Lab 02: The only test this chip has

You'll run SimProc's one shipped test against the good core and all thirteen broken ones and watch every one pass. Then you'll drive the processor from Python, try the harness's example testbench, compare Icarus with Verilator, and write a test that passes on the good core and fails on the one whose `SUB` has its operands swapped.

This lab takes about 1 hour 45 minutes.

Package: `labs/lab-02-the-only-test-this-chip-has/`. It includes `../common/lab.mk`, which lives outside the package, so you need the whole repository checked out and not just this directory. Every lab from here on works the same way.

Before you start

- Finish Lab 01 (`make check` says `VERDICT: PASS`) and cocotb on a counter.
- Read Two simulators and SimProc in one page.
- Keep The SimProc instruction set and The SimProc harness open in other tabs.

In this lab, "the golden core" is the correct SimProc, and the thirteen broken ones are named `B01` to `B13`. Work in a terminal on the noVNC desktop, where `mod dd104` takes you to the course folder, as in Lab 01, Part 1:

```
mod dd104     # in a new terminal, load .designinit first (Lab 01, Part 1)
cd labs/lab-02-the-only-test-this-chip-has
```

Part 1: The shipped test

Which of fourteen processors does the shipped test catch, and what does it actually assert?

1. Run the shipped test against fourteen processors:

   ```
   make shuttle
   ```

   This takes about half a minute, because it builds and runs fourteen cores.

2. Before your first result you will see a wall of lines like this:

   ```
   rtl/simproc_system.sv:552: sorry: constant selects in always_* processes are not fully
   supported (the process will be sensitive to all bits in 'instr_reg_out[7:0]').
   ```

   Icarus prints 28 of these every time it compiles the design, and it will keep doing so for the rest of the course. They're notes rather than errors, and the exit code is 0; a real Icarus failure starts a line with `error:` and stops. Output that is not an error explains why the simulation is still correct, and lists the other harmless messages like this one.

3. You should see:

   ```
   ========================================================================
    THE TEST THIS CHIP SHIPPED WITH
   ========================================================================
    test/test.py, from the SimProc repository, untouched. It sets
    ui_in = 1, waits 100 clock cycles, and asserts uo_out == 3.
    Decoded: uart_tx = 1 (idle), halt = 1 (IDLE), done = 0.
    It asserts that the processor is NOT RUNNING.
   ------------------------------------------------------------------------
   ```

```
    PASS  the golden core                              5.7s
    PASS  B01_sub_operands_swapped                     2.1s
    PASS  B02_nand_is_nor                              2.1s
    PASS  B03_zero_flag_low_nibble                     2.0s
    PASS  B04_alu_writeback_reg_field                  2.0s
    PASS  B06_shift_amount_one_bit                     2.0s
    PASS  B07_load_address_from_ra                     2.0s
    PASS  B05_branch_offset_zero_extended              2.0s
    PASS  B09_branch_clobbers_flags                    2.0s
    PASS  B08_store_drops_done                         2.1s
    PASS  B13_store_falls_to_idle                      2.1s
    PASS  B10_ori_adds_instead_of_ors                  2.0s
    PASS  B11_ori_writes_r1_reads_r0                   2.0s
    PASS  B12_uart_rx_bit_period                       2.0s
-------------------------------------------------------------------------
 13 of 13 deliberately broken processors pass the
 only test this chip has.
=========================================================================
```

`13 of 13` and the names are exact, but the seconds are wall-clock time and will vary. The cores are listed in tier order rather than by number. Each of the thirteen broken cores is a one-line edit to `simproc_system.sv`.

4. Open the test itself, `../common/test_shuttle.py`. It's `test/test.py` from the SimProc repository, unchanged apart from a comment block that explains the pin decode.
5. Write down, in one sentence, the strongest true claim it supports. (This is the first check-yourself question on What a passing test proves.)

   Answer: A1, in Answers at the end.
6. Print the bug catalogue:

   ```
   make bugs
   ```

   You should see thirteen names in five tiers (a tier is how hard the bug is to think of), each with what it hides behind and what it teaches. It doesn't tell you what any of them changes in the design.

