Lab 04: Your first bug hunt

In this lab you're dealt a processor in which one line has been changed, and nobody tells you which line. You'll write tests until one of them goes red, narrow the bug down, fix the line, and then prove the fix. The deliverable is the failing test rather than the fix, and the grader checks for exactly that.

This lab takes about 2 hours 30 minutes, plus whatever time you give the optional Part 8.

Package: `labs/lab-04-your-first-bug-hunt/`

Before you start

- Finish Lab 03, which ends with `3 of 3 ALU bugs caught`.
- Read The golden model, Where bugs hide and Directed and random tests.
- Keep The SimProc instruction set and The SimProc harness open in other tabs.

At the end, `make verdict` grades your work in three rows, in this order:

```
      ok  1. the golden core          must PASS      10 tests, all passed       9.4s
  caught  2. the core you were dealt  must FAIL      10 tests, 1 FAILED         9.3s
      ok  3. core/ as it stands now   must PASS      10 tests, all passed       9.3s
```

Row 1 runs your tests on the golden core, and it comes first because a test that fails on a correct processor is reporting a bug in the test, not in the processor. Until row 1 passes, the other two rows don't tell you anything. Row 3 checks your fix. Row 2 is the deliverable, and it matters most because a patch is something one person did once, whereas a test that fails on the old core is what stops the bug coming back a year from now, in a rewrite, done by someone who has never heard of you. The grader rebuilds the row 2 core fresh from the bug catalogue on every run and never reads it from `core/`, so you can't turn row 2 green by editing the file you're testing.

Don't diff `core/` against `../common/rtl/`. It would work, and take four seconds, but it would skip the whole lab. The skill you're here to practise is finding a bug in a design you can't diff against a known-good copy, and every design you'll ever be paid to verify is like that.

```
mod dd104     # in a new terminal, load .designinit first (Lab 01, Part 1)
cd labs/lab-04-your-first-bug-hunt
```

Part 1: Take delivery

Which bugs could your core have, and what can you already rule out?

1. Deal a mystery core:

   ```
   make new
   ```

   You should see:

   ```
   ============================================================================
    A MYSTERY CORE
   ============================================================================
    Written to core/: the two RTL files and the testbench wrapper.
    Exactly one line of simproc_system.sv differs from the golden core.

    It is one of 5: B04, B05, B06, B07, B09.
    Three are in the decode path or the register file, between the
   ```

```
instruction word and the register it writes. Two are in the branch
path. None of them is in the ALU -- Lab 03 finished the ALU, and
every vector it can be given is already known to be right.
```

Be careful running it again later, because `make new` deletes any `core/` you already have.

2. Note what you can already rule out. Lab 03 drove all 393,216 ALU vectors and checked the value and the flags on every one, so you can stop suspecting the ALU. That eliminates 34 lines before you start, and leaves you choosing between five bugs.

The name of the bug you were dealt is in `core/.sealed`, base64-encoded. Decoding it would throw away the lab, and nothing stops you except the honour system, which the file itself spells out.

Check: `core/` exists and the banner lists five candidates.

Part 2: Start from what you have

What does the shipped suite never drive?

1. Run the shipped suite against your core:

```
make hunt
```

`test_lab04.py` has four ordinary tests, the kind a competent person would write in twenty minutes. You should see either a red row or this:

```
   quiet  core/  the mystery                        4 tests, all passed        3.4s
------------------------------------------------------------------------------
 Nothing your testbench drives can tell this core from a correct
 one. The bug is real and it is in there; your stimulus has not
 reached it yet. Widen the stimulus, not the assertions.
```

2. Read the three questions at the top of `test_lab04.py`, then read `t02` and `t03` with those questions in mind. We wrote both tests with a blind spot on purpose. The shipped tests already assert the right things about what they drive, so the gap is in what they drive. What does each of `t02` and `t03` never drive?

   Answer: A1, in Answers at the end.

Check: you can name the blind spot in `t02` and in `t03`.

Part 3: Narrow it down

Which region of the design is your bug in?

`make hunt` prints a one-line summary and throws away the simulator's output, which keeps each round of the hunt quick. To see what a probe program prints, use `make probe`, which does the same run without throwing anything away.

1. Put your probe programs in `test_probe.py`, which ships as a starting point and is never graded. Keeping them there means `test_lab04.py` stays a proper test suite. Run them with:

```
make probe TB=test_probe
```

You should see the exact `make -f sim.mk …` command it ran, then the simulator's output live, including your `print()` and `dut._log` lines:

```
================================================================================
 PROBE  testbench: test_probe   target: core/
================================================================================
 Showing everything the simulator says. Your print() and dut._log
 lines are in here, in the order they happened.
--------------------------------------------------------------------------------
  $ cd /work/labs/lab-04-your-first-bug-hunt && \
    make -f /work/labs/common/sim.mk sim \
      COMMON=/work/labs/common \
      TB=test_probe \
      ...
--------------------------------------------------------------------------------
...
     0.00ns INFO     cocotb.regression   running test_probe.t_probe (1/1)
PROBE-PRINT regs = (60, 4, 60, 0)
 33290.00ns INFO     cocotb.tb           PROBE-LOG regs = (60, 4, 60, 0)
```

`make hunt VERBOSE=1` does the same thing as `make probe`, and `make verdict VERBOSE=1` shows the full output for all three rows of the grade.

