DD104 — Pre-Silicon Verification

Write cocotb tests that catch all thirteen broken versions of SimProc, an 8-bit processor written by students on the team.

UofT ASIC Team

Contents

1 DD104 — Pre-Silicon Verification 5

I 1. Testbenches

2 Getting started 11

3 What a passing test proves 13

4 What a testbench is 17

5 Your first cocotb test 21

6 The lazy testbench 25

7 Coverage as a question 29

8 Lab 01: The adder you already trust 33

II 2. cocotb

9 Clocks, resets and triggers 41

10 cocotb 2.x and older tutorials 45

11 Driving and checking a design 51

12 Waveforms 55

13 Lab: cocotb on a counter 59

III 3. SimProc

14 Two simulators 67

15 SimProc in one page 71

16 Lab 02: The only test this chip has 77

IV 4. Unit tests

17 Flags as architectural state 87
18 Lab 03: Three blocks before the processor 91

V 5. Bug hunting

19 The golden model 101
20 Where bugs hide 105
21 Directed and random tests 111
22 Lab 04: Your first bug hunt 115

VI 6. Handshakes

23 Handshakes and timing 125
24 Lab 05: Every answer is right 131

VII 7. Capstone

25 Writing the bug report 139
26 Lab 06: Capstone 143

Appendices 151
A The SimProc instruction set 153
B The SimProc harness 159
C Output that is not an error 171

1. DD104 — Pre-Silicon Verification

In this course you find the bugs hidden in a real 8-bit processor, using cocotb, Icarus Verilog and Verilator, before they cost a mask set.

DD104 is part of the UofT ASIC Team education materials, published at https://uoftasic.com/dd104/. The labs run in the workbench: the IIC-OSIC-TOOLS container you start from the `workspace` clone you set up in IC101. You get this course's files with the workbench's `mod` command (`mod add dd104`); Lab 01, Part 1 walks you through it.

Who this course is for

You have finished the digital track up to DD103: you can write SystemVerilog, simulate it, and take it through a flow to layout. None of those steps checked whether the design computes what you meant it to, and that question is what this course is about.

SimProc is an 8-bit processor written by students on this team and wrapped for a real chip (a TinyTapeout shuttle): an ALU, four registers, 64 bytes of memory and a UART, about 1,200 lines of SystemVerilog. It shipped with one test, and that test still passes on a processor whose `SUB` has its operands swapped and computes `B - A`. DD104 ships thirteen broken versions of SimProc, each made by a one-line edit, and all thirteen pass that test. By the end of the course you will have written a test suite that catches all thirteen, and a bug report for one of them.

Track	Digital
Prerequisites	IC101, then DD101, DD102 and DD103, in that order
Tools	cocotb 2.0.1, Icarus Verilog 14.0, Verilator 5.050, Surfer and GTKWave, all in the workbench image `hpretl/iic-osic-tools:2026.08`
Machine	The workbench you already have. This course needs no PDK, no LibreLane and no synthesis. Most runs take under ten seconds; the longest, Lab 06's `make check-all`, takes about four and a half minutes.
Total time	About 19 hours, self-paced. The Course path rows below add up to 18 h 55 min.

Check your cocotb version before you search the web for help: `cocotb-config --version` should print `2.0.1`. That matters because most cocotb tutorials online were written for 1.x, and some of the things they use are errors in 2.x, among them `cocotb.fork` (use `cocotb.start_soon`), `@cocotb.coroutine` (use a plain `async def`) and `dut.sig <= 1` (use `dut.sig.value = 1`). Most of the rest, such as `units=`, `MODULE=` and `.value.integer`, still runs and only prints a deprecation warning. cocotb 2.x and older tutorials has the full list.

Course path

Work down this table in order. A Read row is a set of guide pages, which are for reading only and have no commands in them. A Do row is one lab writeup, and the lab writeups are where every command in the course lives. Each lab's "Before you start" section repeats which pages it expects you to have read. The counter lab in row 4 has no number, but it isn't optional.

#		What	Time
1	Read	Getting started · What a passing test proves · What a testbench is · Your first cocotb test · The lazy testbench · Coverage as a question	1 h 15 min
2	Do	Lab 01: The adder you already trust. Load the course, watch a reasonable testbench pass on a broken adder, then fix the testbench so it catches it.	1 h 30 min
3	Read	Clocks, resets and triggers · cocotb 2.x and older tutorials · Driving and checking a design · Waveforms	1 h
4	Do	Lab: cocotb on a counter. Run the guide pages' cocotb examples on an 8-bit counter, including the 1.x forms that fail in 2.x, then break the counter on purpose and see which tests notice.	1 h 15 min
5	Read	Two simulators · SimProc in one page	45 min
6	Do	Lab 02: The only test this chip has. Run the shipped test against all thirteen broken cores, then write a better one.	1 h 45 min
7	Read	Flags as architectural state	20 min
8	Do	Lab 03: Three blocks before the processor. Unit-test three SimProc modules, including all 393,216 ALU vectors.	2 h
9	Read	The golden model · Where bugs hide · Directed and random tests	1 h
10	Do	Lab 04: Your first bug hunt. Get a mystery core, find the bug, and prove the fix with a test.	2 h 30 min

#		What	Time
11	Read	Handshakes and timing	25 min
12	Do	Lab 05: Every answer is right. Write three checks that catch two cores on which every value is correct, then hunt both.	1 h 15 min
13	Read	Writing the bug report	25 min
14	Do	Lab 06: Capstone. Build one suite that catches all thirteen bugs, and write a bug report for one of them.	3 h 30 min

Keep these reference pages open in a tab while you work. They're for looking things up rather than for reading start to finish:

- The SimProc instruction set
- The SimProc harness
- Output that is not an error

Getting help

Ask in the team Discord; nobody expects you to work all of this out alone. Getting started says what to include in a question so it can be answered quickly.

Next courses

- DD201 — High-Level Synthesis
- DD202 — Computer Architecture

I

1. Testbenches

2 **Getting started 11**
2 Requirements 11
2 Versions 11
2 Reading a lab's output 11
2 How long things take 12
2 Getting help 12

3 **What a passing test proves 13**
3 Three meanings of "works" 13
3 The test SimProc shipped with 13
3 Thirteen broken processors 14
3 What a bug costs 14
3 What to ask of a passing test 15
3 Two kinds of red 15
3 Check yourself 16

4 **What a testbench is 17**
4 Tests that can fail 18
4 Four ways to catch one bug 18
4 Reading a failure message 19
4 The second circuit in cocotb 19
4 The shortest complete DD104 test 20
4 Check yourself 20

5 **Your first cocotb test 21**
5 The design 21
5 The test 21
5 What it prints 22
5 Where the file names live 23
5 Two commands that run nothing 23
5 Reading a value at the clock edge 24
5 Check yourself 24

6 **The lazy testbench 25**
6 The shipped testbench 25
6 What the two tests report 26
6 What "exhaustive" covers 26
6 Reading the count 27
6 When you can't walk everything 27
6 Two habits 28
6 Check yourself 28

7 **Coverage as a question 29**
7 What the coverage run prints 29
7 What it found 30
7 Three traps in that one output 30
7 Functional coverage on a processor 31
7 Before Lab 01 32
7 Check yourself 32

8 **Lab 01: The adder you already trust . 33**
8 Before you start 33
8 Part 1: Load the course 33
8 Part 2: Run the shipped testbench 34
8 Part 3: Fix the testbench 35
8 Part 4: Read the count 36
8 Part 5: Ask a coverage tool 36
8 Expected results 37
8 Finishing 37
8 Answers 38

2. Getting started

This page covers what DD104 needs from your machine, which tool versions it is pinned to, how to read the labs' output, and how to get help. You'll do the actual setup in Lab 01, Part 1.

Requirements

You did the hard part of the setup in IC101, and you have been using the same container since DD102. DD104 is the lightest course in the digital track to set up, because nothing in it is ever mapped to silicon. That means there is no PDK, no liberty file, no `STD_CELL_LIBRARY` and no LibreLane, and the only tools you need are a simulator, Python and a waveform viewer.

The setup has three pieces, all of which you have used before:

- the noVNC desktop, started from your `workspace` clone with `./scripts/start_vnc.sh` (ports, passwords and blank-page fixes are in IC101: Launch the noVNC desktop);
- `. /foss/designs/common/.designinit`, which in DD104 does one thing for you: it defines the `mod` command;
- `mod add dd104`, which clones this repo into `/foss/designs/modules/dd104`. That folder is your `workspace` folder bind-mounted into the container, so your files are on your own disk and deleting the container does not delete your work.

Every DD104 lab also runs with `make` alone, in a bare container with no environment setup, because each lab pins its own paths. So if a setup step fails late at night, type `make` anyway and read what it tells you.

Versions

The course is pinned to `hpretl/iic-osic-tools:2026.08`. These are the versions it ships:

Tool	Version
Icarus Verilog	`14.0 (devel)`
Verilator	`5.050`
cocotb	`2.0.1`
Surfer	`0.6.0`

Of these, the cocotb version matters most, because cocotb 2.0 renamed things that appear in the first few lines of every 1.x example, and most of what a search engine finds for you was written for 1.x. Fortunately, the list of differences that actually break a run is shorter than you might expect:

- Most 1.x forms still run on 2.0.1 and only print a `DeprecationWarning`: `units=` instead of `unit=`, `MODULE=` in a Makefile, and `sig.value.integer`. You should still write the new spellings, but if your run is red, these aren't the reason.
- Some 1.x forms are hard errors, among them `cocotb.fork` (use `cocotb.start_soon`), `@cocotb.coroutine` (use a plain `async def`), and `dut.sig <= 1` (use `dut.sig.value = 1`). None of their error messages mentions a version number, so they won't tell you that the problem is an old example.

When a tutorial disagrees with this course, go with the course, because it describes the cocotb that is installed in your container. The full measured list is cocotb 2.x and older tutorials.

Reading a lab's output

Every lab ends with a verdict block, and a failed verdict exits non-zero on purpose so that a script or CI job can read the result without grepping text. `make` reports that non-zero exit with a line like this:

```
make: *** [Makefile:49: check] Error 1
```

That line is `make` reporting the failed verdict, and it doesn't mean the Makefile is broken; the lab prints three lines just above it to say so. You will see it often, and it goes away once the verdict passes.

Your first run is meant to fail. Lab 01 hands you a testbench that passes on both a good adder and a broken one, so `make` ends on `VERDICT: FAIL`, and the course starts from there.

Lab writeups mark each number as exact or ballpark. Simulated time, vector counts, cycle counts and verdicts are exact, so if one of those doesn't reproduce on your machine, that's a bug in the course and we want to hear about it. Wall-clock seconds are ballpark, and a difference of ±25 % from one run to the next is normal.

How long things take

The image is the same ~19 GB you pulled for IC101, and DD104 adds nothing to it. These are wall-clock times measured in the pinned container, on a machine that was busy with other work at the time:

Job	Wall time
Lab 01 end to end: two tests, against two adders	~5 s
The example testbench against the SimProc core (Icarus)	~6 s
The same six tests under Verilator, including its build	~17 s (~3 s once built)
The shipped chip test against the good core and all 13 broken ones	~30 s
Verilator toggle coverage on the Lab 01 adder, both stimulus sets	~12 s

Because runs are this cheap, you can rerun your tests every time you change a line, and the labs assume you will.

Getting help

- Ask in the team Discord. No question is too basic. In a verification course, much of what looks like your own mistake is really the tool being unhelpful, and in this course a lot of it is a design that is supposed to be broken.
- Quote the exact error and say which core you were running against. "My test fails" and "my test fails on `B07` and passes on the golden core" are different questions, and the second is much easier to answer. (`B07` is one of the thirteen broken cores, and the golden core is the correct one.) In a Python traceback, the last line is the actual error, and everything above it is the call stack, most of which is cocotb's own machinery.
- Paste the whole verdict block rather than a screenshot of one line. The block says which core, which testbench and which tests fired, and that is most of what someone needs to diagnose the problem.
- If a command in a lab doesn't do what the lab says, please report it. We ran every command before writing it down, so a mismatch means something has drifted since.

Requests to avoid

`labs/common/bugs.py` lists all thirteen bugs, in order, with line numbers. So do `labs/simproc/solutions/bug-NN.md` and `labs/simproc/variants/bug-NN/bug.patch`, the second of which is the one-line edit itself. All of them are in your clone, and nothing stops you reading them, but reading them throws away what the course teaches, which is finding a bug you cannot see. You can only practise that on bugs nobody has told you about.

If you are stuck, ask in Discord for help finding a better question to ask the design, rather than for the answer.

Next: What a passing test proves.

3. What a passing test proves

This page answers: *I ran my design and it did the thing. Why isn't that enough?*

By the end of DD103 you had a `.gds` file, a chip layout that was DRC clean and LVS clean, produced by a flow that ran eighty stages and printed a green tick after each one. None of those stages checked whether your design was right. Each tool answers its own question and leaves the others to someone else. That's a division of labour rather than a flaw in the tools, and the gap it leaves is what this course is about.

Tool	The question it answers	The question it does not answer
Yosys	Can this be built out of gates?	Are they the right gates?
Verilator lint	Is this legal SystemVerilog?	Does it compute what you meant?
OpenSTA	Will the signals arrive in time?	Are they the right signals?
LibreLane / DRC / LVS	Can this be manufactured, and does the layout match the netlist?	Does the netlist do anything useful?

Every one of those checks can pass on a processor whose `SUB` instruction computes `B - A`, and what you get at the end is a clean chip that subtracts backwards.

Three meanings of "works"

When you say a design works, you usually mean one of these and hope it implies the others:

1. It compiled, which means the tool accepted it.
2. It ran, without hanging, crashing or emitting `x`.
3. It produced the answer you expected, on the inputs you thought of.

Only the third is about correctness, and it carries a clause people rarely say out loud: *on the inputs you thought of.* Those inputs were picked by the same person who wrote the bug, so they tend to share the bug's blind spots, and being careful doesn't fix that: a case that didn't occur to you while you wrote the design is unlikely to occur to you while you test it.

> If you came from software, tests may feel like the chore that comes after the interesting part, because if you skip them you find out in staging and push a fix. That last step doesn't carry over to hardware: you can't push a fix to a chip that has already been manufactured.
>
> If you came from circuits, you may expect broken things to look broken, the way a floating node reads as X or a shorted output smokes. An RTL bug doesn't damage anything. What you built is a different, well-formed circuit that computes the wrong function at full speed, with clean edges on every signal, so there's nothing to see that would tell you it's wrong. This is DD103's `unique case` lesson again, in a worse form.

The test SimProc shipped with

From Lab 02 on, the design under test is SimProc: an 8-bit processor of about 1,200 lines of SystemVerilog, written by students on this team and wrapped for a real chip. It has registers, an ALU, a memory and a UART you can talk to.

It shipped with exactly one test. Here is `test/test.py` from the SimProc repository, with the boilerplate stripped out:

```
dut.ui_in.value = 1
dut.uio_in.value = 0
await ClockCycles(dut.clk, 100)
assert dut.uo_out.value == 3
```

It sets one input pin high, waits a hundred clocks and checks that the output byte equals 3. To see what that means, decode `3` against the chip's pin map, where `uo_out[0]` is the UART transmit line, `uo_out[1]` is `halt` and `uo_out[2]` is `done`:

```
uo_out = 0b00000011
           |      └── uart_tx = 1   the serial line is idle
           |     └─── halt    = 1   the core is sitting in its IDLE state
           |    └──── done    = 0   no instruction has completed
```

So what the test asserts is that the processor is not running. It sends no command, loads no program and executes no instruction, and it never looks at a register, a flag or a byte of memory. This is the real file, it passes, and for a long time it was the only evidence anyone had that SimProc worked.

The file even admits as much. In the original, the two lines just above the assertion are this comment:

```
    # The following assersion is just an example of how to check the output values.
    # Change it to match the actual expected output of your module:
    assert dut.uo_out.value == 3
```

That comment comes from the project template, typo included. Nobody overrode a warning or ignored a failure here. The placeholder assertion was simply never replaced, and because it kept passing, the tick stayed green and nothing ever looked wrong. That's why this failure mode is so common: almost every first tapeout looks like this.

Thirteen broken processors

DD104 ships thirteen deliberately broken versions of SimProc. Each one differs from the real core by a one-line edit: `SUB` computes `B - A`, `NAND` is wired as `NOR`, the zero flag looks only at the bottom four bits, branches zero-extend their displacement instead of sign-extending it, `LOAD` reads its address from the destination register instead of the pointer, and so on.

Here is what the shipped test prints when it runs against the good core and all thirteen broken ones, which you'll do yourself in Lab 02, Part 1:

```
-------------------------------------------------------------------------
    PASS  the golden core                              5.7s
    PASS  B01_sub_operands_swapped                     2.1s
    PASS  B02_nand_is_nor                              2.1s
    ...
    PASS  B12_uart_rx_bit_period                       2.0s
-------------------------------------------------------------------------
 13 of 13 deliberately broken processors pass the
 only test this chip has.
=========================================================================
```

You could put one missed bug down to bad luck, but not thirteen out of thirteen. The test spends its hundred clock cycles watching a processor sit still, and every one of the thirteen bugs is somewhere the processor only goes when it is running, so the stimulus never reaches the thing the test is supposed to check.

What a bug costs

Where the bug is found	What it costs
In simulation	An edit and a rerun, about eight seconds
In FPGA bring-up	A re-synthesis and an afternoon in the lab

Where the bug is found	What it costs
In silicon	A new mask set: months and a lot of money
In silicon, in a product	A recall, or a customer working around your bug forever

Each row costs roughly a hundred times as much as the row above it, and that ratio is why pre-silicon verification is a job title, why verification engineers outnumber design engineers on most commercial chip teams, and why the field has its own tools and its own idea of evidence.

In this field, a passing test doesn't count as evidence until you know what it would have caught. A green tick claims only as much as the stimulus behind it exercised, so for `test/test.py` the claim is "this chip powers up idle".

What to ask of a passing test

You'll see this question on every page of the course:

> When a test passes, ask what it would have caught. If you cannot name a specific bug that would turn it red, the green has told you nothing.

The mechanical way to answer it is to run your test against a design you know is broken. Every lab package does this with `make check`, and it won't give you a passing verdict until your testbench has both passed the good design and failed a broken one, because a result needs both of those sides.

Two kinds of red

You should be just as suspicious of a failing test, and forgetting that has already given a DD104 student a false PASS. A test that didn't pass hasn't necessarily checked anything, because there are two unrelated ways for a test to come back red:

	What happened	What it tells you
The check fired	You drove the design, compared what came out with what should have, and they differed	Something about the design
The test blew up	`NameError`, `TypeError`, a failed import, a variable spelled two ways	Something about the testbench

Only the first is a caught bug. The second never compared anything, so it would say the same thing about a perfectly correct design.

This kind of failure tends to hide in code that only runs when something is wrong, such as the line that reports a failed check:

```
if bad:
    raise AssertionError(f"{len(bad)} of {checked} wrong: {summary}")    # `summary`
undefined
```

That line doesn't run until the first time the design gives a wrong answer. So the testbench passes the good design, crashes on the broken one, and from the outside looks exactly as if it had caught the bug. cocotb records both outcomes in `results.xml` as a `<failure>` element, and only the `error_type` attribute tells them apart:

```
<failure error_type="AssertionError" error_msg="256 of 512 vectors are wrong" />
<failure error_type="NameError"      error_msg="name 'summary' is not defined" />
```

DD104's graders once read those two lines the same way, and told a student `3 of 3 ALU bugs caught` about a testbench that had crashed all three times. Every checker in the course now goes through `labs/common/verdict.py`, which is the one place that makes the distinction, and it marks a run like that `BROKEN`, never `caught` or `ok`:

```
  BROKEN  B01_sub_operands_swapped           3 tests, 1 BLEW UP
----------------------------------------------------------------------
 VERDICT: FAIL -- 1 of your tests did not run to the end.
```

So the question has a second half: when a test fails, ask whether it really failed or just fell over.

Check yourself

1. In one sentence, write the strongest true claim the shipped test above supports. (You'll compare your answer with the course's in Lab 02, Part 1.)
2. A testbench passes on the good core and raises `TypeError` on a broken one. Has it caught the bug?

Next: What a testbench is.

4. What a testbench is

This page answers: *what am I writing when I write a testbench? It isn't the design, and it isn't quite a program either.*

A **testbench** is a second circuit, and its job is to doubt the first one. We mean "circuit" literally. The design under test, the **DUT**, has inputs and outputs, and something has to sit on the other end of those wires to drive the inputs, watch the outputs and have an opinion about what it sees. That something is a circuit too, even though you never manufacture it.

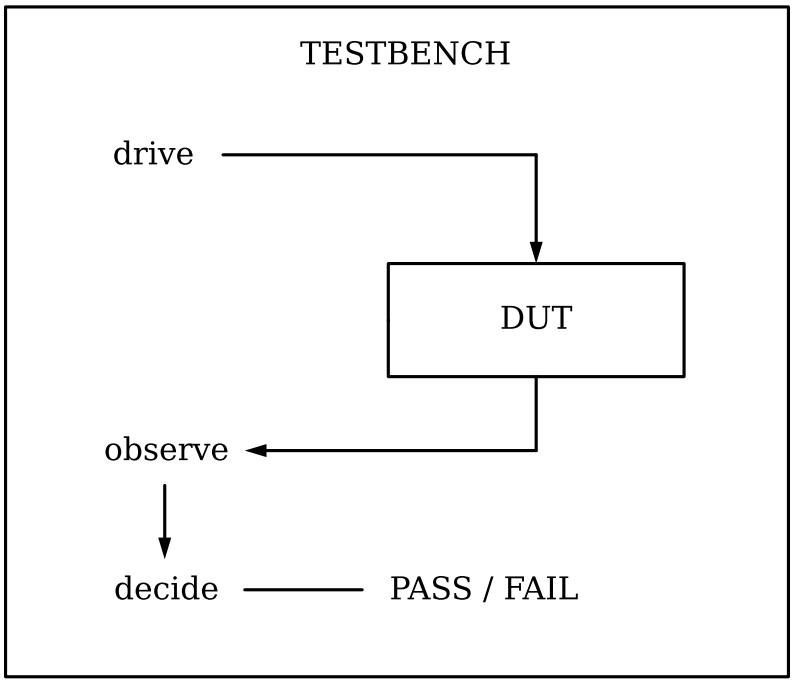


Figure 4.1: The testbench is wrapped around the DUT.

Every testbench does three jobs: drive, observe and decide. A testbench that misses a bug has failed at one of them, so it helps to keep them separate:

Job	The question	How it fails
Drive	Did I put the design in the situation where the bug lives?	The bug is never provoked
Observe	Am I looking at the signal that would show it?	The bug happens off-screen
Decide	Do I have a right answer to compare against?	The bug happens in full view and nothing objects

The shipped SimProc test from the previous page fails at the first job, drive. It never puts the processor in a situation where any of the thirteen bugs can show up, so the observe and decide jobs have nothing to work with.

Tests that can fail

Here is the shortest possible testbench:

```
@cocotb.test()
async def it_works(dut):
    pass
```

It passes on the golden core, on the core whose `SUB` is backwards, on the core that writes every result into the wrong register, and on the core that never signals that an instruction finished. Because it passes on all of them, it can't tell a good core from a broken one.

Every testbench sits somewhere between that and a complete proof, and the only way to find out where yours sits is to make it fail on purpose. Until you have seen a test go red, treat it as a guess about your own code.

That's why the DD104 verdict has two sides, one for the golden core and one for a broken core:

	Must be	Because otherwise
On the golden core	PASS	You are testing the wrong thing, or your expectations are wrong
On a known-broken core	FAIL	You are not testing anything

Neither half means anything on its own, which is why `make check` measures both and, as the previous page said, won't give you a passing verdict until your testbench meets both.

Four ways to catch one bug

Here is `make check` running the shipped example testbench, `labs/common/test_example.py`, against the core whose `SUB` computes `B - A`. You'll run it yourself in Lab 02, Part 3.

```
=========================================================================
 CHECK   testbench: test_example      simulator: icarus
=========================================================================
      ok  the golden core                    6 tests, all passed          7.8s
  caught  B01_sub_operands_swapped           6 tests, 4 FAILED            8.8s
        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
   Your testbench passes the good core, and on B01_sub_operands_swapped a check FIRED:
       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
...
```

Four tests caught the same one-line bug in four different ways, and each is a different kind of observation:

Test	What it noticed	Which job
`t03`	A register holds `199` where `247` was expected	Decide: a value against a known answer
`t04`	The flags say `N=0 Z=0` after an operation that should set `N`	Observe: architectural state that is not a register

Test	What it noticed	Which job
`t05`	The program never reached the address it parks at	Drive: the machine took a path that does not end
`t06`	Instruction 19 disagreed with a reference model	Decide: an independent implementation

`t05` is worth a second look, because nothing in it inspected a wrong value. It ran a loop controlled by a subtraction, the subtraction went the wrong way, and so the loop never ended and the symptom was a timeout rather than a mismatch. A timeout is a perfectly good bug report, as long as your testbench has a deadline. A testbench with no timeout just hangs forever and tells you nothing.

Reading a failure message

You will spend most of this course looking at red text, so it pays to read it well.

```
AssertionError: r1 is 199, expected 247.
```

This is a good failure message, because it tells you what happened (`199`), what should have happened (`247`) and where to look (`r1`). The same bug caught by a worse testbench gives you this:

```
AssertionError
```

It's the same bug, caught at the same instant, but the message tells you nothing about it. The difference is entirely in how the assertion was written:

```
assert sp.reg(1) == 247, f"r1 is {sp.reg(1)}, expected 247."
```

Give every assertion you write in this course a message that names both numbers. When one of your tests fails six weeks from now, whether you wrote that message is what decides whether the failure costs you an hour of confusion.

The wrong number is also a clue in its own right. `t03` runs three instructions on `r1 = 200` and `r2 = 60`:

Step	On the golden core	On `B01`, where SUB is backwards
`ADD r1, r2`	`200 + 60 = 260`, wraps to 4	Same: 4
`SUB r1, r2`	`4 - 60 = -56`, wraps to 200	`60 - 4` = 56
`NAND r1, r2`	`~(200 & 60) = ~8` = 247	`~(56 & 60) = ~56` = 199

The number you're shown, `199`, comes out two instructions after the bug. If you work backwards from it, the middle row gives the bug away: `200 + 56 = 256`, so once you allow for the wrap, the two `SUB` results are exact negations of each other, which is what you get when a subtraction has its operands swapped, so the wrong number points straight at that bug. Reading a wrong number this way is a skill you can practise on every failure you cause.

The second circuit in cocotb

In classic Verilog the testbench is Verilog too: a module with no ports that instantiates the DUT and wiggles its inputs, so it really is a second circuit, written in the same language as the first.

cocotb moves the testbench into Python. Your Python doesn't run inside the simulation; it runs beside it, and the two take turns. `await Timer(1, unit="ns")` looks like a sleep, but what it actually does is hand control back to the simulator, ask it to advance one nanosecond of simulated time, and ask to be woken when it gets there.

That's why a cocotb result line gives simulated time and real time as separate numbers. This one is for `t01_the_obvious_six`, which checks six hand-picked cases and is the first test in the adder testbench of Lab 01:

```
** test_adder.t01_the_obvious_six      PASS         6.00         0.00      7127.11  **
                                                      └─ sim time (ns)  └─ real (s)
└─ ns/s
```

That is six nanoseconds of simulated time, one per case, in almost no real time. Simulated time is deterministic, so you can reason about it: when a DD104 page quotes an exact number of nanoseconds, yours will match to the digit, but your real-time seconds won't match ours.

A cocotb test also needs a simulator underneath it. If you run a test file with `python3` or with `pytest`, neither starts one, so both run nothing and report success; Your first cocotb test shows what they print.

The shortest complete DD104 test

This is a complete test file. It comes from the comment at the top of `labs/common/simproc.py`, the SimProc test harness that every lab imports. The harness also ships six worked tests in `labs/common/test_example.py`, and copying that file is how you are meant to start every SimProc lab.

```
import cocotb
from simproc import SimProc
from simproc_asm import ORI, ADD

@cocotb.test()
async def r1_gets_five(dut):
    sp = SimProc(dut)
    await sp.start()                              # clock, reset, blank memory
    cycles = await sp.run([ORI(5), ADD(1, 0)])    # r0 = 5 ; r1 = r1 + r0
    assert sp.reg(1) == 5, f"r1 is {sp.reg(1)}, expected 5"
    assert cycles == 9, f"took {cycles} cycles, expected 5 + 4"
```

All three jobs are in it: `sp.run` drives, `sp.reg` and the returned cycle count observe, and the two assertions, each with a message, decide. The count is nine because `ORI` takes five cycles and `ADD` takes four. Most people skip checking the cycle count, but it matters, because two of the thirteen bugs leave every register, flag and byte of memory correct and disturb only timing and handshake pins, so a testbench that only compares values can't see them.

The full harness reference is The SimProc harness.

Check yourself

1. Before trusting a green test, what should you do to it?
2. A test reports `r1 is 199, expected 247` on a core you suspect has a swapped `SUB`. Using the table above, which instruction's result would you check first, and what sum would confirm the swap?

Next: Your first cocotb test.

5. Your first cocotb test

This page answers: *a testbench is a Verilog file, isn't it? Why am I writing Python?*

The short answer is that what you actually want from a test, "load this program, run it, and tell me whether register 1 holds 247", takes four lines of Python. In Verilog it takes about forty, and thirty of those are bookkeeping that is easy to get subtly wrong.

cocotb doesn't replace the simulator. Icarus still compiles and runs the SystemVerilog, and cocotb attaches to the running simulation through VPI, a standard interface the simulator already exposes. It hands you the design as a Python object, so you drive a pin by assigning to it and read it back by looking at it.

Every file on this page is in `labs/cocotb-intro/`, and you'll run them yourself in cocotb on a counter.

The design

`rtl/counter.sv` is sixteen lines of logic: an 8-bit counter with an enable, a parallel load and an active-low synchronous reset.

```
always_ff @(posedge clk) begin
    if (!rst_n)      q <= 8'h00;
    else if (load)   q <= d;
    else if (en)     q <= q + 8'd1;
end

assign tc = en & (q == 8'hFF);
```

That excerpt is only the logic. The complete file, with its port list, is `labs/cocotb-intro/rtl/counter.sv`.

The test

This is all of `labs/cocotb-intro/test_01_hello.py` except its module docstring, the comment above the imports and the numbered inline comments:

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import RisingEdge, Timer


@cocotb.test(timeout_time=5, timeout_unit="ms")
async def counts_up(dut):
    """Reset the counter, let it count once, and check that it did."""
    cocotb.start_soon(Clock(dut.clk, 10, unit="ns").start())

    dut.rst_n.value = 0
    dut.en.value    = 0
    dut.load.value  = 0
    dut.d.value     = 0
    await RisingEdge(dut.clk)
    await RisingEdge(dut.clk)

    assert dut.q.value == 0, f"after reset q is {dut.q.value}, expected 0"

    dut.rst_n.value = 1
    dut.en.value    = 1

    await RisingEdge(dut.clk)
```

```
    await Timer(1, unit="ns")

    assert dut.q.value == 1, f"after one enabled clock q is {dut.q.value}, expected 1"
      dut._log.info(f"q = {int(dut.q.value)} after one enabled clock. That is the
whole test.")
```

Every cocotb test you write has the same five parts as this one.

`@cocotb.test()` marks a function as a test. cocotb imports your module and looks for functions with this decorator, so a function without it is just an ordinary Python function: cocotb won't run it, and it won't tell you that it skipped it either. The two arguments in this file put a 5 ms timeout on the test, which Clocks, resets and triggers explains.

`async def` and `await`. Your test is a coroutine. Each `await` means "wake me when this happens", and the simulator runs until it does. No simulated time passes between one `await` and the next, however much Python you run in between.

`dut` is the top-level HDL module as an object, and `dut.q` is the signal named `q`. Sub-modules are attributes too, so in SimProc you will write `dut.tt_um_ieeeuoftasic_simproc.U1.CPU1.curr_state`.

`.value` goes on both sides. `dut.rst_n.value = 0` drives a pin, and `dut.q.value` reads one. In cocotb 1.x, leaving `.value` off a write was a silent bug, because it replaced the attribute with a plain integer and drove nothing at all. cocotb 2.x catches the mistake:

```
AttributeError: Cannot set attribute 'load' on simulation object counter. Did you
forget to add `.value`?
```

So in 2.x, if a signal never changes, a missing `.value` isn't the reason, and you should look for the cause somewhere else.

The clock is a task you start. `Clock(dut.clk, 10, unit="ns")` describes a clock, `.start()` is a coroutine that drives it forever, and `cocotb.start_soon(...)` runs that coroutine in the background. If you leave that line out, the design never moves, because every `await RisingEdge(clk)` waits for an edge that never comes, and the test hangs without printing anything.

What it prints

```
     0.00ns INFO     cocotb                                   Running on Icarus Verilog
version 14.0 (devel)
     0.00ns INFO     cocotb                                 Seeding Python random module
with 1787717251
     0.00ns INFO     cocotb                                   Initialized cocotb v2.0.1
from /usr/local/lib/python3.12/dist-packages/cocotb
     0.00ns INFO     cocotb                                 Running tests
    0.00ns INFO    cocotb.regression                running test_01_hello.counts_up (1/1)
                                                             Reset the counter, let it
count once, and check that it did.
    21.00ns INFO     cocotb.counter                       q = 1 after one enabled clock.
That is the whole test.
    21.00ns INFO     cocotb.regression                    test_01_hello.counts_up passed
   21.00ns INFO    cocotb.regression                ********************************************************
                                                       ** TEST                          STATUS
SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                    ********************************************************************
                                                          ** test_01_hello.counts_up
PASS          21.00           0.00      21998.10  **
                                    ********************************************************************
                                                 ** TESTS=1 PASS=1 FAIL=0 SKIP=0
21.00           0.00       8584.83  **
                                    ********************************************************************
```

The left column is simulated time in nanoseconds, not wall-clock time. The test took 21 ns of simulation, and at the 21,998 simulated nanoseconds per real second in the last column, that was about a millisecond of your time.

`Seeding Python random module with 1787717251` is cocotb reseeding Python's global `random` at the start of every run. When a random test fails, that number lets you get exactly the same run back: set `COCOTB_RANDOM_SEED` to it and run again.

Where the file names live

Nothing in `test_01_hello.py` says which file to compile or which module is the top level. That information lives in `sim.mk`. These are nine of its thirteen working lines. The other four are the `?=` defaults for `HERE`, `TB`, `RTL` and `SIM`, which give the two `$(...)` references their values, and the shipped file also has a comment header:

```
TOPLEVEL_LANG       ?= verilog
COCOTB_TOPLEVEL     ?= counter          # the HDL module `dut` attaches to
COCOTB_TEST_MODULES ?= $(TB)            # the PYTHON module holding your tests
VERILOG_SOURCES     ?= $(RTL)

export PYTHONPATH := $(HERE):$(PYTHONPATH)

ifeq ($(SIM),icarus)
  COMPILE_ARGS += -g2012
endif

include $(shell cocotb-config --makefiles)/Makefile.sim
```

Three of those lines each come with a failure you will run into:

- `COCOTB_TEST_MODULES` is `test_01_hello`, with no `.py` and no path. cocotb imports it as a Python module, so it has to be on `PYTHONPATH`, and that is what the `export` line is for. If you get it wrong, you get `ModuleNotFoundError: No module named 'test_01_hello'` buried inside twenty lines of `importlib` traceback.
- `COMPILE_ARGS += -g2012` is there because Icarus defaults to an older Verilog standard. Without this flag, `logic`, `always_ff` and `always_comb` are syntax errors, and Icarus reports them on lines of perfectly good SystemVerilog.
- `COCOTB_TEST_MODULES` used to be called `MODULE`, and `COCOTB_TOPLEVEL` used to be `TOPLEVEL`. Both old names still work in 2.0.1, but they print a warning. The page on cocotb 2.x and older tutorials covers the renames.

Two commands that run nothing

A cocotb test file is a Python file full of decorated `async def`s, so it looks like something you could run directly or hand to `pytest`. Neither of those runs your test, and neither one tells you that it didn't.

