Lab 06: Capstone

In this lab you bring your tests from Labs 04 and 05 together into one suite and use it to find the three tier-5 bugs, which are the hardest in the catalogue to think of. You finish with a suite that catches all thirteen bugs and a written bug report for one of the three. None of the three is subtle in the RTL. `B10` and `B12` are hard to catch because each one needs a test input that ordinary tests don't happen to produce, and `B11` is easy to catch on this design, for a reason Part 3 explains.

This lab takes about 3 hours 30 minutes.

Package: `labs/lab-06-capstone/`

Before you start

- Finish Lab 05: you have protocol tests as well as architectural ones.
- Read Writing the bug report, and reread Directed and random tests.
- Have your Lab 04 and Lab 05 test files at hand to bring forward.

The three bugs in this lab's pool, and why each survives:

	Survives because
`B10_ori_adds_instead_of_ors`	`0 | imm` equals `0 + imm`, so the first `ORI` after reset is always right, and OR and ADD agree exactly when the operands share no set bits, which is every constant anyone builds on purpose
`B11_ori_writes_r1_reads_r0`	The value it produces is correct and only its destination is wrong, and right after a reset the two registers it could have written hold the same thing
`B12_uart_rx_bit_period`	It has no effect at all at the default parameter value, and the link test that everybody writes first passes even over the broken link

Read the coverage list below as a set of questions to ask about your own suite. Each one has an answer you can measure:

	Ask
Instructions	Which of the sixteen opcodes has my suite never executed, including the three unimplemented ones?
Operands	For each instruction, have I run it with `rA == rB` and with `rA != rB`?
Fields	Have I used every shift amount (4), every branch displacement (16), and every `ORI` immediate that overlaps bits already set?
Registers	Has each of the four been a source, a destination, a pointer, and a bystander that must not move?
Flags	Do I assert the flags after every flag-setting instruction, and assert that they don't move after the ones that leave them alone?

	Ask
Memory	Have I used address 0, address 63, and an address above 63?
Protocol	Do I count `done`, count `halt`, and assert cycle counts?
Parameters	Have I swept `clk_per_bit`, rather than only sampling it at its default?
State	Does anything in my suite run long enough to leave the window where reset has made everything equal?

The last two rows are where `B12` and `B11` hide, and the Fields row is where `B10` hides.

```
mod dd104     # in a new terminal, load .designinit first (Lab 01, Part 1)
cd labs/lab-06-capstone
```

Part 1: Get a baseline

How many of the thirteen does the suite you already have catch?

1. Copy your Lab 04 and Lab 05 tests into `test_capstone.py`. `make check-all` scores whatever that module contains, so copy them as they are; there is no credit for retyping work.
2. Score it against all thirteen:

   ```
   make check-all
   ```

   It takes about four minutes, because it runs fourteen full builds and simulations. You should see a scoreboard and, for each bug you missed, a line saying what it hides behind:

   ```
     caught  tier 1  B01_sub_operands_swapped           t01_arithmetic          20.9s
     ...
     MISSED  tier 5  B12_uart_rx_bit_period
   ------------------------------------------------------------------------
    10 of 13 caught.
    Still hiding from this testbench:
      B12_uart_rx_bit_period
         hides behind: the default baud rate, where it is invisible
   ```

   Your count and list will depend on what you brought in.

This is the only place in the course where a failure comes with a hint, so use the list of missed bugs as your work queue for the rest of the lab.

Check: you have a baseline count.

Part 2: B10, the wrong operator

Why does a good testbench miss an `ORI` that adds instead of ORs?

1. Before you run, predict whether the six-test example testbench catches `B10`. Then, in `labs/common`:

```
cd ../common
make check TB=test_example BUG=B10
cd ../lab-06-capstone
```

You should see `MISSED  B10_ori_adds_instead_of_ors` and the hint `ORing in bits that were not already set — ADD agrees`. The example testbench has six tests and a 29-instruction comparison against the model, and none of those instructions ORs in a bit that was already set.

2. The hint says when `ADD` agrees with `OR`. Before writing a test, work out why that makes this bug hard to catch: try `5 | 3` against `5 + 3`, then think about how a program on this machine builds a constant, and how often it ORs a bit into a register that already has it.

   Answer: A1, in Answers at the end.

3. Write a test that ORs a bit into a register that already has it:

```
await sp.run([ORI(5), ORI(3), ORI(24)])
assert sp.reg(0) == 31
```

4. Run `make check BUG=B10`. You should see `caught`, because the register holds `31` on a correct core and `32` on `B10`.