Check: `13 of 13` on your screen, and a one-sentence claim written down.

Part 2: Make the chip do something

What does it take to make the processor run one instruction and check the result?

1. Run the lab's testbench, which ships working:

   ```
   make sim
   ```

   You should see:

   ```
   ** test_lab02.t01_the_test_this_chip_shipped_with      PASS        1150.00 **
   ** test_lab02.t02_what_that_number_meant               PASS        1160.00 **
   ** test_lab02.t03_one_instruction_that_actually_runs   PASS       33300.00 **
   ** TESTS=3 PASS=3 FAIL=0 SKIP=0                                   35610.00 **
   ```

and, in the log, `uo_out = 3, and the core is IDLE`, then `halt=1 done=0 state=IDLE — this is a processor doing nothing, and t01 asserted exactly that`, then `one ORI, 5 clock cycles, r0 = 5`. The four simulated times and the `5 clock cycles` should match yours exactly.

2. Read `t03` in `test_lab02.py`. It takes only three lines of Python to make the processor run an instruction, and a fourth to check the result. `sp.run()` loads the program, adds a parking loop after it (a `JMP -1`, because the processor has no halt instruction), runs it, and returns once it reaches that loop:

```
sp = SimProc(dut)
await sp.start()                # clock, reset, RAM blanked
await sp.run([ORI(5)])          # load a program, run it to its parking loop
assert sp.reg(0) == 5
```

3. Compare `t01` and `t02`. They assert the same thing about the same idle chip. Is `t02` a stronger test?

   Answer: A2, in Answers at the end.

The docstring at the top of `test_lab02.py` is a two-page summary of the harness API.

Check: `TESTS=3 PASS=3` and `35610.00` ns.

Part 3: The example testbench

In how many different ways can a testbench catch one bug?

The harness directory, `labs/common`, ships six worked tests in `test_example.py`, and the intended way to start a testbench of your own is to copy that file.

1. Run the example against the golden core and `B01`:

```
cd ../common
make check
```

   `make check` uses `BUG=B01` unless you name another core. You should see `ok` on the golden core and `caught` on `B01_sub_operands_swapped`, with four tests failing in four different ways (long messages are cut to fit the table, and yours will be cut the same way):

```
      ok  the golden core                   6 tests, all passed             9.6s
  caught  B01_sub_operands_swapped          6 tests, 4 FAILED               7.3s
        t03_arithmetic_on_different_values
            AssertionError: r1 is 199, expected 247.
        t04_flags_are_state
            AssertionError: after r1 = 255 the flags are N=0 Z=0, expected N=1 Z=0
        t05_a_loop_and_its_cycle_count
            TimeoutError: the program did not reach its parking loop at address 16 within
20000 clock
        t06_against_the_golden_model
            AssertionError: instruction 19, at address 19 (0x66 = SUB   r1, r2), left the
machine in a
-------------------------------------------------------------------------
 VERDICT: PASS
```

2. Match each failure to drive, observe or decide, using the table in What a testbench is. Why does `t05` time out instead of reporting a wrong value?

Answer: A3, in Answers at the end.

3. Make your own copy and run it both ways:

```
cp test_example.py test_mine.py
make sim   TB=test_mine
make check TB=test_mine BUG=B01
```

`TB` takes the name of a Python module rather than a file, so write `TB=test_mine`, never `TB=test_mine.py`. You should see six tests pass under `make sim`, and `make check` should give the same `caught` verdict as in step 1, this time under the name `test_mine`.

Check: `make check` in `labs/common` ends on `VERDICT: PASS`.

Part 4: The golden model on its own

Can you predict a program's exact cycle count from the instruction costs?

1. Still in `labs/common`, run the golden model with no simulator. It runs the demo program built into `golden.py`, a loop that sums 1 to 5 and stores the result at `0x20`:

```
make model
```

You should see:

```
16 instructions in memory, 28 executed, 109 clock cycles
registers: r0=32 r1=15 r2=1 r3=32
mem[0x20] = 15   (all 28 instructions executed)
```

2. Derive the 109 by hand, using the program listing that `make model` prints above those three lines and the cycle-count table in SimProc in one page: seven setup instructions, a three-instruction loop run five times, and six to finish. `t05_a_loop_and_its_cycle_count` in `test_example.py` asserts the same number on the simulated hardware.