```
python3 test_01_hello.py
```

This prints nothing and exits 0. Python loaded the module, defined the function and stopped, because with no simulator running there was nothing for the test to run against.

```
python3 -m pytest test_01_hello.py
```

```
============================= test session starts ==============================
platform linux -- Python 3.12.3, pytest-9.0.3, pluggy-1.6.0
rootdir: /work/labs/cocotb-intro
plugins: anyio-4.13.0
collected 0 items

============================ no tests ran in 0.03s =============================
```

This exits 5. `collected 0 items` means pytest looked for functions named `test_*` and found none, because a cocotb test is named whatever you called it and is marked with `@cocotb.test()` instead. (cocotb does have a pytest integration, through `cocotb_tools.runner`, but it needs a Python file that builds and launches the simulator, so you can't just point pytest at the test file.)

A cocotb test only runs when a simulator runs it, and in this course it's `make` that starts the simulator. If you didn't see `Running on Icarus Verilog version 14.0` and a `TESTS=` line, nothing was tested, whatever the exit code said.

Reading a value at the clock edge

If you take out the `Timer` line, the test reads the counter at the same instant as the clock edge:

```
    await RisingEdge(dut.clk)
    assert dut.q.value == 1, f"after one enabled clock q is {dut.q.value}, expected 1"
```

You'll try this yourself in cocotb on a counter, Part 1, where it prints this:

```
     20.00ns WARNING  cocotb.Test counts_up.counts_up    after one enabled clock q is
00000000, expected 1
                                                          assert LogicArray('00000000',
Range(7, 'downto', 0)) == 1
                                                        +  where LogicArray('00000000',
Range(7, 'downto', 0)) = LogicArrayObject(counter.q).value
                                                    +    where LogicArrayObject(counter.q)
= HierarchyObject(counter).q
                                                             Traceback (most recent
call last):
                                                           File "/work/labs/cocotb-
intro/test_norace.py", line 58, in counts_up
                                                           assert dut.q.value == 1,
f"after one enabled clock q is {dut.q.value}, expected 1"
                                                        AssertionError: after one
enabled clock q is 00000000, expected 1
     20.00ns WARNING  cocotb.regression                 test_norace.counts_up failed
```

The test doesn't hang or crash. It reads a value that is off by exactly one clock. `await RisingEdge(clk)` returns at the edge, which is the same instant the flip-flop is deciding what its new value will be, and under Icarus you get the old value. Clocks, resets and triggers measures both values side by side and gives three ways to avoid the race.

Two things in that output will appear on every failure you read.

The `assert ... +  where ...` lines aren't part of your message. They come from pytest's assertion rewriting, which cocotb 2.x turns on, and which takes your `assert` apart and shows what each piece of it evaluated to. The line `LogicArrayObject(counter.q) = HierarchyObject(counter).q` is telling you where the value came from. It's useful once you expect it, though it can be alarming the first time you see it.

The message also says `q is 00000000` rather than `q is 0`. That's because `dut.q.value` is a `LogicArray`, and printing one gives you its bit pattern, since a `LogicArray` can hold `X` and `Z`, which have no integer value. `int(dut.q.value)` gives `0`, and on a signal that is partly `X` it raises `ValueError: Can't convert LogicArray to int: it contains non-0/1 values`. Both behaviours are correct, and you'll want both of them.

The race itself comes from the way simulators schedule a clock edge, not from cocotb: it's the difference between the value on a wire at the edge and its value after the clock event has been processed. It's the most common reason a first testbench reports numbers that are all correct but one cycle late.

Check yourself

1. Your test file has no `@cocotb.test()` on one of its functions. What happens to that function when you run `make`?
2. You forgot `cocotb.start_soon(...)` on the clock. What does your test do?
3. Why does the test above need `await Timer(1, unit="ns")` after the rising edge?

Next: The lazy testbench.

6. The lazy testbench

This page answers: *how careful do I actually have to be? My tests look reasonable.*

They probably do look reasonable, and that's the problem, because a test can look reasonable and still miss a bug. The cheapest place to see this happen is in a circuit you already understand completely.

You wrote a 4-bit ripple-carry adder in DD102 and measured it in DD103. It's nine lines long and you know exactly what it does, which makes it the last design under test in this course that you'll be able to say that about. It ships as `labs/lab-01-the-adder-you-already-trust/src/ripple4.sv`:

```
module ripple4 (
    input  logic [3:0] a, b,
    input  logic       cin,
    output logic [3:0] sum,
    output logic       cout
);
    logic [4:0] c;
    assign c[0] = cin;
    genvar i;
    generate
        for (i = 0; i < 4; i++) begin : slice
            fulladder fa (.a(a[i]), .b(b[i]), .cin(c[i]),
                          .sum(sum[i]), .cout(c[i+1]));
        end
    endgenerate
    assign cout = c[4];
endmodule
```

The Lab 01 package also ships `broken/ripple4.sv`, which is that file with one line changed:

```
    assign c[0] = 1'b0;             // <-- was: assign c[0] = cin;
```

With that change, the carry-in never reaches the bottom slice. From Lab 04 on we'll ask you not to read the broken designs, but you're allowed to read this one (the file says so at the top), because Lab 01 isn't a bug hunt. You already know where the bug is; the question here is what kind of testbench notices it.

It's a real kind of bug, too. A port that was tied off during bring-up and never reconnected is one of the most common defects in industrial RTL, and it survives review because the line looks deliberate.

The shipped testbench

The testbench, `test_adder.py`, ships with two tests. Whoever wrote the first one, `t01_the_obvious_six`, was thinking about addition:

```
    cases = [
        # a,  b,  cin
        (0,  0,  0),
        (1,  1,  0),
        (2,  3,  0),
        (7,  8,  0),
        (15, 0,  0),
        (15, 1,  0),
    ]
```

The list covers zero, one, a middling pair, a carry that walks between slices, the top of the range and the wrap. That's a defensible choice, and it's what most people come up with when asked for "a few good test cases".

The second test is the improvement you'd probably reach for next:

```
@cocotb.test()
async def t02_every_a_and_every_b(dut):
    """Every value of a against every value of b.  256 vectors."""
```

It runs every value of `a` against every value of `b`, which is the complete cross product, sixteen by sixteen. If someone asked you whether that testbench was thorough, you'd say yes, and you'd probably call it exhaustive. That answer would be wrong, and the reason it's wrong is worth remembering.

(These are excerpts only. The complete file is `labs/lab-01-the-adder-you-already-trust/test_adder.py`, and it ships in working order.)

What the two tests report

Run against the good adder, both tests pass, and the log reports how much of the input space they covered:

```
   262.00ns INFO     cocotb.ripple4                  checked 256 of the 512 possible
input vectors
```

Run against the adder in `broken/`, they pass as well. `make check` runs the testbench on both adders, and it passes only if the testbench passes the good one (`ok`) and fails the broken one (`caught`). A testbench that passes both gets `MISSED` and `VERDICT: FAIL`. This is what it prints for the shipped testbench, which you'll see for yourself in Lab 01, Part 2:

```
=============================================================================
 CHECK   testbench: test_adder      simulator: icarus
=============================================================================
      ok  src/  the good adder           2 tests, all passed              1.9s
  MISSED  broken/  carry-in dropped      2 tests, all passed              1.9s
-----------------------------------------------------------------------------
 VERDICT: FAIL -- your testbench passes the broken adder too.
   broken/ripple4.sv drops the carry-in on the way into slice 0, so
   every vector with cin = 1 comes out one too small. Every vector
   with cin = 0 is perfectly correct.
   Look at the third column of your stimulus. If cin is 0 in every
   row, this bug is not reachable by anything you drove -- and you
   drove 262 of the 512 vectors this module has.
=============================================================================
```

`MISSED` means that every vector the testbench drove came out green on an adder that's broken. The verdict counts 262 vectors driven, 6 from `t01` and 256 from `t02`, but the six from `t01` are repeated inside the 256.

What "exhaustive" covers

Look again at the line the run printed: `checked 256 of the 512 possible input vectors`. The testbench is telling you itself that it covered only half the inputs. Four bits of `a`, four bits of `b` and one bit of `cin` make $4 + 4 + 1 = 9$ bits of input, and $2^9 = 512$. The cross product of `a` and `b` is complete over eight of those nine bits, which is why it reaches only 256 of the 512 vectors.

Careful people make this mistake too, because it happens whenever you list the inputs you happen to be thinking about. The author of `t02` was thinking "an adder takes two numbers", which is what an adder is in every context except testing it. `cin` is plumbing rather than one of the numbers: it's the port you use when you chain adders, and you don't notice it unless you're chaining them.

> Looping over your inputs makes you thorough only over the input list you wrote down, and that list came from the same mental model as the design, so it has the same blind spots. When you call a testbench "exhaustive", you're making a claim about an input space rather than about the testbench, and the claim is only as good as the space you chose.

If you add `cin` to the sweep and collect the failures instead of stopping at the first one, the test reports how many vectors are wrong. (The reference solution also renames `t02` to `t02_exhaustive`.) Now `t02` fails on the broken adder, so the testbench can tell the two adders apart, and the verdict is PASS:

```
  caught  broken/  carry-in dropped      2 tests, 1 FAILED                1.9s
        t02_exhaustive
             AssertionError: 256 of 512 vectors are wrong. First five: 0+0+1=0 (want
1), 0+1+1=1 (want
----------------------------------------------------------------------
 VERDICT: PASS
   512 vectors on the good adder, all correct.
   512 vectors on the broken one, and the failure is a count, not
   an anecdote:
      t02_exhaustive: AssertionError: 256 of 512 vectors are wrong. First five: 0+0+1=0
(want 1), 0+1+
   That is a test worth committing: it will still catch this bug in
   a year, on a rewrite, done by somebody who has never met you.
```

Making that change is what you do in Lab 01, and it takes four lines.

Reading the count

The fixed test doesn't stop at the first disagreement. It walks all 512 vectors, counts every wrong one and reports the total, which here is exactly half.

Exactly half is an unlikely count for a random mistake; a scrambled carry chain or a bad XOR would give you some awkward fraction instead. A count of exactly 256 out of 512 tells you that the failures track a signal that's high exactly half the time. In a space of $16 \times 16 \times 2$, `cin` has that shape, but so does each bit of `a` and `b`, so the count narrows the search, and the wrong vectors themselves do the rest.

The reference solution's failure message lists the first few wrong vectors and then states the pattern outright, because it was written to:

```
Every wrong vector above has cin=1 and every one is short by exactly 1.
```

No vector is wrong with `cin = 0`, every vector is wrong with `cin = 1`, and each wrong one is short by one. That's enough to diagnose the bug, and you have it before you've read a line of RTL.

Shapes worth recognising

The count	What it usually means
Exactly half	The failure tracks one input that is high half the time
Exactly one quarter	Two such inputs, both high together
Exactly one	An edge: the maximum, the wrap, a single decoded value
All but a handful	The operation is fundamentally wrong; the survivors are cases where right and wrong happen to agree
An odd, un-round number	Probably a real functional bug rather than a wiring mistake; start reading

You'll meet the fourth row soon. One of the SimProc bugs swaps the operands of `SUB`, and of all 65,536 `(A, B)` pairs it gets 65,024 wrong. The 512 pairs that survive aren't random: `A - B` and `B - A` agree exactly when `A == B` (256 pairs) or when the difference is 128 (256 more). `SUB rX, rX`, the usual way to clear a register, is always among those 512, so the most common use of the instruction is one of the few that hide the bug.

When you can't walk everything

Nine input bits give 512 vectors, which is 512 ns of simulated time and about a hundredth of a second of real time. When you're in that situation, take advantage of it.

> If the input space is small enough to walk, walk it. Don't hand-pick cases for a combinational block with fewer than about 2^{20} input combinations, because you won't choose better than a `for` loop that tries everything.

SimProc's ALU is still small enough to walk, at 6 operations × 256 × 256 = 393,216 vectors. You'll do that in Lab 03, where brute force turns up three bugs.

The processor around the ALU isn't. Its four registers, 64 bytes of memory, program counter, two flags and UART make a state space far too big to walk, and because its behaviour depends on sequences, enumerating states wouldn't be enough even if you could. From Lab 04 on, you need the two things that replace exhaustive testing:

- a reference model to compare the design against, instruction by instruction: The golden model;
- a way to ask what your stimulus actually reached, which is not the same as counting how many tests you wrote: Coverage as a question, next.

Two habits

> Sweep the input space instead of sampling it, and count the space first. Add up the input bits, raise two to that power, and make your testbench print how many of them it checked. A line like `256 of the 512 possible input vectors` is enough on its own to catch this bug.

> Read the count before you read the values. A count of exactly half, exactly one or exactly 2^k points at the signal the failure tracks, and it gets you there faster than a waveform will.

Check yourself

1. A module has an 8-bit input, a 3-bit select and an enable. How many input vectors does it have?
2. A sweep of that module reports exactly half of its vectors wrong. What kind of signal do you look for first?

Next: Coverage as a question.

7. Coverage as a question

This page answers: *how do I know when I have tested enough?*

You can't know that for certain, and no coverage number will tell you. Everything useful about coverage follows once you accept this, rather than looking for a way around it.

It helps to treat coverage as a tool that raises questions about your stimulus, rather than as a score to push up. Each question is only worth as much as the answer you give it, and the percentage the tool attaches to them is a side effect.

The quickest way to see this is to point a real coverage tool at the two stimulus sets from The lazy testbench and watch it find the blind spot without being told anything about adders.

What the coverage run prints

The Lab 01 package has a `coverage/` add-on that runs Verilator's toggle coverage on the six hand-picked cases and on all 512 vectors. Icarus has no coverage instrumentation, so this is the one place early in the course where only the stricter simulator, Verilator, can answer the question. You'll run it in Lab 01, Part 5, and this is what it prints:

```
======================================================================
 COVERAGE   what did the stimulus actually reach?
======================================================================

----------------------------------------------------------------------
 six hand-picked cases
----------------------------------------------------------------------
cov_six.dat
  design toggle points : 48
  never toggled        : 5
  design line points   : 0
  the ones that never moved:
    ripple4.sv:3  b[2]:1->0
    ripple4.sv:4  cin:0->1
    ripple4.sv:4  cin:1->0
    ripple4.sv:8  c[0]:0->1
    ripple4.sv:8  c[0]:1->0

        // 4-bit ripple-carry adder built from four full adders.
        module ripple4 (
%000002     input  logic [3:0] a, b,
%000000     input  logic       cin,
%000002     output logic [3:0] sum,
%000001     output logic       cout
        );
%000002     logic [4:0] c;
            assign c[0] = cin;

----------------------------------------------------------------------
 all 512
----------------------------------------------------------------------
cov_all.dat
  design toggle points : 48
  never toggled        : 1
  design line points   : 0
  the ones that never moved:
```

```
    ripple4.sv:3  a[3]:1->0

================================================================
 Coverage did not say whether the adder is right.  It said which
 parts of it your stimulus never moved.  Different question.
================================================================
```

Both runs use `../src/ripple4.sv`, the correct adder, so there is no bug here for anything to find. Coverage isn't looking for one, and that's the first thing to understand about it.

What it found

Start with the `%000000` on the `cin` line of the annotated source.

`verilator_coverage --annotate` rewrites your source with a count in the left margin and a flag for anything it considers under-covered: `%` when a point falls below `--annotate-min` (10 by default). You don't need the full legend, only the two facts on that line: the flag is `%`, and the count is `000000`.

The list above it says the same thing more plainly. Of the 48 toggle points in the design, five never fired, and two of them are `cin:0->1` and `cin:1->0`. The carry-in never moved in either direction during the whole run, and the tool reported it on the line of source where `cin` lives.

It did that without knowing what an adder is, without a reference model, without an assertion, and without being told `cin` was interesting. It asked one mechanical question of every bit in the design, *did this go 0⟶1, and did it go 1⟶0?*, and five of the answers were no.

That's what coverage is good for. It has no idea what the correct answer is, but it knows which parts of the design your stimulus never touched, and those untouched places are where bugs survive.

The other three cold points either follow from those two or are incidental. `c[0]` is the wire that `cin` is assigned to, so it never moved either. `b[2]:1->0` is a real hole, but an incidental one: none of the six cases happens to take that bit from 1 back to 0.

Three traps in that one output

1. `design line points : 0`

Line coverage is what most people mean by "coverage", and it's what software teams track. Here it found zero line points inside `ripple4.sv` and `fulladder.sv`, which means there were no lines to measure at all, not that none of them were covered. The design is made of continuous assignments, and a continuous assignment isn't a statement that executes, so line coverage has nothing to count.

A report saying "100 % line coverage" on this design would therefore be true but would tell you nothing. It would be just as true of the six-case run as of the exhaustive run, so it can't tell the two apart.

> A coverage number means nothing unless you name the metric. "We have 92 % coverage" is not a useful sentence. "We have 92 % statement coverage and have never measured functional coverage" is.

2. The stimulus files check nothing

Open `cov_six.sv`. It has no assertions at all: it drives six values and prints them, and that's the whole file.

Even so, it gets exactly the score that a version checking every result would get. Coverage instruments the design, so it can only measure what your stimulus reached; it has no way of knowing whether you looked at the answer. A testbench that drives every input, toggles every bit and asserts nothing would be 100 % covered and still useless. This happens in practice, too: it's what a large, old, unmaintained regression suite tends to drift into.

	Asserts nothing	Asserts properly
Poor coverage	Useless, and it looks it	Useless, and it looks fine
Good coverage	Useless, and it looks excellent	The only useful combination

The dangerous cell is the bottom-left one, because it's the one with a good number on it.

3. Even the exhaustive run has a hole

In the second block, all 512 vectors ran and one point still never moved: `a[3]:1->0`.

That doesn't mean anything went untested, because every input combination was applied. The sweep ends with `a = 15` and stops there, so the top bit of `a` goes up once and never comes back down. Toggle coverage counts transitions rather than values, which makes the order of your loops part of your stimulus.

The same package has a sharper version of this. `cov_all.sv` puts `cin` in the innermost loop on purpose. If you wrote the same sweep with `cin` outermost, 256 vectors at `cin = 0` and then 256 at `cin = 1`, you would visit every value of `cin` while toggling it only once, so `cin:1->0` would stay cold in a test that really is exhaustive.

> Treat a coverage hole as a question to answer. Uncovered does not mean untested, and covered does not mean correct, so find out which one you are looking at.

Some holes are real gaps, like `cin` in the six-case run, and the fix is to improve your stimulus. Some are artefacts of how the metric is defined, like `a[3]:1->0`, and you can note them and move on. Some are unreachable by construction, and those are the most valuable, because a point your design cannot reach is either dead logic to delete or a sign that part of your understanding is wrong. Chasing a number would have you paper over all three kinds the same way.

Functional coverage on a processor

Toggle coverage on a nine-line adder is a good teacher, because you can hold the whole design in your head. On SimProc it's nearly useless, because almost any program toggles nearly every bit, and you could sit at 99 % with `SUB` subtracting its operands the wrong way round.

What you need there is **functional coverage**: coverage of things you named yourself because they matter to the design's specification, as opposed to its wiring. You write that list yourself. For SimProc it starts like this:

The question	Why it is on the list
Have I executed every opcode?	Eleven exist; three encodings are unimplemented on purpose
Have I executed `SUB` with two different operands?	`SUB rX, rX` is symmetric and hides an operand swap
Have I executed `NAND` with two different operands?	`NAND rX, rX` is how you write NOT, the one case where NAND and NOR agree
Have I used all four shift amounts, in both directions?	Shifting by 1 is what everybody writes
Have I taken a backward branch?	One sign-extension bug still gets forward displacements right
Have I taken and not taken every branch?	A branch never taken has never been tested
Have I run two branches off one compare?	Flags are supposed to survive a branch
Have I checked for one `done` pulse per retired instruction?	One bug disturbs nothing else
Have I run the UART at more than one baud rate?	One bug is invisible at the default rate

You could answer each of those today with a `for` loop and a set, without any coverage tool, because functional coverage is really a list you maintain. Writing the list is the engineering part; the counting afterwards is clerical work.

Read the table again and you'll see it is organised around the ways an operation can be symmetric, defaulted or one-sided, rather than around the design's structure. Those are the places where a wrong implementation and a right one give the same answer, and that is where the thirteen SimProc bugs live. Where bugs hide has the full taxonomy, and every entry in it is a hole that this kind of list is designed to notice.

> Never quote a coverage number without naming the metric and the assertions behind it. When you find a hole, ask a question about it instead of just closing it. Asking “Why did nothing ever set `cin`?” finds the bug, and asking “How do I get this to 100 %?” hides it.

Before Lab 01

You now have the four ideas that the rest of DD104 builds on:

1. A green tick is a claim exactly as large as the stimulus behind it.
2. A testbench drives, observes and decides, and it is only a testbench once you have seen it fail.
3. If you can walk the input space, walk it. When you can’t, the shape of a failure still tells you where to look.
4. Coverage tells you where your stimulus has not been, but never whether the design was right in the places it has been.

Check yourself

1. A colleague reports “100 % line coverage” on a design made entirely of `assign` statements. What has been measured?
2. Why can a testbench with no assertions score full toggle coverage?

Next: Lab 01: The adder you already trust.

8. Lab 01: The adder you already trust

You'll load the course, then take the 4-bit ripple-carry adder from DD102, a copy of it with one line changed, and a testbench that cannot tell them apart. You'll fix the testbench so it catches the broken adder and reports how many vectors are wrong, then see which parts of the adder a coverage tool says your stimulus never moved.

This lab takes about 90 minutes.

Package: `labs/lab-01-the-adder-you-already-trust/`

Before you start

- Read Getting started, What a passing test proves, What a testbench is, Your first cocotb test, The lazy testbench and Coverage as a question.
- You need IC101 done (the noVNC desktop opens and you can get a terminal) and DD102 done (you have written a `ripple4.sv`; the lab ships its own copy in `src/`).
- `broken/ripple4.sv` isn't hidden, and its header says what's wrong with it, so you may read the changed line. This lab is about the testbench rather than a bug hunt; the bug hunts start in Lab 04.

Part 1: Load the course

Can you reach DD104's files, on the tool versions this course's outputs were recorded with? If you haven't set up the workbench yet, do IC101 Lab 03 first.

1. From your `workspace` clone, start the desktop (`./scripts/start_vnc.sh`, or `scripts\start_vnc.bat` on Windows), open http://localhost/ and open a terminal on the desktop. Type every command from here on in that terminal.
2. Load the environment and get the course files:

   ```
   . /foss/designs/common/.designinit   # in every new terminal
   mod add dd104                    # first time only: clones github.com/uoftasic/dd104
   mod dd104                            # every time, including the first
   ```

   You should see the `ASIC-EDU: PDK=sky130A ...` banner, then `OK  module ready: /foss/designs/modules/dd104`, then `cwd: /foss/designs/modules/dd104`. DD104 doesn't use a PDK, so in this course the only thing `.designinit` does for you is define `mod`.
3. Check the tool versions, because every exact number in this course depends on them:

   ```
   iverilog -V | head -1
   verilator --version
   cocotb-config --version
   surfer --version
   ```

   You should see:

   ```
   Icarus Verilog version 14.0 (devel) (s20260301-391-g64f13540-dirty)
   Verilator 5.050 2026-07-01 rev v5.050
   2.0.1
   surfer 0.6.0 (git: v0.6.0)
   ```

   Two `Unable to get version from ...` lines after the Icarus line are harmless; they appear because `head` stops reading early. If `cocotb-config` prints a version starting with `1.`, you are not in the

workbench. Don't use `gtkwave --version` as a check: outside the noVNC desktop it fails with `Could not initialize GTK! Is DISPLAY env var/xhost set?`, which doesn't mean GTKWave is broken.

If you see	Do this
`bash: /foss/designs/common/.designinit: No such file or directory`	The workspace isn't mounted, so relaunch `start_vnc` from your `workspace` clone. In the meantime you can run every lab from a plain `git clone` of `github.com/uoftasic/dd104` inside the workbench image, because `make` needs neither `.designinit` nor `mod`; the `docker run` line for that is in DD104's README.
`bash: mod: command not found`	You haven't loaded `.designinit` in this terminal (the first line of step 2). Every new tab starts without it, so you have to load it again in each one.
`mod: no such module: dd104`	You ran `mod dd104` before `mod add dd104`.
`fatal: could not read Username for 'https://github.com'`	git couldn't read the repo anonymously and is asking you to log in. Check that you typed `dd104`; if you did, ask in Discord. Don't type your password.

For anything else, see IC101's Troubleshooting page.

Check: `cocotb-config --version` prints `2.0.1` and you are in `/foss/designs/modules/dd104`.

Part 2: Run the shipped testbench

Does the shipped testbench catch the broken adder?

1. Before you run anything, write down two answers. You may read the lab's testbench first, `labs/lab-01-the-adder-you-already-trust/test_adder.py`. First: will its two shipped tests catch `broken/ripple4.sv`? Second: if you ran every input the adder has, how many would come out wrong? Give a number, even a guess; you'll check it in Part 3.

2. Run the package:

   ```
   cd labs/lab-01-the-adder-you-already-trust
   make
   ```

   This takes about five seconds. `make` runs `make sim` (your testbench against the good adder in `src/`) and then `make check` (against both adders).

3. Read the `make sim` half. You should see this summary table:

   ```
   ** TEST                              STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO
   (ns/s) **
   ********************************************************************************************
   ** test_adder.t01_the_obvious_six         PASS              6.00              0.00
   7527.92  **
   ** test_adder.t02_every_a_and_every_b    PASS            256.00              0.01
   21811.16  **
   ********************************************************************************************
   ** TESTS=2 PASS=2 FAIL=0 SKIP=0                           262.00              0.01
   19078.99  **
   ```

   and, in the log above it, `checked 256 of the 512 possible input vectors`. The simulated times (`6.00`, `256.00`, `262.00`) and the vector count are exact, so yours should match. The `REAL TIME` and `RATIO` columns measure wall-clock time, so yours will differ.

4. Read the `make check` half. You should see:

```
======================================================================
 CHECK   testbench: test_adder      simulator: icarus
======================================================================
      ok  src/  the good adder           2 tests, all passed              1.9s
  MISSED  broken/  carry-in dropped      2 tests, all passed              1.9s
----------------------------------------------------------------------
 VERDICT: FAIL -- your testbench passes the broken adder too.
   broken/ripple4.sv drops the carry-in on the way into slice 0, so
   every vector with cin = 1 comes out one too small. Every vector
   with cin = 0 is perfectly correct.
   Look at the third column of your stimulus. If cin is 0 in every
   row, this bug is not reachable by anything you drove -- and you
   drove 262 of the 512 vectors this module has.
======================================================================
```

The `262` counts `t01`'s six vectors as well as `t02`'s 256, even though the six are among the 256 and so get driven twice. After the block, `make` prints `make: *** [Makefile:49: check] Error 1`. That line comes from the check exiting with a non-zero status on purpose when the verdict fails (Getting started explains why), and it goes away once the verdict passes.

5. Compare with your first prediction. `t02` is called "every a and every b". Why does it check only 256 of the 512 vectors?

   Answer: A1, in Answers at the end.

Check: the verdict says `MISSED` on `broken/` and `VERDICT: FAIL`.

Part 3: Fix the testbench

Can you make the testbench catch the broken adder and report how many vectors are wrong?

1. Open `test_adder.py` and find the TODO in `t02`. It asks for two changes.
2. Sweep `cin` as well. The shape you want is:

```
for cin in range(2):
    for a in range(16):
        for b in range(16):
            ...
```

3. Stop asserting on the first bad vector. Instead, collect the bad ones in a list, run all 512, and assert once at the end, with `len(bad)` and the total in the message (the reference solution says `256 of 512 vectors are wrong`). Part 4 explains why the count is worth having. Make both changes before you run anything, because `make check` won't pass with only the first one.
4. Run `make sim`. You should see `checked 512 of the 512 possible input vectors` in the log and `518.00` ns total simulated time (6 for `t01` plus 512 for `t02`).
5. Run `make check`. You should see:

```
      ok  src/  the good adder           2 tests, all passed              1.9s
  caught  broken/  carry-in dropped      2 tests, 1 FAILED                1.9s
        t02_exhaustive
            AssertionError: 256 of 512 vectors are wrong. First five: 0+0+1=0 (want
1), 0+1+1=1 (want
----------------------------------------------------------------------
 VERDICT: PASS
```

The test name in that block is whatever you called your test; the reference solution in `solutions/` renames `t02` to `t02_exhaustive`. The `AssertionError` line is cut off on purpose, because `make check` truncates long messages to fit its table. Compare `256` with your second prediction from Part 2.

If the verdict is still FAIL, read its message. `check.py` measures how many vectors each run drove, and from that it tells you which half of the TODO is missing. If you do only the sweep, `broken/` fails, but the run stops at the first wrong vector, so the failure tells you about that one vector instead of giving you a count:

```
VERDICT: FAIL -- the second half of the TODO is not done.
  You swept cin: on the good adder you drove all 512 vectors.
  On the broken adder you drove 263 and then stopped, because you are
  still asserting on the first vector that is wrong instead of collecting
  them.
```

With the loop order from step 2, the 263 is `t01`'s six vectors plus the 257 that `t02` drives up to and including the first one with `cin = 1`.

`check.py` can print one more verdict. The code that builds your failure message runs only once something is wrong, so a misspelled variable in it goes unnoticed on the good adder and crashes the test on the broken one. A checker that only counted failures would score that crash as a catch, so `check.py` reports it as a test that blew up:

```
  BROKEN  broken/  carry-in dropped      2 tests, 1 BLEW UP           1.3s
        t02_every_a_and_every_b   <- this test BLEW UP; it checked nothing
            NameError: name 'summary' is not defined
----------------------------------------------------------------------
VERDICT: FAIL -- 1 of your tests did not run to the end.
```

Python stopped before it reached the comparison, so the run tells you nothing about either adder; fix the name and run it again. The code that tells a test that failed an assertion from a test that crashed is in `labs/common/verdict.py`, if you want to see how DD104 does it.

Check: `make check` ends on `VERDICT: PASS`, and your failure message on `broken/` reports 256 wrong vectors out of 512.

Part 4: Read the count

What does `256 of 512` tell you about the bug before you open the RTL?

1. Look at the count in your failure message: 256 of 512, exactly half. Which input in the port list is high in exactly half of all input vectors? Write it down.

   Answer: A2, in Answers at the end.

2. Compare with the first-bad-vector verdict quoted in Part 3. What does each style of test report on the broken adder?

   Answer: A3, in Answers at the end.

A failure count tells you something on any design, not just on adders. Here are the ones you'll meet in Lab 03 and what each says:

Count	What it says
Exactly half	The fault tracks a signal that is high half the time
65,024 of the 65,536 `SUB` vectors	512 vectors come out right despite the bug, and they share a pattern
0 values wrong, 25,726 flags wrong	The arithmetic is right, so look at the flag logic

Check: you can name the signal the 256 failures track.

Part 5: Ask a coverage tool

Which parts of the adder did the six hand-picked cases never move?

1. Before you run it, write down whether you expect the all-512 run, which drives every input combination, to leave any toggle point cold.
2. Run the coverage add-on, a small package in the lab's `coverage/` folder:

```
cd coverage
make
```

It takes about twelve seconds because it builds the design twice with Verilator. Both runs use the correct adder in `../src/`, and neither uses your `test_adder.py`: the stimulus comes from the add-on's own files, `cov_six.sv` (the six hand-picked cases) and `cov_all.sv` (all 512).

3. In the six-case block, you should see `design toggle points : 48`, `never toggled : 5` and these five points:

```
    ripple4.sv:3  b[2]:1->0
    ripple4.sv:4  cin:0->1
    ripple4.sv:4  cin:1->0
    ripple4.sv:8  c[0]:0->1
    ripple4.sv:8  c[0]:1->0
```

`cin` never moved in either direction. Why is `c[0]` cold too?

Answer: A4, in Answers at the end.

4. In the all-512 block, you should see `never toggled : 1`, the point `a[3]:1->0`. Every input combination ran, so why does one point stay cold?

Answer: A5, in Answers at the end.

5. Note `design line points : 0` in both runs. What would a report of "100 % line coverage" on this adder tell you?

Answer: A6, in Answers at the end.

The add-on has its own `README.md`.

Check: six cases leave 5 points cold, including both `cin` transitions; all 512 leave 1.

Expected results

Check	Expected	
`make check` final line	`VERDICT: PASS`	exact
`make sim` total simulated time	`518.00` ns	exact
The log line in your sweep	`checked 512 of the 512 possible input vectors`	exact
The count in your failure on `broken/`	256 of 512	exact
`make` wall clock	about 5 s	ballpark
`coverage/`, six cases	48 toggle points, 5 never toggled, 0 line points	exact
`coverage/`, all 512	48 toggle points, 1 never toggled, 0 line points	exact

Finishing

You're done when `make check` on your `test_adder.py` ends on `VERDICT: PASS` and your failure message on `broken/` reports 256 wrong vectors out of 512.

Questions:

1. The reference solution keeps `t01_the_obvious_six`, even though your sweep drives every vector it drives and 506 more. Would you delete it? Decide before you open the answer.

Answer: A7, in Answers at the end.

Submit: nothing for this lab.

Next on the Course path: read Clocks, resets and triggers, cocotb 2.x and older tutorials, Driving and checking a design and Waveforms, then do Lab: cocotb on a counter.

Answers

A1. Part 2: Run the shipped testbench

`t02` drives every value of `a` and every value of `b`, so it's exhaustive over two of the three inputs, but `cin` stays at 0 the whole time. The module has $4 + 4 + 1 = 9$ bits of input, so it has $2^9 = 512$ input vectors, not 256, and `t02` drives only the half with `cin = 0`.

A2. Part 4: Read the count

It's `cin`, the only one-bit input. A failure count of exactly half means the fault tracks a signal that is high in half the vectors. Each bit of `a` and `b` is also high in half the vectors, so the count alone doesn't pick out `cin`, but the wrong vectors the reference solution's message lists (`0+0+1`, `0+1+1`, ...) all have `cin = 1`. You have narrowed the bug to one wire without opening the RTL.

A3. Part 4: Read the count

A test that asserts on the first bad vector stops the run there (at index 256, the first vector with `cin = 1`) and reports that one wrong answer. A test that collects the failures reports `256 of 512`, and only that count tells you where to look for the bug, as step 1 showed.

A4. Part 5: Ask a coverage tool

`c[0]` is the internal carry wire that `cin` drives (`assign c[0] = cin;` in `ripple4.sv`), so it's cold for the same reason. The tool knows nothing about adders and never checked an answer, yet it still pointed at `cin`, the input the bug drops. The `b[2]:1->0` point is a real gap but an incidental one, because none of the six cases happens to lower that bit.

A5. Part 5: Ask a coverage tool

`cov_all.sv` has `a` in its outermost loop (and `cin` innermost), so the loops end with `a` at 15: `a[3]` rises once, and the test finishes before it can fall again. The cold point comes from the loop order rather than from a gap in the stimulus, which did drive every input combination. Compare this with your prediction from step 1.

A6. Part 5: Ask a coverage tool

It would tell you nothing. The design is made only of continuous assignments, so line coverage has nothing in it to count. The claim would be true but empty, and it would be just as true of the six-case run as of the exhaustive one.

A7. Finishing

You can defend either choice. The reference keeps it because a test suite is read as often as it's run, and six named cases with a comment document what the module is for. `solutions/README.md` makes the full case, and describes one more decision that's worth disagreeing with.

II 2. cocotb

9 Clocks, resets and triggers 41
9 `Clock`, and the edge at t = 0 41
9 Reading at the clock edge 41
9 `Timer`: absolute time, no clock needed 42
9 `ClockCycles` 42
9 Reset 43
9 Waiting for something that never happens 43
9 Triggers 44
9 Check yourself 44

10 cocotb 2.x and older tutorials 45
10 Forms that still work 45
10 Forms that fail loudly 46
10 A filter that runs nothing 48
10 Conversion table 48
10 Ask the installed cocotb 49
10 Check yourself 49

11 Driving and checking a design 51
11 The two counters 51
11 The six tests 51
11 Three test shapes worth copying 52
11 Assertion messages 52
11 Whole-sequence and random checks 53
11 Reference models 53
11 Check yourself 54

12 Waveforms 55
12 Getting a waveform out 55
12 FST or VCD 56
12 Opening a viewer 56
12 Surfer 56
12 GTKWave 57
12 What to look at first 57
12 A bug you can only see in the picture 58
12 Check yourself 58

13 Lab: cocotb on a counter 59
13 Before you start 59
13 Part 1: Your first test 59
13 Part 2: Triggers 60
13 Part 3: The 1.x traps 60
13 Part 4: A two-sided verdict 61
13 Part 5: Waveforms 62
13 Expected results 63
13 Finishing 63
13 Going further (optional) 63
13 Answers 64

9. Clocks, resets and triggers

This page is for when your test hangs, or reads the wrong value, or finishes instantly, and you want to know *what `await` is actually waiting for*.

A cocotb test is always in one of two states: it is either running your Python or waiting for the simulator, and simulated time moves only while it is waiting. Every bug on this page comes from being wrong about which of the two your test is doing at a given moment.

All the output below was measured on `hpretl/iic-osic-tools:2026.08` (cocotb 2.0.1, Icarus Verilog 14.0) by `labs/cocotb-intro/test_02_triggers.py`. It has six tests, which between them simulate 1832.51 ns and take well under a second of real time. You'll run it in cocotb on a counter, Part 2.

`Clock`, and the edge at t = 0

```
cocotb.start_soon(Clock(dut.clk, 10, unit="ns").start())
```

That line does three separate things:

`Clock(dut.clk, 10, unit="ns")`	Creates an object that describes a clock, but doesn't drive anything.
`.start()`	Gives you a coroutine that toggles the signal forever. That still drives nothing, because a coroutine doesn't run until something runs it.
`cocotb.start_soon(...)`	Schedules that coroutine as a background task and returns immediately.

The test checks that `start_soon` really does return immediately, with no simulated time passing:

```
t_before = get_sim_time("ns")
cocotb.start_soon(Clock(dut.clk, 10, unit="ns").start())
assert get_sim_time("ns") == t_before
```

> Always give `unit=`. The default is `unit="step"`, and one step here is one picosecond, so `Clock(dut.clk, 10)` is a 10 picosecond clock. It runs without any errors, and every timing number in your test comes out off by a factor of a thousand.

Now count ten clock edges starting from `t = 0`, and then ten more:

```
    90.00ns INFO  cocotb.counter  10 edges counted from t=0 took 90 ns (nine periods
-- the edge at t=0 was free)
   190.00ns INFO  cocotb.counter  10 more edges, now starting on an edge: 100 ns
```

`Clock.start()` drives the signal high at t = 0 (`start_high=True` is the default). That first `0`⟶`1` transition is a rising edge and takes no time, so ten edges counted from `t = 0` span only nine periods. The next ten start on an edge, so they span ten periods, as you'd expect.

You don't need to work around this. Reset the design first, and after the reset you are at a known phase of the clock, so the arithmetic stops surprising you.

Reading at the clock edge

The race that costs everyone an hour sooner or later looks like this in the test:

```
await RisingEdge(dut.clk)
at_the_edge = str(dut.q.value)
await Timer(1, unit="ns")
one_ns_later = str(dut.q.value)
```

```
   221.00ns INFO  cocotb.counter  q read AT the rising edge : 00000000
   221.00ns INFO  cocotb.counter  q read 1 ns after the edge: 00000001
   225.00ns INFO  cocotb.counter  q read on the FALLING edge : 00000001 (settled, no
Timer needed)
```

The flip-flop changes because of the edge, and your test wakes up because of the same edge. Your test doesn't get to choose which of the two happens first, and on Icarus 14.0 you see the old value, from before the flip-flop changed.

Seen from the outside, this failure doesn't raise an exception and doesn't hang. It gives you values that are all correct but one clock late, which is exactly what a design that really is one clock late would give you, so you go and read the RTL looking for the bug, which is where the hour goes.

There are three fixes, listed here from crudest to best:

```
await RisingEdge(dut.clk); await Timer(1, unit="ns")   # crude, obvious, fine
await RisingEdge(dut.clk); await ReadOnly()         # the scheduler's own settled phase
await FallingEdge(dut.clk)                                  # sample mid-cycle, like
real hardware
```

The third does what real hardware does, launching data on one edge and sampling it far from that edge, and it needs no extra `await`. It's the one the labs use.

If a testbench ever reports values that are all correct but one cycle out of step, add `await Timer(1, unit="ns")` after the edge before you change anything else. If the numbers then snap into place, the bug was in the testbench rather than in the design.

`Timer`: absolute time, no clock needed

```
   226.00ns INFO  cocotb.counter  Timer(1, unit='ns') advanced 1.0 ns
   226.50ns INFO  cocotb.counter  Timer(500, unit='ps') advanced 0.5 ns
  1226.50ns INFO  cocotb.counter  Timer(1, unit='us') advanced 1000.0 ns
```

That test doesn't start a clock, because `Timer` counts absolute simulated time and doesn't need one. Use it for settle time, for modelling a real-world delay, and for the reset experiment below.

Two errors are worth meeting here, before you run into them in your own tests:

```
  1226.50ns INFO  cocotb.counter  Timer(0, unit='ns') -> ValueError: Timer argument
time must be positive
  1226.50ns INFO  cocotb.counter  Timer(1, unit='s')  -> ValueError: Invalid unit
(s) provided
```

The legal units are `"step"`, `"fs"`, `"ps"`, `"ns"`, `"us"`, `"ms"` and `"sec"`. Note that seconds are spelled `sec`, not `s`, which is what the second error is about.

`ClockCycles`

```
  1336.50ns INFO  cocotb.counter  ClockCycles(clk, 5) from 1 ns past an edge: 49 ns
  1336.50ns INFO  cocotb.counter  ClockCycles(clk, 5) starting on an edge   : 50 ns
  1361.50ns INFO  cocotb.counter  ClockCycles(clk, 3, edge_type=FallingEdge): 25 ns
```

Five edges take five periods only if you start on an edge. If you start 1 ns past one, the first of the five is only 9 ns away, so the total is 49 ns.

`edge_type=FallingEdge` counts falling edges instead. (`ClockCycles(clk, n, rising=False)` also works; it's the older spelling.)

Reset

`rst_n` is active low and synchronous, which means that pulling it low does nothing until a rising clock edge arrives. SimProc, the course's 8-bit processor, uses the same convention. You don't have to take it on trust, because you can test it in two lines:

```
dut.rst_n.value = 0            # asserted, but no clock edge yet
await Timer(3, unit="ns")      # a Timer moves time WITHOUT a clock edge
```

```
  1385.50ns INFO  cocotb.counter  rst_n low, no clock edge yet: q = 0x5A
  1392.50ns INFO  cocotb.counter  one rising edge later     : q = 0x00
```

`0x5A` survives three nanoseconds of asserted reset and is cleared on the first edge. If the first line had printed `0x00`, this would be an asynchronous reset, which is a different circuit with different rules about when reset may be released.

Reset style is hard to change late in a project, and you can't see it in a block diagram, so it's worth spending these two lines of testbench to settle it, once for each design.

This is the reset routine the labs use:

```
async def reset(dut, cycles=2):
    dut.rst_n.value = 0
    dut.en.value    = 0
    dut.load.value  = 0
    dut.d.value     = 0
    await ClockCycles(dut.clk, cycles)
    dut.rst_n.value = 1
    await Timer(1, unit="ns")
```

It drives every input, not just the interesting ones, because an input you never drive stays `X`. The `X` propagates through the design, and you end up debugging a signal that has no value at all rather than one with a wrong value.

Waiting for something that never happens

```
await RisingEdge(dut.tc)
```

If `tc` never rises, this waits forever. A hung test is the worst failure there is, because it gives you no message, no line number and no traceback, and in CI it becomes a job that uses up its whole time budget before anyone learns anything.

Either of these fixes it in one line:

```
await with_timeout(RisingEdge(dut.tc), 200, "ns")     # raises SimTimeoutError
winner = await First(RisingEdge(dut.tc), Timer(200, unit="ns"))   # tells you which
arrived
```

```
   1613.51ns INFO   cocotb.counter   with_timeout(RisingEdge(dut.tc), 200, 'ns') ->
SimTimeoutError, as it should: en is low, so tc never rises
  1813.51ns INFO  cocotb.counter  First(RisingEdge(dut.tc), Timer(200, unit='ns')) ->
the Timer won
  1832.51ns INFO  cocotb.counter  tc went high at t = 1833 ns with q = 0xFF
```

Use `with_timeout` when the event not arriving means the test has failed, and `First` when either outcome is legitimate and you want to branch on which one happened. There is also a version that puts a timeout on a whole test:

```
@cocotb.test(timeout_time=50, timeout_unit="us")
async def something_that_might_hang(dut):
    ...
```

Put a timeout on any test that waits on a signal the design generates. In this course that includes every SimProc test, since they all wait for `done`.

Triggers

Trigger	Wakes when
`Timer(n, unit=...)`	`n` units of simulated time have passed
`RisingEdge(sig)` / `FallingEdge(sig)`	That edge arrives on a 1-bit signal
`Edge(sig)` / `ValueChange(sig)`	The signal changes at all
`ClockCycles(clk, n)`	`n` edges of `clk` have passed
`First(a, b, ...)`	The first of several triggers fires; it returns the one that won
`Combine(a, b, ...)`	All of the triggers have fired
`with_timeout(trig, n, unit)`	`trig` fires, or raises `SimTimeoutError` if `n` units pass first
`ReadOnly()`	The settled phase of the current time step: values final, writes forbidden
`ReadWrite()`	The phase where writes are allowed
`NextTimeStep()`	The next time step, however far away
`Event()`	Another coroutine calls `.set()`, for tests that talk to each other

Import them from `cocotb.triggers`. (`Join`, for waiting on a task you started, moved to `cocotb.task` in 2.x. Importing it from `cocotb.triggers` still works, but gives a warning.)

Check yourself

1. `Clock(dut.clk, 10)` with no `unit=`. What is the clock period?
2. You start on a rising edge and `await ClockCycles(dut.clk, 3)` with a 10 ns clock. How much simulated time passes? What if you started 2 ns after the edge?
3. Your test reads `q` right after `await RisingEdge(dut.clk)` and every value is one count behind. Name two ways to fix the testbench.

Next: cocotb 2.x and older tutorials.

10. cocotb 2.x and older tutorials

This page answers: *I copied a cocotb example off the internet and it does not work, or it works but prints warnings. Which of us is wrong?*

Neither of you. The example is right for cocotb 1.x, but this course runs cocotb 2.0.1, and the 2.0 release renamed and removed a lot of the 1.x interface.

Almost every cocotb tutorial, blog post, Stack Overflow answer, chatbot completion and university lab handout was written against 1.x. Some of those examples still run on 2.0.1, and they are the dangerous ones, because they give you no reason to suspect them. That's why this page starts with them.

We ran everything below in the workbench and pasted the output unedited. The four demonstration files ship in `labs/cocotb-intro/oldapi/`, and you'll run them in cocotb on a counter, Part 3.

Forms that still work

`units=` instead of `unit=`

`units=` is the 1.x form you'll come across most often, and cocotb 2.0.1 still accepts it:

```
cocotb.start_soon(Clock(dut.clk, 10, units="ns").start())
await Timer(20, units="ns")
```

```
     0.00ns WARNING  py.warnings   /work/labs/cocotb-intro/oldapi/test_1x_units.py:24:
DeprecationWarning: The 'units' argument has been renamed to 'unit'.
                           cocotb.start_soon(Clock(dut.clk, 10, units="ns").start())
...
   41.00ns INFO    cocotb.regression  test_1x_units.the_old_spelling_still_runs passed
** TESTS=1 PASS=1 FAIL=0 SKIP=0                                  41.00           0.00
19633.12  **
```

The test passes. The warning is a single `WARNING` line among the INFO lines, so it scrolls past, and nothing else in the run tells you that this code will stop working. A deprecation means the old spelling is going to be removed in a later release, so write `unit=`.

`.integer`, `.binstr`, `.signed_integer`, `.buff`

These value getters behave the same way: they still work and they warn, and each warning names the replacement to use.

```
    41.00ns WARNING  py.warnings   DeprecationWarning: `logic_array.integer` getter is
deprecated. Use `logic_array.to_unsigned()` instead.
    41.00ns INFO     cocotb.counter   v.integer = 42
    41.00ns WARNING  py.warnings   DeprecationWarning: `logic_array.binstr` getter is
deprecated. Use `str(logic_array)` instead.
    41.00ns INFO     cocotb.counter   v.binstr  = 00101010
```

1.x	2.x
`sig.value.integer`	`int(sig.value)` or `sig.value.to_unsigned()`
`sig.value.binstr`	`str(sig.value)`
`sig.value.signed_integer`	`sig.value.to_signed()`
`sig.value.buff`	`sig.value.to_bytes(byteorder="big")`

`sig.value.is_resolvable` is not deprecated, and it's the right way to ask whether a value contains any `X` before you convert it.

`MODULE` and `TOPLEVEL` in the Makefile

1.x makefiles say:

```
TOPLEVEL = counter
MODULE   = test_tiny
```

Under 2.0.1's `Makefile.sim` these still work, with one warning:

```
/usr/local/lib/python3.12/dist-packages/cocotb_tools/makefiles/simulators/
Makefile.icarus:66: Using MODULE is deprecated, please use COCOTB_TEST_MODULES instead.
...
** TESTS=1 PASS=1 FAIL=0 SKIP=0                 1.00          0.00       1178.18  **
```

The 2.x names are `COCOTB_TOPLEVEL` and `COCOTB_TEST_MODULES`. Use those, as DD104's `sim.mk` does.

> The translation from the old names to the new ones is done by cocotb's makefiles, not by cocotb itself. If you drive a simulation without the makefiles (with the Python runner, `pytest` or a hand-written `vvp` command line), only `COCOTB_TEST_MODULES` is read, and `MODULE` is ignored without any warning.

`COCOTB_TESTCASE`

`COCOTB_TESTCASE` is deprecated in favour of `COCOTB_TEST_FILTER`, which takes a regular expression:

```
     0.00ns WARNING  py.warnings   DeprecationWarning: COCOTB_TESTCASE is deprecated
in favor of COCOTB_TEST_FILTER
```

Read A filter that runs nothing before you use either.

Forms that fail loudly

These produce a traceback with a line number, so once you have seen each one you'll be able to fix it in seconds.

`cocotb.fork` → `cocotb.start_soon`

```
cocotb.fork(Clock(dut.clk, 10, unit="ns").start())
```

```
      0.00ns WARNING  ..b.Test fork_is_gone.fork_is_gone    module 'cocotb' has no
attribute 'fork'
                                                             Traceback (most recent
call last):
                                                           File "/work/labs/cocotb-
intro/oldapi/test_1x_fork.py", line 22, in fork_is_gone
                                                         cocotb.fork(Clock(dut.clk,
10, unit="ns").start())   # AttributeError
                                                          ^^^^^^^^^^
                                                    AttributeError: module 'cocotb'
has no attribute 'fork'
```

The replacement is `cocotb.start_soon(...)`, with the same argument. (There is also `cocotb.start(...)`, which is the version you `await`; it lets the new task run before your code continues.)

`dut.sig <= value` → `dut.sig.value = value`

1.x let you assign to a signal with `<=`, the way you would in Verilog:

```
dut.rst_n <= 1
```

```
     0.00ns WARNING  ..re.le_is_not_assignment_any_more   '<=' not supported between
instances of 'LogicObject' and 'int'
                                                              Traceback (most recent
call last):
                                                             File "/work/labs/cocotb-
intro/oldapi/test_1x_assign.py", line 25, in le_is_not_assignment_any_more
                                                                    dut.rst_n <= 1
# TypeError
                                                                 ^^^^^^^^^^^^^^^
                                                            TypeError: '<=' not supported
between instances of 'LogicObject' and 'int'
```

A `TypeError` about `<=` reads like a bug in a comparison, because to Python the line is a comparison. It means that the author used the old assignment form, which 2.0 removed.

`cocotb.binary` and `cocotb.result`

```
from cocotb.binary import BinaryValue
from cocotb.result import TestFailure
```

These fail while cocotb is importing your file, before any test runs, so there is no test name in the output and no results table:

```
  (this one never reached 'Running tests' -- it failed while cocotb was
   still importing the file, so no test ran at all)
Traceback (most recent call last):
  ...
  File "/work/labs/cocotb-intro/oldapi/test_1x_imports.py", line 20, in <module>
    from cocotb.binary import BinaryValue     # ModuleNotFoundError, at IMPORT time
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ModuleNotFoundError: No module named 'cocotb.binary'
```

`cocotb.result` is harder to spot. The module still exists but is now empty, so the error is about a missing name rather than a missing module, and it reads like a typo:

```
AttributeError: module 'cocotb.result' has no attribute 'TestFailure'
```

1.x	2.x
`from cocotb.binary import BinaryValue`	`from cocotb.types import LogicArray`
`raise TestFailure("...")`	`assert cond, "..."`
`raise TestSuccess()`	`cocotb.pass_test()`, or just `return`
`@cocotb.coroutine` + `yield`	`async def` + `await`
`cocotb.regression.TestFactory`	`@cocotb.parametrize(...)` (`TestFactory` still exists)

The second traceback

When cocotb fails during discovery (a missing module, an import error, no tests found), it prints a second traceback after the real one:

```
Traceback (most recent call last):
    File "/usr/local/lib/python3.12/dist-packages/cocotb/_init.py", line 117, in
_sim_event
    cocotb._regression_manager._fail_simulation(msg)
  File "/usr/local/lib/python3.12/dist-packages/cocotb/regression.py", line 899, in
_fail_simulation
    self._running_test.abort(self._sim_failure)
```

```
    ^^^^^^^^^^^^^^^^^^
AttributeError: 'RegressionManager' object has no attribute '_running_test'
     0.00ns ERROR    gpi   Passing event to upper layer failed
```

You can ignore this one. It's cocotb's own error handler failing while it reports your error, because there was no running test for it to abort. The real problem is in the traceback above it. You can tell the two apart because `RegressionManager`, `_running_test`, `gpi` and `_sim_event` are not names from your code.

A filter that runs nothing

None of the forms above will cost you a day, but this one can.

`make nothing` in `labs/cocotb-intro` runs the real six-test suite with two letters swapped in the test filter:

```
COCOTB_TESTCASE = chekcing
```

```
     0.00ns INFO     cocotb   Initialized cocotb v2.0.1 from /usr/local/lib/python3.12/
dist-packages/cocotb
     0.00ns WARNING  py.warnings   DeprecationWarning: COCOTB_TESTCASE is deprecated
in favor of COCOTB_TEST_FILTER
     0.00ns INFO     cocotb   Running tests
     0.00ns WARNING  cocotb.regression   No tests left after filtering with: chekcing$

   exit status 0, no FAIL anywhere, and zero tests ran.
```

There is no results table, so there's no `TESTS=`, no `PASS=` and no `FAIL=`. The whole run, including the lines not shown above, is six INFO lines, two WARNINGs and a clean exit, which in CI shows up as a green tick.

`results.xml` does record what happened, if anyone reads it:

```
<testcase name="load_then_count" classname="test_03_checking" ... >
  <skipped />
</testcase>
```

All six tests are marked `<skipped/>`, and none of them ran.

If you take one habit from this page, make it this one: after every cocotb run, read the `TESTS=` line and check that the count matches the number of tests you wrote. The colour of the output and the exit code won't tell you; only the count will. A cocotb run that tested nothing looks just like one that tested everything, apart from one missing line.

This is DD104′s argument on a small scale. SimProc shipped with one test that passes on thirteen broken processors, and a filter typo passes on all thirteen and on the good one too. That's why "My tests pass" doesn't count as a result here, while "Six tests ran, they passed on the good core, and this one fails on the broken core" does.

Conversion table

You found (1.x)	Write this (2.0.1)	What happens if you don't
`Timer(1, units="ns")`	`Timer(1, unit="ns")`	Works, `DeprecationWarning`
`Clock(clk, 10, units="ns")`	`Clock(clk, 10, unit="ns")`	Works, `DeprecationWarning`
`Clock(clk, 10)`	`Clock(clk, 10, unit="ns")`	Works, as a 10 ps clock
`cocotb.fork(coro)`	`cocotb.start_soon(coro)`	`AttributeError: module 'cocotb' has no attribute 'fork'`
`dut.sig <= 1`	`dut.sig.value = 1`	`TypeError: '<=' not supported between instances of 'LogicObject' and 'int'`

You found (1.x)	Write this (2.0.1)	What happens if you don't
`dut.sig = 1`	`dut.sig.value = 1`	`AttributeError: ... Did you forget to add `.value`?` (silent in 1.x)
`sig.value.integer`	`int(sig.value)`	Works, `DeprecationWarning`
`sig.value.binstr`	`str(sig.value)`	Works, `DeprecationWarning`
`sig.setimmediatevalue(v)`	`sig.value = Immediate(v)`, from `cocotb.handle`	Works, `DeprecationWarning`
`from cocotb.binary import BinaryValue`	`from cocotb.types import LogicArray`	`ModuleNotFoundError` at import
`raise TestFailure("x")`	`assert cond, "x"`	`AttributeError` on `cocotb.result`
`@cocotb.coroutine` / `yield`	`async def` / `await`	`AttributeError: module 'cocotb' has no attribute 'coroutine'`
`from cocotb.triggers import Join`	`from cocotb.task import Join`	Works, `DeprecationWarning`
`MODULE=`	`COCOTB_TEST_MODULES=`	Works in makefiles and warns; ignored elsewhere
`TOPLEVEL=`	`COCOTB_TOPLEVEL=`	Works, warns
`TESTCASE=` / `COCOTB_TESTCASE=`	`COCOTB_TEST_FILTER=` (a regex)	Works, warns

Ask the installed cocotb

The installed version's own function signatures are the only reference that is guaranteed to match the cocotb you are running. `inspect.signature(cocotb.test)`, for example, prints the keywords that the `@cocotb.test` decorator accepts:

```
(obj:    Union[Callable[...,    Coroutine[cocotb._base_triggers.Trigger,    NoneType,
NoneType]],
 cocotb._decorators.Parameterized, NoneType] = None, *, timeout_time: Optional[float]
= None,
 timeout_unit: Literal['step', 'fs', 'ps', 'ns', 'us', 'ms', 'sec'] = 'step',
    expect_fail:    bool    =    False,    expect_error:    Union[Type[BaseException],
Tuple[Type[BaseException],
 ...]] = (), skip: bool = False, stage: int = 0, name: Optional[str] = None) -> ...
```

(We wrapped it here to fit the page; it comes out as one long line.) If a keyword you read about somewhere isn't in that list, it's from 1.x. You'll run this check yourself in the lab.

Check yourself

1. A tutorial's test runs, passes and prints `DeprecationWarning: The 'units' argument has been renamed to 'unit'.` Is anything wrong with your result?
2. A run prints no results table and exits 0. What happened?
3. `TypeError: '<=' not supported between instances of 'LogicObject' and 'int'`. What did the author of that line mean to write?

Next: Driving and checking a design.

11. Driving and checking a design

This page answers: *my testbench passes. Is that good?*

On its own, a pass tells you almost nothing. SimProc's one shipped test asserts `uo_out == 3`, which decodes to "the processor is not running", so it passes on the real core and on all thirteen broken ones, including the one whose `SUB` computes `B - A`. When a testbench passes, all you have learned is that it accepted the design, which is a claim about the testbench rather than about the design. The claim you want to be able to make about a testbench has two sides:

It passes on the design that is right, and it fails on the design that is wrong.

Running `make` in `labs/cocotb-intro` checks both sides, which is why the package ships a broken counter alongside the real one. This is the verdict it prints:

```
----------------------------------------------------------------------
 VERDICT
----------------------------------------------------------------------
   test_03_checking.py PASSED on rtl/counter.sv
   test_03_checking.py FAILED on rtl/counter_broken.sv

   Both halves matter. A testbench that only passes has not been
   shown to be able to fail, and a testbench that cannot fail is
   not evidence of anything.
----------------------------------------------------------------------
```

You'll run it in cocotb on a counter, Part 4.

The two counters

`rtl/counter_broken.sv` differs from `rtl/counter.sv` by one term:

```
good:    assign tc = en & (q == 8'hFF);
broken:  assign tc =       (q == 8'hFF);
```

With the `en &` missing, every value of `q` is still correct, on every cycle and from every starting point, so the count itself is perfect. Only `tc` is wrong, and only while `en` is low.

Most real bugs look like this: the design as a whole still works, and one output is wrong in one situation that nobody tests.

The six tests

`test_03_checking.py` has six tests, and their names say what they are for:

Test	What it is for	Catches the `tc` bug?
`load_then_count`	The ordinary path: load a value, count eight times	No
`enable_holds_the_count`	The case where the design must do nothing	No
`load_wins_over_enable`	Priority between two controls asserted together	No
`terminal_count_needs_enable`	`tc` at `q = 0xFF` with `en` low	Yes, every time

Test	What it is for	Catches the `tc` bug?
`the_wrap_is_where_the_bugs_live`	The exact cycle 255 → 0, as `(q, tc)` pairs	No
`against_a_reference_model`	600 random clocks against three lines of Python	No

Five of the six pass on the broken counter, and one of those five is the 600-clock random test.

Three test shapes worth copying

The first is a test of the case where nothing should happen. `enable_holds_the_count` drives `en = 0`, clocks twenty times, and checks that the count did not move. Half of any specification is about when the design must hold still, and few people write a test for it.

The second tests priority explicitly. `load_wins_over_enable` raises `load` and `en` together, so one of them has to win. Which one wins is decided by the order of two `else if` branches, and someone tidying up the code in an editor can swap those without any tool warning them.

The third tests the output nobody looks at, which here is `tc`. `terminal_count_needs_enable` is that test, and it's the only one in the file that catches the bug. Flags, status bits, `done` pulses and `valid` lines are architectural state just as much as a data value is, and a testbench that checks only data has no way to see a bug in a flag.

Assertion messages

Here is the assertion most people write:

```
assert dut.tc.value == 0
```

and here is the one in the lab:

```
assert tc == 0, (
    f"q is 0xFF and en is 0, so the count is NOT about to wrap, but tc is {tc}.\n"
    f"  tc should be `en & (q == 8'hFF)`. A tc of 1 here means the `en &`\n"
    f"  is missing and tc is really just `q == 8'hFF`.\n"
    f"  Everything about the COUNT is still correct -- only the flag is wrong,\n"
    f"  and only while en is low.\n  {state(dut)}")
```

`make broken` prints it:

```
   404.00ns WARNING  ..able.terminal_count_needs_enable q is 0xFF and en is 0, so the
count is NOT about to wrap, but tc is 1.
                                                            tc should be `en & (q ==
8'hFF)`. A tc of 1 here means the `en &`
                                                          is missing and tc is really
just `q == 8'hFF`.
                                                           Everything about the COUNT
is still correct -- only the flag is wrong,
                                                             and only while en is low.
                                                               q=0xFF tc=1 en=0 load=0
rst_n=1 at t=404 ns
                                                      assert Logic('1') == 0
```

When you read that, you know which line of RTL to open, without rerunning anything, adding a print or opening a waveform. That's the standard to aim for: write the message you would want to see if this test failed at 2 a.m. six months from now, on someone else's machine. Three habits get you there.

The first is a `state()` helper whose output you paste into every message:

```
def state(dut):
    return (f"q=0x{int(dut.q.value):02X} tc={dut.tc.value} "
            f"en={dut.en.value} load={dut.load.value} "
            f"rst_n={dut.rst_n.value} at t={get_sim_time('ns'):.0f} ns")
```

It prints one line with every input, every output and the simulation time, and you only have to write it once per design.

The second is to say in the message what each possible wrong answer would mean:

```
assert got == 0x11, (
    f"with load AND en both high, q became 0x{got:02X}.\n"
    f"  0x11 means load won (correct for this design).\n"
    f"  0x12 means load won and en also applied.\n"
    f"  anything else means en won.\n  {state(dut)}")
```

With messages like that, a failure hands you a diagnosis that you wrote before you needed it.

The third is to report the difference as well as the two values. `expected 0x35, got 0x36 — a difference of +1` names the bug, whereas `expected 0x35, got 0x36` leaves the reader to do the subtraction at 2 a.m.

Whole-sequence and random checks

Comparing a whole sequence tells you more than comparing one value. `the_wrap_is_where_the_bugs_live` collects five `(q, tc)` pairs and compares the list in a single assertion, so when it fails you see the whole trajectory:

```
    486.00ns INFO  cocotb.counter  q, tc across the wrap: (254,0) (255,1) (  0,0)
(  1,0) (  2,0)
```

Random stimulus finds bugs only by accident. `against_a_reference_model` runs 600 clocks against a six-line Python model of the counter and checks `q` and `tc` on every one. It is the most thorough test in the file, yet it passes on the broken counter, and its own log shows why:

```
  6507.01ns INFO  cocotb.counter  600 random clocks, q and tc agreed with the model on
every one (final q = 0x7C)
  6507.01ns INFO  cocotb.counter  of those 600 clocks, 0 had q = 0xFF with en low --
the only ones where a broken tc would have shown
```

In 600 random clocks, the one situation that exposes the bug occurred zero times, because the counter only reaches `0xFF` while it is running, and while it is running the broken `tc` and the correct `tc` agree.

The seed is pinned (`random.Random(104)`), so that zero means the same thing on your machine as it does here. If you unpin it, you get a test that passes on Tuesday and fails on Wednesday, and you have no way to tell whether you changed the design or just drew a different hand.

When a random or constrained-random test passes, ask how many of its vectors reached the situation you care about, and count them, which takes a counter in the loop and one extra log line. If the answer is zero, the test told you nothing about that situation, however many vectors it ran. Running a lot of vectors is not the same as covering the cases that matter.

Reference models

The model in `against_a_reference_model` is three lines:

```
if load:      model_q = d
elif en:      model_q = (model_q + 1) & 0xFF
model_tc = 1 if (en and model_q == 0xFF) else 0
```

It is written from the specification, in a different language, by someone who is allowed to be lazy about how. It's worth having because it checks things you did not think to check: a directed test looks at one signal at a time, while a model looks at all of them on every cycle.

It also has a limit that matters just as much: the model can only be wrong in the same ways you are. This one can't see `tc` depending on `en` in any situation the stimulus never reached, and it has no idea that `done` or `halt` exist. In SimProc that gap is a whole class of bug, because a core that computes every answer correctly and gets its handshake wrong passes a model comparison, and that's why a suite needs more than one kind of test.

Check yourself

1. Why does `against_a_reference_model` pass on the broken counter even though it checks `tc` on every clock?
2. Write the assertion message you would want for a counter that holds when it should count. What values and signals does it name?

Next: Waveforms.

12. Waveforms

This page answers: *the test says `q is 0x32, expected 0x31`. I have read the RTL four times and it looks right. Now what?*

The next step is to look at the wires themselves, in a waveform. An assertion tells you that something was wrong at one instant, while a waveform shows what everything was doing for the whole run. You reach for one when the assertion has stopped narrowing things down, usually because the wrong value came from somewhere you weren't watching, or because the value was right but arrived at the wrong time.

Most bugs in this course are quicker to find by improving the assertion message than by opening a viewer, and if you make a habit of opening the waveform first, it will cost you hours. Some bugs, though, show up only in the picture, and this page is about those. You'll do everything it describes in cocotb on a counter, Part 5.

Getting a waveform out

Running `make waves` in `labs/cocotb-intro` runs `test_04_waves.py` with waveform dumping turned on, and prints this:

```
    11.00ns INFO  cocotb.counter  reset released at t = 11 ns
   111.00ns INFO  cocotb.counter  counted to 0x0A at t = 111 ns
   121.00ns INFO  cocotb.counter  loaded 0xFF at t = 121 ns  <-- zoom here
   151.00ns INFO  cocotb.counter  held 3 clocks with en low, still 0xFF, t = 151 ns
   152.00ns INFO  cocotb.counter  en high again at t = 152 ns; the wrap is the next edge
   161.00ns INFO  cocotb.counter  wrapped to 0x00 at t = 161 ns
   211.00ns INFO  cocotb.counter  finished at t = 211 ns with q = 0x05
...
FST info: dumpfile /work/labs/cocotb-intro/build/waves/counter.fst opened for output.

   wrote  /work/labs/cocotb-intro/counter.fst   (699 bytes)
   wrote  /work/labs/cocotb-intro/counter.vcd   (1098 bytes)
```

The `FST info: dumpfile ... opened for output.` line appears after the results table because it comes from Icarus closing the file at the end of the run. It isn't an error, and by the time it's printed the test has already passed.

All of this comes from one make variable, `WAVES=1`, passed to cocotb's makefile. With it set, cocotb's Icarus makefile generates a dump module for you and adds it to the compile, so you never have to write `$dumpfile` or `$dumpvars` yourself:

```
/foss/tools/bin/iverilog  ...  -s  cocotb_iverilog_dump   rtl/counter.sv  build/waves/
cocotb_iverilog_dump.v
```

The dump is written to `$(SIM_BUILD)/$(COCOTB_TOPLEVEL).fst`, which here is `build/waves/counter.fst`. The package's `make waves` then copies it up to `counter.fst` and converts it with `fst2vcd`, so you end up with both formats.

> If you drive `sim.mk` by hand and change `RTL=`, change `SIM_BUILD=` along with it. cocotb's makefile relinks only when the sources are newer than `sim_build/sim.vvp`, so a second run against a different design can quietly rerun the first one. The test goes green, but it has tested something other than what you meant, and that's why every package target passes its own `SIM_BUILD`.

> If you search for "cocotb waveform" you'll find `COCOTB_PLUSARGS=+WAVES` and hand-written `$dumpvars` testbench shells, and both are real techniques. SimProc's `tb.v` uses the `$dumpvars` approach, because it already has a Verilog shell to put it in. For a design that cocotb attaches to

directly, `WAVES=1` does the same job with one variable, so write the shell yourself only when you need to dump a subset of a large design.

FST or VCD

The two formats hold the same information in different encodings. Use FST unless some tool refuses to read it.

	This counter	SimProc, `labs/common` → `make waves`
Signals in the file	7	190 (across 11 scopes)
VCD	1,098 bytes	3,007,592 bytes
FST	699 bytes	12,152 bytes

As FST, the SimProc dump is 247 times smaller. VCD is plain text, with every value change written out in ASCII, while FST is compressed and indexed, so a viewer can seek to any point in it without reading the whole file. `fst2vcd` and `vcd2fst`, both on the workbench `PATH`, convert in either direction.

VCD is still worth knowing, because everything reads it, and because when you suspect the viewer rather than the data you can open a VCD file with `less` and look at it directly.

Opening a viewer

Start Surfer and GTKWave by typing their names in a terminal on the noVNC desktop, never by clicking a desktop icon. Both viewers print their real complaints to the terminal they were started from. If you start one from a menu there is no such terminal, so a viewer that can't open your file tells you nothing and just sits there looking normal.

Surfer

The workbench has Surfer 0.6.0. Here is `counter.fst` open in it:

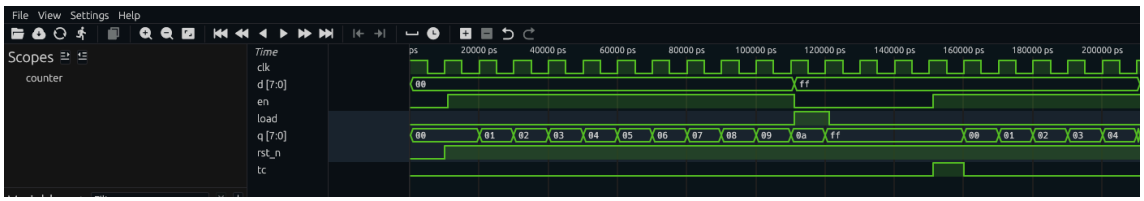


Figure 12.1: Surfer showing the counter's full run

That's the whole 211 ns run. Reset releases, the counter walks from `00` to `0a`, `load` pulses and `d` puts `ff` in, and `en` drops for three clocks while `q` sits at `ff`. Then `en` comes back, `tc` goes high for exactly one clock, and the count wraps to `00`.

Using it takes three steps:

1. In the Scopes pane, top left, click the module (`counter`). For SimProc you walk down `tb` → `tt_um_ieeeuoftasic_simproc` → `U1` → `CPU1`.
2. In the Variables pane, bottom left, click signals to add them, or type `module_add counter` in the command box to add every signal in a scope at once.
3. Drag on the time ruler to zoom. `Ctrl` + scroll also zooms.

The time axis is in picoseconds, because cocotb's default simulation precision is 1 ps. That's why the ruler in the screenshot reads `20000 ps`, `40000 ps` and so on, and `20000 ps` is 20 ns. Your test logs times in nanoseconds, so multiply them by 1000 before you go looking for them on the ruler.

Surfer can also read a command file, which saves you re-adding the same signals every time. We made the screenshot that way, with a file containing `module_add counter` passed to Surfer with `surfer -c`.

GTKWave

GTKWave (`v4.0.0-prealpha` on the workbench) is the older of the two. It reads both formats, and prints `FSTLOAD | Processing 7 facs. / Built 7 signals and 0 aliases.` when it loads FST and `[0] start time. / [211000] end time.` when it loads VCD.

GTKWave has one trap that you should know about before you run into it, which is that it opens fully zoomed in, at time zero:

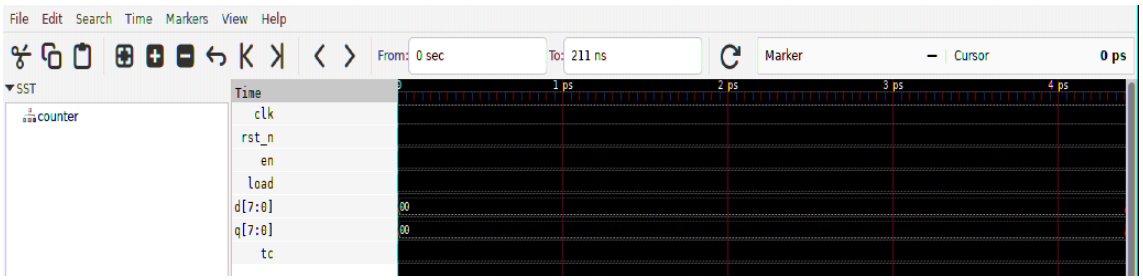


Figure 12.2: GTKWave opening at 0-4 ps, showing nothing

The waveform loaded correctly: the header even says `To: 211 ns`, and the seven signals are listed on the left. The wave pane looks empty because it's showing only the first four picoseconds of the 211 ns run, before anything has happened. The first clock edge after `t = 0` is at 10 ns, which is 2,500 screens to the right.

The fix is Zoom Fit, which is under `View → Zoom → Zoom Best Fit`, on `Ctrl Alt F`, and on the toolbar as the magnifier with the square in it. Do it first, every time you open a file, before you conclude anything from what's on the screen.

If GTKWave opens with an empty signal list instead, that's the other half of the same trap. The signals are in the SST tree at the top left, and nothing appears in the wave pane until you `Append` them to it. A `.gtkw` save file remembers that choice, so you only have to make it once. `counter.gtkw` ships with the package, and it's just a plain-text list of signal names.

Use Surfer unless you have a reason to prefer GTKWave: Surfer opens at a sensible zoom, its keyboard handling is better, and it's useful without a save file. GTKWave is the one to use when you want its analogue rendering or transaction filters, or when you're following someone else's `.gtkw`.

What to look at first

A whole processor has a lot of signals (SimProc's dump has 190), and if you add all of them you'll spend your time scrolling instead of debugging. Start with these four kinds of signal, in this order.

1. The clock and the reset. They aren't interesting in themselves, but when everything is wrong, they're usually the first thing that went wrong. A clock that never toggles means you forgot `cocotb.start_soon`, and a reset that never releases means you drove `rst_n` and never raised it. Either one gives you a screen of flat lines, and either takes seconds to fix.
2. Anything `X` or red. In both Surfer and GTKWave, red means an undriven or conflicting signal. An `X` propagates (one undriven input turns a whole datapath red three cycles later), so trace it back to the first signal that is red and fix that one, because everything downstream of it is only a symptom.
3. The state register. In a design with a control FSM, it carries more information than any other signal. In SimProc it's `CPU1.curr_state`, which decodes to `IDLE` and `CYCLE_1` to `CYCLE_5`. If you watch it walk `CYCLE_1 → CYCLE_2 → CYCLE_3 → CYCLE_1`, you know that a three-cycle instruction retired, before you've looked at any data.
4. Whatever your assertion named, at the time it named. That's why `test_04_waves.py` logs `loaded 0xFF at t = 121 ns  <-- zoom here`. If you know the timestamp before you open the viewer, you can go straight to it and read the waveform there, instead of scrolling through the whole run looking for the problem.

Before you add a fifth signal, ask yourself what you expect it to show. If you can't say, you're browsing rather than debugging, and you'd do better to go back to the assertion and make it print more.

A bug you can only see in the picture

Run against the broken counter (`rtl/counter_broken.sv`, the same file with `en &` deleted from `tc`), `test_04_waves` still passes, and its log is identical line for line to the good run's, with the same times, the same values and the same `PASS`. That's because `test_04_waves` never checks `tc`, so nothing in the text output can tell the two counters apart. In the waveform the difference is easy to see:

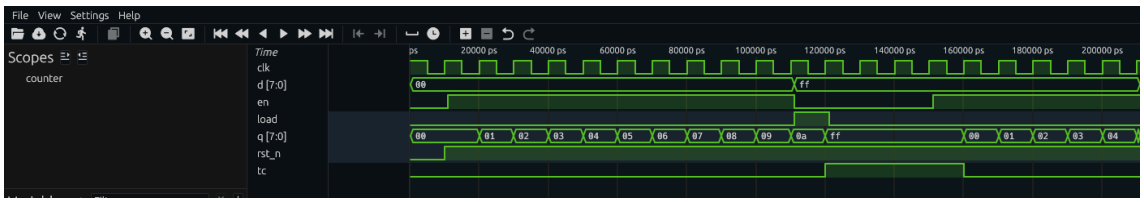


Figure 12.3: Surfer showing the broken counter: tc high for the whole parked stretch

Compare the bottom trace with the earlier screenshot. On the good counter, `tc` is one narrow pulse, one clock wide, at t = 152–161 ns: the one clock where `q` is `0xFF` and `en` is high. On the broken counter, `tc` goes high at t = 121 ns, the instant `q` becomes `0xFF`, and stays high for the whole four-clock stretch, including the three clocks where the counter isn't moving.

Everything else on the screen is pixel-identical, so the only difference between the two runs is one trace that is four times too wide, and the log doesn't show it at all. Some bugs have the shape of a waveform, and recognising those shapes is the reason to learn the viewer before you need it rather than during the hour you need it.

Check yourself

1. GTKWave shows seven flat signals and the header says `To: 211 ns`. What do you do first?
2. You open a SimProc waveform to chase a wrong register value. Which signals do you add, in what order?

Next: cocotb on a counter.

13. Lab: cocotb on a counter

In this lab you run every cocotb example from the guide pages on an 8-bit counter: your first test, the clock-edge race, the triggers, the 1.x forms that warn and the ones that fail, a two-sided verdict, and the counter's waveforms in Surfer and GTKWave. Nothing here is graded; the lab is practice with the tools before the processor arrives in Lab 02. Twice along the way you'll break something on purpose and check that the right tests notice.

This lab takes about 75 minutes.

Package: `labs/cocotb-intro/`

Before you start

- Finish Lab 01.
- Read Your first cocotb test, Clocks, resets and triggers, cocotb 2.x and older tutorials, Driving and checking a design and Waveforms.
- Work in a terminal on the noVNC desktop (Part 5 opens graphical viewers), in the package directory:

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

The design is `rtl/counter.sv`: an 8-bit counter with an enable, a parallel load that takes priority over the enable, and an active-low synchronous reset. Its `tc` output is high on the cycle before the count wraps. `rtl/counter_broken.sv` is the same file with `en &` deleted from the expression for `tc`.

Part 1: Your first test

What does the smallest cocotb test print, and what goes wrong if it reads `q` at the clock edge?

1. Run the minimal test:

   ```
   make hello
   ```

   You should see `Running on Icarus Verilog version 14.0 (devel)`, then `q = 1 after one enabled clock. That is the whole test.` at `21.00ns`, and `TESTS=1 PASS=1 FAIL=0 SKIP=0` with `21.00` ns of simulated time.

2. Try the two commands that look like they should run the test:

   ```
   python3 test_01_hello.py; echo "exit=$?"
   python3 -m pytest test_01_hello.py; echo "exit=$?"
   ```

   The first prints nothing and `exit=0`, and the second prints `collected 0 items` and `no tests ran`, with `exit=5`. Neither of them started a simulator, so neither ran a test; in this package, only `make` starts one.

3. Next, make the clock-edge race happen by copying the test and deleting its `Timer` line. Start with the copy:

   ```
   cp test_01_hello.py test_norace.py
   ```

   Open `test_norace.py` and delete the line `await Timer(1, unit="ns")`, the one between the last `await RisingEdge(dut.clk)` and the final `assert`. Before you run it, write down what you expect the `assert` to read from `q`. Then run it through `sim.mk` directly, with its own build directory:

```
make -f sim.mk sim TB=test_norace SIM_BUILD=build/norace
```

You should see the test fail at `20.00ns` with `after one enabled clock q is 00000000, expected 1`, then `TESTS=1 PASS=0 FAIL=1`. The `make` error lines after the table are cocotb's makefile reporting the failed test.

4. Read the failure. Which lines are your assertion message, and which are pytest's assertion rewriting (`assert ... + where ...`)? Why does the message print `00000000` and not `0`?

   Answer: A1, in Answers at the end.

Check: `make hello` passes, and `test_norace` fails with `q is 00000000` one clock early.

Part 2: Triggers

How long does each trigger really wait, and is the counter's reset synchronous?

1. Run the trigger tests:

```
make triggers
```

You should see `TESTS=6 PASS=6 FAIL=0 SKIP=0` and `1832.51` ns of simulated time.

2. Find these lines in the log and match each to its section of Clocks, resets and triggers:

```
10 edges counted from t=0 took 90 ns (nine periods -- the edge at t=0 was free)
q read AT the rising edge : 00000000
q read 1 ns after the edge: 00000001
ClockCycles(clk, 5) from 1 ns past an edge: 49 ns
rst_n low, no clock edge yet: q = 0x5A
one rising edge later     : q = 0x00
First(RisingEdge(dut.tc), Timer(200, unit='ns')) -> the Timer won
```

3. Why did ten edges counted from t = 0 take 90 ns and not 100?

   Answer: A2, in Answers at the end.

4. The two reset lines also tell you something about the design: whether the counter's reset is synchronous or asynchronous. Write down which it is, and which line tells you.

   Answer: A3, in Answers at the end.

Check: six tests pass in 1832.51 ns, and you can say why ten edges from t = 0 took 90 ns.

Part 3: The 1.x traps

Which cocotb 1.x forms still run on cocotb 2.0.1, and which fail?

1. Before you run them, write down which of the four files in `oldapi/` you expect to pass. The file names say which 1.x form each one uses. Then run them:

```
make oldapi
```

You should see four results in order:

File	Result
`oldapi/test_1x_units.py`	`DeprecationWarning: The 'units' argument has been renamed to 'unit'.`, then `TESTS=1 PASS=1`
`oldapi/test_1x_fork.py`	`AttributeError: module 'cocotb' has no attribute 'fork'`, `TESTS=1 PASS=0 FAIL=1`

File	Result
`oldapi/test_1x_assign.py`	`TypeError: '<=' not supported between instances of 'LogicObject' and 'int'`, `FAIL=1`
`oldapi/test_1x_imports.py`	`ModuleNotFoundError: No module named 'cocotb.binary'`, and no results table, because it failed while cocotb was importing the file

2. Which of the four is the most dangerous to find in old code?

   Answer: A4, in Answers at the end.

3. Run the suite with a mistyped test filter:

   ```
   make nothing
   ```

   You should see `No tests left after filtering with: chekcing$`, no results table, and `exit status 0, no FAIL anywhere, and zero tests ran.`

4. Look at what cocotb recorded:

   ```
   grep -c "<skipped" build/nothing.xml
   ```

   You should see `6`: six tests, all skipped.

5. Ask the installed cocotb which keywords `@cocotb.test()` takes:

   ```
   python3 -c "import cocotb, inspect; print(inspect.signature(cocotb.test))"
   ```

   You should see one long line starting `(obj: Union[Callable[..., Coroutine[...` and listing `timeout_time`, `timeout_unit`, `expect_fail`, `expect_error`, `skip`, `stage` and `name`.

Check: you can say which of the four 1.x forms passed, and why `make nothing` exits 0.

Part 4: A two-sided verdict

Does the checking testbench pass the good counter, fail the broken one, and notice a counter you break yourself?

1. Run everything:

   ```
   make
   ```

   This runs `hello`, `triggers`, `sim` and `broken`, which comes to four simulator runs and nineteen tests, with 0.3 s of simulation. The whole command takes 8–14 s of wall-clock time, depending on how busy the machine is. You should see:

   ```
   ********************************************************************************
    VERDICT
   ********************************************************************************
      test_03_checking.py PASSED on rtl/counter.sv
      test_03_checking.py FAILED on rtl/counter_broken.sv
   ```

2. Scroll up to the `make broken` section, where one test fails and five pass. Find the failing test's message, which starts `q is 0xFF and en is 0, so the count is NOT about to wrap, but tc is 1.` Which line of RTL does it send you to?

   Answer: A5, in Answers at the end.

3. Find the two log lines from `against_a_reference_model`. How many of its 600 random clocks had `q = 0xFF` with `en` low?

Answer: A6, in Answers at the end.

4. Now break the good counter yourself. Open `rtl/counter.sv` and change

```
else if (en)     q <= q + 8'd1;
```

to `q <= q + 8'd2`. Before you run, write down which of the six tests in `test_03_checking.py` you expect to fail. Then run:

```
make sim
```

You should see three of the six tests fail: `load_then_count`, `the_wrap_is_where_the_bugs_live` and `against_a_reference_model`, with `TESTS=6 PASS=3 FAIL=3`. The first failure is on the first comparison:

```
31.00ns WARNING  .. load_then_count.load_then_count counting from 0x30: expected
0x31, got 0x32 -- a difference of +1
```

The `assert 50 == 49` line under it is pytest's assertion rewriting showing the same numbers in decimal. `the_wrap_is_where_the_bugs_live` shows the whole trajectory, `[(255, 1), (1, 0), (3, 0), (5, 0), (7, 0)]`, which is the counter stepping by two.

5. What do `a difference of +1` and the `q=0x32 ... at t=31 ns` in the full message tell you?

Answer: A7, in Answers at the end.

6. Undo the edit and run `make sim` again. You should see `TESTS=6 PASS=6 FAIL=0 SKIP=0`.

Check: `make` prints the verdict above, and `rtl/counter.sv` is back to `q + 8'd1`.

Part 5: Waveforms

The broken counter prints the same log as the good one, so what does its bug look like in a waveform?

1. Make a waveform:

```
make waves
```

You should see the log lines from `reset released at t = 11 ns` to `finished at t = 211 ns with q = 0x05`, including `loaded 0xFF at t = 121 ns  <-- zoom here`, then `FST info: dumpfile ... opened for output.` after the results table (which is normal), and two `wrote` lines for `counter.fst` and `counter.vcd`.

2. Open it in Surfer, from this terminal:

```
surfer counter.fst
```

Click `counter` in the Scopes pane, then add `clk`, `rst_n`, `en`, `load`, `q` and `tc` from the Variables pane. Then zoom to 121 ns; Surfer's ruler is in picoseconds, so look near `121000 ps`.

You should see `q` reach `ff` at 121 ns, hold for three clocks with `en` low, and `tc` go high for exactly one clock before `q` wraps to `00`.

3. A command file saves you all that clicking next time:

```
printf 'module_add counter\n' > view.sf
surfer -c view.sf counter.fst
```

You should see every signal in `counter` added at once.

4. Open the same file in GTKWave, with the shipped save file:

```
gtkwave counter.fst counter.gtkw
```

If the wave pane looks empty, that's because it is showing only the first few picoseconds. Use `View → Zoom → Zoom Best Fit` (`Ctrl Alt F`), and you should then see the same 211 ns run.

5. Make a waveform of the broken counter. Before you open it, sketch what you expect `tc` to do from 121 ns, while `q` sits at `0xFF` with `en` low.

```
make -f sim.mk sim TB=test_04_waves WAVES=1 RTL=rtl/counter_broken.sv \
     SIM_BUILD=build/bw COCOTB_RESULTS_FILE=build/bw.xml
surfer build/bw/counter.fst
```

The test passes, and its log is identical line for line to the good run's. Add the same six signals.

You should see `tc` go high at 121 ns, when `q` becomes `0xFF`, and stay high for four clocks, including the three where `en` is low and the counter is not moving. On the good counter it was one clock wide.

Check: you have seen `tc` one clock wide on the good counter and four clocks wide on the broken one.

Expected results

Check	Expected	
`make hello`	`TESTS=1 PASS=1`, `21.00` ns	exact
`make triggers`	`TESTS=6 PASS=6`, `1832.51` ns	exact
`make nothing`	no results table, exit 0, six `<skipped` in `build/nothing.xml`	exact
`make`, verdict	`PASSED` on `rtl/counter.sv`, `FAILED` on `rtl/counter_broken.sv`	exact
`make` wall clock	8–14 s	ballpark
`make sim` with `q + 8'd2`	`TESTS=6 PASS=3 FAIL=3`	exact
`make waves`, VCD size	`1098` bytes	exact

Finishing

You're done when your `test_norace` fails one clock early, your `q + 8'd2` counter fails 3 of 6 tests, and `make` on the restored `rtl/counter.sv` prints both halves of the verdict.

Submit: nothing for this lab. `make clean` removes `build/`, the waveform files and the caches; it does not remove `test_norace.py`, `view.sf` or a `test_mine.py` from Going further, so delete those yourself if you want a clean package.

Going further (optional)

To write your own variant of the checking tests, copy the file and run it through `sim.mk`, giving each run its own build directory:

```
cp test_03_checking.py test_mine.py
make -f sim.mk sim TB=test_mine SIM_BUILD=build/mine
make -f sim.mk sim TB=test_mine RTL=rtl/counter_broken.sv SIM_BUILD=build/mine-broken
```

You should see `TESTS=6 PASS=6` on the first and `TESTS=6 PASS=5 FAIL=1` on the second. The failure on the second run is the half of the two-sided verdict that shows your testbench can see the bug. (`make sim TB=...` won't work for this, because the package's `sim` target always runs `test_03_checking`.)

Next on the Course path: read Two simulators and SimProc in one page, then do Lab 02: The only test this chip has.

Answers

A1. Part 1: Your first test

Your message is the text after `AssertionError:`, and it also appears on the first line. The `assert ... + where ...` lines are pytest taking your `assert` apart and showing what each piece evaluated to. The message prints `00000000` because `dut.q.value` is a `LogicArray`, which can hold `X` and `Z`, so printing it gives the bit pattern. `int(dut.q.value)` would give `0`. Your first cocotb test has the full explanation.

A2. Part 2: Triggers

`Clock.start()` drives the clock high at t = 0. That first `0⟶1` transition is a rising edge and takes no time, so ten edges from t = 0 span nine periods.

A3. Part 2: Triggers

Synchronous. `rst_n low, no clock edge yet: q = 0x5A` shows `q` keeping its value with reset asserted, and `one rising edge later` shows it cleared on the first edge. If the first line had printed `0x00`, the reset would be asynchronous.

A4. Part 3: The 1.x traps

`units=`, the one that passed. Old code that still runs gives you no sign that it will break in a later release, and the only hint is a one-line deprecation warning that scrolls past.

A5. Part 4: A two-sided verdict

The `tc` assignment. The message goes on to say that `tc` should be `en & (q == 8'hFF)` and that a `tc` of 1 here means the `en &` is missing, which is exactly the difference between `rtl/counter.sv` and `rtl/counter_broken.sv`.

A6. Part 4: A two-sided verdict

None: `of those 600 clocks, 0 had q = 0xFF with en low`. The counter only reaches `0xFF` while it is running, and while it is running the broken `tc` and the correct one agree, so the most thorough test in the file never reaches the one situation that exposes the bug.

A7. Part 4: A two-sided verdict

The `+1` tells you what went wrong: one counted edge moved `q` one step further than it should have. The `t=31 ns` tells you when, which was on the first counted edge, so the error was there from the start rather than drifting in later.

III 3. SimProc

14 Two simulators 67
14 The same six tests, twice 67
14 The build cost 67
14 Where the crossover is 68
14 Lint and X 68
14 Which one to use 69
14 `-Wno-fatal` 69
14 Identical simulated time 70
14 Check yourself 70

15 SimProc in one page 71
15 The architectural state 71
15 One byte per instruction 71
15 The FSM and its timing 72
15 The memory 73
15 The chip from outside 73
15 Where the pieces live 74
15 Five facts that will cost you 75
15 Check yourself 75

16 Lab 02: The only test this chip has . . 77
16 Before you start 77
16 Part 1: The shipped test 77
16 Part 2: Make the chip do something 78
16 Part 3: The example testbench 79
16 Part 4: The golden model on its own 80
16 Part 5: Two simulators 80
16 Part 6: A waveform 81
16 Part 7: Write a test that catches `B01` 81
16 Part 8: The scoreboard 82
16 Expected results 83
16 Finishing 83
16 Going further (optional) 83
16 Answers 83

14. Two simulators

This page answers: *why does this course ship two simulators, and which one should I run?*

The short answer is that you should run both, for different reasons. Every DD104 lab runs on Icarus Verilog unless you say otherwise, and every lab also accepts `SIM=verilator`. The two tools are good at different things, and by the end of this page you should be able to say which one to reach for in a given situation.

People often say "Verilator is the fast one", which is true as far as it goes, but the measurements below give you a more useful picture of when it's faster and when it isn't. We took all of them in `labs/common`, the shared harness directory, and you'll repeat the important ones in Lab 02, Part 5.

The same six tests, twice

`make sim SIM=icarus` and `make sim SIM=verilator` in `labs/common` both run `test_example.py` against the golden core, and both pass. Here is the last line of each results table:

```
icarus     ** TESTS=6 PASS=6 FAIL=0 SKIP=0     512010.01     4.38     116812.47 **
verilator  ** TESTS=6 PASS=6 FAIL=0 SKIP=0     512010.01     1.92     267302.99 **
```

The three numbers after the test counts each measure something different. The third is roughly the first divided by the second, so the first two are the ones that matter here.

The first, `512010.01`, is simulated time in nanoseconds, and it is identical on both tools, to the hundredth of a nanosecond, on every run. The design is the same state machine whichever tool runs it, so a cycle count you measure on one simulator holds on the other, which is why the course quotes timing numbers without saying which tool produced them.

The second, `4.38` against `1.92`, is how many seconds each simulator spent simulating, so Verilator did the same work in 44 % of the time. Over three runs each, Icarus measured 4.38 / 4.57 / 4.71 s and Verilator 1.90 / 1.92 / 1.97 s, which is a steady ratio of 2.3×.

The number that decides which tool to use, though, is wall-clock time. We measured it by putting `time` around the whole `make`, starting from a clean directory:

	Wall clock	Of which simulating	The rest is
Icarus	6.8 / 5.8 / 5.9 s	4.4–4.7 s	Starting `vvp`, loading Python
Verilator	16.3 / 16.4 / 16.0 s	1.9–2.0 s	Building a C++ program

So Verilator simulates 2.3× faster and still finishes about ten seconds later. The reason is that Verilator is really a compiler, not a simulator in the sense Icarus is. It reads your SystemVerilog and writes out C++, and `g++` has to turn that C++ into a binary before the first clock edge can happen. Icarus just reads your SystemVerilog into a bytecode file and starts running it.

> "Compiled is faster than interpreted" is true per unit of work, but it tells you nothing about a job so small that the compile time dominates. You already know this about C++ against Python for a ten-line script, and it's the same trade here, with simulated time as the unit of work.

The build cost

The Verilator step itself takes almost no time:

```
- Verilator: Walltime 0.035 s (elab=0.003, cvt=0.022, bld=0.000)
```

That's 35 milliseconds to read 1,178 lines of SystemVerilog and write six C++ files. Nearly all the rest of the build time is `g++`, and you can see this in the statistics from `ccache`, which caches `g++` output. Here they are for two Verilator builds in the same container, with `make clean` between them:

```
Hits:               0 /   6 ( 0.00%)      <- first build: 16.04 s
Hits:               6 /  12 (50.00%)      <- second build: 3.60 s
```

The first build compiled six C++ files with six cache misses and took sixteen seconds. The second found all six in the cache and took three and a half seconds. The only thing that differed between them was `g++`, and you pay that cost once per container session rather than once per run. The cache lives inside the container, so it survives `make clean` but is lost when the container stops.

A more accurate version of "Verilator is slow to start", then, is that the first build in a fresh container costs about fifteen seconds of `g++`. If you start the workbench once in the morning and work inside it all day, you pay that once. If you start a new container for every command, you pay it every time. Tool comparisons rarely mention this, and it changes which tool comes out ahead.

Where the crossover is

You're trading a fixed build cost against a cost per nanosecond of simulation, so which tool wins depends on how much simulated time is in the run. DD104 ships a deliberately long testbench, `test_values_only.py`, with 1,889,970 ns of simulated time, which is 3.7× as much as the example. Here is the wall-clock time we measured for both:

Simulated time in the run	Icarus	Verilator, cold cache	Verilator, warm cache
`test_example`: 512,010 ns	5.8–6.8 s	16.0–16.4 s	3.2–3.6 s
`test_values_only`: 1,889,970 ns	22.5 s	22.4 s	8.5 s

At half a millisecond of simulated time, Icarus wins by ten seconds. At two milliseconds the two tie on a cold cache, and Verilator is 2.6× faster on a warm one. On this design, then, the crossover is just under 2 ms of simulated time, and it moves lower every time you rerun in the same container, because more of the build is already cached.

`make check-all` builds and runs fourteen separate cores, which is the worst case for Verilator, since each core is a separate build:

```
Icarus     77.18 s      10 of 13 caught
Verilator  84.94 s      10 of 13 caught
```

Even with fourteen full C++ builds, Verilator comes within 10 % of Icarus, because `ccache` catches most of the files that don't change from one core to the next. More importantly, the two tools reach the same verdict: each catches 10 of the 13.

Lint and X

Verilator can also lint your code, that is, check it for likely mistakes without simulating it. Run with `--lint-only` on the three SimProc source files, it prints twelve warnings in 0.077 seconds, ending with:

```
%Error: Exiting due to 12 warning(s)
```

Five of the twelve say `Bit extraction`, and four of those five are the same finding, reported at four different lines of the same module:

```
%Warning-WIDTHTRUNC: rtl/simproc_system.sv:848:16: Bit extraction of array[63:0]
requires
                     6 bit index, not 8 bits.
```

`dp_ram` has 64 entries but the address bus is 8 bits wide, so addresses 64–255 are off the end of the array. Verilator found that just by reading the file, before simulating anything. Icarus compiles the same file

without mentioning it; it prints twenty-eight `sorry:` notes about something else and exits with status 0 (see Output that is not an error).

It's worth seeing what each tool does with that defect at run time. `labs/common/test_two_simulators.py` writes `0x5A` to address 36, then reads address 36 and address 100 back out through the chip's UART and prints the raw bits that came off the pin.

Here is what Icarus prints:

```
READ  36  (in range)      -> 01011010
READ 100  (off the end)   -> XXXXXXXX
```

And here is Verilator:

```
READ  36  (in range)      -> 01011010
READ 100  (off the end)   -> 01011010
```

Both runs pass, and both simulate exactly 60,520.00 ns, yet they disagree about what the chip did.

Icarus is a 4-state simulator, which means every bit can be `0`, `1`, `x` or `z`. Reading past the end of the array gives `x`, the `x` propagates through the read port and the UART, and eight `X` characters come out of the chip. That's an obvious and correct result, as long as your stimulus happens to read address 100.

Verilator is a 2-state simulator, so every bit is either 0 or 1. It truncated the address to the six bits the array has, `100 & 63 = 36` (the same as 100 − 64), and you got the byte at address 36, which is the same `0x5A` you wrote there. That answer looks plausible, and Verilator gives no sign that it's wrong.

Neither tool is broken; they're answering different questions.

This tells you when to trust a passing run. A design that reads uninitialised state, indexes past the end of an array or leaves a signal undriven will often work perfectly under Verilator and produce X under Icarus. If you run the same suite on both simulators, any disagreement between them is a finding, and it's one you can't get from a single passing run.

Which one to use

You are	Use	Because
Editing a test and rerunning it every ninety seconds	Icarus	There's no build, so it's six seconds from saving the file to a result
Running a long suite, or the same design many times in one session	Verilator	It's 2.3× faster on simulated time, and the build is already cached
About to trust a passing run	Both	Any disagreement between them is a finding
Looking for width, truncation and case-coverage mistakes	`verilator --lint-only`	It takes 0.077 s and needs no testbench
Chasing an `X` on a pin	Icarus	Verilator is 2-state, so it has no X to show you
In a CI job that runs everything once	Either	It's 77 s against 85 s here, so choose on lint rather than speed

Icarus is the default in `sim.mk` only because of the first row of that table. That doesn't make it the better simulator; it has the faster edit-and-run loop, and that loop is where you'll spend most of this course.

`-Wno-fatal`

`sim.mk` passes `-Wno-fatal` when `SIM=verilator`. Without it, Verilator refuses to build this design:

```
%Error: Exiting due to 12 warning(s)
```

Those are the same twelve lint warnings. Verilator treats warnings as fatal by default, which is a reasonable policy for code you own and an impossible one for code you've been handed to test, because you can't fix

the warnings without changing the design under test. So don't read `-Wno-fatal` as "the warnings do not matter". Read it as "this design does not lint clean, and that is a finding", and you'll look at that finding in Lab 02.

Identical simulated time

cocotb writes a JUnit XML file with one `sim_time_ns` value per test. If you run both simulators, extract those values and diff them, you'll find no difference at all. These are the six values for `test_example`:

```
sim_time_ns="13680.0"
sim_time_ns="11220.0"
sim_time_ns="33370.0"
sim_time_ns="99580.0"
sim_time_ns="33820.0"
sim_time_ns="320340.0"
```

You should do the same with your own testbench. If a simulated-time number differs, stop and find out why. It means your testbench depends on something the two tools model differently, which is almost always an X, an undriven signal, or a race between two things scheduled in the same delta, and so one of the two passing results isn't telling you the truth about the design.

Check yourself

1. You will run a 10 µs test fifty times this afternoon in one container. Which simulator, and why?
2. A test passes on both simulators but its `sim_time_ns` differs between them. What does that suggest?
3. Why can Verilator never show you an `X` on a pin?

Next: SimProc in one page.

15. SimProc in one page

This page answers: *what is this processor, exactly? I need to know what it should do before I can say it does not.*

`rtl/simproc_system.sv` is 1,178 lines, and you don't need to read it. The machine is small enough to hold in your head, and everything DD104 asks you to do is a question about the machine rather than the file, so this page describes the whole machine instead.

The architectural state

```
r0  r1  r2  r3        four 8-bit registers              32 bits
PC                    one 8-bit program counter          8 bits
N   Z                 two flags                          2 bits
mem[0 .. 63]          sixty-four bytes of RAM          512 bits
                                                       ────────
                                                       554 bits
```

Those 554 bits are the processor's entire architectural state, so if two SimProcs agree on them after every instruction, no program can tell them apart. That is the claim the golden model is built to check, and Lab 05 shows you that the claim is not enough.

> If you have met RISC-V or MIPS, you know that register zero reads as zero and ignores writes. `r0` here is an ordinary register. It holds whatever you last put in it, and `ORI`, the only instruction that can introduce a constant, writes only `r0`. Assuming otherwise will cost you an evening; Lab 03 has you check it in two lines of test.

Three things the machine does not have shape every program you write:

- No carry flag and no overflow flag. Eight-bit arithmetic wraps around without telling you, so `200 + 60` gives `4`.
- No compare instruction. You compare with `SUB`, which writes the difference into `rA`, so comparing two values destroys one of them.
- No `NOT`, no `HALT`, no stack, no call and no interrupt. You invert a register with `NAND rX, rX`, and a finished program parks itself in `JMP -1`.

One byte per instruction

Every instruction is one byte, and three field layouts share the encoding space:

```
register form     A A B B o o o o      rA = instr[7:6], rB = instr[5:4]
branch form       d d d d o o o o      d = 4-bit SIGNED displacement
3-LSB forms       A A D S S 0 1 1      SHIFT: D = direction, SS = amount
                  i i i i i 1 1 1      ORI:   5-bit immediate
```

In the register form, `rA` is both a source and the destination: `ADD(1, 2)` means `r1 = r1 + r2`, and there are no three-operand instructions.

SimProc has eleven instructions. The full encoding table, with what each does and what it costs, is The SimProc instruction set; keep it open in another tab. The short version:

Arithmetic and logic	`ADD SUB NAND SHL SHR`: all write `rA`, all set `N` and `Z`
The constant	`ORI imm5`: `r0 <- r0 | imm`, sets flags

Memory	`LOAD rA, rB` reads `mem[rB]`; `STORE rA, rB` writes `mem[rB]`
Control	`JMP` always; `BZ BNZ BPZ` on the flags. None of them touch the flags
Nothing	`0xC 0xD 0xE` are unimplemented and behave like a slow no-op

The FSM and its timing

The core is a six-state machine: `IDLE`, then `CYCLE_1` to `CYCLE_5`. It is not pipelined, has no cache and speculates nothing, so a program's cycle count is the sum of its instructions' cycle counts and you can add it up by hand before you run it.

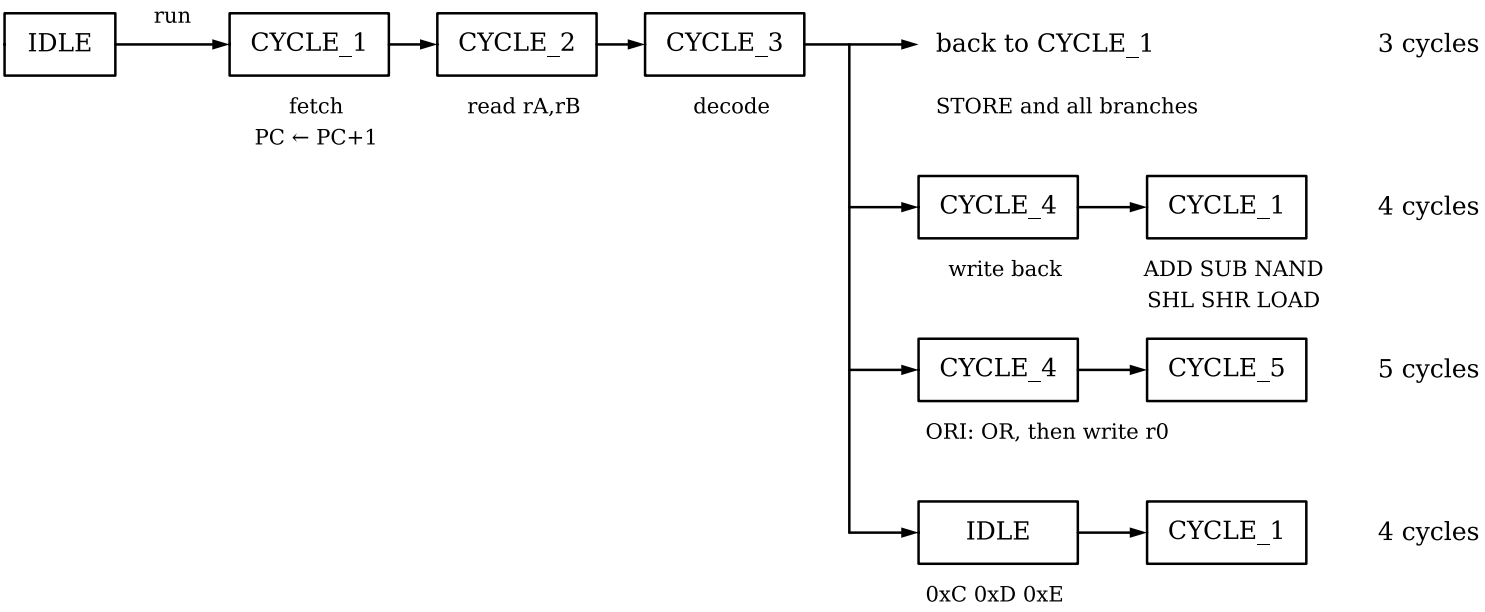


Figure 15.1: The core's states, and the four paths an instruction can take out of `CYCLE_3`.

Two details of that diagram matter more than they look.

`CYCLE_2` reads the register file for every instruction, using `instr[7:6]` and `instr[5:4]` as the two register numbers, before anything has decided what the instruction is. On a branch those bits are half of the displacement, but the read happens anyway and its result is thrown away. So a bug in the register-select path can show up on instructions that don't obviously use registers.

An unimplemented opcode leaves through `IDLE`. It matches no clause in `CYCLE_3`, so the FSM takes its default exit, spends one cycle in `IDLE`, and is sent back to `CYCLE_1` by `run`, which is still asserted. We measured it on a program of four filler bytes (`0x0C`):

Clock cycles	16: four each, not three
`done` pulses	0
Cycles with `halt` high	4: one per instruction

So the byte the harness fills unwritten memory with is also the one byte that makes `halt` blink in the middle of a running program. If you write the assertion "`halt` must stay low while `run` is asserted", as Lab 05 asks you to, it will fire on any program that runs off its own end into filler. When that happens, your test is working correctly and the machine is behaving as designed, and it's better to learn that distinction here than at midnight.

Cycle counts

Instruction	Cycles
`ORI`	5
`ADD SUB NAND SHL SHR LOAD`	4
`STORE JMP BZ BNZ BPZ`	3
`0xC 0xD 0xE` (unimplemented)	4, no `done`, one cycle of `halt`

The golden model can run a program on its own, with no simulator underneath it, and its demo program prints this:

```
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)
```

You'll run it in Lab 02, Part 4. That 109 is exact, and you can derive it rather than take it on trust. Seven setup instructions cost `4+5+4+4+4+5+4 = 30`; a three-instruction loop body costs `4+4+3 = 11` and runs five times, for `55`; the finish costs `4+5+4+4+4+3 = 24`. $30 + 55 + 24 = 109$. The simulated hardware agrees to the cycle, and `test_example.t05_a_loop_and_its_cycle_count` asserts exactly this. If your hardware ever reports 114, the five extra cycles over five iterations point to one extra cycle per iteration, which is a much better starting point for debugging than "it is slow".

The memory

`dp_ram` is 64 bytes, dual-ported, with combinational reads and no reset.

You'll rely on the missing reset constantly. Resetting the core clears the registers, the PC and the flags, but leaves your program and its results in memory where they were, and that is what lets a test load a program, reset, and run from a known state.

The size is a trap, because the address bus is 8 bits wide but the array has only 64 entries:

- Reads above 63 return `X`. A UART `READ` of address `0x45` puts eight undefined bits on the chip's TX pin, and in cocotb 2.x, calling `int()` on that value raises an exception instead of guessing. That is the right behaviour, but the first time you see it, it will look like a harness bug.
- Writes above 63 are dropped without any warning, and they don't wrap around into low memory. When we wrote `0x99` to `0x45`, `mem[0x05]` was left untouched.
- Verilator's linter reports the mismatch just from reading the file (`Bit extraction of array[63:0] requires 6 bit index, not 8 bits`, four times), while Icarus shows it only at run time, as `X` on a pin. Both are right, but only Verilator tells you before you run anything. See Two simulators.

The harness refuses out-of-range addresses for you, with a message explaining all of the above, so you will normally meet this through a program whose pointer register went somewhere you did not intend.

The chip from outside

SimProc is wrapped in a TinyTapeout pin frame. Seen from outside the package, its whole interface is one serial input pin, one serial output pin, two status pins and a baud divider.

Pin	Direction	What it is
`ui_in[0]`	In	`uart_rx`: everything goes in here
`uo_out[0]`	Out	`uart_tx`: everything comes out here
`uo_out[1]`	Out	`halt`: high while the core sits in `IDLE`
`uo_out[2]`	Out	`done`: one-cycle pulse as an instruction retires

Pin	Direction	What it is
`uo_out[7:3]`	Out	Tied to zero
`uio_in[7:0]`	In	The baud divider: `clk_per_bit = uio_in × 4`
`uio_out`, `uio_oe`	Out	Tied to zero; every bidirectional pin is an input
`rst_n`	In	Reset, active low
`ena`	In	Always 1 when powered; ignore it

The chip's only test checks `uo_out`. `uo_out == 3` is `0b0000_0011`: `uart_tx = 1` (an idle serial line sits high), `halt = 1` (the core is in `IDLE`) and `done = 0`. So checking for that value only asserts that the processor is not running, and Lab 02 is where you watch all thirteen broken cores pass it.

The debug protocol

The only way in from outside is a 3-byte UART command, always three bytes (`cmd addr data`) even when `addr` and `data` are ignored. The serial format is 8N1, with no parity.

Byte	Command	What it does
`0xA3`	`READ`	Replies with `mem[addr]`
`0x5C`	`WRITE`	`mem[addr] = data`
`0xF0`	`PING`	Replies `0xAA`
`0x3F`	`RUN`	Start free-running
`0xC7`	`HALT`	Stop
`0x6D`	`STEP`	Execute exactly one instruction
`0x99`	`SET_PC`	`PC = addr`, only while the core is in `IDLE`
`0xB2`	`GET_PC`	Replies with the PC

Serial is slow, and we measured how slow. All of these numbers are at the default `clk_per_bit = 32`:

	Clock cycles
One 3-byte command	1,104
A command with a reply (`PING`)	1,353
Loading a 64-byte program over the wire	70,656
The same program through the harness's memory backdoor	2
The head start `RUN` gets: the core is executing this long before the command finishes going out	57

Those numbers are why the harness has a backdoor into the RAM array, and why everything in `simproc.py` that counts something starts its monitor before sending `RUN`. They are also why you cannot stop this processor after exactly N instructions: `HALT` takes 1,104 cycles to arrive and the core runs the whole time. Programs park themselves in a `JMP -1` instead, the way real bare-metal test code does.

Where the pieces live

If you do open the file, this is its shape, and these are the paths the harness reaches through:

```
tb
└── tt_um_ieeeuoftasic_simproc          the pin frame
    └── U1                              simproc_system      lines 1–326
        ├── RAM1.mem[0..63]             64 bytes            lines 827–859
        ├── UART1                       8N1, no parity      lines 863–1178
        └── CPU1                        the core            lines 330–739
            ├── curr_state              IDLE / CYCLE_1..CYCLE_5
            ├── n_flag_out, z_flag_out
            ├── RF1.rf[0..3]            the registers       lines 741–771
            ├── PC1.pc_out                                  lines 773–788
            └── ALU1                    6 ops, N and Z      lines 790–823
```

Three of those modules are 16, 31 and 34 lines, which is why Lab 03 can test them completely, including all 393,216 ALU vectors, before you reason about the processor as a whole.

`simproc.py` wraps every one of these paths, so you shouldn't need to type them yourself.

Five facts that will cost you

Everything above is background. These five facts are the ones most likely to turn into a wrong answer at 1 a.m.:

1. `ORI` only sets bits. Run `SUB rX, rX` first, or you get the OR of everything that register has ever held. This is the most common way to write a SimProc program that is wrong and looks right.
2. `ORI` always writes `r0`, whatever else the program is doing. It has no register field; all five spare bits are the immediate.
3. Branch displacements are measured from the next instruction, because the PC has already been incremented when the branch computes its target. `JMP(0)` falls through and `JMP(-1)` loops on itself. The range is `-8` to `+7`.
4. Only `ADD`, `SUB`, `NAND`, `SHIFT` and `ORI` set flags. `LOAD`, `STORE` and every branch leave `N` and `Z` exactly as they were, which lets you compare once and branch on the result twice. Relying on that is also an assumption, though, and assumptions get tested.
5. Shift amounts are two bits: 0, 1, 2 or 3. You cannot shift by 4 in one instruction, and the assembler refuses rather than silently truncating.

Each of those is a place where a wrong implementation and a right one agree on the inputs you would naturally pick, and that is the subject of Where bugs hide.

Check yourself

1. Add up the cycles for `SUB r0, r0 ; ORI 5 ; ADD r1, r0 ; STORE r1, r2`.
2. A branch at address 10 has displacement `-3`. Where does it go if taken?
3. Your test reads address 70 through the UART and gets an exception from `int()`. Is the harness broken?

Next: Lab 02: The only test this chip has.

16. 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.

IV 4. Unit tests

17 Flags as architectural state 87
17 A one-character bug 87
17 The exhaustive sweep 87
17 A value-only testbench 88
17 Four lines that catch it 89
17 Branching on a wrong flag 89
17 A flag that changes when it should not 90
17 What to put in your own suite 90
17 Check yourself 90

18 Lab 03: Three blocks before the processor 91
18 Before you start 91
18 Part 1: The program counter 91
18 Part 2: The register file 92
18 Part 3: All 393,216 ALU vectors 92
18 Part 4: Two bugs, and the count 93
18 Part 5: Value-only checks against `B03` 94
18 Part 6: Add the flags 94
18 Part 7: An undefined opcode 95
18 Expected results 95
18 Finishing 96
18 Answers 96

17. Flags as architectural state

This page answers: *I checked the answer and it was right. What else is there to check?*

The answer is the flags. SimProc has two: `N` is bit 7 of the last result, and `Z` is 1 when that result was zero. `ADD`, `SUB`, `NAND`, `SHIFT` and `ORI` write them, and `LOAD`, `STORE` and every branch leave them alone. Each flag is a single bit, but the flags are **architectural state**: they are as much a result of an instruction as the byte that lands in a register, and on this machine they are the only thing a branch can see.

> It's natural to assume that if the numbers are right, the machine is right. On this processor you can get every number right, in every register and every byte of memory, on every instruction, and still ship a chip that takes the wrong branch.

A one-character bug

`B03_zero_flag_low_nibble` changes one expression:

```
good:    assign Z = ~(|ALUout);
broken:  assign Z = ~(|ALUout[3:0]);
```

The zero detector now looks at the bottom four bits of the result instead of all eight, so the broken core says "that was zero" for 16, 32, 64, 128 and 240, and for every other non-zero byte whose low nibble happens to be zero.

Every value the broken core computes is still correct: `ALUout` is untouched, the register and memory get the right byte, the program counter is right, and so is `N`. The only thing wrong is one bit, and only for some results.

The exhaustive sweep

The ALU is 34 lines long, and its whole input space is 393,216 vectors, few enough that Icarus walks through all of them in about twenty seconds. That walk is what you do in Lab 03. With a sweep that checks values and flags in separate buckets, `B03` produces this (you'll get there in Lab 03, Part 6):

```
      ok  the golden ALU                    3 tests, all passed            19.4s
  caught  B03_zero_flag_low_nibble          3 tests, 1 FAILED              18.6s
       t02_every_vector_the_alu_can_be_given
           AssertionError: Every value is correct, and 25726 of 393216 vectors have
the wrong FLAGS
```

The assertion message says what this page is about: every value is correct, and 25,726 of the 393,216 vectors have the wrong flags.

`make check` truncates the message to fit its table, but the run itself prints all of it:

```
393216 vectors checked -- 0 value mismatches , 25726 flag mismatches
    {'ADD': 3840, 'SUB': 3840, 'OR': 255, 'NAND': 255, 'SHL': 14400, 'SHR': 3136}
  First one: ADD(0x00, 0x10) = 0x10 with N=0 Z=1, expected N=0 Z=0.
```

Before you open any RTL, you can read three things off this output.

First, the value count is zero for all six operations, so the adder adds, the subtractor subtracts and the shifter shifts.

Second, the flag count is non-zero for all six. A fault that affects every operation but no value can't be inside any one operation. It has to be in something all six share and none of them computes, and this module has exactly two such things: `N` and `Z`.

Third, the counts factor, and the factors describe the bug. Each of the six counts is exactly the number of vectors whose result is non-zero but has a zero low nibble, which is the only prediction the hypothesis "`Z` is wired to `ALUout[3:0]`" makes:

	Count	=	Why
ADD, SUB	3,840	15 × 256	15 results `0x10`…`0xF0`, each reachable from 256 operand pairs
OR	255	16 × 16 − 1	`a|b` ends in four zeros only if both operands do, minus the one case where the result really is zero
NAND	255	16 × 16 − 1	Mirror image: both operands must end in `0xF`
SHL	14,400	960 + 1,920 + 3,840 + 7,680	Shifts of 0, 1, 2 and 3; each larger shift creates more low zero bits
SHR	3,136	960 + 896 + 768 + 512	Shifting right removes low zeros instead, so the count shrinks
	25,726		

The column adds up to exactly 25,726. When a failure count factors into a sentence about the design, you have a hypothesis you can act on; when it doesn't, you have a number you still have to explain. The `SHL`/`SHR` asymmetry alone rules out every explanation except "something is looking at the bottom of the byte", because nothing else would care which way the bits moved.

The last line, `First one: ADD(0x00, 0x10) = 0x10 with N=0 Z=1`, gives you a three-line reproducer.

So from one number and its factorisation you've narrowed the bug down to one of two wires, without opening the RTL. That is what we mean by "the count is the diagnosis", and it's the difference between a bug report a designer can act on and one that says "the ALU is broken".

A value-only testbench

`labs/common/test_values_only.py` is a reasonable testbench, not a strawman. It has three tests: `ADD` and `SUB` over twelve operand pairs each (zero, one, the halves of the byte, the wrap); `NAND` over the same twelve plus twenty shifts; and a sixteen-instruction program checked for both its answer and its exact cycle count.

```
@cocotb.test()
async def v01_arithmetic_over_a_spread_of_operands(dut):
    """ADD and SUB, twelve operand pairs each, every result checked."""
    sp = SimProc(dut)
    await sp.start()

    for a, b in PAIRS:
        await sp.run(set_reg(1, a) + set_reg(2, b) + [ADD(1, 2)])
        assert sp.reg(1) == (a + b) & 0xFF, (
            f"{a} + {b}: r1 is {sp.reg(1)}, expected {(a + b) & 0xFF}.\n"
            f"  {sp.describe()}")
```

It runs for 1,889,970 ns of simulated time (3.7× as long as the example testbench) and takes twenty-two seconds of wall clock, and it's more thorough than most first attempts. Against `B03` it gives:

```
      ok  the golden core                    3 tests, all passed         21.3s
  MISSED  B03_zero_flag_low_nibble           3 tests, all passed         20.1s
-------------------------------------------------------------------------------
 VERDICT: FAIL — your testbench passes B03_zero_flag_low_nibble too.
```

```
A test that passes on a broken core is not testing that core.
That bug hides behind: every test that checks results and not flags
```

The testbench reports `MISSED`, and more luck or more vectors wouldn't have helped, because it cannot see the bug at all. Every value this processor computes is visible in a register or in memory, but `N` and `Z` show up in neither. A test that only reads registers and memory is testing a smaller machine than the one on the die, so you could run it for a week and it would still miss `B03`. You'll run it yourself in Lab 03, Part 5.

So the rule is: for every instruction that writes a flag, assert the flag, and for every instruction that must not write one, assert that it did not move. Treat it as a per-instruction rule, the same way you already check the result of every instruction.

Four lines that catch it

Here are the four lines that make the difference, from the fourth test in `test_example.py`:

```
    # r1 = 16, and make that the LAST flag-setting result.
    await sp.run(set_reg(1, 16))
    assert sp.reg(1) == 16, f"r1 is {sp.reg(1)}, expected 16"
    n, z = sp.flags()
    assert (n, z) == (0, 0), (
        f"after r1 = 16 the flags are N={n} Z={z}, expected N=0 Z=0.\n"
        f"  16 is 0b00010000: not negative, and not zero. If Z is 1 here, "
        f"something is looking at only part of the byte.")
```

Against `B03`:

```
caught  B03_zero_flag_low_nibble          6 tests, 2 FAILED           2.7s
      t04_flags_are_state
          AssertionError: after r1 = 16 the flags are N=0 Z=1, expected N=0 Z=0.
```

What makes this test work is the value 16. Its low four bits are zero but the byte as a whole isn't, so a full-byte zero detector and a nibble-wide one give different answers for it. 32, 64, 128 and 240 would work too, but 5, 255 and 0 would not, because both detectors agree on them. If you pick the values for a flag test without thinking about the flag, the bug tends to agree with the correct design on every one of them.

Branching on a wrong flag

So far the bug is a wrong bit in a status register that nobody reads. What makes it a chip you can't ship is a branch that reads it, as in the last four lines of `R07` in `test_reference.py`:

```
    # And the consequence, which is what actually bites: a branch on a bad flag.
    prog = set_reg(1, 16) + [SUB(0, 0), ADD(1, 3), BZ(1), ORI(1), SUB(3, 3)]
    await sp.run(prog)
    assert sp.reg(0) == 1, (
        f"r0 is {sp.reg(0)}, expected 1: the BZ after r1 = 16 was TAKEN, so the "
        f"core thinks 16 is zero.")
```

`r1` holds 16. `ADD r1, r3` adds zero, so `r1` is still 16 and the flags now describe 16. Then `BZ +1` asks "was that zero?" and skips the next instruction if it was.

On the good core `Z` is 0, the branch is not taken, `ORI 1` runs, and `r0` is 1. On the broken core `Z` is 1, the branch is taken, `ORI 1` is skipped, and `r0` is 0.

Two instructions later the machine is running different code, even though no register and no byte of memory was ever wrong. The program went the other way at the branch, so everything after it is wrong, and none of that looks like a flag bug any more.

A flag set late or set wrong is invisible until something branches on it. The gap between the defect and the symptom is however many instructions run between the compare and the branch, and in real firmware that can be a hundred.

A flag that changes when it should not

The mirror-image bug is `B09_branch_clobbers_flags`: one added line that makes a branch write the flags with the result of its own PC arithmetic.

SimProc's branches don't set flags, and that's deliberate. The ISA note gives the reason in one sentence: *"you can compare once and branch on it twice."* That is a real idiom (`BZ here; BNZ there`), and it's the only stimulus that can see this bug, because with one branch per compare everything still works.

This is the reference suite against `B09`:

```
  caught  B09_branch_clobbers_flags          11 tests, 2 FAILED             18.1s
        R05_flags_survive_a_branch
            AssertionError: r0 is 8, expected 1.
        R07_zero_and_negative_flags
            AssertionError: after r1 = 128 the N flag is 0; N is bit 7 of the result
```

The program in `R05` is eight lines and reads like a proof:

```
        SUB   r1, r1        # r1 = 0  ->  Z = 1, N = 0
        BZ    taken         # taken
        ORI   16            # skipped
taken:  BNZ   wrong         # must NOT be taken: Z is still 1
        ORI   1             # r0 = 1, the proof we got here
        JMP   done
wrong:  ORI   8             # r0 = 8 if the flags were clobbered
done:   SUB   r3, r3
```

`r0 == 1` means the flags survived the first branch, and `r0 == 8` means they did not, so a single register with two possible values tells you which of the two things happened. Write your own assertions the same way, so that a wrong answer reads as a sentence and not just as a number.

What to put in your own suite

Here are four rules, each of which costs about one line of code per instruction:

1. After every flag-setting instruction, assert `N` and `Z`. `sp.flags()` returns the pair.
2. After every instruction that must leave the flags alone (`LOAD`, `STORE`, all four branches), assert they did not move: read them before and after, and compare.
3. Pick values that tell the two implementations apart. For `Z`: something non-zero whose low nibble is zero (16, 32, 64, 128, 240). For `N`: something ≥ 128, and something just below it. A value like `5` tells you nothing about either flag.
4. Then branch on the flag and check where you landed. A flag assertion catches the defect and a branch catches its consequence, and the consequence is what turns up in silicon.

`set_reg()` clobbers `N` and `Z`, because it is built from `SUB`, `ORI` and `SHIFT`, all of which set flags. So set your registers up before the compare, never between the compare and the branch. The harness docstring warns about this too, and it's the most common way for a flag test to end up testing `set_reg` instead of the design.

Check yourself

1. Which of these values would catch `B03`'s nibble-wide zero detector: 0, 8, 48, 255?
2. A suite reads every register after every instruction. Why can it still miss `B09`?
3. You set up `r1` with `set_reg` between a `SUB` and a `BZ`. What is the `BZ` now testing?

Next: Lab 03: Three blocks before the processor.

18. 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`	

Check	Expected	
`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.

V

5. Bug hunting

19 The golden model 101
19 What a golden model is 101
19 Using it in four lines 101
19 What a model gives you that a hand-written expectation can't 102
19 What a model cannot see 103
19 When the model is the one that's wrong 103
19 When the model and the design disagree 104
19 Check yourself 104

20 Where bugs hide 105
20 The 99.2 % case 105
20 Disagreement sets 105
20 Six places a disagreement set hides 106
20 The evidence 108
20 Reading the file 108
20 The question list 109
20 Check yourself 109

21 Directed and random tests 111
21 Directed tests 111
21 Random tests 111
21 What the generator reaches 111
21 Why `R10` catches `B05` 113
21 The gap 113
21 Making the generator better 114
21 Check yourself 114

22 Lab 04: Your first bug hunt 115
22 Before you start 115
22 Part 1: Take delivery 115
22 Part 2: Start from what you have 116
22 Part 3: Narrow it down 116
22 Part 4: Let the model find the first disagreement 117
22 Part 5: The bugs one test catches 118
22 Part 6: What random stimulus reaches 118
22 Part 7: Fix it and get the third row 119
22 Part 8: Do it again (optional) 119
22 Expected results 120
22 Finishing 120
22 Answers 120

19. The golden model

This page answers: *the input space is too big to walk. What do I compare against instead?*

In Lab 03 you drove all 393,216 legal vectors through SimProc's ALU in about twenty seconds, and afterwards there was no unexplored corner left for a bug. That is the best position verification ever puts you in, and you should take it whenever a block is small enough.

Now count the processor: 554 bits of architectural state, and a program is 64 bytes, so the space of programs alone is 256^{64}, and testing all of it is out of the question.

So instead of checking every input, you check every instruction as it runs, by comparing the design against a second opinion.

What a golden model is

A **golden model**, or reference model, is a second implementation of the same specification, written in a language where you can see what it does. You run the design and the model on the same stimulus and compare them after every step. When they disagree, one of them is wrong.

DD104's golden model is `labs/common/golden.py`, about 120 lines of plain Python with no imports and no simulator behind it. You ran it on its own in Lab 02, Part 4, and it printed this:

```
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)
```

That's a whole SimProc program running with no Verilog involved. It runs in no measurable time, and because it's plain Python you can read it and put a `print` in it.

The interface is deliberately small:

```
def snapshot(self):
    """(regs, pc, N, Z) — the same shape `SimProc.snapshot()` returns, so
    the two can be compared with `==` and nothing else."""
    return (tuple(self.regs), self.pc, self.n, self.z)
```

Because both sides return the same tuple, comparing them takes a single `==`. You don't check field by field or keep a list of things to remember to look at, so the comparison also checks the registers you weren't thinking about.

> The rule that `spec()` followed in Lab 03 applies here too: write the model from the ISA, not from the Verilog. A model written by reading the RTL agrees with the design about everything, including its bugs, so it passes forever and proves nothing. Lab 03 called this the most common way a verification effort wastes six months, and it's easy to do by accident, so it's worth writing at the top of the file.
>
> `golden.py` is written from the instruction set. Where the ISA was ambiguous, somebody had to make a decision, and the end of this page describes one.

Using it in four lines

```
await sp.load(prog)
model = Golden(prog + [PARK])
steps, problem = await trace(sp, 29, model=model)
assert problem is None, f"{problem}\n\n{listing(prog)}"
```

`trace()` single-steps the real design, steps the model alongside it, and compares the full snapshot after every instruction. The program here is the one in `test_example.t06`: thirty instructions, of which 29 execute, because a taken branch skips one. So those four lines check all four registers, the PC and both flags, twenty-nine times over, and that includes everything you didn't think to check.

`trace()` doesn't look at memory, so the shipped test follows those four lines with one call to `compare_memory(model, sp)`, which compares all 64 bytes. When both pass, this one line is all it reports:

```
29 instructions, every register / PC / flag / byte of memory agreed with golden.py
```

When it fails, you get a message like this one, from `test_example.t06` run on the core whose `SUB` computes `B - A`:

```
AssertionError: instruction 19, at address 19 (0x66 = SUB   r1, r2), left the machine
in a different state:
        model: r=(48, 195, 29, 48) PC=20 N=1 Z=0
          DUT: r=(48, 61, 29, 48) PC=20 N=0 Z=0

the program:
  00: 06   SUB   r0, r0
  01: C7   ORI   24
  ...
  12: 64   ADD   r1, r2
  13: 66   SUB   r1, r2
```

You didn't write any of that message yourself, and every part of it is useful:

- which instruction failed: number 19, counting from 0, rather than "somewhere in the program";
- where that instruction lives: address 19;
- what it was: `0x66`, already disassembled to `SUB r1, r2`, so you don't have to decode the hex;
- the state of the whole machine on both sides. Here $195 + 61 = 256$, which is the signature of a negated result, so you know the operands went in backwards before you open anything;
- the flags as well: `N=1` against `N=0`, which the same comparison caught without any extra assertion.

What a model gives you that a hand-written expectation can't

There's nothing wrong with directed assertions. `assert sp.reg(1) == 5` is fine, and you should write plenty of them. A model, though, gives you three things that a hand-written expected value can't.

1. It checks the registers you were not looking at

A directed test asserts on the register that the instruction names, and says nothing about the other three. "The answer is correct" and "nothing else moved" are different claims, and a directed test only makes the first one.

On the same broken core, the directed test in the same file reports:

```
AssertionError: r1 is 199, expected 247.
```

That message is true and useful, and it points you at `r1`, where the answer was supposed to go. But one bug in the catalogue writes the right value into the wrong register, and against that bug a directed assertion on the destination reports a symptom in a register that was never the problem. The model comparison catches it immediately, because a bystander register that changed fails `==` just as loudly as a destination register that didn't.

> Check the state that should not have changed. It costs nothing when you compare snapshots, and it is almost impossible to remember when you don't.

2. It names the first disagreement, not the last

Once a processor's state has diverged, everything downstream is wrong too. A test that checks state at the end of a program hands you the symptom (a wrong byte, forty instructions later, in a register nobody cared about) and leaves you to walk it back by hand.

`trace()` stops at instruction 19 of 29, while the machine's state is still only one instruction away from correct. Everything before that instruction is known to be good, so you start debugging at instruction 19 instead of walking back from the end.

3. It lets you run stimulus you can't work out by hand

You can't write down the expected final register file for thirty random instructions, but the model can, and that opens up a whole category of test. The limits of that kind of test are the subject of Directed and random tests.

What a model cannot see

A model checks only what it models, so write down what it leaves out next to the model itself, rather than discovering it at midnight. `golden.py` models architectural state: four registers, the PC, two flags and 64 bytes of RAM. It has no concept of:

`done`	The one-cycle retire pulse
`halt`	The pin that says the core is in `IDLE`
Cycle counts	It publishes them, but nothing compares them by default
The UART	Baud rate, framing, the debug protocol
`X`	A model has no undefined value

A core that computes every answer correctly but gets its handshake wrong passes a model comparison, because a correct answer and a correct handshake are different kinds of property. Two bugs in the catalogue are exactly that kind, and finding them is the work of Lab 05. Running more vectors won't help with them; you need a different kind of check.

When the model is the one that's wrong

Everyone gets caught out by a wrong model the first time, so here is a real example from this repository.

The unimplemented opcodes, `0xC`, `0xD` and `0xE`, take four cycles each, and `golden.py` used to say they took three. What they do is easy to describe: the PC advances, nothing else changes, and `done` never pulses. The model gave them three cycles, the same as every branch and `STORE`, which is what the RTL looks as though it should do. This is what we measured on four filler bytes:

Clock cycles	16, four for each byte
`done` pulses	0
Cycles with `halt` high	4

These opcodes match no clause in `CYCLE_3`, so the FSM takes its default exit to `IDLE`, spends a cycle there, and is sent back to `CYCLE_1` by `run`, which is still asserted. That adds up to four cycles, one of them with `halt` high in the middle of a running program.

There are three lessons in that:

1. The model was the one that was wrong. It's a second implementation written by a person, not an oracle. "The DUT disagrees with the model" is a statement about the pair of them, so the next question is always which one is wrong. Answer it from the specification or, when the specification is silent, by measuring the hardware and writing down what it does.
2. The model was still right about what it was mostly used for. Its architectural state was correct all along, and the number it got wrong was a timing figure that it also happened to publish. A model can be trustworthy in one dimension and wrong in another, so know exactly which dimensions yours covers.
3. The mistake was found by measuring a case that nobody tests. `0x0C` is filler, and no program runs it on purpose, so it's exactly the kind of "obviously nothing happens here" behaviour that never gets an assertion. That is the subject of Where bugs hide, applied here to the testbench instead of the design.

When the specification has not decided

SimProc's `ORI` has no register field (all five spare bits are the immediate), so the register it accumulates into is implicit in the hardware. For a year the RTL carried a comment saying it might be the other register.

We built and simulated four variants to settle it, and found that `r0` and `r1` are exactly isomorphic in this machine, meaning that nothing treats them differently: a register index never leaves the instruction word,

`r0` is not hardwired to zero, and no flag, addressing mode, reset value or debug command treats either one specially. The same program with `r0` and `r1` swapped produces the same memory and mirrored registers.

Since the hardware can't tell the two registers apart, no measurement can break the tie. Somebody has to decide, write the decision down and make the model match, and from then on the decision is part of the specification rather than an accident of the RTL. DD104 specifies `r0`.

Which register you pick is a matter of taste, but the two lines that implement the decision must change together. If you apply it to only one of them, you get "OR the immediate into one register, write the answer to a different one", which is an instruction nobody specified. That is one of the three tier-5 bugs, the hardest in the catalogue to think of. It isn't hard to catch, though: after a reset both candidate registers hold zero, so the first `ORI` of a program puts the right value into the machine, but in the wrong register, and any test that checks `r0` afterwards finds zero there. Lab 02's one-`ORI` test catches it, and so do seven of the eleven reference tests in Lab 06.

When the model and the design disagree

1. Read the failing instruction rather than the final state; `trace()` has already told you which instruction it is.
2. Decide which side is wrong from the specification, not by siding with whichever one you wrote.
3. If the specification is silent, measure the hardware, write down what it does, and say in the model that this records behaviour rather than a requirement.
4. If the specification is ambiguous rather than silent, the ambiguity is a hazard in its own right, so decide it, document it, and check that everything implementing the decision changed together.
5. Then ask what a model comparison could not have seen on this design, and write that test separately.

Compare snapshots, not values. If both sides can produce the same tuple, compare the tuples: it is less code than a hand-written expectation, it checks state you were not thinking about, and it fails with a location instead of a symptom.

Keep the model's blind spots written down next to it. A model that everyone trusts and nobody has bounded is worse than no model, because it turns "we did not check that" into "it passed".

Check yourself

1. A `trace()` failure says the model has `r2 = 10` and the DUT has `r2 = 10`, but `r3` differs. The instruction was `ADD r2, r1`. What kind of bug do you suspect?
2. Name two things a SimProc core can get wrong that `golden.py` will never notice.

Next: Where bugs hide.

20. Where bugs hide

This page answers: *the bugs I am hunting are one line each. Why can't I just read the file?*

The short answer is that these one-line bugs aren't typos. Each one is a line that looks correct, written in the right style and in the right place, doing something the surrounding code also does, and it gives a wrong answer only on inputs you wouldn't naturally write. This page gives the evidence for that, and then a taxonomy that turns it into a way of working.

> This page doesn't tell you where any bug is. DD104 ships thirteen broken cores, and Lab 04 deals you one without saying which. Every catalogue name below, along with what each one hides behind, is something `make bugs` already prints, and no line numbers or RTL appear here. The skill we want you to build is finding a bug in a design you can't diff against a known-good copy, and a diff would let you skip it.

The 99.2 % case

`B01_sub_operands_swapped` makes the ALU compute `B - A` instead of `A - B`. Drive every legal vector through the ALU, as Lab 03 does in about twenty seconds, and it is wrong on

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

That is 65,024 of the 65,536 `SUB` vectors, or 99.2 % of them, so this processor gets subtraction wrong ninety-nine times in a hundred.

Even so, the bug passed code review, and it passed the only test the chip ever shipped with, while DD104's own six-test example suite catches it four times over, with four separate assertions.

Those facts are consistent with each other, and seeing why is the single most useful idea in verification.

Disagreement sets

It helps to think of a bug as the set of inputs on which the wrong implementation and the right one give different answers, rather than as a broken line. We'll call that its disagreement set.

- If that set is everything, you find the bug just by looking. The design never works, the first person to run it notices, and it never reaches a shuttle.
- If that set is empty, there is no bug. The two implementations are the same machine, and which one you shipped is a matter of taste. (This does happen here: `ORI` could accumulate in `r0` or in `r1`, and the two machines are exactly isomorphic, so DD104 had to pick one, because no measurement could decide it.)
- Every bug that survives to silicon lies in between, and it survives because its disagreement set leaves out the inputs people actually write.

So for `B01` the question to ask is what the 512 survivors are, rather than how nobody noticed 99.2 % of subtractions coming out wrong.

`A - B == B - A` exactly when $2(A - B) \equiv 0 \pmod{256}$, that is, when $A \equiv B$ or $A - B \equiv 128$. Each condition has 256 solutions, 512 in total, and $65,536 - 512 = 65,024$, which is the number the sweep printed.

Now look at what is in that set. SimProc has four registers and no `MOV` or `CLR`, so `SUB rX, rX` is how you zero a register, and it appears in nearly every program on this machine. That instruction is one of the 512, which means the bug is invisible on the most common way anyone writes the instruction it breaks.

This is the general pattern: a bug survives review when its disagreement set misses the idioms, because the idioms are what a reviewer's eye is trained on.

Six places a disagreement set hides

Every bug in the DD104 catalogue has one of these six shapes, and so does most of what you will find in industry. Each one comes with a habit, and the habit is the part you'll carry to other designs.

1. Aliasing

Any instruction with two register fields has a case where both fields name the same register, and in that case it doesn't matter which field the hardware reads, because there is no wrong one to pick.

Catalogue	`B01`, `B02`, `B04`, `B07`
Measured	`B02` disagrees on 65,280 of 65,536 `NAND` vectors; the 256 survivors are exactly `A == B`
Why it survives	The machine has no `NOT`, so `NAND rX, rX` is how you invert, and `SUB rX, rX` is how you clear

`B07` has the best history in the course. The first `LOAD`/`STORE` test written for this core used `LOAD r1, r1`, which loads through a pointer into the register that held the pointer. That's a real idiom, and you use it when you've finished with the address. The test passed on a core that takes the address from the wrong register field, because when both fields name the same register there is no wrong one. Nobody noticed until the bug was injected on purpose and the suite stayed green.

> The habit: make every operand distinct on purpose, and check the register that should not have moved as well as the one that should.

2. Identity elements and reset

`x | 0 == x + 0`, and `0 | imm == 0 + imm`. Operators that differ in general agree on their identity elements, and a reset sets every register in the machine to zero, which is the identity element of almost everything.

Catalogue	`B10`, `B11`
Measured	`ORI 5 ; ORI 3` gives 8 on a core whose `ORI` adds; it should give 7. `ORI 8 ; ORI 3` gives 11 on both
Why it survives	The first `ORI` after reset is `0 | imm`, which is the right value under both bugs. Building a constant from disjoint bit groups, the natural way to assemble a value from a 5-bit immediate, never overlaps, so it never diverges

`B11` is just as hard to think of, but it is the easier of the two to catch. It writes the right value to the wrong register: after a reset `r0` and `r1` both hold zero, so the first `ORI` computes `0 | imm` correctly and puts it in `r1`, and the machine holds the right number under the wrong name. A test that checks `r0` after a single `ORI` finds zero there and fails, which is how Lab 02's `t03` catches it, and seven of the eleven reference tests fail on `B11` against two on `B10`. A test that only looks for the value somewhere in the registers misses it, but only until the second `ORI`, which reads an `r0` that the first one never wrote.

> The habit: never test an operation only from a reset state, and never test it only once. Once you do two of anything (two `ORI`s, two branches off one compare, two writes to the same register), the state the second one sees can no longer be right by coincidence.

3. Parameters left at their default

A parameter that is only ever driven at its default has been tested at one point of a range, and the silicon will be used across the whole of that range.

Catalogue	`B12`, and `B06` in spirit

Measured	`B12` scores 10/10 on the full systematic suite at `clk_per_bit = 32`, indistinguishable from a correct core, and 0/10 at `clk_per_bit = 8`
Why it survives	`clk_per_bit` is set by eight external pins, every lab uses one value, and at that value the receiver's drift still lands inside the bit

There is a second trap on top of the first. At `clk_per_bit = 16`, `PING` still answers `0xAA` over a link that can't carry data. `PING` sends `0xF0`, whose top bit is already 1, so a receiver that samples one bit-time late reads the stop bit in its place, which is also 1, and the command decodes perfectly. In other words, a link test can pass over a broken link.

`B06` has the same shape one level down. The shift-amount field is two bits wide, amounts 0 and 1 are correct, and 1 is the amount everybody writes.

> The habit: sweep the field instead of sampling it. Four shift amounts, sixteen branch displacements and five baud rates are all small enough to walk through, and testing every one of them is simply finishing the job.

4. One half of a signed range

Catalogue	`B05`
Measured	All eight forward displacements, `+0` to `+7`, land exactly right; every backward one lands 16 too far forward
Why it survives	"Skip the next instruction" is the first branch anybody writes, and it works. What breaks first is the first loop, because a loop is the only reason to branch backwards

Half the number line is correct, and it's the half you test first, so the bug survives every test that only branches forwards.

> The habit: when a field is signed, the sweep must cross zero.

5. Outputs that are not values

In this shape every value the design computes is correct, and what's wrong is some output that isn't a value.

Catalogue	`B03` (flags), `B08` (`done`), `B13` (`halt` and one cycle)
Measured	The exhaustive ALU sweep on `B03` reports 0 value mismatches and 25,726 flag mismatches out of 393,216
Why it survives	A testbench that reads registers and memory is testing a smaller machine than the one on the die

`B03` is the one to think about. A testbench that checks only values can't see it at all, however many vectors you add, so adding stimulus won't help; what it needs is a different kind of check. Flags as architectural state covers that check.

`B08` does the opposite of what you expect a bug to do. If your scoreboard triggers on `done` and one instruction stops pulsing it, the scoreboard doesn't report an error, because it was never woken up for that instruction. It compares everything it was asked to compare, finds no mismatches, and reports that truthfully, having made one comparison fewer than there were instructions. So a missing trigger doesn't make your test fail; it quietly stops your test from testing, which is worse, because a failure at least shows up.

The habit: list the design's outputs. Every one you never assert on is unverified, and the list is shorter than you think. For SimProc it is `done`, `halt`, `uart_tx` and the clock.

6. Using state twice

State that is correct when you use it once can be destroyed before you use it twice.

Catalogue	`B09`, and `B11` again
Measured	One branch after a compare works every time. A second branch off the same compare reads garbage
Why it survives	Most programs compare once and branch once, and this machine's flags are read combinationally in the same cycle the branch decides, so the first branch always sees the right value

In a machine whose `SUB` doubles as its compare, it matters how long the flags stay valid, and reading the source won't tell you. "The comparison is still valid" is an assumption, and an assumption needs a test.

The habit: use each piece of state twice, and assert that the second use sees what the first did.

The evidence

All thirteen broken cores pass the only test the chip has: you saw `13 of 13` in Lab 02, Part 1.

A real testbench isn't necessarily enough against a real bug either. `test_example.py` has six worked tests: arithmetic on distinct operands, flags, a loop with a hand-checked cycle count, and a full golden-model comparison over 29 instructions. Here is what it reports against `B10`:

```
      ok  the golden core                    6 tests, all passed          6.6s
  MISSED  B10_ori_adds_instead_of_ors        6 tests, all passed          5.7s
------------------------------------------------------------------------------
 VERDICT: FAIL — your testbench passes B10_ori_adds_instead_of_ors too.
   A test that passes on a broken core is not testing that core.
   That bug hides behind: ORing in bits that were not already set — ADD agrees
   Write a test whose stimulus does not do that.
```

That's six good tests, with twenty-nine instructions checked register by register against a reference model, and the bug is still `MISSED`, because none of those instructions happened to OR in a bit that was already set. You'll run this yourself in Lab 06, Part 2.

The test that catches it is two lines long, and it works because of its stimulus:

```
await sp.run([ORI(5), ORI(3), ORI(24)])
assert sp.reg(0) == 31          # 5 | 3 = 7, 7 | 24 = 31.  Added: 5 + 3 + 24 = 32.
```

Verification is mostly a matter of choosing inputs well, rather than of machinery, assertion count or framework. The reference suite that catches all thirteen bugs is eleven ordinary tests, each a short program, a run and a comparison, and they work because their stimulus steps outside the idioms: subtracting a number from something other than itself, shifting by more than one, branching backwards as well as forwards, and ORing in bits that were already set.

Reading the file

Reading the source tells you where to look, but it can't tell you that you're finished. On a design you didn't write it often can't even tell you which of two plausible lines is wrong, because the wrong line usually looks like a correct line from four clauses further down.

There are three reasons the "read it carefully" instinct fails on this class of bug:

1. You don't have the good version to diff against. That's normal in industry, and it's what Lab 04 is built around. With a diff you would find all thirteen in four seconds and learn nothing.
2. A one-line bug is written in the file's own style. One of the thirteen adds a line that looks like it belongs, in the right style and the right place, which shows that an injected bug doesn't have to be a typo.
3. The effort of reading grows with the size of the file, while the effort of testing grows with the size of the machine. The file is 1,178 lines, and the machine is 554 bits of state and eleven instructions. On your next project one of those numbers will grow and the other won't.

The question list

This is the same taxonomy, turned into a list you can work down while a core is in front of you. Each question either narrows the search or rules out a region of the design.

	Question	What a "yes" rules in	Shape
1	Which instruction classes are wrong: arithmetic, shift, load/store, branch?	The decode path for that class	
2	Is it wrong for all operands, or only some?	A data-dependent path: the ALU, or a width	
3	Does it depend on rA and rB being different?	A register-select field, read or write	1
4	Which register ends up wrong: the destination, a source, a bystander?	The write-back address	1
5	Does it survive a reset? Is the first one right and the second not?	Somewhere reset makes two things equal	2
6	Are the values right and something else wrong?	Flags, `done`, `halt`, the cycle count	5
7	Does a field you have only driven at one setting change it?	A shift amount, a displacement, a baud divider	3, 4

Work through them in order. Each answer takes one small program, and what separates a five-minute debug from a two-hour one is rarely insight; it's whether you spent the first ten minutes narrowing the search or reading the file.

> Before you write a test, name the input class it won't reach: which operand pattern, which field value, which half of which range, which output pin. Write the answer in the docstring. It's the most valuable line in most testbenches, and almost nobody writes it.

There is a corollary, and `make check` automates it: a test you have never seen fail is a claim you have never checked. Break the thing your test is supposed to watch, and see whether the test notices. That is how `B07` was found in this repository, by the person who had written the test that missed it.

Check yourself

1. A broken `ADD` gives the right answer whenever one operand is zero. Which of the six shapes is that?
2. A suite tests every branch with displacement `+1`. Which bug in the catalogue can it never see?
3. For `NAND`, how many of the 65,536 operand pairs survive a NAND-is-NOR bug, and which ones?

Next: Directed and random tests.

21. Directed and random tests

This page answers one question: *should I write tests by hand, or generate them?*

You need both, because neither is enough on its own:

> Directed tests find what you thought of. Random tests find what you did not think of, as long as it lies inside the distribution you chose. Neither kind finds what neither covers, and that third category is where the expensive bugs live.

DD104 ships the tools to measure all three of those, so most of this page is measurements.

Directed tests

A directed test drives stimulus that you chose and checks the result against an answer you worked out. Nine of the eleven tests in the reference suite are directed, and none of them is clever: each one is a short program, a run and a comparison. What makes them work is the choice of stimulus.

Directed testing has one way of failing, which is that the test shares your blind spot. Because you choose the stimulus, an input class you never thought of is missing twice over: from the tests, and from your own idea of what needs testing.

The history of `B07`, told on Where bugs hide, is an example. The first `LOAD`/`STORE` test written for this core used `LOAD r1, r1`, so it passed on a core that takes the address from the wrong register field, and nobody found out until the bug was injected on purpose and the suite stayed green. That page describes the habit that prevents this. What matters here is that the fix was another directed test, written on purpose once somebody had named the input class. You can't generate your way out of a blind spot you haven't named.

Random tests

`R10_random_programs` generates eight programs of twenty instructions each, runs each one on the chip, and compares the final register file with the golden model. The whole test is about forty lines, generator included, and you couldn't work out a single one of its expected answers by hand.

It does find bugs. When we inject the bugs from the catalogue one at a time, it catches these three:

Bug	Random programs that disagreed with the model
`B01_sub_operands_swapped`	4 of 8
`B10_ori_adds_instead_of_ors`	5 of 8
`B05_branch_offset_zero_extended`	7 of 8

That is three of the thirteen bugs, found by stimulus nobody wrote and checked against answers nobody had to work out by hand, which is a good return on forty lines. Random testing doesn't replace thinking, and it isn't worthless either. The rest of this page measures what it didn't do.

What the generator reaches

The generator is designed around **windows**. A window is the input condition a bug needs before it can be seen at all. `SUB rX, rY` with `rX != rY` is the window for a swapped subtract, because outside it a backwards `SUB` still gives the right answer.

`gap.py` runs the same generator that `R10` uses (it imports it from the same file, so there is only one copy) through `golden.py`, and counts how often each of nine windows is entered. No simulator is involved, so it takes about a second. You'll run it as `make gap` in Lab 04, Part 6:

```
the eight seeds R10 actually runs
------------------------------------------------------------------------
window                                                 entered   programs
  B01  SUB, operands differ by neither 0 nor 128             6      5 of 8
  B02  NAND with two different operands                     11      5 of 8
  B06  shift by 2 or 3                                      13      8 of 8
 *B05  branch with a negative displacement                   0      0 of 8
  B09  two branches off one comparison                       7      5 of 8
 *B07  any LOAD or STORE                                     0      0 of 8
  B03  non-zero result whose low nibble is zero              2      2 of 8
  B10  ORI that sets a bit r0 already had                   15      5 of 8
  B11  ORI executed while r0 is non-zero                    19      6 of 8
------------------------------------------------------------------------
8 programs, 160 instructions written, 2 of 9 windows never entered
```

Every number in that table is exact, because the generator is seeded and so writes the same 160 instructions on every machine.

The table shows three different situations, and the next three sections take them in turn.

Windows it enters constantly

`B06`'s window, a shift by 2 or 3, is entered thirteen times, in all 8 programs, and `B10`'s is entered fifteen times. Random stimulus covers these windows without anyone having thought about shift amounts at all. That is the case for random testing, and it is a strong one: it's cheap, and it doesn't depend on what you failed to imagine.

Windows it enters rarely, and one it enters and still misses

`B03`'s window is a flag-setting result that is non-zero but has a zero low nibble. The eight seeds enter it twice in 160 instructions. Over a thousand programs it is entered 338 times in 20,000 instructions, which is about 1.7 % of them.

Here is the reference suite run against `B03`:

```
      ok  the golden core                    11 tests, all passed        19.2s
  caught  B03_zero_flag_low_nibble           11 tests, 1 FAILED          16.2s
        R07_zero_and_negative_flags
            AssertionError: after r1 = 16 (0b00010000) the Z flag is 1.
```

One test of the eleven catches it, and it's `R07`, not the random one, even though the random test entered the window twice.

The reason it missed is the most useful point on this page. `R10` checks `model.snapshot()[0] != sp.regs()`, which compares the registers and nothing else. The flags are in the snapshot, but that comparison ignores them. A wrong `Z` can only reach a register if a later branch uses it and changes the control flow, and none of those eight programs did that.

> Coverage of your stimulus and coverage of your checks are two different questions, and you have to ask both. A generator can walk into a window and back out again, and if nothing in the checker can see the difference, the run exercised the design without testing it.

Windows it can never enter

The two rows marked `*` are still at zero after a thousand programs and twenty thousand instructions. That isn't bad luck, and more seeds won't change it, because the generator never writes those instructions at all. Its docstring says why:

```
def random_program(seed, n=20):
    """Arithmetic, shifts, ORI, and forward branches that cannot loop.

    No LOAD or STORE: a random register value is a random address, and half of
    them are outside the 64-byte RAM. No backward branches: a random loop is an
```

```
    infinite loop. Both of those are worth testing — with stimulus you designed
    rather than stimulus you rolled."""
```

Both exclusions are good engineering, and nobody made a mistake here. A random 8-bit register value is an address outside the 64-byte RAM more often than not, and a randomly placed backward branch is an infinite loop that turns your regression into a timeout.

Look at what those constraints cost, though. `B07` lives in `LOAD`, and `B05` lives in backward branches. The generator's necessary and sensible exclusions mark out exactly the directed tests you still have to write, and the reference suite's `R06` and `R04` are those two tests.

> A constrained random generator's constraints describe what it will never find, so write them down and treat them as a to-do list.

Why `R10` catches `B05`

`R10` does fail on `B05`, even though the table says the generator never emits a backward branch. Both of those are true, and it's worth two minutes to see why.

The harness's `load()` appends a parking loop, a single `JMP -1` (byte `0xF1`), after every program, because this machine has no `HALT` instruction and a program has to stop itself. On a core that zero-extends branch displacements instead of sign-extending them, that byte decodes as `JMP +15`, which doesn't loop at all.

So the program runs off its own end and through the filler, its 8-bit PC wraps around the 64-byte RAM, and it executes a second time on top of its own results. Here is the result on a three-instruction program:

	`r1`
Model	10
Chip	20

The random test catches `B05` because the harness added one backward branch to every program and the bug broke it. Nothing the generator chose had anything to do with the catch. That leaves you with two things to keep in mind:

1. This is real coverage and a real catch, so take the failure seriously; it is telling you the truth about the core.
2. You can't report it as evidence that your random testing covers backward branches, and you can't rely on it. If someone changes the parking idiom, the catch disappears and the bug stays.

Being able to say why a test failed, and not only that it failed, is what separates a verification result from a lucky one.

The gap

Putting the three together gives this:

	Finds	Misses	Costs
Directed	Exactly what you named	Everything you did not name, including whatever your model of the design gets wrong	One test per input class, written by someone who understands the machine
Random	Things nobody named, inside the distribution you chose	Anything the constraints exclude; anything your checker cannot observe	A generator, a model, and honesty about what it did not reach
The gap		What neither covers, which is where the bugs that reach silicon live	A written list

The only thing that closes the gap is a list of questions that you maintain by hand. That list is what functional coverage means. It isn't a tool, and Coverage as a question shows what one looks like. For this machine, the list starts like this:

The question	Why it is on the list
Have I executed every opcode, including the three unimplemented ones?	
For every instruction, `rA == rB` and `rA != rB`?	Four bugs live here
All four shift amounts, both directions?	Shifting by 1 is what everybody writes
All sixteen branch displacements, taken and not taken?	Half of the displacements are negative, and a bug can hide in that half
Two branches off one comparison?	Flags are supposed to survive a branch
An `ORI` whose immediate overlaps bits `r0` already has?	`OR` and `ADD` agree until the bits overlap
A second `ORI`, once `r0` is no longer zero?	After reset, several wrong machines look right
`done` counted, `halt` counted, cycle counts asserted?	Two bugs disturb nothing else
`clk_per_bit` swept, not sampled?	One bug does not exist at the default

You can answer every line today with a `for` loop and a set, without any tool. Writing the list is where the engineering goes; the counting is clerical work.

Making the generator better

If you want to improve `R10` rather than replace it, there are three separate improvements you can make, and each one fixes a different problem:

1. Constrain instead of excluding. `LOAD` and `STORE` are excluded because a random address is usually out of range. If you force the pointer register into `0..63` before the access, they can come back. That is what "constrained random" means, and it reaches the window where `B07` lives. Backward branches come back the same way, as a loop with a counter that the generator sets up.
2. Widen the checker rather than the stimulus. Compare the whole snapshot (registers, PC, `N` and `Z`) instead of only the register tuple. That is a one-line change, from `model.snapshot()[0] != sp.regs()` to `model.snapshot() != sp.snapshot()`, and it makes every flag window the generator already enters count.
3. Report what you reached. `gap.py` is the smallest possible coverage model: a list of named windows, a counter for each one, and a `*` on the ones that stayed cold. Without something like it, "we ran ten thousand random programs" tells people more about your CPU budget than about your design.

Do (2) first, because it's the cheapest, and it turns stimulus you already have into checking you don't have yet.

> After every random run, ask what it could not have reached, and write the answer next to the seed count. "Ten thousand programs, none of which contained a memory access" is a useful sentence. "Ten thousand programs" on its own isn't a result, for the same reason that a green tick is not a result.

Check yourself

1. A random generator never emits `STORE`. Which bugs in the catalogue can it never reach directly, and what directed test would you write?
2. `R10` entered `B03`'s window twice and still passed. What one change to its checker fixes that?

Next: Lab 04: Your first bug hunt.

22. 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.

VI 6. Handshakes

23 Handshakes and timing 125
23 One cycle late 125
23 The same idea on the wire 126
23 How long the receiver is busy 126
23 Sending without the gap 127
23 `PING` without the gap 127
23 Two numbers your testbench already depends on 128
23 Other bytes 128
23 Check yourself 129

24 Lab 05: Every answer is right 131
24 Before you start 131
24 Part 1: The architectural suite against both cores 131
24 Part 2: Count the pulses 132
24 Part 3: The shipped file against `B08` 132
24 Part 4: Assert the number 133
24 Part 5: Watch `halt` 133
24 Part 6: Assert the clock 134
24 Part 7: The UART receiver's busy time 134
24 Part 8: Hunt them properly 135
24 Expected results 135
24 Finishing 135
24 Answers 136

23. Handshakes and timing

This page answers: *every register, every flag and every byte of memory is correct. How can it still be broken?*

It can, because a block also has to produce its results at the agreed moment, and a register dump shows you only what the values are, not when they appeared.

Two of the thirteen bugs in this course are of that kind. Here is one of them run against the most thorough value-checking testbench in the repository:

```
      ok  the golden core                    3 tests, all passed             20.0s
  MISSED  B08_store_drops_done               3 tests, all passed             19.2s
-----------------------------------------------------------------------------
 VERDICT: FAIL — your testbench passes B08_store_drops_done too.
   That bug hides behind: checking values instead of the handshake
```

`B08_store_drops_done` changes one line so that `STORE` no longer pulses the `done` output. The byte still lands in memory, and the address, the data, the program counter, the flags and the cycle count are all right; the only difference is that the `done` pin stays still during a `STORE`.

The shipped six-test example also reports it as `MISSED`. The eleven-test reference suite catches it, but with exactly one test:

```
  caught  B08_store_drops_done               11 tests, 1 FAILED              18.2s
        R08_done_pulses_once_per_instruction
             AssertionError: 15 instructions retired in 60 clock cycles, but `done`
pulsed 12 times.
```

Fifteen instructions gave twelve pulses because three of the fifteen were `STORE`. The other ten reference tests pass on this core. One of them steps a program and compares every register, the PC and both flags against a Python model after every instruction, and it still passes, because the model (`golden.py`) has no idea `done` exists.

> In software, "returned the right value" is usually the whole contract, and when it returned is a performance question. In hardware, the timing is part of the specification, and that part has a name: the handshake. A downstream block that is waiting for a pulse that never comes doesn't get a late answer; it gets no answer at all.

This matters more than it might seem, for the reason Where bugs hide gave: `done` is the signal a scoreboard triggers on, and a missing pulse doesn't make such a test fail. Instead the scoreboard stops scoring, and the test keeps reporting green about a stretch of the program it is no longer checking.

One cycle late

`B13_store_falls_to_idle` belongs to the same family, and its effect is even smaller. `STORE` drops back through the core's idle state on its way to the next instruction instead of going straight there. Every result and every byte of memory is identical, and the only cost is one extra clock cycle per `STORE`. Here it is against the example testbench:

```
  caught  B13_store_falls_to_idle            6 tests, 1 FAILED               5.8s
        t05_a_loop_and_its_cycle_count
            AssertionError: the program took 110 clock cycles, expected 109.
```

That one-cycle difference between 110 and 109 is the only symptom, so the test that catches it is the one that asserts a cycle count worked out by hand:

```
    setup    4+5+4+4+4+5+4               = 30
    loop     (ADD 4 + SUB 4 + BNZ 3) x 5  = 55
```

```
    finish   4+5+4+4+4+3                = 24
                                          ---
                                          109
```

An assertion on a number derived from the datasheet costs one line, and it catches a class of bug that no value comparison can reach. The size of the difference also tells you where to look. If the answer had been 114 instead of 110, the difference of five would have pointed to a cycle lost once per loop iteration. A cycle count is a property of the design, and properties like that are cheap to assert.

The same idea on the wire

SimProc talks to the outside world over a UART. The harness in `simproc.py` contains a **bus functional model** (BFM), which is Python code that pretends to be the device at the other end of a wire; it knows the protocol and nothing else about the design. Its `send_byte()` sends a start bit, eight data bits and a stop bit, then holds the line idle for one more bit period, and a comment in the code says that gap is required.

`labs/common/test_handshake_timing.py` measures why the gap is needed. It has four tests and runs 90,910 ns of simulated time in under a second of wall clock, and you'll run it in Lab 05, Part 7.

How long the receiver is busy

`t01` puts a monitor on `UART_RX1.state` and sends one byte at the default speed (`uio_in = 8`, so `clk_per_bit = 32`):

```
async def watch_rx(sp, cycles):
    """Record (state_name, how_many_clocks) for the receiver, for `cycles`."""
    rx = sp.sys.UART1.UART_RX1
    runs = []
    for _ in range(cycles):
        await RisingEdge(sp.dut.clk)
        name = RX_STATES[int(rx.state.value)]
        if runs and runs[-1][0] == name:
            runs[-1][1] += 1
        else:
            runs.append([name, 1])
    return [(n, c) for n, c in runs]
```

It prints:

```
clk_per_bit = 32; receiver states: START=16  RECEIVE=288  DONE=32
the frame on the wire is 320 clocks; the receiver is busy for 336 -- 16 clocks LONGER,
which is 0.50 of a bit period
```

You can work out each of those numbers from the RTL:

State	Clocks	Why
`START`	16	It leaves when `clkCount == (clk_per_bit-1)>>1` = 15, counting from 0, so it stays for 16 clocks. That is half a bit period, which puts the sampling point in the middle of the start bit
`RECEIVE`	288	9 × 32: the eight data bits, then one more bit period, which ends in the middle of the stop bit. The receiver's RTL keeps that ninth period as the slot for a parity bit, but SimProc's UART is built with no parity

State	Clocks	Why
`DONE`	32	One more bit period, during which it holds the byte before it looks at the line again

That adds up to 16 + 288 + 32 = 336 clocks. The frame on the wire was 10 bit periods (start, eight data, stop), which is 320 clocks, so the receiver is busy for exactly half a bit period longer than each frame it receives. The chip is ready again 16 clocks after your frame ends, rather than as it ends, and 16 is `clk_per_bit / 2`.

The test asserts the formula rather than just the number, so it still holds at other baud rates. Here it is swept across four settings:

`uio_in`	`clk_per_bit`	START	RECEIVE	DONE	Busy	Frame	Excess
4	16	8	144	16	168	160	8
8	32	16	288	32	336	320	16
12	48	24	432	48	504	480	24
16	64	32	576	64	672	640	32

The excess is `clk_per_bit / 2` in all four rows.

Sending without the gap

`t02` sends the same three bytes twice: once through the harness's `send_byte()`, which leaves the gap, and once through `send_frame_only()`, which is `send_byte()` with its last two lines deleted. That second version is the one almost everybody writes first:

```
async def send_frame_only(sp, value):
    """One 8N1 frame and NOTHING else -- no trailing idle."""
    frame = [0] + [(value >> i) & 1 for i in range(8)] + [1]
    for b in frame:
        sp.dut.ui_in.value = b
        await ClockCycles(sp.dut.clk, sp.clk_per_bit)
```

The command is `SET_PC 12`: three bytes, `0x99 0x0C 0x00`. The test prints:

```
sent    : 0x99 0x0C 0x00
received: 0x99 0x86 0x80
with the inter-frame gap: PC = 12.  Without it: PC = 134.
```

Nothing produced an error or a warning, and every bit was on the wire for exactly one bit period, but the program counter ended up at 134 instead of 12.

The corruption has a pattern, and you can read the cause straight from it:

- `0x99` arrived intact. The receiver was idle before it, so it caught the start bit on time.
- `0x0C` arrived as `0x86`. `0x0C` is `0000 1100` and `0x86` is `1000 0110`, which is `0x0C` shifted right by one with a `1` in the new top bit: `(0x0C >> 1) | 0x80 = 0x86`. Every bit 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 came from the stop bit, which is always 1.
- `0x00` arrived as `0x80`, which is `(0x00 >> 1) | 0x80` by the same rule. For these three bytes the test pins the result exactly (`assert got == late and bad == late[1]`), because a prediction you can check is worth more than a shrug.
- `134` is `0x86`, the corrupted address byte, loaded into the program counter.

The receiver was still in `DONE` when the second start bit began, so it missed that falling edge. It picked up the next falling edge instead and never recovered, which is why every later frame is late by the same amount.

`PING` without the gap

`t03` sends `PING` the same broken way:

```
gapless PING answered 0xAA -- a green light on a link that just corrupted two of its
three bytes
```

`PING` is `0xF0`, and it ignores its address and data bytes, so the two frames that get mangled are the two that nothing reads, and the reply is a fixed `0xAA` either way. Your link test passes while every command that carries an argument is broken.

> A link test that passes over a broken link is worse than having no link test. It has the same flaw as the shipped test that asserts the processor is not running: it is a green light that was never wired to the thing it claims to check.

Two numbers your testbench already depends on

`t04` measures the two constants the harness is built on, and checks each against arithmetic you can do on paper:

```
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
```

The 1,104 is three frames of eleven bit periods each (ten on the wire plus the gap), plus the 48 clocks the harness waits at the end of a command: 3 × 11 × 32 = 1,056, and 1,056 + 48 = 1,104. This is why you can't stop this processor after exactly *N* instructions: `HALT` takes 1,104 cycles to arrive, and the core keeps running the whole time. It is also why `run()` appends a parking loop.

The 57 is the core's head start. The chip acts on `RUN` before the last stop bit has finished going out, so a monitor that you start after `start_running()` returns has already missed about fourteen instructions. For that reason every counting call in the harness (`run()`, `run_cycles()`, `count_done_pulses()`, `count_halt_cycles()`) starts its watcher first. If you write your own counting call, do the same; the harness's `_go()` is the four lines to copy.

Other bytes

`COMMAND` is one line at the top of the test file, and nothing in `t02` is a constant, so you can change the data and watch the corruption change its pattern. Here are four runs, all of which pass:

Address, data	Received	The "one period late" prediction	PC
`0x0C, 0x00` (shipped)	`0x99 0x86 0x80`	`0x99 0x86 0x80` ✓	134
`0x34, 0x12`	`0x99 0x9A 0x89`	`0x99 0x9A 0x89` ✓	154
`0x0F, 0x33`	`0x99 0x87 0xE6`	`0x99 0x87 0x99` ✗	135
`0x55, 0xFF`	`0x99 0xAA`, two bytes only	`0x99 0xAA 0xFF` ✗	0

They all pass because `t02` asserts only what is true for any three bytes: the first frame arrives intact, something later is corrupted, and the command did not do what you asked.

The last two rows show that the clean one-bit-period slip is only the common case, and you can't rely on it.

`0x0F, 0x33` slips cleanly on the second frame and then lands somewhere else on the third. The receiver resynchronised onto a different edge, and `0xE6` is neither the byte you sent nor what a uniform slip would give.

`0x55, 0xFF` is worse, because only two bytes arrive. After its start bit, `0xFF` is a solid run of ones, with no falling edge for a resynchronising receiver to latch onto. The receiver never sees a third byte, the chip is left waiting for it, and the program counter stays at 0. A lost frame and a corrupted frame look completely different in a log, even though they come from the same defect.

> When a protocol has a recovery or turnaround period, find its length in the RTL, make your BFM wait it out, and write a test that measures it, so that you find out when someone changes it. When you pick test data, avoid bytes that are fixed points of the corruption you are hunting. `0x00`, `0xFF` and

`0xAA` survive more transformations than any other bytes, and they happen to be the three everybody reaches for first.

Check yourself

1. A core gets every register right and pulses `done` once for every two instructions. Which of `golden.py`, a cycle-count assertion and a `done` counter will notice?
2. At `clk_per_bit = 48`, how many clocks after the end of a frame is the receiver ready again?
3. Why does `PING` make a poor link test?

Next: Lab 05: Every answer is right.

24. 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.

VII Capstone

25 **Writing the bug report** 139
25 The five parts, in the order the reader needs them 139
25 Four ways a bug report fails 141
25 Check yourself 142

26 **Lab 06: Capstone** 143
26 Before you start 143
26 Part 1: Get a baseline 144
26 Part 2: B10, the wrong operator 144
26 Part 3: B11, the wrong destination register . . . 145
26 Part 4: B12, the baud rate 146
26 Part 5: Hunt one properly 146
26 Part 6: The suite 147
26 Part 7: Practise the report on B01 147
26 Part 8: Write the report 148
26 Part 9: Close the course 148
26 Expected results 149
26 Finishing 149
26 Going further (optional) 149
26 Answers 149

25. Writing the bug report

This page answers: *I found it. What do I actually hand to the designer?*

You hand them a reproducer, which is a test they can run. A description, a patch or a paragraph in a chat window doesn't do the same job. The capstone is built on one idea that surprises people:

> The test is the deliverable, not the fix.

You met the reason in Lab 04, when the grade's row 2 turned out to be the deliverable, and it comes down to which of the two lasts. A patch is something one person did once, in one file, on one afternoon. A test that fails on the broken design and passes on the fixed one is what stops the bug coming back next year, in a rewrite, done by someone who has never heard of you. Of those two things, only the test will still be doing any work in 2031.

DD104 builds that idea into the grader. `make verdict` prints three rows, and the middle one is your deliverable:

```
      ok  1. the golden core          must PASS      3 tests, all passed        4.2s
  caught  2. the core you were dealt  must FAIL      3 tests, 1 FAILED          3.4s
       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          3.4s
```

The core in row 2 is rebuilt fresh from the bug catalogue every time, rather than read from `core/`, so when you apply the fix, row 3 goes green and row 2 stays exactly as it was:

```
      ok  1. the golden core          must PASS      3 tests, all passed        4.2s
  caught  2. the core you were dealt  must FAIL      3 tests, 1 FAILED          3.4s
      ok  3. core/ as it stands now   must PASS      3 tests, all passed        3.5s
-----------------------------------------------------------------------------
 VERDICT: PASS
   Row 2 is the part worth keeping. Commit that test.
```

Editing the file the test runs against can't turn row 2 green; only a test that catches the bug can do that. That makes "the test is the deliverable" a fact about the grader rather than an opinion.

In Lab 06, Part 7 you'll go through all of this end to end on a bug you already know, before you write the real report.

The five parts, in the order the reader needs them

`BUG-REPORT-template.md` ships in the capstone package. Next to it is a complete worked example, `BUG-REPORT-EXAMPLE.md`, filled in for `B01_sub_operands_swapped`, which we deliberately chose from outside the capstone's three bugs. Read it to see the shape of a report, then write yours. The five parts below are in the order the reader needs them.

1. The smallest program that reproduces it

Minimise the program: cut instructions until it stops failing, then put the last one back.

A ten-line repro and a three-line repro show the same bug, but they cost very different amounts of someone else's day, because every instruction you leave in is one the reader has to rule out.

The worked example is two `set_reg` calls, which come to twelve instructions of setup, and one `SUB`, and it ships complete as `labs/common/test_b01_repro.py`:

```
    sp = SimProc(dut)
    await sp.start()
```

```
    await sp.run(set_reg(1, 200) + set_reg(2, 100) + [SUB(1, 2)])
    got = sp.reg(1)
```

There is no loop, no branch, no memory access, no model, and no UART traffic beyond what the harness needs to start the core. The repro also comes with the command that runs it, from `labs/common`: `make check TB=test_b01_repro BUG=B01`.

If the reader can't run a repro, it's only a description, so give both the file and the command, and make sure both work from a clean clone, because the reader doesn't have your shell history.

2. Expected against actual, as numbers, with the arithmetic

"r1 is wrong" isn't enough. Give the number you expected, the number you got, and the sum that produces the expected one. With the sum included, the report stops being a complaint and becomes a claim that someone can check.

Put all of this in the assertion message as well as in the report, so that it appears on someone else's machine without anyone having to work it out again:

```
    assert got == 100, (
        f"SUB r1, r2 with r1 = 200 and r2 = 100 left r1 = {got}.\n"
        f"  expected  200 - 100 = 100\n"
        f"  got       {got}\n"
        f"  note      (100 - 200) mod 256 = {(100 - 200) & 0xFF}, which is what a "
        f"core computing rB - rA would give\n"
        f"  {sp.describe()}")
```

`make verdict VERBOSE=1` prints all of it:

```
 SUB r1, r2 with r1 = 200 and r2 = 100 left r1 = 156.
     expected  200 - 100 = 100
     got       156
      note       (100 - 200) mod 256 = 156, which is what a core computing rB - rA
would give
     PC=13  r0=100 r1=156 r2=100 r3=  0  N=1 Z=0  IDLE
 assert 156 == 100
```

From that message alone you know which line of RTL to open, without rerunning anything, adding a print or opening a waveform. That is the standard to aim for, and it's cheap to meet: the last line of the message, `sp.describe()`, is a single call.

3. The shape

This part says how many things are wrong, out of how many, and what that count tells you before anyone opens the RTL.

A count becomes a diagnosis when it factors, that is, when you can explain exactly where the number comes from. Here is the ALU driven exhaustively, 393,216 vectors in about nineteen seconds:

```
ALU ADD  value_mismatch=     0 flag_mismatch=     0
ALU SUB  value_mismatch= 65024 flag_mismatch= 65024
ALU OR   value_mismatch=     0 flag_mismatch=     0
ALU NAND value_mismatch=     0 flag_mismatch=     0
ALU SHL  value_mismatch=     0 flag_mismatch=     0
ALU SHR  value_mismatch=     0 flag_mismatch=     0
```

65,024 of the 65,536 `SUB` pairs are wrong, and every other operation is perfect. The next step is to account for the 512 pairs that survive, because that is where the diagnosis is. `A - B == B - A (mod 256)` exactly when `2(A - B) == 0 (mod 256)`, which happens when `A - B` is `0` or `128`. That gives 256 pairs with `A == B` plus 256 with `|A - B| == 128`, and `65,536 − 512 = 65,024` exactly. Only "the operands are the wrong way round" produces exactly 512 survivors, so a single number has already narrowed the bug down to one explanation.

Some shapes are worth learning to recognise:

The count says	It means
`0` value mismatches, `25,726` flag mismatches	The arithmetic is fine; look at the flag logic
Exactly half	The fault tracks a signal that is high half the time
A count that factors into a sentence	You have a hypothesis; go and confirm it
A count that does not factor	You have a number, and you still owe an explanation for it

The sweep output above also gives you a finding for free: `65024` appears in both columns. `N` and `Z` are computed from `ALUout`, so they are wrong exactly when it is. That clears the flag logic, and you didn't need a separate test to find that out.

4. Where it is not

People tend to skip this part, but it's what makes a report credible, and it also narrows down where the fix has to go.

- Which instructions are unaffected? (`ADD`, `OR`, `NAND`, `SHL`, `SHR`: zero of 65,536 each.)
- Which registers, addresses and flags are unaffected?
- Does it survive a reset? (This bug is combinational, so reset makes no difference, and saying so rules out a whole class of explanations.)
- Does it depend on the operands being different? (Yes. `SUB rX, rX` is the usual way to clear a register, which is why this bug gets through almost every program in this course unnoticed.)
- Which of your other tests pass on this core, and what does that rule out?

The worked example ships two tests that pass on the broken core on purpose, `t02_the_idiom_that_hides_it` and `t03_the_512_survivors`. If a report contains only failing tests, nobody has checked which parts of the design still work.

> In software testing, a passing test in a bug report is noise. Here it is evidence, because each one marks a region of the design the reader can stop suspecting, and narrowing the search is most of debugging.

5. The habit the suite gained

Write one line giving the rule a person can carry to the next design, and name the test that enforces it here.

> Reflex: *never test a binary operation with `op(x, x)`. Aliased operands make a swapped, mirrored or wrong-source implementation indistinguishable from a correct one, and the aliases you write by habit (`SUB rX, rX` for clear, `NAND rX, rX` for not, `LOAD rX, rX` for a pointer dereference in place) are exactly the ones you will never suspect.*
>
> Enforced by: `t01_sub_with_two_distinct_operands` in `labs/common/test_b01_repro.py`, which fails on this core and passes on the golden one.

A demonstration with a one-line rule attached becomes a habit, while a demonstration on its own stays an anecdote. The rule has to be short enough that you still remember it six months later, and it has to name a test, because otherwise nothing enforces it.

Four ways a bug report fails

The first is "It doesn't work." A report like that has no numbers in it, so nobody can tell whether they have fixed the bug.

The second is a patch with no test, which is the quickest way to end up with the same bug twice. If you did fix the bug as well, that's good: put the diff at the bottom, under the reproducer.

The third is a repro that only runs on your machine, because it uses a file path that isn't in the repo, a testbench that imports something you wrote at 1 a.m., or a `make` invocation with four variables set. Everything in your report must run from a clean clone with the command you printed.

The fourth is a failing test with no passing one, and that is what the first row of `make verdict` is there for. If a test fails on the broken core and also fails on the golden one, it has found a problem in your testbench rather than your bug, so row 1 must be `ok` before row 2 means anything.

Check yourself

1. Your repro needs a loop and a branch to fail. What do you try before you write the report?
2. Why does the worked example include two tests that pass on the broken core?

Next: Lab 06: Capstone.

26. 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?
Memory	Have I used address 0, address 63, and an address above 63?

	Ask
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:
   a. The smallest program that reproduces it. Cut instructions until it stops failing, then put the last one back.
   b. Expected against actual, as numbers, with the arithmetic shown.
   c. The shape: how many vectors or cycles are wrong, out of how many, and what the count implies before anyone opens the RTL.
   d. Where it isn't: what still works.
   e. 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
`B12` at `clk_per_bit >= 24`	indistinguishable from a correct core	exact
`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?

Appendices

A. The SimProc instruction set

This page tells you what every SimProc opcode does, how it is encoded, and how many cycles it takes. It's meant to be kept open in a tab while you work. If you want the machine explained rather than tabulated, read SimProc in one page instead.

You can check everything here against two files that ship with the course: the assembler, `labs/common/simproc_asm.py`, and the model, `labs/common/golden.py`. Both run at a plain Python prompt, without a simulator.

Encoding

Every instruction is one byte, and three different field layouts share those eight bits:

```
register form     A A B B o o o o      rA = instr[7:6]   rB = instr[5:4]
                  7 6 5 4 3 2 1 0      op = instr[3:0]

branch form       d d d d o o o o      d  = instr[7:4], 4-bit SIGNED
                  7 6 5 4 3 2 1 0      op = instr[3:0]

SHIFT             A A D S S 0 1 1      D  = instr[5]: 1 = left, 0 = right
                  7 6 5 4 3 2 1 0      SS = instr[4:3], the amount, 0..3

ORI               i i i i i 1 1 1      i  = instr[7:3], a 5-bit immediate
                  7 6 5 4 3 2 1 0
```

`SHIFT` and `ORI` are selected by their low three bits, while every other instruction is selected by the full low nibble.

The table

`rA` is both a source and the destination. The flags are `N`, which is bit 7 of the result, and `Z`, which is set when the whole result was zero. A dash in the flags column means the instruction leaves both flags exactly as they were.

op	byte	mnemonic	what it does	flags	cycles	assembler
`0000`	`0x?0`	`LOAD rA, rB`	`rA <- mem[rB]`; rB is the pointer	—	4	`LOAD(1, 2)`
`0001`	`0x?1`	`JMP d`	`PC <- PC+1+d`, always	—	3	`JMP(-3)`
`0010`	`0x?2`	`STORE rA, rB`	`mem[rB] <- rA`; same order as `LOAD`	—	3	`STORE(1, 3)`
`0011`	`0x?3`	`SHL rA, n` / `SHR rA, n`	`rA <- rA << n` or `rA >> n`, `n` in 0..3	`N Z`	4	`SHL(0, 3)`
`0100`	`0x?4`	`ADD rA, rB`	`rA <- rA + rB`, wraps at 8 bits	`N Z`	4	`ADD(1, 0)`

op	byte	mnemonic	what it does	flags	cycles	assembler
`0101`	`0x?5`	`BPZ d`	if `N == 0`: `PC <- PC+1+d`	—	3	`BPZ(2)`
`0110`	`0x?6`	`SUB rA, rB`	`rA <- rA - rB`, wraps at 8 bits	`N Z`	4	`SUB(1, 2)`
`0111`	`0x?7`	`ORI imm5`	`r0 <- r0 | imm5`; the destination is always r0	`N Z`	5	`ORI(5)`
`1000`	`0x?8`	`NAND rA, rB`	`rA <- ~(rA & rB)`	`N Z`	4	`NAND(1, 1)`
`1001`	`0x?9`	`BNZ d`	if `Z == 0`: `PC <- PC+1+d`	—	3	`BNZ(-3)`
`1010`	`0x?A`	`BZ d`	if `Z == 1`: `PC <- PC+1+d`	—	3	`BZ(1)`
`1011`	`0x?B`	*(`SHIFT` again)*	the second encoding of `SHIFT`	`N Z`	4	
`1100`	`0x?C`	*(unimplemented)*	`PC+1` and does nothing else	—	4	`UNIMP()`
`1101`	`0x?D`	*(unimplemented)*	`PC+1` and does nothing else	—	4	
`1110`	`0x?E`	*(unimplemented)*	`PC+1` and does nothing else	—	4	
`1111`	`0x?F`	*(`ORI` again)*	the second encoding of `ORI`	`N Z`	5	

The `?` stands for whatever the register or displacement fields happen to hold. The byte `0x44` is `ADD r1, r0` and the byte `0x04` is `ADD r0, r0`, but both are `ADD`, because the low nibble is `4` in each.

All 256 bytes

There are no illegal encodings, because every byte decodes to something, and no two opcodes collide. We checked this by single-stepping all 256 instruction bytes and classifying each one by its cycle count, `done`, PC delta, register write and memory write:

	encodings	which
the nine 4-bit opcodes	16 each, 144 total	`0x?0` `0x?1` `0x?2` `0x?4` `0x?5` `0x?6` `0x?8` `0x?9` `0x?A`
`SHIFT`	32	`0x?3` and `0x?B`
`ORI`	32	`0x?7` and `0x?F`
unimplemented	48	`0x?C` `0x?D` `0x?E`
	256	

Because `SHIFT` and `ORI` are chosen by only three bits, each of them owns two low nibbles. Nothing else is assigned to `0x?B` or `0x?F`, so the aliasing does no harm. It does mean, though, that the byte `0xF1` is a `JMP` while `0xFF` is an `ORI`, and the two differ in one bit that most people don't look at.

Disassembling a byte from the command line

The assembler also works as a disassembler. When a test fails and a log line shows you a hex byte, this is the quickest way to find out which instruction it was:

```
cd labs/common
python3 simproc_asm.py 0x2F 0x44 0xD9
```

```
0x2F   ORI   5
0x44   ADD   r1, r0
0xD9   BNZ   -3
```

The script imports nothing and needs no simulator, and it will decode as many bytes as you give it.

Here are some bytes worth learning to recognise on sight:

byte		
`0x0C`	`(unimplemented)`	what the harness fills unwritten RAM with
`0xF1`	`JMP -1`	the parking loop, which is a jump to itself
`0x06`	`SUB r0, r0`	the idiom for "clear `r0`"
`0x2F`	`ORI 5`	
`0x3B`	`SHL r0, 3`	
`0x8B`	`SHR r2, 1`	note the low nibble, `B`, which is `SHIFT`'s second encoding

Branch displacements

The PC is incremented during `CYCLE_1`, *before* the branch computes its target in `CYCLE_3`, so the displacement is measured from the instruction **after** the branch:

$$\text{target} = (\text{address of the branch}) + 1 + d \tag{1.1}$$

`JMP(0)`	falls through to the next instruction
`JMP(1)`	skips one instruction
`JMP(-1)`	an infinite loop on itself, which is the parking idiom
range	`-8` to `+7`, and nothing outside it

The field is four bits wide and **signed**. If you branch to a label further away than that, the assembler stops with an error rather than quietly wrapping the displacement:

```
ValueError: line 12: 'BNZ loop': 'loop' is -11 instructions away and a SimProc
branch only reaches -8..+7. Move the label closer, or JMP to a JMP.
```

Flags

set `N` and `Z`	leave them alone
`ADD SUB NAND SHL SHR ORI`	`LOAD STORE JMP BZ BNZ BPZ`

`N` is bit 7 of the result, and `Z` means the whole 8-bit result was zero. There is no carry flag and no overflow flag, so arithmetic wraps without telling you.

The branches leave the flags alone by specification, not by accident, and that is what lets you compare once and then branch on the result twice:

```
        SUB   r1, r1        # r1 = 0  ->  Z = 1
        BZ    taken         # taken
        ORI   16            # skipped
taken:  BNZ   wrong         # must NOT be taken: Z is still 1
```

Idioms

This instruction set lacks several things you're used to, and each gap has a standard workaround. Think of these as the only way to write each operation on SimProc rather than as tricks.

you want	you write	why
clear `rX`	`SUB rX, rX`	there is no `MOV` and no `CLR`
`NOT rX`	`NAND rX, rX`	there is no `NOT`
compare `rA` against `rB`	`SUB rA, rB`	there is no `CMP`, and the compare destroys `rA`
stop the program	`JMP -1`	there is no `HALT` instruction
a constant in `r0`	`SUB r0, r0`, then `ORI`	`ORI` only *sets* bits
a constant larger than 31	`ORI` the high bits, `SHL`, `ORI` the low bits	the immediate is 5 bits
a constant in `r1`/`r2`/`r3`	build it in `r0`, then `SUB rX, rX ; ADD rX, r0`	`ORI` only writes `r0`

> Every row in that table is also an entry in the bug catalogue. `SUB rX, rX` is symmetric, so it gives the right answer even on a core whose subtract is backwards, and `NAND rX, rX` is the one operand pattern for which `NAND` and `NOR` agree. The idiom a programmer reaches for first is the one that hides the bug. Where bugs hide covers this at length.

Building a constant by hand is easy to get wrong, so the assembler has a helper that does it for you:

```
>>> from simproc_asm import set_reg, listing
>>> print(listing(set_reg(1, 0xB7)))
  00: 06   SUB   r0, r0
  01: B7   ORI   22
  02: 3B   SHL   r0, 3
  03: 3F   ORI   7
  04: 56   SUB   r1, r1
  05: 44   ADD   r1, r0
```

The helper always emits four instructions for `r0` and six for `r1`–`r3`, so the cycle count stays predictable. It does clobber `N` and `Z`, because `SUB`, `ORI` and `SHL` all set flags, so set your registers up before a compare, never between the compare and the branch.

Cycle counts

The core is not pipelined and nothing is speculative, so a program's cycle count is the sum of the cycle counts of its instructions, and you can check it by hand.

instruction	cycles	
`ORI`	5	`CYCLE_1`…`CYCLE_5`, writes `r0` in `CYCLE_5`

instruction	cycles	
`ADD SUB NAND SHL SHR LOAD`	4	write-back in `CYCLE_4`
`STORE JMP BZ BNZ BPZ`	3	retires in `CYCLE_3`
`0xC 0xD 0xE`	4	and no `done` pulse; see below

A branch costs 3 cycles whether or not it is taken.

Unimplemented opcodes: four cycles

These opcodes match no clause in `CYCLE_3`, so the FSM takes its default exit to `IDLE`, spends a cycle there, and is sent back to `CYCLE_1` by `run`, which is still asserted. When we measured four filler bytes in a row, they took 16 clock cycles, produced 0 `done` pulses, and had `halt` high on 4 of the 16.

That last figure matters. `0x0C` is what `sp.start()` fills unwritten RAM with, so a program that runs off its own end makes the `halt` pin blink. "`halt` must stay low while `run` is asserted" is an assertion Lab 05 asks you to write, so if it fires in that situation, your test is working correctly on a machine that is behaving as designed.

`golden.py` said three cycles until somebody measured it, and The golden model tells that story.

Writing a program

There are three ways to write a program, shown here from least to most readable:

```
prog = [0x2F, 0x44, 0xD9]              # ... which of these is the loop?
```

```
from simproc_asm import ORI, ADD, STORE
prog = [ORI(5), ADD(1, 0), STORE(1, 2)]
```

```
from simproc_asm import assemble
prog = assemble("""
        ORI   5              # r0 = 5
        ADD   r1, r0         # r1 = 5
loop:   SUB   r1, r0         # r1 -= 5
        BNZ   loop           # ... until it hits zero
        STORE r1, r2
""")
```

The text front end works out the branch displacements for you. That is the only reason to use it, but it's a good one, because almost everybody measures a displacement from the wrong place the first time.

`listing(prog)` prints a dump with the address, hex byte and mnemonic of each instruction. If you put it in your failure messages, it will save you twenty minutes when a test fails:

```
  00: 06   SUB   r0, r0
  01: 0F   ORI   1
  02: A6   SUB   r2, r2
```

Traps

Each of these has cost somebody an evening.

1. `ORI` ORs its immediate into `r0` and never clears any bits. Do `SUB r0, r0` first, or you get the OR of everything `r0` has ever held.
2. `ORI` always writes `r0`. There is no register field to get wrong, and no way to make it write anything else.
3. Shift amounts are 0–3. `SHL(0, 4)` raises an error rather than quietly truncating the amount: `shift amount 4 does not fit in 2 bits. SimProc shifts by 0, 1, 2 or 3 — to shift by 4, use two instructions.`
4. The branch range is `-8` to `+7`, measured from the instruction after the branch.
5. Arithmetic wraps without any warning. `200 + 60` is `4`, and there is no carry flag to check.

6. Addresses above 63 don't exist. A read from one returns `X` and a write to one is dropped, and neither of them aliases into low memory.
7. `LOAD rA, rB` and `STORE rA, rB` take the data register first and the pointer second. The order is the same for both, and it is the opposite of what many readers expect.
8. `SUB` is your compare, and it destroys `rA`. If you need the value afterwards, copy it first, and remember that a copy costs two instructions and overwrites both flags.

Links

- SimProc in one page: the machine, the FSM, the pins, the protocol
- The SimProc harness: what the test harness gives you
- `labs/common/simproc_asm.py`: this page as executable code, plus the disassembler
- `labs/common/golden.py`: this page as a reference model

B. The SimProc harness

This page answers: *what can I ask the harness for, and what does it already do for me?*

Everything on this page lives in `labs/common/`, and you can import it from any lab, because `sim.mk` puts both your lab directory and `common/` on `PYTHONPATH`.

Every worked example below is a real line from `test_api_tour.py`, and every output block is what that line printed. The whole tour is nine tests and 535,650 ns of simulated time, and it runs in about five seconds:

```
cd labs/common
make sim TB=test_api_tour
```

If a signature on this page stops matching the code, that file stops running, so the page and the code can't drift apart without someone noticing.

The five files

file	what it is
`simproc.py`	The harness: `SimProc`, `trace()`, `bit()`, and the command constants.
`simproc_asm.py`	The assembler: instruction helpers, `set_reg`, `assemble`, `disasm`, `listing`. It imports nothing, so it runs at a plain Python prompt.
`golden.py`	The reference model: the instruction set in about 120 lines of Python. It imports nothing.
`bugs.py` · `inject.py` · `check.py`	The bug catalogue, the injector that puts a bug into the core, and the checker that gives the PASS/FAIL verdict. You run these through `make` rather than calling them by hand.
`test_example.py` · `test_reference.py` · `test_api_tour.py`	Worked testbenches. Copy the first to start your own, read the second once you've written yours, and treat the third as the source of every example on this page.

The shortest complete test

```
import cocotb
from simproc import SimProc
from simproc_asm import ORI, ADD

@cocotb.test()
async def r1_gets_five(dut):
    sp = SimProc(dut)
    await sp.start()                              # clock, reset, blank memory
    cycles = await sp.run([ORI(5), ADD(1, 0)])    # r0 = 5 ; r1 = r1 + r0
    assert sp.reg(1) == 5, f"r1 is {sp.reg(1)}, expected 5"
    assert cycles == 9, f"took {cycles} cycles, expected 5 + 4"
```

```
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 name, so write `TB=test_mine`, never `TB=test_mine.py`.

Setup

`SimProc(dut, clock_ns=10, uio=8)`

`SimProc` wraps the top-level handle that cocotb gives you. Creating it doesn't touch the simulation; nothing happens until you call `start()`.

`clock_ns` is the clock period. `uio` sets the baud divider, with `clk_per_bit = uio * 4`, so the default of 8 gives 32 clocks per bit. That's fast enough for tests to finish quickly, and slow enough to keep the UART comfortably inside its spec.

The object also gives you hierarchical handles that reach past the pins and into the design: `sp.chip` (the TinyTapeout wrapper), `sp.sys` (`simproc_system`), `sp.cpu` (the core), `sp.ram` (the 64-byte array) and `sp.rf` (the four registers). A real chip has nothing like these, but a simulation does, and if you refuse to use them on principle, each bug will cost you an afternoon.

`await sp.start(fill=FILLER)`

Starts the clock, resets the chip, writes `fill` into all 64 bytes of RAM, and starts the background UART receiver. Always call it first.

```
sp = SimProc(dut, clock_ns=10, uio=8)
await sp.start()
```

```
clock period 10 ns; uio=8 so clk_per_bit=32
after start(): PC= 0  r0=  0 r1=  0 r2=  0 r3=  0  N=0 Z=0  IDLE
RAM filler is 0x0C, RAM depth is 64, the parking instruction is 0xF1 = JMP   -1
```

The filler matters because `0x0C` is an unimplemented opcode (it takes four cycles, advances the PC by one and changes no state), so a program that runs off its own end walks harmlessly through the filler. Without it, unwritten RAM reads `XXXXXXXX` under Icarus, the core decodes an X, and you get a hang with no error message at all.

`await sp.reset()`

Pulses `rst_n` low for 10 cycles. That clears the registers, the PC and the flags, but not the RAM: `dp_ram` has no reset, so whatever you loaded into it survives.

```
sp.poke(0, 0x5A)
await sp.reset()
```

```
after reset(): PC= 0  r0=  0 r1=  0 r2=  0 r3=  0  N=0 Z=0  IDLE, and mem[0] is
still 0x5A
```

Because reset leaves memory alone, you can load a program, reset the core, and then run the program from a known state.

`await sp.set_baud(uio)`

Changes the divider in the middle of a test, and `clk_per_bit` becomes `uio * 4`. The chip reads the divider from `uio_in` on every cycle, so the change takes effect immediately, and the harness's own receiver switches to the new rate along with it.

```
await sp.set_baud(16)                  # clk_per_bit = 64
```

```
set_baud(16): clk_per_bit is now 64; uart_ping() -> 0xAA
```

If you only ever test a parametric design at its default parameter, you've tested one point of a range. One of the thirteen bugs is invisible only at the default setting.

The backdoor

`poke`, `peek` and `load` write and read the RAM array directly, in zero simulated time. The UART does the same job through the chip's real pins, at 1,104 clock cycles per command. Use the backdoor for setting up a test, and use the UART when the UART is the thing you're testing.