In the eleven-test reference suite, 2 of the 11 tests catch `B10`.

Check: `make check BUG=B10` says `caught`.

Part 3: B11, the wrong destination register

Why can a value be right and still be wrong?

1. See how the reference suite does against it:

```
make check TB=test_reference BUG=B11
```

   You should see:

```
  caught  B11_ori_writes_r1_reads_r0         11 tests, 7 FAILED              14.2s
        R01_alu_asymmetric_operands
        R02_ori_overlapping_immediates
        R03_shift_amount_sweep
        R05_flags_survive_a_branch
```

   Seven failing tests out of eleven is an exact count, and it's the highest of the three bugs in this pool by some distance.

2. Compare what a five-instruction constant builder leaves in the registers on the two cores (both rows are exact):

	r0	r1
Golden	20	0
B11	0	4

   On `B11`, `ORI` reads `r0` but writes `r1`, so the shift in the middle of the builder shifts `r0`, a register that never received anything.

3. Make sure your suite catches it with `make check BUG=B11`.
4. Why is `B11` easy to catch in this design, and why would a short test that only checks the value miss it?

   Answer: A2, in Answers at the end.

`B11` is also upstream's own to-do, applied to only one of the two places it touches. The SimProc source shipped with a comment suggesting exactly this change, and we removed that comment from DD104′s fork on purpose, because in a course whose premise is "there are hidden bugs in this file" it reads as a planted clue pointing at code that is in fact correct. `ORI-DECISION.md` has the full reasoning, including why the shipped behaviour is the specified one. Neither a source comment nor a to-do is a specification, and `B11` shows what happens when a to-do is applied halfway.

Check: your suite catches `B11`.

Part 4: B12, the baud rate

Does the UART work at every speed, or only at the default?

The UART's bit period, `clk_per_bit`, is `uio_in × 4`. Every lab so far has run at the default `uio_in = 8`, which gives `clk_per_bit = 32`.

1. Write a test that sweeps the bit period and, at each speed, sends a `PING` and also writes and reads back a byte that is not a fixed pattern:

```
for uio in (4, 8, 16, 32, 64):
    await sp.set_baud(uio)
    reply = await sp.uart_ping()
    await sp.uart_write(0x3F, 0x5A)
    got = await sp.uart_read(0x3F)
```

2. `B12` adds one clock to every bit the UART receives. Before you run, predict whether it starts failing at short bit periods or at long ones. Then run the test with `make sim`, and after that with `make check BUG=B12`. On the golden core every speed works, and on `B12` you should see this pattern (every row is exact):

`clk_per_bit`	`PING`	A data byte (`0x5A`, written then read back)
8	`TimeoutError`, no reply at all	
16	`0xAA`	`ValueError`, because the reply is X bits
24	`0xAA`	`0x5A`
32 (the labs' default)	`0xAA`	`0x5A`
64, 128, 256	`0xAA`	`0x5A`

3. At `clk_per_bit = 16`, `PING` gets a perfect `0xAA` reply over a link that cannot carry a data byte. Why?

   Answer: A3, in Answers at the end.

4. Why does `B12` fail at short bit periods first, and look correct at 24 and above?

   Answer: A4, in Answers at the end.

Check: `make check BUG=B12` says `caught`.

Part 5: Hunt one properly

Can you find which of the three your dealt core has?

1. Deal yourself a core with one of the three bugs and hunt for it:

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

   The grade is the same three rows as in Lab 04, and row 2 is still the deliverable. The Lab 04 tools are here too: `make probe`, which shows every `print()` and `dut._log` line as it happens, and `make hunt VERBOSE=1` and `make verdict VERBOSE=1`.

Knowing that three bugs exist doesn't tell you which one you have. The pool description in `test_capstone.py` describes all three without naming the line any of them touches, which is about as much as a bug report tells you in real life.

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

Part 6: The suite

Does your suite catch all thirteen, and which tests do the catching?