Answer: A4, in Answers at the end.

Check: your sum is 109.

Part 5: Two simulators

Do Icarus and Verilator agree about this design?

You stay in `labs/common` for this part.

1. Lint the design:

```
verilator --lint-only --timescale 1ns/1ps --top-module tb \
  rtl/tt_um_ieeeuoftasic_simproc.sv rtl/simproc_system.sv rtl/tb.v
```

You should see twelve warnings in well under a second, including `rtl/simproc_system.sv:848:16: Bit extraction of array[63:0] requires 6 bit index, not 8 bits.`, ending with `%Error: Exiting due to 12 warning(s)`. That last line is expected, because Verilator treats warnings as fatal, and here the warnings are what you ran it to find. What does line 848 tell you about addresses 64 to 255?

Answer: A5, in Answers at the end.

2. Read past the end of the RAM on each simulator. Before you run, write down what you expect each one to return for address 100.

```
make clean && make sim TB=test_two_simulators SIM=icarus
make clean && make sim TB=test_two_simulators SIM=verilator
```

You should see `READ 100  (off the end)  -> XXXXXXXX` from Icarus and `READ 100  (off the end)  -> 01011010` from Verilator, with `READ  36` giving `01011010` on both, `TESTS=2 PASS=2` and `60520.00` ns on both. Explain the Verilator value using `100 & 63`.

Answer: A6, in Answers at the end.

3. Run the example testbench on both simulators and check that simulated time is identical, test by test:

```
make clean && make sim TB=test_example SIM=icarus  >/dev/null 2>&1 && cp build/results.xml /tmp/ic.xml
make clean && make sim TB=test_example SIM=verilator >/dev/null 2>&1 && cp build/results.xml /tmp/vl.xml
diff <(grep -o 'sim_time_ns="[^"]*"' /tmp/ic.xml) \
     <(grep -o 'sim_time_ns="[^"]*"' /tmp/vl.xml) && echo "identical -- every test, both simulators"
```

You should see no diff output, then `identical -- every test, both simulators`.

4. Go back to the lab directory:

```
cd ../lab-02-the-only-test-this-chip-has
```

Check: the two simulators disagree about address 100 and agree on every simulated time.

Part 6: A waveform

What does one `ORI` look like inside the core?

1. Dump a waveform of the lab's three tests:

```
make waves
```

It takes about two seconds and writes `waves.vcd`, which is 337,999 bytes. The size is exact, because the file is a dump of a deterministic simulation.

2. Open it from your terminal on the noVNC desktop:

```
surfer waves.vcd        # or: gtkwave waves.vcd
```

3. Add `clk` and `uo_out`. Then open `tb` → `tt_um_ieeeuoftasic_simproc` → `U1` → `CPU1` and add `curr_state`. The trace holds all three tests: `t03` runs from 2,310 ns to 35,610 ns, and its `ORI` comes after the UART commands that start the core.

You should see `curr_state` walk from `IDLE` through `CYCLE_1` … `CYCLE_5` and back, with `done` (bit 2 of `uo_out`) pulsing as the instruction retires. It's worth learning to move around the viewer now, while the trace is only 35,610 ns long, because later traces run to millions of nanoseconds.

Check: you can point at the five cycles of the `ORI`.

Part 7: Write a test that catches `B01`

Can you write a test that passes on the golden core and fails on the core whose `SUB` computes `B - A`?