2. Work down the question list from Where bugs hide. You can answer each question with one small program, for example:

```
# Q1 -- which classes are wrong?
await sp.run(set_reg(1, 200) + set_reg(2, 60) + [ADD(1, 2)])   # arithmetic
await sp.run(set_reg(1, 0x0C) + [SHL(1, 2)])                   # shift, amount != 1
await sp.run(set_reg(2, 0x2A) + [LOAD(1, 2)])                  # load, rA != rB
```

```
# Q4 -- which register ends up wrong?
print(sp.regs())          # all four at once. A tuple, not a list.
```

3. Keep a written list of what each probe ruled in or out. If arithmetic with `rA == rB` is right and arithmetic with `rA != rB` is wrong, for instance, you have answered questions 1, 3 and 4 at once, and you're down to four lines of RTL.

As Where bugs hide put it, what separates a five-minute debugging session from a two-hour one is rarely insight. It usually comes down to whether the first ten minutes went on narrowing the problem down or on reading the RTL.

If you've spent a real hour on one core and are stuck, `make reveal` tells you the bug's name and what it hides behind, though not the line. You still have to write the test that catches it, because knowing a bug's name is not the same as having a test for it, and only the test stays with the project. Using `make reveal` after a genuine hour is a reasonable engineering decision, so if you get there, take the hint, write the test and move on.

Check: your list of candidates has shrunk to one region of the design.

Part 4: Let the model find the first disagreement

Where does your core first disagree with the golden model?

1. Add a golden-model trace to a test:

```
await sp.load(prog)
model = Golden(prog + [PARK])
steps, problem = await trace(sp, len(prog), model=model)
assert problem is None, f"{problem}\n\n{listing(prog)}"
```

2. Run it with `make probe TB=...` or `make hunt`. When it fails you should see a message naming the instruction number, its address, the disassembled instruction and the state that disagreed:

   ```
   instruction 19, at address 19 (0x66 = SUB   r1, r2), left the machine in a ...
   ```

   The message names the first instruction at which the core and the model disagree, not the last, so it points you at where things started to go wrong. Those four lines of code check every register, the PC and both flags after every instruction, and they're shorter than the directed tests around them.

3. What can this trace not see? Write the answer down next to the test.

   Answer: A2, in Answers at the end.

Check: you have a trace test, and you know what it cannot see.

Part 5: The bugs one test catches

Which bugs does only a test written on purpose catch?

1. Whichever bug you were dealt, add these two tests to `test_lab04.py`:

   - `LOAD` with the pointer in a different register from the destination, checking the pointer afterwards to prove it was not overwritten;
   - one compare followed by two branches, `BZ x; BNZ y`, off a single comparison.

2. Run them against two named bugs:

   ```
   make check BUG=B07
   make check BUG=B09
   ```

   You should see `caught` on both.

3. Compare your results with the reference ten-test suite, which gives this scoreboard (all five counts are exact):

Bug	Tests that fail, of 10
`B04_alu_writeback_reg_field`	8
`B05_branch_offset_zero_extended`	2
`B06_shift_amount_one_bit`	8
`B07_load_address_from_ra`	1
`B09_branch_clobbers_flags`	1

   Two of the five bugs are caught by exactly one test each, and in both cases that test had to be written on purpose after everything else had already passed.

4. The shipped `t03` tests `LOAD` as `LOAD r1, r1`. Why does it pass on `B07`, which reads the load address from the wrong register field?

   Answer: A3, in Answers at the end.

Check: `make check BUG=B07` and `make check BUG=B09` both say `caught`.

Part 6: What random stimulus reaches

What can random stimulus reach, and what can it never reach?

`labs/common/test_reference.py` catches all thirteen bugs with eleven tests. One of those tests, `R10`, runs eight random programs and compares the registers with the golden model after each one.

1. See which windows its generator enters:

   ```
   cd ../common
   make gap
   ```

   It takes about a second, because it doesn't run the simulator. You should see the table from Directed and random tests, ending `8 programs, 160 instructions written, 2 of 9 windows never entered`, and a second table over a thousand seeds in which the same two windows, `B05` and `B07`, are still at zero.

2. Read the docstring of `random_program` in `test_reference.py`. Why does it never emit a `LOAD`, a `STORE` or a backward branch?

   Answer: A4, in Answers at the end.

3. `make gap` shows the random programs entering `B03`'s window. Before you run anything, predict whether `R10` catches `B03`. Then run the reference suite against it:

   ```
   make check TB=test_reference BUG=B03
   cd ../lab-04-your-first-bug-hunt
   ```

   It takes about 35 seconds. You should see:

   ```
        ok  the golden core                   11 tests, all passed           20.8s
    caught  B03_zero_flag_low_nibble          11 tests, 1 FAILED             16.7s
          R07_zero_and_negative_flags
              AssertionError: after r1 = 16 (0b00010000) the Z flag is 1.
   ```

   One test catches it, `R07_zero_and_negative_flags`, rather than `R10`. Why doesn't `R10` catch it?