1. Run `make check-all` until it says:

```
----------------------------------------------------------------------
 13 of 13 caught.
 Every bug in the catalogue, every one of them caught by a check
 that fired. Nothing in this course hides from this testbench any
 more.
======================================================================
```

2. Look at which test catches each bug. In the reference capstone suite, the catches are:

```
  caught  tier 1  B01_sub_operands_swapped          t01_arithmetic
  caught  tier 1  B02_nand_is_nor                   t01_arithmetic
  caught  tier 1  B03_zero_flag_low_nibble          t08_against_the_golden
  caught  tier 2  B04_alu_writeback_reg_field       t01_arithmetic
  caught  tier 2  B06_shift_amount_one_bit          t01_arithmetic
  caught  tier 2  B07_load_address_from_ra          t07_load_through_a_sep
  caught  tier 3  B05_branch_offset_zero_extended   t04_a_loop
  caught  tier 3  B09_branch_clobbers_flags         t10_a_compare_survives
  caught  tier 4  B08_store_drops_done              p01_done_pulses_once_p
  caught  tier 4  B13_store_falls_to_idle           t04_a_loop
  caught  tier 5  B10_ori_adds_instead_of_ors       t08_against_the_golden
  caught  tier 5  B11_ori_writes_r1_reads_r0        t01_arithmetic
  caught  tier 5  B12_uart_rx_bit_period            c03_baud_sweep
```

   The reference suite has sixteen tests: ten from Lab 04, three from Lab 05 and three new ones. Most of the work was already done before this lab began, but the three new tests took the longest.

3. Two general tests, `t01_arithmetic` and `t08_against_the_golden_model`, catch seven of the thirteen between them, and five bugs are caught by exactly one test each. Is that a well-balanced suite, or one with a long tail

of special cases bolted on? Write down your answer and what you would write next. Either reading can be defended.

Check: `13 of 13 caught`.

Part 7: Practise the report on B01

Can you take a bug you already know through the whole report loop?

1. Run the worked reproducer in `labs/common`:

   ```
   cd ../common
   make check TB=test_b01_repro BUG=B01
   ```

   You should see `ok` on the golden core and `caught  B01_sub_operands_swapped  3 tests, 1 FAILED`, with the message `SUB r1, r2 with r1 = 200 and r2 = 100 left r1 = 156.`

2. Deal `B01` in the Lab 04 package and grade the reproducer against it. `make new` deletes the `core/` already in `labs/lab-04-your-first-bug-hunt`, so finish anything you still want from your Lab 04 core first.

   ```
   cd ../lab-04-your-first-bug-hunt
   make new BUGPICK=B01
   make verdict TB=test_b01_repro
   ```

   The `make new` banner still lists Lab 04's five-bug pool, but the core it wrote has `B01`. You should see rows 1 and 2 green and row 3 `FAIL`, which means the test exists and the design is still broken.

   ```
        ok  1. the golden core          must PASS      3 tests, all passed        7.8s
    caught  2. the core you were dealt  must FAIL      3 tests, 1 FAILED          4.2s
          t01_sub_with_two_distinct_operands
              AssertionError: SUB r1, r2 with r1 = 200 and r2 = 100 left r1 = 156.
      FAIL  3. core/ as it stands now   must PASS      3 tests, 1 FAILED          4.1s
   ```

3. Fix line 811 of `core/simproc_system.sv`:

   ```
              SUB:        ALUout = B - A;     ->     SUB:        ALUout = A - B;
   ```

4. Grade it again, with the full failure message shown:

   ```
   make verdict TB=test_b01_repro VERBOSE=1
   ```

   You should see `VERDICT: PASS` with row 2 unchanged, and the whole assertion message from Writing the bug report, including `note       (100 - 200) mod 256 = 156`.

5. Go back to the capstone package:

   ```
   cd ../lab-06-capstone
   ```

Check: row 2 stayed `caught` after the fix.

Part 8: Write the report

Can someone act on your bug without redoing your work?

1. Copy the template:

```
cp BUG-REPORT-template.md BUG-REPORT.md
```

2. Fill it in for one of `B10`, `B11` and `B12`, using `BUG-REPORT-EXAMPLE.md` as a model. Five parts:

    1. The smallest program that reproduces it. Cut instructions until it stops failing, then put the last one back.
    2. Expected against actual, as numbers, with the arithmetic shown.
    3. The shape: how many vectors or cycles are wrong, out of how many, and what the count implies before anyone opens the RTL.
    4. Where it isn't: what still works.
    5. The habit the suite gained, in one line, and the test that enforces it, by name.

A red test is only where the report starts; the report itself is what lets someone act on the bug without redoing your week. Part 4 of the report is the one people skip, and it's the one designers read first, because a sentence like "every `ORI` whose immediate is disjoint from `r0` is correct, all 393,216 ALU vectors are correct, and both branches and all memory accesses are unaffected" tells them where not to look, and that turns a report into a map. The report is finished when a test in the suite fails before the fix and passes after it, which is what `make verdict` checks.

