Lab 05: Every answer is right

In this lab you'll meet two broken processors that get every register, flag, PC value and byte of memory right, for every program you can write. The golden model agrees with both of them completely, and an architecturally complete suite built from Lab 04's passes on both. You'll write three short checks that do catch them, run a test that measures the UART receiver's turnaround time, and then hunt both bugs.

This lab takes about 75 minutes.

Package: `labs/lab-05-every-answer-is-right/`

Before you start

- Finish Lab 04, so that you have a suite you trust.
- Read Handshakes and timing.
- Keep The SimProc harness open in another tab, for `count_done_pulses`, `count_halt_cycles`, and what `run()` returns.

SimProc's contract has three parts that `golden.py` says nothing about:

	What it is	Correct behaviour
`done`	One-cycle pulse as an instruction retires	Exactly one pulse per retired instruction
`halt`	High while the core sits in `IDLE`	Never high while `RUN` is asserted
The clock	`sp.run()` returns an exact count	`ORI` 5; `ADD`/`SUB`/`NAND`/`SHL`/`SHR`/`LOAD` 4; `STORE` and every branch 3

Anything outside the core that wants to know whether an instruction finished (a debugger, a performance counter, a testbench scoreboard) watches `done`. Anything that wants to know whether the processor is running watches `halt`. The cycle count is part of the contract as well, because a design that produces every right answer one cycle late has still missed its timing budget.

```
mod dd104     # in a new terminal, load .designinit first (Lab 01, Part 1)
cd labs/lab-05-every-answer-is-right
```

Part 1: The architectural suite against both cores

Does a suite that checks every architectural result catch these two cores?

1. Lab 04's finished suite compares every register, flag and byte of memory with the golden model. Before you run it against this lab's two cores, write down whether you expect it to catch either. Then run:

   ```
   make blind
   ```

   It takes about 41 seconds. You should see:

   ```
   ============================================================
    BLIND SPOT   test_architecture.py -- ten architecturally complete
                 tests -- against both cores in this lab's pool.
   ============================================================
         ok  the golden core                          10 tests, all passed      13.4s
     MISSED  B08                                      10 tests, all passed      13.5s
   ```

```
 MISSED  B13                                          10 tests, all passed      13.4s
--------------------------------------------------------------------------------
 Three identical green rows. Every architectural result on all three
 of those processors is correct, and two of them are broken.

 What test_architecture.py never reads: the `done` pin, the `halt`
 pin, and the number of clock cycles anything took. Those are the
 three things this lab is about.
```

`test_architecture.py` is Lab 04's finished suite with its one cycle-count assertion removed (question 1 in Finishing explains why). Its ten tests compare all four registers, the PC, both flags and every byte of memory with the golden model after every instruction of a mixed program. They also cover every shift amount in both directions, all sixteen branch displacements, a compare followed by two branches, and `LOAD` and `STORE` through pointers held in a different register from the data. The suite catches every bug in Labs 03 and 04.

2. Would more vectors, programs or register assertions help it catch these two?

   Answer: A1, in Answers at the end.

Check: you see three green rows, and two of them are for broken cores.

Part 2: Count the pulses

How many clock cycles should this lab's program take, and how many `done` pulses?

1. Run the shipped `test_lab05.py` against the golden core:

   ```
   make sim
   ```

   You should see `TESTS=2 PASS=2` and `88810.00` ns, and in the log:

   ```
   the program ran in 37 clock cycles
   a done-triggered scoreboard would have made 9 comparisons for a 9-instruction program
   ```

   Both `37` and `9` are exact.

2. Every test in this lab runs the same nine-instruction program, `PROGRAM` at the top of `test_lab05.py`, and one of its instructions is a `STORE`. Work out the 37 yourself from the instruction costs in the table at the top of this page, and write the sum down, because you'll assert it in Part 6.

   Answer: A2, in Answers at the end.

The shipped test `t02` in `test_lab05.py` reports the pulse count but asserts almost nothing about it, only that at least one instruction retired. We left it that way on purpose, and the next part shows why.

Check: the log shows `37 clock cycles` and `9 comparisons`.

Part 3: The shipped file against `B08`

What happens to a scoreboard triggered by `done` on a core that drops a `done` pulse?

1. Before you run, predict whether the shipped file, which only asserts that at least one instruction retired, will catch `B08`. Then run it:

   ```
   make check BUG=B08
   ```

You should see `MISSED`, which means the shipped file passes on the broken core.

2. `make check` doesn't show the simulator's log, so to see the log of the broken run, call the checker directly with `-v`:

```
python3 ../common/check.py --tb test_lab05 --bug B08 --dir . -v
```

You should see the same `MISSED` verdict followed by the log, which includes `a done-triggered scoreboard would have made 8 comparisons for a 9-instruction program`.

3. Nothing went red and no assertion failed. What would a scoreboard triggered by `done` have reported on this core, and why is that worse than a failure?

Answer: A3, in Answers at the end.

Check: the log of the broken run says 8 comparisons.

Part 4: Assert the number

Can you make the pulse count fail loudly, with a message that points at the culprit?

