that latched the first write and then stuck would pass too. The file's `write()` and `read()` helpers each do one write or one read, and `t01` and `t02` show them in use. `make rf` runs against the correct register file, so if your assertion fails it was your expectation that was wrong, and you should fix the test rather than the RTL.

3. Do `t04` and `t05`. They test two properties you'll see everywhere: one write changes one register, and an enable that isn't asserted writes nothing. Run `t05` for several cycles rather than one, because "it didn't write on the first edge" and "it doesn't write" are different claims, and only the second is the requirement.

4. Run `make rf` again. Was your prediction about `r0` right?

   Answer: A2, in Answers at the end.

Check: `make rf` runs five tests, all passing, and you know whether `r0` is hardwired.

Part 3: All 393,216 ALU vectors

Is the ALU right on every vector it can be given?

This is all of the ALU's logic, from `labs/common/rtl/simproc_system.sv`, module `ALU`, lines 790–823 (port list and `aluop_t` enum omitted):

```
always @ (*) begin
    case(ALUop)
        ADD:     ALUout = A + B;
        SUB:     ALUout = A - B;
        OR:      ALUout = A | B;
        NAND:    ALUout = ~(A & B);
        SHL:     ALUout = A << B[1:0];
        SHR:     ALUout = A >> B[1:0];
        default: ALUout = 8'h00;
    endcase
end
assign N = ALUout[7];
assign Z = ~(|ALUout);
```

The shifts use only `B[1:0]`, so the shift amount is 0–3. `N` and `Z` depend only on the result: `N` is bit 7 of the output, and `Z` is set when the output is all zeroes. There is no carry or overflow flag.

> Write your specification model from the ISA, not from the Verilog. A model you write by reading the RTL agrees with the design about everything, bugs included, and that is the most common way a verification effort wastes six months. `spec()` in `test_alu.py` is written from the ISA in exactly this sense, and every check in this lab uses it as the model.

1. Run the ALU tests:

   ```
   make alu
   ```

   It takes about 23 seconds, 20 of them in `t02`. You should see:

   ```
   393216 vectors checked, 0 value mismatches
   ** test_alu.t01_one_vector_per_operation            PASS          6.00 **
   ** test_alu.t02_every_vector_the_alu_can_be_given   PASS     393216.00 **
   ** test_alu.t03_the_undefined_opcodes               PASS          2.00 **
   ** TESTS=3 PASS=3 FAIL=0 SKIP=0                                393224.00 **
   ```

   `393216`, `0 value mismatches` and `393224.00` are exact.

That is six operations × 256 × 256, every legal vector the module can be given, each checked against a specification model, so no corner of the input space is left unexplored (although Part 5 shows what checking only the value misses). Usually you can't test exhaustively. A pair of 8-bit operands is 65,536 vectors, which is fine, and an 8-bit ALU with a 3-bit opcode is 524,288, which is also fine, but a 32-bit adder is 2^{65}, which you'll never get through. When a block is small enough, though, brute force beats cleverness.

Check: `393216 vectors checked, 0 value mismatches`.

Part 4: Two bugs, and the count

What does the number of wrong vectors tell you about a bug?

1. Run the check against the first two ALU bugs:

```
make check BUG=B01
make check BUG=B02
```

Each takes about 45 seconds. You should see both `caught`, with a count in the message:

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

```
65280 of 393216 vectors have the wrong value: {'NAND': 65280}.
  First one: NAND(0x00, 0x01) = 0xFE, expected 0xFF.
```

`65024`, `65280` and the per-operation dictionaries are exact. No other operation appears in either dictionary, and that's part of the diagnosis, because it tells you which operations aren't broken.

2. Derive both counts by hand. The exercise is the derivation, and the counts from the run are only there for you to check it against. `B01` swaps the operands of `SUB`. Which `SUB` vectors survive it?

   Answer: A3, in Answers at the end.

3. `B02` makes `NAND` compute NOR. Which `NAND` vectors survive it?

   Answer: A4, in Answers at the end.

4. What do the two sets of survivors have in common?

   Answer: A5, in Answers at the end.

Check: your derivations give 65,024 and 65,280.

Part 5: Value-only checks against `B03`

Can a testbench that checks only values see a bug in the `Z` flag?

1. `B03` breaks the zero flag. The shipped `test_alu.py` checks `ALUout` on all 393,216 vectors and ignores the flags. Before you run it, write down whether you think it will catch `B03`. Then run:

   ```
   make check BUG=B03
   ```

   You should see:

   ```
        ok  the golden ALU                    3 tests, all passed            18.6s
    MISSED  B03_zero_flag_low_nibble          3 tests, all passed            18.7s
  ----------------------------------------------------------------------
   VERDICT: FAIL -- test_alu passes B03_zero_flag_low_nibble too.
     That bug hides behind: every test that checks results and not flags
   ```

   The testbench drove all 393,216 vectors and every value was correct, even though the ALU is broken. Would adding more vectors help?