1. Before running anything, decide: would `t03`, which runs `ORI 5` and checks `r0 == 5`, catch a processor whose subtractor has its operands backwards?
2. Run the two-sided check:

   ```
   make check
   ```

   You should see:

   ```
         ok  the golden core                    3 tests, all passed             2.2s
    MISSED  B01_sub_operands_swapped            3 tests, all passed             2.2s
   ------------------------------------------------------------------------------
    VERDICT: FAIL — your testbench passes B01_sub_operands_swapped too.
      A test that passes on a broken core is not testing that core.
      That bug hides behind: SUB rX, rX — the idiom for 'clear a register'
      Write a test whose stimulus does not do that.
   ==============================================================================
   ```

   followed by `make: *** [../common/lab.mk:66: check] Error 1`, which is make reporting that the verdict exited non-zero. Your testbench misses `B01` because `t03` never subtracts.
3. Read the hint. `SUB rX, rX` is how you clear a register, so it appears in nearly every program on this machine. For how many of the 65,536 operand pairs do `A - B` and `B - A` give the same byte?

   Answer: A7, in Answers at the end.
4. Do the TODO in `t03`'s docstring: add a new test function that subtracts two registers holding different values and checks the result. `set_reg(1, 200) + set_reg(2, 60) + [SUB(1, 2)]` leaves `r1` at $200 - 60 = 140$ on a correct core. Work out what a swapped subtractor gives before you run it. `test_lab02.py` already imports `SUB` and `set_reg`; a test has this shape:

   ```
   @cocotb.test()
   async def t04_subtract_two_different_numbers(dut):
       sp = SimProc(dut)
       await sp.start()
       await sp.run(...)          # your program
       assert ...                 # with a message that names both numbers
   ```

   Answer: A8, in Answers at the end.
5. Assert the flags too: `sp.flags()` returns `(N, Z)`. Predict `N` and `Z` after `200 - 60` before you write the assertion.

   Answer: A9, in Answers at the end.
6. Run `make check` again. You should see:

   ```
         ok  the golden core                    4 tests, all passed          3.6s
     caught  B01_sub_operands_swapped           4 tests, 1 FAILED            3.2s
           t04_subtract_two_different_numbers
               AssertionError: r1 is 116, expected 140 (200 - 60).
   ```

```
----------------------------------------------------------------------
 VERDICT: PASS
```

The test name and the message will be whatever you wrote, but `140` and `116` are exact. If you asserted the wrong `N`, the golden-core row reads `FAIL` instead and the verdict says `VERDICT: FAIL — your testbench does not pass the GOLDEN core.` That's the right order for things to go wrong in, because a testbench that fails on the golden core is telling you about a mistake in the testbench itself.

Check: `make check` says `ok` on the golden core and `caught` on `B01`, with your new test asserting both `r1` and the flags.

Part 8: The scoreboard

How many of the thirteen does your testbench catch so far?

1. Run your testbench against all thirteen broken cores:

   ```
   make check-all
   ```

   It takes about 22 seconds and prints, for each core you missed, one line about what it hides behind. Keep the output, because it's your scoreboard for the rest of the course: by Lab 06 the goal is `13 of 13 caught`.

2. On the shipped file the count is `1 of 13 caught`, and the one it catches is:

   ```
     caught  tier 5  B11_ori_writes_r1_reads_r0         t03_one_instruction_th   2.2s
   ```

   With your Part 7 test added, `B01` is caught too; with the reference solution's `t04`, the count is `5 of 13 caught`. Your count depends on what your test checks. Expect most of the thirteen to stay missed for now, because four short tests aren't meant to catch thirteen bugs.

3. `t03` runs a single `ORI 5` and checks `r0`. Why is that enough to catch `B11`, a tier-5 bug?

   Answer: A10, in Answers at the end.

Check: you have the `check-all` output saved.

Expected results

Check	Expected	
`make shuttle` final count	`13 of 13`	exact
`make shuttle` wall clock	about 30 s	ballpark
`make sim`, shipped file	`35610.00` ns total	exact
The `ORI` cycle count in the log	`5 clock cycles`	exact
`make waves` output file	`waves.vcd`, 337,999 bytes	exact
Icarus `sorry:` notes per compile	28	exact
`make check`, shipped file	`MISSED`, `VERDICT: FAIL`	exact
`make check`, when you are done	`caught`, `VERDICT: PASS`	
`make check-all`, shipped file	`1 of 13 caught`	exact
`make check-all`, reference solution	`5 of 13 caught`	exact

Finishing

You're done when `make check` says `ok` on the golden core, `caught` on `B01` and `VERDICT: PASS`, with your new test asserting both `r1` and the flags.