Check: `BUG-REPORT.md` exists and has all five parts.

Part 9: Close the course

What claim does your suite now support?

1. Write down, in one sentence, the claim your suite now supports. Start it with "This processor…" and keep going until the sentence is true.

2. Put it next to the claim the chip shipped with:

```
assert dut.uo_out.value == 3
```

    which supports only "this processor is not running". How long is your sentence, and why?

    Answer: A5, in Answers at the end.

Check: you have your sentence written next to the shipped claim.

Expected results

Check	Expected	
`make check-all`	`13 of 13 caught`	exact
`make check-all` wall clock	about 4 min 35 s	ballpark
Reference suite on golden	16 of 16, `2118170.02` ns	exact
Tests failing, reference 11-test suite: `B10` / `B11` / `B12`	2 / 7 / 1	exact
`B10`: `ORI 5, ORI 3, ORI 24`	golden `31`, broken `32`	exact
`B11`: five-instruction constant builder	golden `r0 = 20`; broken `r0 = 0, r1 = 4`	exact
`B12` at `clk_per_bit = 16`	`PING` returns `0xAA`; a data byte does not survive	exact

Check	Expected
`B12` at `clk_per_bit >= 24`	indistinguishable from a correct exact core
`BUG-REPORT.md`	exists, five parts

Finishing

You're done when `make check-all` on your `test_capstone.py` says `13 of 13 caught` and your `BUG-REPORT.md` has all five parts.

Submit: `test_capstone.py` and `BUG-REPORT.md`.

Going further (optional)

Read `labs/common/test_reference.py`, which catches all thirteen in eleven tests, and find the three choices you disagree with.

Next on the Course path: this is the last lab. The digital track continues with DD201 (High-Level Synthesis) and DD202 (Computer Architecture).

Answers

A1. Part 2: B10, the wrong operator

`5 | 3 = 7` but `5 + 3 = 8`; then `7 | 24 = 31` but `8 + 24 = 32`. OR and ADD give the same answer on every pair of operands that share no set bits. Building a constant on this machine only ever ORs in bits that aren't set yet, which is exactly the case where the two agree, and that is what `set_reg` does:

```
SUB r0, r0     # clear
ORI 0x16       # the top five bits
SHL r0, 3      # shove them up
ORI 0x07       # the bottom three -- disjoint from what is already there
```

Every one of those `ORI`s gives the right answer on a core whose `ORI` adds. The first `ORI` after a reset is safe too, because it computes `0 | imm`, which equals `0 + imm`, so even an overlapping immediate gives the right answer the first time.

A2. Part 3: B11, the wrong destination register

It is easy to catch here because `ORI` is the only way to get a constant into the machine, so every `set_reg` in every test goes through it (not every design is that obliging). A short test that only checks the value misses it because the value is right and only the destination is wrong. Right after a reset `r0` and `r1` hold the same thing, so a machine that reads one and writes the other produces a number that is correct in every respect except where it ends up. Reset is a state in which several things are temporarily indistinguishable, and a test that stays inside that window can't tell them apart. Your programs are short because short programs are readable, and that is what keeps them inside the window.

A3. Part 4: B12, the baud rate

It's because of the particular bytes involved. The `PING` command, `0xF0`, already has its top bit set, and the reply `0xAA` is a fixed pattern, so a receiver that mis-samples bit 7 as the stop bit still gets both of them right. A link test that passes over a broken link is worse than having no link test, because you believe it.

A4. Part 4: B12, the baud rate

The extra clock per received bit makes the sample point drift further off centre with each bit of a frame, and a long bit period has room to absorb that drift. At the default speed this core behaves like a correct one, and the rest of your suite scores the same on it as on the golden core, so the pattern of which speeds work is your diagnosis. Only 1 of the 11 tests in the reference suite catches `B12`, and it's the only one that sweeps the bit period. When a design has a parameter, test it at more than one value.

A5. Part 9: Close the course

It should be uncomfortably long. It will have clauses about operands that are not equal to each other, about every value of a two-bit field and both halves of a four-bit signed one, about flags that move and flags that must not, about a pin that pulses once per instruction and a pin that never rises, about an exact number of clock cycles, and about a serial link at five different speeds. Each clause is there because a bug got past you first. A test suite is a way to make a long, specific, checkable claim, and you write one because nobody, including you next year, can hold a claim that long in their head.

The part of this course you will use everywhere is the question to ask of any green test, including your own: what would this have caught?