   Answer: A6, in Answers at the end.

2. The whole processor has the same blind spot, and you can see it there too. In `labs/common`, run two testbenches against `B03`: first `test_values_only.py`, a thorough testbench that checks only values, and then `test_example.py`, the six-test example from Lab 02:

```
cd ../common
make check TB=test_values_only BUG=B03
make check BUG=B03
cd ../lab-03-three-blocks-before-the-processor
```

The first takes about 40 seconds and should say `MISSED  B03_zero_flag_low_nibble`. The second should say `caught  B03_zero_flag_low_nibble  6 tests, 2 FAILED`, with `t04_flags_are_state` reporting `after r1 = 16 the flags are N=0 Z=1, expected N=0 Z=0.` Flags as architectural state explains why 16 is the value that catches it.

Check: `MISSED` on `B03` from both value-only testbenches, and `caught` from `test_example`.

Part 6: Add the flags

What does a sweep that checks the flags report on `B03`, and can you account for every count?

1. Do the TODO in `t02` of `test_alu.py`. `spec()` already returns `(out, N, Z)`, so compare all three, and keep value mismatches and flag mismatches in separate counts.
2. Run all three ALU bugs at once:

   ```
   make check-alu-bugs
   ```

   It takes about 90 seconds, because it does four full builds and four exhaustive sweeps. You should see `3 of 3 ALU bugs caught`, and `B03` failing with:

   ```
   393216 vectors checked -- 0 value mismatches , 25726 flag mismatches
   {'ADD': 3840, 'SUB': 3840, 'OR': 255, 'NAND': 255, 'SHL': 14400, 'SHR': 3136}
   ```

   All the numbers are exact. Your message can be worded differently, and it doesn't have to break the count down by operation, but the totals should match. What does "0 value mismatches, 25,726 flag mismatches" tell you that a single combined count of 25,726 would not?

   Answer: A7, in Answers at the end.
3. If instead you see `NO COUNT`, your testbench crashed rather than caught anything:

   ```
   NO COUNT. 3 of the 3 runs did not finish:
     B01_sub_operands_swapped: t02_every_vector_the_alu_can_be_given BLEW UP
        NameError: name 'value_per_op' is not defined
   ```

   The code that builds your two-count report only runs when something is wrong, and that's the one condition you never test it under, since against the correct ALU it never runs. A student's `t02` once crashed like this on all three broken ALUs, and an older `check.py` counted the three crashes as three catches. Fix the name and run it again.
4. Account for the counts by hand. The bug makes `Z` wrong exactly when the result is non-zero but its low four bits are zero, which is true of 15 possible results: `0x10`, `0x20`, … `0xF0`. Start with `ADD` (3,840) and `OR` (255).

   Answer: ADD: A8, in Answers at the end.

   Answer: OR: A9, in Answers at the end.

5. Do `NAND` yourself; it's the same argument as `OR`, with `0xF` in place of `0`. `SHL` (14,400) and `SHR` (3,136) each need one sum over the four shift amounts, and they're worth ten minutes.

   Answer: NAND: A10, in Answers at the end.

   Answer: SHL and SHR: A11, in Answers at the end.

Check: `3 of 3 ALU bugs caught`, and your hand counts match.

Part 7: An undefined opcode

Why keep a test for behaviour that no specification requires?

1. Read `t03_the_undefined_opcodes` in `test_alu.py`, which ships passing. It checks that `ALUop` 6 and 7 both return `0x00`, with `N=0` and `Z=1`.
2. Is that behaviour correct? Those two opcodes aren't in the enum. Write down your answer, and what you think `t03` is for.

   Answer: A12, in Answers at the end.

Check: you can say why `t03` exists even though no specification requires the behaviour it checks.

Expected results

Check	Expected	
`make pc`	`TESTS=4 PASS=4`, `2674.00` ns	exact
`make rf`, shipped	`TESTS=2 PASS=2`, `51.00` ns	exact
`make rf`, finished	`TESTS=5 PASS=5`	
`make alu`	`393216 vectors checked, 0 value mismatches`, `393224.00` ns	exact
`make alu` wall clock	about 23 s	ballpark
`B01`	`{'SUB': 65024}` value mismatches	exact
`B02`	`{'NAND': 65280}` value mismatches	exact
`B03`, values only	`MISSED`, 0 mismatches	exact
`B03`, with flags	0 value, 25726 flag: `{'ADD': 3840, 'SUB': 3840, 'OR': 255, 'NAND': 255, 'SHL': 14400, 'SHR': 3136}`	exact
`make check-alu-bugs`, when done	`3 of 3 ALU bugs caught`	
`make check-alu-bugs` wall clock	about 90 s	ballpark

Finishing

You're done when `make check-alu-bugs` on your `test_alu.py` says `3 of 3 ALU bugs caught` and `make rf` runs your five tests, all passing.