1. Do the first TODO in `test_lab05.py`: replace `assert pulses > 0` with the exact count, and put the count in the message.

2. Run `make check BUG=B08` again. You should see:

```
  caught  B08_store_drops_done              4 tests, 1 FAILED           3.5s
        AssertionError: `done` pulsed 8 times for a 9-instruction program.
```

The output blocks here and in Part 5 come from the finished file, which has all four tests, so until you've written the other two, your output will show fewer tests. The reference message continues:

> The shortfall is 1. Look at the instructions that appear exactly 1 time(s) in this program: one of them retired without pulsing `done`.

A message like `8, expected 9` is far more help than "some pulses are missing", because it tells you the shortfall is one instruction, so you know to look at the instructions that appear only once in the program: the one `SHL` and the one `STORE`. A failure message that does the arithmetic for the reader costs one line to write and can save you an hour.

Check: `make check` reports `caught` on `B08`.

Part 5: Watch `halt`

Is `halt` ever high while the core is running?

1. Write the second TODO as a new test. It uses `await sp.count_halt_cycles(n)`, which runs the loaded program for `n` cycles of execution and returns how many of those cycles had `halt` high. For `n`, use the cycle count that `sp.run()` returns, and call `sp.reset()` before counting, the same way `t02` does for `count_done_pulses`. Then assert that the result is zero.

2. Run it against `B13`:

```
make check BUG=B13
```

You should see:

```
caught  B13_store_falls_to_idle           4 tests, 2 FAILED           3.6s
      AssertionError: `halt` was high on 1 of the 38 cycles the core spent running.
      AssertionError: the program took 38 clock cycles, expected 37.
```

The second line only appears once you've done Part 6 as well. The numbers `1`, `38` and `37` are exact.

This is the cheapest useful assertion in the course. It compares one number with one expected value and needs no arithmetic, and zero is the only correct answer, because while `RUN` is asserted the core is not supposed to visit `IDLE` at all.

3. Every value `B13` produces is correct, in the right order and the right place. Is it a real bug? Decide before you open the answer. The answer also says what `B13` changes, so leave it closed if you'd rather find that out yourself in Part 8.

   Answer: A4, in Answers at the end.

Check: `make check` reports `caught` on `B13`, with `halt` high on 1 cycle.

Part 6: Assert the clock

Does the program take exactly as many cycles as the instruction costs say?

1. Do the third TODO: assert `cycles == 37`, with the sum from Part 2 in the failure message.
2. Run `make check BUG=B13` again. You should see the second failure from Part 5, `the program took 38 clock cycles, expected 37.`
3. What does 38 − 37 tell you about where to look?

   Answer: A5, in Answers at the end.

Check: both `B13` failures appear.

Part 7: The UART receiver's busy time

What happens to a serial command when the sender doesn't wait for the receiver to finish?

1. In `labs/common`, run the handshake-timing tests:

   ```
   cd ../common
   make sim TB=test_handshake_timing
   ```

   You should see `TESTS=4 PASS=4` and `90910.00` ns, with these lines in the log:

   ```
   clk_per_bit = 32; receiver states: START=16  RECEIVE=288  DONE=32
   sent    : 0x99 0x0C 0x00
   received: 0x99 0x86 0x80
   with the inter-frame gap: PC = 12.  Without it: PC = 134.
   gapless PING answered 0xAA -- a green light on a link that just corrupted two of its
   three bytes
   one 3-byte command costs 1104 clock cycles = 3 x 11 x 32 (frame + gap) + 32 + 16 = 1104
   the core fetched its first instruction 57 clock cycles BEFORE uart_cmd(RUN) returned
   ```

2. Explain `0x86` from `0x0C` using one rule.

   Answer: A6, in Answers at the end.

3. Now change the command bytes. Before you run anything, use your rule to predict what `0x34, 0x12` will arrive as. Then run it:

```
sed 's/^COMMAND = (CMD_SET_PC, 12, 0)/COMMAND = (CMD_SET_PC, 0x34, 0x12)/' \
    test_handshake_timing.py > test_gap_mine.py
make sim TB=test_gap_mine
```

   You should see:

```
sent    : 0x99 0x34 0x12
received: 0x99 0x9A 0x89
with the inter-frame gap: PC = 52.  Without it: PC = 154.
if every frame after the first were sampled ONE BIT PERIOD LATE, they would arrive as: 0x99
0x9A 0x89
```

   The prediction and the measurement agree.

4. Repeat step 3 with `0x0F, 0x33` and with `0x55, 0xFF` in place of `0x34, 0x12`. Which one delivers three bytes but doesn't follow the one-bit-period rule, and why does `0x55, 0xFF` deliver only two bytes?

   Answer: A7, in Answers at the end.

5. Delete your copy and go back to the lab directory:

```
rm test_gap_mine.py
cd ../lab-05-every-answer-is-right
```

Check: you can explain the received bytes for at least the first two commands.

Part 8: Hunt them properly

Can you find and fix both bugs with your new checks?