   Answer: A5, in Answers at the end.

Check: you can say why `R10` misses `B03` and can never reach `B07`.

Part 7: Fix it and get the third row

Can you fix the line and prove the fix with a test?

1. Once you have a test that fails on your core, find and edit the line in `core/simproc_system.sv`.

2. Run the grade:

   ```
   make verdict
   ```

   You should see three green rows. For example, on `B07`:

   ```
        ok  1. the golden core          must PASS      10 tests, all passed        9.4s
    caught  2. the core you were dealt  must FAIL      10 tests, 1 FAILED          9.3s
   ```

```
        t07_load_through_a_separate_pointer
            LOAD r1, r2 with r2 = 0x2A gave r1 = 0x06, expected 0x77.
      ok  3. core/ as it stands now   must PASS      10 tests, all passed        9.3s
-----------------------------------------------------------------------
 VERDICT: PASS
   The bug was B07_load_address_from_ra.
   It hides behind: LOAD rX, rX — loading through a pointer into itself
   It teaches: aliasing hides bugs — make every operand distinct on purpose

   Row 2 is the part worth keeping. Commit that test.
```

If you fix `core/` before you have a test that catches the bug, row 3 goes green, row 2 stays red, and the verdict still says FAIL:

```
VERDICT: FAIL -- row 2. Your testbench cannot tell the dealt core
  from a correct one, so it is not testing whatever is wrong with it.
  This is the row that matters and it is the one still open.
```

The lab won't accept a fix that nobody can prove, and a code review should treat a patch with no test the same way.

Check: `make verdict` says `VERDICT: PASS`.

Part 8: Do it again (optional)

Can you find the next one faster?

1. Deal another core with `make new`. There are five in the pool, and you'll find each one quicker than the last. To practise a particular kind of bug rather than be surprised, pick one:

   ```
   make new BUGPICK=B05
   ```

2. Hunt it with `make hunt`, `make probe` and `make verdict`, as before.

Your first hunt mostly teaches you the harness. By the third you're learning the question list, which you'll carry over to every design you verify after this one.

Expected results

Check	Expected	
`make sim`, shipped file	`TESTS=4 PASS=4`, `200060.00` ns	exact
`make verdict`, when done	three green rows, `VERDICT: PASS`	
Tests failing on `B07`, ten-test suite	1	exact
Tests failing on `B09`, ten-test suite	1	exact
Tests failing on `B04` / `B06` / `B05`	8 / 8 / 2	exact
`make hunt` wall clock	3–10 s depending on suite size	ballpark
`make probe TB=…`, testbench that calls `print()`	the `print()` appears in the terminal	

As in earlier labs, the `sorry:` notes, and `make: *** [...] Error 1` under a FAIL verdict, are normal. Output that is not an error explains them.

Finishing

You're done when `make verdict` on the core you were dealt shows three green rows and `VERDICT: PASS`, with row 2 caught by a test you wrote.

Submit: nothing for this lab. Keep `test_lab04.py`; you'll bring it into Lab 05 and Lab 06.

Next on the Course path: read Handshakes and timing, then do Lab 05: Every answer is right.

Answers

A1. Part 2: Start from what you have

`t02` only ever shifts by 1, so the two-bit amount field is only ever tried at one value (question 1). `t03` loads with `LOAD r1, r1`, so the pointer and the destination are the same register (question 2). Neither of them branches; the only backward branch in the file is the loop in `t04_a_loop`.

A2. Part 4: Let the model find the first disagreement

`golden.py` models architectural state only, and has no idea that `done` or `halt` exist, so a core that gets every answer right but its handshake wrong passes the trace. That kind of bug is the subject of Lab 05.

A3. Part 5: The bugs one test catches

In `LOAD r1, r1` both fields name the same register, so reading the address from the wrong field still gives the right address. `LOAD r1, r1` is a real idiom, used when you're walking a list and no longer need the address, and nothing about it looks like an incomplete test. A golden-model comparison over 27 instructions missed `B07` too. The habit that catches it is to make every operand distinct on purpose, and to check the register that shouldn't have changed as well as the one that should.

A4. Part 6: What random stimulus reaches

A random register value used as a pointer is a random address, and most addresses are outside the 64-byte RAM: addresses 64 to 255 are three quarters of the 256 values an 8-bit register can hold. (The docstring says half.) A random backward branch is an infinite loop. Loads, stores and backward branches are all worth testing, but with stimulus you designed rather than stimulus you rolled.

A5. Part 6: What random stimulus reaches

`R10` compares only the registers, so a wrong `Z` flag gets past it even in a program that reaches the window. `B07` is similar: exactly one test (`R06_load_store_distinct_registers`) catches it, and that isn't the random one either, because the random generator never emits a `LOAD`. Random stimulus with no coverage model only samples the input space, so it can't replace thinking about which inputs matter.