Questions:

1. The catalogue has thirteen bugs and this lab reached three. Why can't a unit test of the ALU see the other ten?

   Answer: A13, in Answers at the end.

Submit: nothing for this lab.

Next on the Course path: read The golden model, Where bugs hide and Directed and random tests, then do Lab 04: Your first bug hunt.

Answers

A1. Part 1: The program counter

`t03` drives junk on `pc_in` with `pc_wr` low and checks that the PC didn't change. A PC that loads correctly but never holds its value would pass `t02` perfectly. `t04` raises `rst` and `pc_wr` together, because the priority between two control inputs is a decision somebody made. You can't see that decision from the port list, and the next person to touch the file can reverse it just by swapping two branches.

A2. Part 2: The register file

`r0` is an ordinary register here, and it holds whatever you last put in it. `ORI`, the only instruction that can bring a constant into a register, writes `r0` and nothing else, so every `ORI` in this course overwrites whatever `r0` was holding. It's better to learn that now than at 1 a.m. in Lab 06.

A3. Part 4: Two bugs, and the count

The survivors are the pairs where $A - B$ and $B - A$ are the same byte, which happens when $2(A - B) \equiv 0 \pmod{256}$, that is, when $A - B \equiv 0$ or $A - B \equiv 128$. Each congruence has 256 solutions, so there are $256 + 256 = 512$ survivors and $65,536 - 512 = 65,024$ failures.

A4. Part 4: Two bugs, and the count

$\neg(A \wedge B) = \neg(A \vee B)$ exactly when $A \wedge B = A \vee B$, which is exactly when $A = B$. That is 256 survivors, and $65,536 - 256 = 65,280$.

A5. Part 4: Two bugs, and the count

Each contains the most idiomatic use of the instruction. `SUB rX, rX` is the idiom for clearing a register, it appears in nearly every program on this machine, and it is one of `B01`'s 512 survivors. On a four-register machine with no `NOT`, `NAND rX, rX` is how you write `NOT`, which makes it the most common `NAND` anyone writes, and it is exactly the case where `B02` is invisible. A bug that hides in the commonest use of an instruction can survive review, which is why these two are in the bug catalogue.

A6. Part 5: Value-only checks against B03

No. A value-only testbench can't see this bug however many vectors you add, because the values aren't what's wrong. To catch it you need a different kind of check, one that looks at the flags.

A7. Part 6: Add the flags

It tells you that the arithmetic is fine and the flag logic is broken, before anyone has opened a waveform. That's the reason for keeping two counts.

A8. Part 6: Add the flags (Answer: ADD)

For any target byte t, exactly 256 pairs (A, B) satisfy $A + B \equiv t$. Fifteen targets at 256 pairs each is $15 \times 256 = 3840$. `SUB` is the same.

A9. Part 6: Add the flags (Answer: OR)

$A \vee B$ has a zero low nibble only when $A[3:0] = B[3:0] = 0$, which is $16 \times 16 = 256$ pairs. One of those, $A = B = 0$, gives a genuinely zero result, and a low-nibble detector gets that one right by accident, so the count is $256 - 1 = 255$.

A10. Part 6: Add the flags (Answer: NAND)

$\neg(A \wedge B)$ has a zero low nibble only when `A[3:0]` and `B[3:0]` are both `0xF`, which is 256 pairs. One of those, `A = B = 0xFF`, gives a genuinely zero result, so the count is $256 - 1 = 255$.

A11. Part 6: Add the flags (Answer: SHL and SHR)

Only `B[1:0]` matters, so each shift amount s covers 64 values of `B`. Count the values of `A` whose shifted result is non-zero with a zero low nibble, for $s = 0, 1, 2, 3$:

- `SHL`: 15, 30, 60 and 120, for a total of $64 \times 225 = 14,400$.
- `SHR`: 15, 14, 12 and 8, for a total of $64 \times 49 = 3,136$.

A12. Part 7: An undefined opcode

The `case` has `default: ALUout = 8'h00;`, so both opcodes return zero, which also sets `Z`. A zero result with `Z` set looks like an ordinary answer, and nothing downstream can tell "the answer was zero" from "that wasn't an instruction". The behaviour is defined, but no specification asks for it; it's simply what the RTL happens to do.

`t03` pins that behaviour down, so that if somebody changes it (to X, to a trap, to a passthrough) a test goes red and a person has to decide whether the change was intended. A test can record a decision as well as check a requirement, and the comment above it tells the next reader which of the two it's doing.

A13. Finishing

Nine of the other ten are in the control path around the ALU (decode, write-back, branching, the handshake), and the tenth, `B12`, is in the UART receiver. `make check-alu-bugs` says so in its closing line. Finishing a block proves that the block is right, but it says nothing about whether the processor uses the right block, with the right operands, at the right time. Testing that is the subject of Lab 04.