Submit: nothing for this lab. Keep the `check-all` output you saved in Part 8.

Going further (optional)

See what Verilator's build costs and what `ccache` saves. In `labs/common`:

```
make clean && make sim SIM=icarus
make clean && make sim SIM=verilator && ccache -s | grep Hits
make clean && make sim SIM=verilator && ccache -s | grep Hits
```

Both simulators give `TESTS=6 PASS=6 FAIL=0 SKIP=0` and `512010.01` ns, with Verilator's `REAL TIME` (the column after the simulated time) about half of Icarus's.

`ccache` keeps the compiled C++ files, and its `Hits` line counts across everything compiled in this container. Each Verilator build compiles six files. If you did Part 5 in this container, its Verilator builds have already filled the cache, so both builds here are fast, and the line reads `Hits: 12 / 18`, then `Hits: 18 / 24`. The cache is lost when the container stops, so to see a cold build, run these commands first thing after a restart. The first Verilator build then takes about 16 s end to end, against about 6 s for Icarus, because Verilator builds a C++ program first, and the line reads `Hits: 0 / 6`; the second takes a few seconds, and the hits go up by six, to `Hits: 6 / 12 (50.00%)`. Two simulators explains the numbers.

Next on the Course path: read Flags as architectural state, then do Lab 03: Three blocks before the processor.

Answers

A1. Part 1: The shipped test

After reset and 100 clock cycles with `ui_in[0]` high, this design drives `uo_out` to `0x03`. That's a real claim, and even a useful one: it would catch a core that came up running by accident, or a UART that sent garbage at power-on. It's just a very small claim, and nothing about the way the test is written tells you how small it is.

A2. Part 2: Make the chip do something

No. Both are exactly as weak as the shipped test, because both check the same idle chip. What `t02` improves is the message it gives when it fails, which names the pin instead of handing you an integer to decode. Asserting on the decoded meaning rather than the bus value costs nothing, so do it.

A3. Part 3: The example testbench

`t05` runs a loop controlled by a subtraction. On `B01` the subtraction goes the wrong way, so the loop never ends and the program never reaches its parking loop. That's why the symptom is a timeout rather than a mismatch, and it's a useful bug report only because the test sets a deadline.

A4. Part 4: The golden model on its own

Setup: `4+5+4+4+4+5+4 = 30`. The loop body costs `4+4+3 = 11` and runs five times, for 55. The finish costs `4+5+4+4+4+3 = 24`. $30 + 55 + 24 = 109$.

A5. Part 5: Two simulators

The RAM, module `dp_ram` in `simproc_system.sv`, has 64 entries and the address bus is 8 bits, so addresses 64–255 are off the end of the array. Verilator found that just by reading the file, before simulating anything.

A6. Part 5: Two simulators

Verilator is 2-state, so it truncates the address to the six bits the array has: `100 & 63 = 36`, and you get the byte at address 36, which is the `0x5A` the test wrote there before reading. Icarus is 4-state, so reading past the end gives `x`, and eight `X` characters come off the pin.

A7. Part 7: Write a test that catches B01

For 512 of them. They agree for every pair with `A == B` (256 pairs, `SUB rX, rX` among them), and for every pair whose difference is 128, because 128 is its own negation in eight bits (256 more). That's only 0.78 % of the input space, but the most natural subtraction test lands inside it.

A8. Part 7: Write a test that catches B01

`r1` ends up at 116. A swapped subtractor computes $60 - 200 = -140$, and -140 in eight bits is 116.

A9. Part 7: Write a test that catches B01

`N = 1`, `Z = 0`. $140 = 0b10001100$, so bit 7 is set. The machine has no opinion about whether that byte means 140 or −116, because `N` is simply bit 7 of the result. See Flags as architectural state.

A10. Part 8: The scoreboard

`ORI` is the only way to get a constant into this machine, so a bug on that path is on the path of every program. The tier is how hard a bug is to think of, not how hard it is to trip over. You'll meet `B11` again in Lab 06.