1. Hunt both bugs in the pool, one at a time:

```
make new
make hunt
make verdict
```

   The three-row grade is the same as Lab 04's, and row 2 is still the deliverable. `make hunt` and `make verdict` run `test_lab05.py`, which is where your checks from Parts 4–6 are. With an architectural suite alone, `make hunt` will say `quiet` however much stimulus you add, and the way out is a different kind of check.

2. As in Lab 04, `make probe` shows everything the simulator printed, including a probe program's `print()` output. `make hunt VERBOSE=1` and `make verdict VERBOSE=1` do the same.

3. Run `make new` again until you've been dealt both bugs and fixed them.

Check: `make verdict` passes on both bugs.

Expected results

This table is the complete difference between a working processor and these two:

	Golden	B08	B13
done pulses over the program	9	8	9
halt high, of the running cycles	0	0	1
Clock cycles, sp.run()	37	37	38
Every register, flag, byte of memory, PC	Correct	Correct	Correct

All twelve entries are exact, and three of them differ from the golden core's.

Check	Expected	
make blind	ok, MISSED, MISSED	exact
make blind wall clock	about 41 s	ballpark
make sim, shipped file	TESTS=2 PASS=2, 88810.00 ns	exact
make sim TB=test_handshake_timing in labs/common	TESTS=4 PASS=4, 90910.00 ns	exact
make verdict, when done	three green rows, on both bugs	

Finishing

You're done when make verdict shows three green rows and VERDICT: PASS on both B08 and B13, with row 2 caught by your checks from Parts 4–6.

Questions:

1. test_architecture.py is Lab 04's suite with one assertion removed: the exact cycle count in t04_a_loop. With it left in, Lab 04's suite catches B13 by accident, and make blind doesn't work. So was Lab 04's cycle assertion a protocol test living in an architectural suite?

   Answer: A8, in Answers at the end.

2. What would have made any scoreboard catch B08, however it was triggered?

   Answer: A9, in Answers at the end.

Submit: nothing for this lab. Keep test_lab05.py; you'll bring it into Lab 06.

Next on the Course path: read Writing the bug report, then do Lab 06: Capstone.

Answers

A1. Part 1: The architectural suite against both cores

No. Nothing is wrong with the suite: it reads architectural state, and neither of these bugs is in architectural state, so more vectors, programs or register assertions won't reach them. Catching them takes a different kind of check.

A2. Part 2: Count the pulses

```
SUB 4 + ORI 5 + SHL 4 + SUB 4 + ADD 4 + SUB 4 + ORI 5 + ADD 4 + STORE 3 = 37
```

A3. Part 3: The shipped file against B08

It would have compared eight machine states with the model, found all eight correct (they are), and reported a clean run. The ninth instruction did execute, and it wrote the right byte to the right address, but it never signalled that it had finished, so nothing outside the core noticed it. Nine instructions retired with only eight pulses, and that is the whole of the observable defect.

This is worse than a failure because a missing trigger doesn't make your test fail. It stops part of your test from running, and nothing on the screen tells you so, whereas a failure at least shows up.

A4. Part 5: Watch halt

Yes. `B13` drops the core into `IDLE` for one cycle after every `STORE` and then restarts it at once, because `RUN` is still asserted. Its only symptoms are one wasted cycle and a one-cycle blip on a pin, but on a real chip that is a performance bug (one store in a tight loop, run a million times, is a million wasted cycles) and also a glitch any debugger can see, because anything watching `halt` sees the processor stop and start again. "Functionally equivalent" doesn't mean "correct": it's a claim about outputs, made by someone who has already decided which outputs count.

A5. Part 6: Assert the clock

$38 - 37 = 1$, so there is one extra cycle, and it comes from an instruction that runs once. The program has one `SHL` and one `STORE`, so the count narrows the search to those two. By the same division, in a program with five `STORE`s and no other instruction that runs five times, five extra cycles would point straight at the stores. This check turns "the answer is right and arrived late" into a red row with a number attached, and all it costs is one addition, done once, from the instruction costs in the ISA reference.

A6. Part 7: The UART receiver's busy time

Every bit after the first frame was sampled one bit period late, so bit *i* of the received byte is bit *i+1* of the sent one, and the top bit comes from the stop bit, which is always 1: `(0x0C >> 1) | 0x80 = 0x86`. `0x00` becomes `0x80` by the same rule. Handshakes and timing has the full explanation.

A7. Part 7: The UART receiver's busy time

`0x0F, 0x33` arrives as `0x99 0x87 0xE6`. The second frame slips cleanly, but then the receiver resynchronises onto a different edge, so the third byte is neither what you sent nor what a uniform slip would give (`0x99`). `0x55, 0xFF` arrives as `0x99 0xAA`, only two bytes, because after its start bit `0xFF` is a solid run of ones with no falling edge for the receiver to latch onto. The third frame never happens, and the PC stays 0. The table in Other bytes has all four runs.

A8. Finishing

It was, and it's still a good test that belongs in Lab 04. The labels "architectural" and "protocol" are useful because they tell you what kinds of check you're missing; they don't mean every test has to fit in exactly one box.

A9. Finishing

Asserting how many times the scoreboard fired, as well as that every comparison it made agreed. That takes one counter and two lines, and it turns the worst failure mode in verification into an ordinary red row.