`sp.poke(addr, value)` · `sp.peek(addr)`

These move one byte each way. `peek` returns an `int`, or raises `ValueError` if the byte contains an X. Both raise `IndexError` for an address outside 0–63.

```
sp.poke(0x30, 0xA5)
dut._log.info(f"... -> 0x{sp.peek(0x30):02X}   <- the OLD value")
await RisingEdge(dut.clk)
dut._log.info(f"one clock later, peek(0x30) -> 0x{sp.peek(0x30):02X}")
```

```
poke(0x30, 0xA5), peek(0x30) in the same instant -> 0x0C   <- the OLD value
one clock later, peek(0x30) -> 0xA5
```

> Watch out for this. `poke()` writes a simulator signal, and a signal written at time *T* can't be read back until the simulator has stepped forward. If you read it in the same instant, you get the old value, with no error and no warning. That's why `load()` ends with two clock cycles, whereas a bare `poke()` doesn't wait at all. Put a trigger (an `await` such as `RisingEdge(dut.clk)`) between a poke and the peek that checks it.

An address out of range gets this message:

```
poke(64, 0) raises: address 64 (0x40) is outside SimProc's 64-byte RAM. The hardware
addresses it with 8 bits but only has 64 entries, so on real silicon this read returns X
and this write is dropped. Verilator's linter says so statically; Icarus only shows
it at
run time. See Lab 02.
```

`await sp.load(program, at=0, park=True)`

Writes a list of instruction bytes into memory and returns the parking address.

```
park = await sp.load([ORI(5), ADD(1, 0)])
```

```
load() put the program at 0..1 and the parking loop at 2 (0xF1 = JMP   -1)
```

Unless you pass `park=False`, it also writes one `JMP -1`, a jump to itself, right after your last instruction. You need that, because there's no way to stop this processor after exactly *N* instructions: `HALT` arrives over the UART and takes 1,104 clock cycles to get there, and the core keeps running the whole time. So programs park themselves, the way real bare-metal test code does, and `run()` watches for the parking address.

`load()` doesn't reset the core and doesn't start it, and it takes 2 clock cycles.

`sp.memory(lo=0, hi=64)`

Returns the RAM contents as a dict, for dumping when something has already gone wrong. A byte containing an X comes back as the string `'XXXXXXXX'` instead of raising an exception, because a dump you print after a failure shouldn't itself crash on the entry you most want to see.

```
memory(0, 8) minus the filler: 0:0x2F 1:0x44 2:0xF1
```

Reading the machine

All of these are synchronous, so you call them without `await` and they take no simulated time.

call	returns	example
`sp.reg(i)`	one register as an `int`	`200`
`sp.regs()`	all four as a tuple	`(200, 200, 0, 0)`
`sp.pc()`	the program counter	`6`
`sp.flags()`	`(N, Z)`	`(1, 0)`
`sp.fsm()`	the core's state, as a name	`IDLE`
`sp.halt`	the `halt` pin, as a character	`'1'`
`sp.done`	the `done` pin, as a character	`'0'`
`sp.snapshot()`	`(regs, pc, N, Z)`	`((200, 200, 0, 0), 6, 1, 0)`
`sp.describe()`	one line of everything	`PC= 6  r0=200 r1=200 r2=  0 r3=  0  N=1 Z=0  IDLE`

Three of these need a note:

`halt` and `done` are characters rather than integers: `'0'`, `'1'` or `'x'`. We did that on purpose. In cocotb 2.x, `int()` on a value containing an X raises `ValueError`, and a test that dies with a stack trace has told you something real in the least helpful way it could.

`snapshot()` returns the same shape that `Golden.snapshot()` returns, so you can compare the two with a plain `==`. That comparison is the whole interface to model-based checking.

`describe()` belongs in every failure message you write, because one call gives you every register, both flags, the PC and the state.

`bit(signal, index)`

`bit` is a free function rather than a method of `SimProc`, and it returns one bit of any signal as the character `'0'`, `'1'`, `'x'` or `'z'`. Use it instead of `(int(sig.value) >> i) & 1` on any signal that a broken design might drive to X.

```
dut._log.info(f"bit(dut.uo_out, 0) -> '{bit(dut.uo_out, 0)}'")
```

```
bit(dut.uo_out, 0) -> '1'  (the UART TX pin, idle high)
```

Making it go

SimProc has two ways to execute a program, and you can't swap one for the other. `RUN` free-runs until you send `HALT`, which is fast, but it gives you no way to stop after exactly *N* instructions. `STEP` executes one instruction and drops back to IDLE, which is exact, but it costs a whole UART command for every instruction.

`await sp.run(program=None, at=0, timeout_cycles=20000)` → cycles

This is the call you'll use most. It loads the program, adds the parking loop, starts the core, and watches the fetch address until it reaches the parking loop. Pass `program=None` to run something you loaded earlier.

```
cycles = await sp.run([ORI(5), ADD(1, 0)])
```

```
run([ORI(5), ADD(1,0)]) -> 9 cycles, r1 = 5   (ORI is 5, ADD is 4)
```

The returned count is exact, and you can check it by hand. It's the number of clocks between fetching your first instruction and fetching the parking loop, which is exactly the sum of your program's instruction times.

On a timeout it raises an exception whose message is a diagnosis rather than a bare number:

```
the program did not reach its parking loop at address N within 20000 clock cycles.
  IDLE means the core never started. A PC far past your program means it ran off the
end —
   check your branch displacements. A PC that is not moving means an infinite loop
you did
  not intend.
```

`await sp.step(n=1)`

Executes exactly `n` instructions, sending one `STEP` command for each, and the core comes to rest in IDLE after every one. Use this mode when you want to inspect the state after each instruction.

```
await sp.load([ORI(1), ORI(2), ORI(4)])
await sp.set_pc(0)
await sp.step(1)
await sp.step(2)
```

```
after step(1):  r0 = 5, PC = 1, state IDLE
after step(2) more: r0 = 7, PC = 3
```

(`r0` starts at 5, left over from the previous `run()`. `step()` doesn't reset anything, and that's deliberate, since the point of stepping is to carry on from the state you're in.)

`await sp.run_cycles(n)`

Free-runs while counting `n` clock cycles of execution. The count starts at the first instruction fetch, not when the `RUN` command was sent. Use it when your question is about time rather than about the answer, for example *"after 40 clocks, how far has it got?"*

```
await sp.load([ORI(1), ORI(2), ORI(4), ORI(8)])
await sp.set_pc(0)
await sp.run_cycles(10)
```

```
after run_cycles(10): r0 = 15, PC = 4
```

> `run_cycles()` counts `n` cycles, but it doesn't freeze the machine after them. When the count is reached the harness sends `HALT`, and `HALT` is a UART command that takes time to arrive, so the core keeps executing until it does. Here is what we measured on a program of 40 unimplemented opcodes with no parking loop:

```
same call on a program that does NOT park: PC = 20.
```

At the instant the count reached ten, the PC was 3, and by the time the core came to rest it was 20. If you need the state at exactly cycle *n*, read it inside a watcher of your own (copy `_go()`), or give the program a parking loop and use `run()`. The test `a04` in `test_api_tour.py` measures both.

`await sp.set_pc(addr)` · `start_running()` · `stop_running()`

These are the three raw controls. `set_pc` only takes effect while the core is in IDLE. `start_running` sends `RUN`, and `stop_running` sends `HALT`, which takes effect when the current instruction retires.

If you write your own free-running measurement, start your monitor before you call `start_running()`. The chip acts on `RUN` 57 clock cycles before `uart_cmd` returns (we measured this in `test_handshake_timing.py`), so a monitor started afterwards has already missed about fourteen instructions. That's why every counting call in the harness goes through one four-line helper, `_go()`, and why you should copy it rather than write your own.

Protocol counters

These check the handshake rather than the data. A core can compute every answer correctly and still get its handshake wrong, and two of the thirteen bugs do exactly that.

`await sp.count_done_pulses(cycles)` → count

Runs the loaded program and counts rising edges of `done` over the first `cycles` clocks of execution. `done` should pulse once per retired instruction.

```
await sp.load([ORI(1), ADD(1, 0), SUB(2, 2), NAND(3, 3)])
await sp.set_pc(0)
pulses = await sp.count_done_pulses(17)          # 5 + 4 + 4 + 4
```

```
count_done_pulses(17) over 4 instructions -> 4
```

Unimplemented opcodes never pulse `done`, so keep them out of a program whose pulses you're counting, or put them in on purpose to check that they don't:

```
two unimplemented opcodes, 6 clocks -> 0 done pulses: 0x0C retires nothing
```

If your scoreboard is triggered by `done`, a missing pulse doesn't make your test fail. Instead your test silently stops scoring, with no message at all, which is worse than a failure.

`await sp.count_halt_cycles(cycles)` → count

Counts how many of the first `cycles` clocks of execution had `halt` high. While `RUN` is asserted and the program runs only implemented opcodes, the answer must be zero:

```
count_halt_cycles(17) -> 0   (implemented opcodes never pass through IDLE)
```

Unimplemented opcodes are the exception here too. Each one spends a cycle in `IDLE` with `halt` high, so four filler bytes (`0x0C`) give 4 halt cycles in 16 clocks, as the instruction set reports. Keep them out of a program whose halt cycles you're counting, or expect one per unimplemented opcode.

A core that blips back into IDLE between instructions can still compute every answer correctly, but it's breaking the handshake, and this counter is how you catch it.

The wire

The harness contains a bus functional model (BFM): Python that plays the device at the other end of the serial line. It knows the protocol (8N1: one low start bit, eight data bits sent least significant first, and one high stop bit), and it knows nothing else about the design.

The receiver runs as a background task, which `start()` launches, and it watches `uo_out[0]` from before the first command goes out. It has to be running that early, because the chip can begin its reply while the

harness is still finishing the last stop bit of the command, and a receiver that only starts looking afterwards gets a plausible but wrong byte instead of an error.

call	what it does	replies?
`await sp.uart_ping()`	`PING`, returns the reply	yes, `0xAA`
`await sp.uart_read(addr)`	`READ`, returns the byte	yes
`await sp.uart_write(addr, data)`	`WRITE`	no
`await sp.uart_get_pc()`	`GET_PC`, returns the PC	yes
`await sp.uart_cmd(cmd, addr=0, data=0)`	any command, no reply waited for	—
`await sp.send_byte(value)`	one raw 8N1 frame plus the mandatory idle gap	—
`await sp.recv_byte(timeout_cycles=20000)`	the next byte the chip sent	—

```
dut._log.info(f"uart_ping()      -> 0x{await sp.uart_ping():02X}")
await sp.uart_write(0x21, 0x3C)
dut._log.info(f"... uart_read(0x21) -> 0x{await sp.uart_read(0x21):02X}")
await sp.set_pc(9)
dut._log.info(f"set_pc(9) then uart_get_pc() -> {await sp.uart_get_pc()}")
```

```
uart_ping()      -> 0xAA
uart_write(0x21, 0x3C) then uart_read(0x21) -> 0x3C
set_pc(9) then uart_get_pc() -> 9
three raw send_byte() calls -> recv_byte() 0xAA
```

Four things about the UART will each cost you an hour if you don't know them:

Every command is three bytes (`cmd`, `addr`, `data`), even when the last two are ignored. If you send only two, the chip is still waiting for the third.

`WRITE`, `RUN`, `HALT`, `STEP` and `SET_PC` don't reply. If you wait for a reply to one of those, you wait forever, and `recv_byte` says so in its timeout message.

`send_byte()` leaves one bit period of idle line after the frame, and that gap is required, because the chip's receiver stays busy for half a bit period longer than the frame it just received. If you send the next start bit immediately, the receiver misses it, and instead of an error you get a byte shifted by one. Handshakes and timing measures all of this.

A working `PING` doesn't prove that the link works. `0xF0` already has its top bit set and `0xAA` is a fixed pattern, so a receiver whose bit period is slightly wrong still gets both of them right.

The assembler — `simproc_asm.py`

The assembler imports nothing, so it runs at a plain Python prompt, and `python3 simproc_asm.py 0x2F 0x44` disassembles bytes from the command line.

helper	form	byte	disassembles as
`ORI(5)`	`R0 |= imm5`, always r0	`0x2F`	`ORI  5`
`ADD(1, 0)`	`rA += rB`	`0x44`	`ADD  r1, r0`
`SUB(2, 3)`	`rA -= rB`	`0xB6`	`SUB  r2, r3`
`NAND(1, 1)`	`rA = ~(rA & rB)`	`0x58`	`NAND  r1, r1`
`SHL(0, 3)`	`rA <<= amt`, amt 0–3	`0x3B`	`SHL  r0, 3`
`SHR(0, 1)`	`rA >>= amt`	`0x0B`	`SHR  r0, 1`
`LOAD(0, 1)`	`rA = mem[rB]`	`0x10`	`LOAD  r0, r1`

helper	form	byte	disassembles as
`STORE(1, 2)`	`mem[rB] = rA`	`0x62`	`STORE r1, r2`
`JMP(-1)`	signed 4-bit displacement	`0xF1`	`JMP  -1`
`BZ(2)`	branch if `Z`	`0x2A`	`BZ   +2`
`BNZ(-3)`	branch if not `Z`	`0xD9`	`BNZ  -3`
`BPZ(1)`	branch if `N == 0`	`0x15`	`BPZ  +1`
`UNIMP()`	unimplemented opcode	`0x0C`	`(unimplemented)`

`set_reg(reg, value)`

Returns the instructions that put an arbitrary 0–255 byte in a register. You need a helper for this because `ORI` is the only way to get a constant into this machine, and it has three limits: it ORs rather than loads, its immediate is only 5 bits, and it only ever writes `r0`. So for any other register, `set_reg` builds the value in `r0` first and then copies it across:

```
listing(set_reg(1, 0xB7))
```

```
  00: 06   SUB   r0, r0
  01: B7   ORI   22
  02: 3B   SHL   r0, 3
  03: 3F   ORI   7
  04: 56   SUB   r1, r1
  05: 44   ADD   r1, r0
```

It's always 4 instructions for `r0` and 6 for the other registers, so the cycle count of a test that uses it stays predictable.

> `set_reg` clobbers `N` and `Z`, because `SUB`, `ORI` and `SHIFT` all set flags. Set your registers up before a compare, never between the compare and the branch. Getting this wrong is the most common way for a flag test to end up testing `set_reg` instead of the design.

`assemble(text, base=0)`

Lets you write the program as text, with labels, so that the assembler does the displacement arithmetic for you:

```
prog = assemble("""
        SUB   r1, r1
top:    ORI   1
        BZ    top
        SUB   r3, r3
""")
```

```
  00: 56   SUB   r1, r1
  01: 0F   ORI   1
  02: EA   BZ    -2
  03: F6   SUB   r3, r3
```

`BZ top` became `BZ -2` without you having to count anything. Branch displacements are 4-bit and signed, and when a label is out of reach, `assemble` raises an error instead of truncating the displacement:

```
ValueError line 12: 'BZ top': 'top' is -11 instructions away and a SimProc branch only
reaches -8..+7. Move the label closer, or JMP to a JMP.
```

`disasm(word)` · `listing(prog, base=0)`

`disasm` turns one byte back into text, and `listing` produces the block of address, hex and mnemonic shown above. If you put a `listing()` in a log line when a test fails, you'll save yourself twenty minutes. If you put a `disasm()` in the failure message, whoever reads it doesn't have to look anything up: `expected 0x14, got 0x24` tells nobody anything, while *"at PC 6 the core ran `ORI 4` and the model ran `ADD r0, r1`"* tells them where to look.

The golden model — `golden.py`

The golden model is a second implementation of the same specification, written in a language where you can see what it does. It imports nothing, and `python3 golden.py` runs a demo program on its own.

call	what it does
`Golden(memory, pc=0)`	A machine holding a program. It pads memory to 64 bytes with the same `0x0C` filler the DUT gets.
`g.step()`	Runs one instruction and returns its cycle count, or a string if it can't continue.
`g.run(n, stop_on_reason=True)`	Runs `n` steps and returns `(cycles, reason_or_None)`.
`g.snapshot()`	`(regs, pc, N, Z)`, the same shape that `sp.snapshot()` returns
`g.memory(lo, hi)`	a dict, like `sp.memory()`
`compare_memory(model, sp, lo=0, hi=64)`	every address where they disagree, as `(addr, model, dut)`

```
prog = set_reg(1, 20) + set_reg(2, 6) + [SUB(1, 2), STORE(1, 2)]
g = Golden(prog + [PARK])
cycles, reason = g.run(len(prog))
await sp.run(prog)
diffs = compare_memory(g, sp)
```

```
Golden.run(14) -> 59 cycles, reason=None
Golden.snapshot() -> ((6, 14, 6, 0), 14, 0, 0)
SimProc.snapshot() -> ((6, 14, 6, 0), 14, 0, 0)
compare_memory() -> []   (an empty list is the good answer)
```

We made `step()` return a string rather than raise an exception on purpose, so that the caller can print the model's reason next to the DUT's state instead of having to catch an exception.

The model's limitation is stated at the top of its own file: it has no concept of `done`, `halt`, cycle counts or the UART, so a bug in any of those is invisible to it. That isn't a flaw in model-based testing, but it is the reason a test suite needs more than one kind of test.

`await trace(sp, n_instr, model=None)` ⟶ `(steps, problem)`

Single-steps `n_instr` instructions and records the state after each one. `steps` is a list of `(index, address, instruction_word, snapshot)` tuples, and `problem` is either `None` or a string describing the first place the DUT disagreed with the model. This example, from `a09_trace_against_the_model` in `test_api_tour.py`, traces a seven-instruction program on a fresh model:

```
prog = set_reg(0, 9) + [ADD(1, 0), SHL(1, 1), NAND(2, 2)]
g = Golden(prog + [PARK])
await sp.load(prog)
await sp.set_pc(0)
steps, problem = await trace(sp, len(prog), model=g)
```

```
trace() ran 7 instructions; problem = None
   0  @00  0x06 SUB   r0, r0     regs=(0, 0, 0, 0) PC=1 N=0 Z=1
   1  @01  0x0F ORI   1          regs=(1, 0, 0, 0) PC=2 N=0 Z=0
   2  @02  0x3B SHL   r0, 3      regs=(8, 0, 0, 0) PC=3 N=0 Z=0
   3  @03  0x0F ORI   1          regs=(9, 0, 0, 0) PC=4 N=0 Z=0
   4  @04  0x44 ADD   r1, r0     regs=(9, 9, 0, 0) PC=5 N=0 Z=0
   5  @05  0x6B SHL   r1, 1      regs=(9, 18, 0, 0) PC=6 N=0 Z=0
   6  @06  0xA8 NAND  r2, r2     regs=(9, 18, 255, 0) PC=7 N=1 Z=0
```

When the model and the DUT disagree, `problem` tells you exactly which instruction went wrong, instead of leaving you with *"something is wrong somewhere"*:

```
instruction 19, at address 19 (0x66 = SUB   r1, r2), left the machine in a different
state:
      model: r=(...) PC=... N=... Z=...
        DUT: r=(...) PC=... N=... Z=...
```

Stepping is slower than free-running, because every instruction costs a whole UART command. Use `run()` when you want the answer, and `trace()` when you want to know where it went wrong.

Constants

```
from simproc import (MEM_DEPTH, FILLER, PARK, STATE_NAMES,
                     CMD_READ, CMD_WRITE, CMD_PING, CMD_RUN,
                     CMD_HALT, CMD_STEP, CMD_SET_PC, CMD_GET_PC)
```

	value	
`MEM_DEPTH`	`64`	`dp_ram` is 64 bytes, so addresses 64–255 read X
`FILLER`	`0x0C`	an unimplemented opcode: 4 cycles, PC+1, no `done`
`PARK`	`0xF1`	`JMP -1`, the parking loop
`CMD_READ` `CMD_WRITE`	`0xA3` `0x5C`	`READ` replies, `WRITE` doesn't
`CMD_PING` `CMD_GET_PC`	`0xF0` `0xB2`	both reply
`CMD_RUN` `CMD_HALT` `CMD_STEP` `CMD_SET_PC`	`0x3F` `0xC7` `0x6D` `0x99`	none of them reply
`STATE_NAMES`	`IDLE, CYCLE_1 … CYCLE_5`	the list `fsm()` looks names up in

Timing you can assert against

These are counts of simulated clock cycles, not wall-clock times, so they're deterministic and you can check each of them in a testbench.

instruction	cycles
`ORI`	5
`ADD` `SUB` `NAND` `SHL` `SHR` `LOAD`	4
`STORE` `JMP` `BZ` `BNZ` `BPZ`	3
an unimplemented opcode (`0xC`, `0xD`, `0xE`)	4, no `done` pulse, and one cycle of `halt`

the UART, at the default `clk_per_bit = 32`	cycles
one 3-byte command	1,104 = 3 × 11 × 32 + 32 + 16
a command with a reply (`PING`)	1,353
loading a 64-byte program over the wire	70,656
the same program through the backdoor	2
`RUN`'s head start: the core is executing this long before `uart_cmd` returns	57

Two things that are not obvious

`load()` writes a `JMP -1` after your program. Nothing on this hardware can stop a program after N instructions, so programs park themselves, the way real bare-metal test code does.

A reset clears the registers but not the RAM. That's what lets you load a program, reset the core and start from a known state, and it's why the random-program test in `test_reference.py` can call `sp.reset()` between programs and still compare memory afterwards.

Make targets

target	what it does	wall clock
`make`	`sim`, then `check`	20 s
`make sim`	your testbench against the golden core	~6 s
`make waves`	the same run, dumping `waves.vcd` (3,007,592 bytes)	~6 s
`make check`	the verdict: PASS on the good core, FAIL on `BUG`	~12 s
`make check-all`	your testbench against all thirteen broken cores	~77 s
`make bugs`	prints the bug catalogue: names, tiers, and what each one teaches	instant
`make model`	runs `golden.py` on its own, with no simulator involved	instant
`make verify`	confirm every catalogue anchor still matches `rtl/`	instant
`make shuttle`	the test the chip shipped with, against everything	~30 s

```
make sim   TB=test_mine          # your module instead of test_example
make check BUG=B07               # which broken core to check against
make sim   SIM=verilator         # the strict simulator
```

The wall-clock figures are ballpark and move by a second or two between runs. The simulated-time numbers don't move at all, on either simulator; see Two simulators for more on that.

For the warnings all of this prints along the way, see Output that is not an error.

C. Output that is not an error

This page answers: *the screen is full of red and yellow. Did something break?*

Usually nothing has. This page goes through every wall of alarming output that DD104 produces. For each one it gives the exact count, so that you can confirm you're seeing the same thing, and a one-line rule for telling it apart from a real failure.

Here are the rules in short:

Tool	It is fine when	It is real when
Icarus compile	Lines say `sorry:`, exit code 0	A line starts `error:` and compilation stops
Verilator	You passed `-Wno-fatal` and the build finished	It says `%Error:` and no binary was produced
cocotb	`TESTS=n PASS=n FAIL=0`	`FAIL` is non-zero, or the results table is missing
`make check`	`VERDICT: PASS`	`VERDICT: FAIL`. The `make: *** Error 1` under it is only `make` reporting that the verdict exited non-zero

1. Icarus prints 28 `sorry:` lines

Compile the design the way the labs do:

```
cd labs/common/rtl
iverilog -g2012 -o /tmp/sp.vvp tt_um_ieeeuoftasic_simproc.sv simproc_system.sv tb.v
```

You should see twenty-eight lines like this, and nothing else:

```
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]').
```

To confirm the count, send the output to a file and count lines in it:

```
iverilog -g2012 -o /tmp/sp.vvp tt_um_ieeeuoftasic_simproc.sv simproc_system.sv tb.v
> /tmp/ic.log 2>&1
echo "exit code: $?"                    # run it UNPIPED, or you get grep's exit code
grep -c 'sorry:' /tmp/ic.log              # 28
grep -c 'error:' /tmp/ic.log              # 0
wc -l < /tmp/ic.log                       # 28 -- there is nothing else
grep -oE '^[^ ]+:[0-9]+:' /tmp/ic.log | sort -u | wc -l    # 22 distinct locations
```

> After a pipeline, `$?` holds the exit code of the last command in the pipeline, not the compiler's. That's why the commands above write `cmd > file 2>&1`, check `$?` on its own line, and only then grep the file. If you write `cmd | grep -c ...` instead, a compiler failure comes out as a count of zero that looks perfectly plausible.

`sorry:` lines	28
Distinct source locations	22 (six lines produce two each)

Total lines of output	28, so there is nothing else
`error:` lines	0
`warning:` lines	0
Exit code	0

Icarus means these as notes rather than warnings. Where the code selects `instr_reg_out[3:0]` inside an `always_comb`, Icarus makes the process sensitive to all eight bits instead of the four that were selected. That gives the block more sensitivity than it needs rather than less, so it re-evaluates more often than it strictly has to, and the simulation is still correct. All 22 locations are in the decoder, between lines 543 and 677, which is where a processor slices an instruction word into fields.

> Rule: a real Icarus failure starts a line with `error:` and stops. `sorry:` means "I am doing this a less clever way"; `error:` means "I am not doing this".

For contrast, here is a real failure. If you drop `-g2012`, the flag that turns on SystemVerilog-2012, the same three files produce:

```
simproc_system.sv:122: syntax error
simproc_system.sv:122: error: Invalid module instantiation
```

	With `-g2012`	Without it
Lines of output	28	414
`error:` lines	0	288
Exit code	0	152
`/tmp/sp.vvp`	Written	Not written

If you ever see `syntax error` on a line of perfectly good SystemVerilog, the flag you're missing is `-g2012`. You only have to remember it when you run `iverilog` by hand, because `sim.mk` already passes it.

2. Verilator prints 12 warnings and refuses to build

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

Verilator prints twelve warnings in 0.077 seconds and then stops with:

```
%Error: Exiting due to 12 warning(s)
```

The exit code is 1. To count the warnings by kind:

```
verilator --lint-only --timescale 1ns/1ps --top-module tb \
  rtl/tt_um_ieeeuoftasic_simproc.sv rtl/simproc_system.sv rtl/tb.v > /tmp/v.log 2>&1
grep '^%Warning' /tmp/v.log | sed 's/^%Warning-\([A-Z]*\).*/\1/' | sort | uniq -c
```

```
      2 CASEINCOMPLETE
      4 WIDTHEXPAND
      6 WIDTHTRUNC
```

Five of the twelve say `Bit extraction`, and they're the only ones that are findings rather than complaints about tidiness:

Line	Module	What it says
`simproc_system.sv:848`	`dp_ram`	`Bit extraction of array[63:0] requires 6 bit index, not 8 bits`
`simproc_system.sv:851`	`dp_ram`	Same, the other write port
`simproc_system.sv:856`	`dp_ram`	Same, read port A
`simproc_system.sv:857`	`dp_ram`	Same, read port B
`simproc_system.sv:1022`	`UART_RX`	`Bit extraction of var[7:0] requires 3 bit index, not 4 bits`

Four of the five are the same defect, reported at four lines: the RAM has 64 entries and the address bus is 8 bits wide, so addresses 64–255 are off the end of the array. Verilator found it just by reading the file, before it simulated anything. Icarus never mentions it, so under Icarus you only see it at run time, as eight X bits on a pin, and only if your stimulus happens to read, say, address 100. Two simulators measures that run-time side, and you run it yourself in Lab 02.

Verilator treats warnings as fatal by default. That is a reasonable policy for code you own and an impossible one for code you have been handed to test, so `sim.mk` passes `-Wno-fatal` when `SIM=verilator`. You will see the same twelve warnings scroll past during a normal `make sim SIM=verilator`, and then the build finishes.

> Rule: with `-Wno-fatal`, a `%Error: Exiting due to N warning(s)` line at the end of a build that still produced a binary is just noise. Without `-Wno-fatal`, the same line means no binary was produced and nothing ran. Either way, look for the results table rather than for the word `Error`.
>
> Don't take `-Wno-fatal` to mean that the warnings don't matter, though. It means this design doesn't lint clean, and that is itself a finding.

`-Wall`: 19 warnings, or 27 with the testbench

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

With `-Wall`, the design on its own gives 19 warnings:

Kind	Count	Is it a finding?
`WIDTHTRUNC`	6	Yes: the five `Bit extraction` warnings, plus one slip in a packet width
`WIDTHEXPAND`	4	No: `index <= 1'b0` assigning 1 bit to a 4-bit register
`UNUSEDSIGNAL`	4	No
`CASEINCOMPLETE`	2	Worth a look, because an incomplete `case` is how latches and X get into a design
`UNUSEDPARAM`	1	No
`PINCONNECTEMPTY`	1	No
`DECLFILENAME`	1	No: it only says the module isn't named after its file

6 + 4 + 4 + 2 + 1 + 1 + 1 = 19.

If you add the course's `rtl/tb.v` and make `tb` the top module, as the twelve-warning command in section 2 does, `-Wall` reports 27:

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

The extra eight are 5 `UNDRIVEN` and 3 `UNUSEDSIGNAL`, all inside `tb.v`, which is a testbench wrapper with pins it deliberately doesn't drive. 19 + 8 = 27, and checking that the counts add up like this is how you know you haven't found something new by accident. Adding `tb.v` to the first command and keeping `--top-module tt_um_ieeeuoftasic_simproc` still gives 19, because `tb.v` sits above that top module rather than under it.

> Rule for `-Wall` output: sort it by warning kind and look at the width and case warnings first, because `WIDTHTRUNC` and `CASEINCOMPLETE` describe the design doing something you didn't write. `UNUSED*`, `DECLFILENAME` and `PINCONNECTEMPTY` only describe a design that isn't tidy.

Ten more you can cause yourself

If you leave `--timescale 1ns/1ps` off the lint command, you get ten extra `TIMESCALEMOD` warnings, one per module, because `tb.v` has a timescale and the RTL files don't. cocotb's own makefiles pass `--timescale`, so you only see these when you run `verilator` by hand, and they're caused by your command line rather than by anything in the design.

3. cocotb deprecation warnings

In a cocotb log, `WARNING` is only the log level of the message, and it says nothing about whether the test passed. These are the warnings you'll actually see:

```
0.00ns WARNING  py.warnings  .../test.py:24: DeprecationWarning: The 'units' argument
                             has been renamed to 'unit'.
0.00ns WARNING  py.warnings  DeprecationWarning: `logic_array.integer` getter is
                             deprecated. Use `logic_array.to_unsigned()` instead.
               Using MODULE is deprecated, please use COCOTB_TEST_MODULES instead.
```

All three old spellings still work: cocotb 2.0.1 accepts `units=`, `MODULE=` and `.integer`, does the right thing with each, and prints a warning. The test runs and passes, and the only cost is a noisier log.

> Don't go looking for a failure that isn't there. If a page tells you that `MODULE=` causes a silent run with no tests, it's wrong: we measured it in this container, and it works, with a warning. Write the new spellings anyway, but if your test is failing, a `DeprecationWarning` isn't the reason.
>
> Some old forms are hard errors, though, among them `cocotb.fork`, which gives `AttributeError`, `@cocotb.coroutine`, which gives `AttributeError`, and `dut.sig <= 1`, which gives `TypeError`. Four runnable demonstrations ship in `labs/cocotb-intro/oldapi/`, and cocotb 2.x and older tutorials has the full breakdown.

Some other lines look like problems but aren't:

Line	What it is
`-.--ns INFO gpi ... VPI registered`	cocotb attaching to the simulator. `-.--ns` means the line was printed before simulation time started
`Seeding Python random module with 1787752037`	The seed for this run, so you can reproduce it
`- :0: Verilog $finish`	Verilator's normal end-of-simulation line
`sys:1: RuntimeWarning: coroutine '_do_writes' was never awaited`	Python cleaning up cocotb's writer task at shutdown

4. `make: *** Error 1` after a failed verdict

```
 VERDICT: FAIL — your testbench passes B03_zero_flag_low_nibble too.
   A test that passes on a broken core is not testing that core.
========================================================================

  The verdict above is FAIL, so this target exits non-zero and make
  prints an error line under this one. That is the verdict talking,
  not a broken Makefile.
make: *** [lab.mk:66: check] Error 1
```

A failed verdict exits non-zero on purpose, so the labs can be used from a script or a CI job without anyone grepping the text, and `make` reports a non-zero exit from anything it runs. The lab prints the paragraph above so that you can tell a failed verdict from a broken Makefile.

> Rule: read the `VERDICT:` line, not the `make:` line. If there is a `VERDICT:` line at all, the machinery worked, whatever the verdict says.

The same applies to a cocotb test that fails: the results table still prints, `FAIL` is non-zero, and `make` reports `Error 2` from `Makefile.icarus`. That's just the test failing, which is what you asked it to do when it finds a problem.

---

5. Green output that ran nothing

Everything above is noise that looks like failure. The runs in this section are the opposite: their output looks fine even though nothing was tested, and they're worse, because nothing in them makes you look twice.

Running `python3 -m pytest test_example.py` gives:

```
collected 0 items
============================ no tests ran in 0.10s =============================
```

The exit code is 5. It looks like the obvious command to run and nothing in the output is red, but it ran nothing, because a cocotb test isn't a pytest test.

Running `python3 test_example.py` prints nothing at all and exits with 0. A cocotb test isn't a standalone Python program either: it needs a simulator running underneath it, and here only `make` starts one.

The third case is a typo in `COCOTB_TESTCASE`:

```
make sim TB=test_example COCOTB_TESTCASE=chekcing
```

```
0.00ns WARNING  cocotb.regression   No tests left after filtering with: chekcing$
```

The exit code is 0, there's no results table, and `build/results.xml` contains six `<skipped/>` entries. The simulator started and cocotb attached, but the name you gave matches no test, so every test was filtered out.

> Rule for all three: a run that tested something prints a results table ending in `TESTS=n PASS=n FAIL=0 SKIP=0`. No table means no tests ran, whatever the exit code says. Check the table, then check that `n` is the number you expected.

If the table says `TESTS=5` when you wrote six tests, that's the same failure in a quieter form.

Checklist

Before you decide that a run was fine, check these:

1. Is there a results table? If there isn't, no tests ran.
2. Does `TESTS=n` match the number of `@cocotb.test()` functions you wrote?

3. Is `FAIL=0`?
4. If a verdict target ran, does it say `VERDICT: PASS`?
5. Did the build produce something: a `.vvp`, or a `Vtop`?

None of these checks is the exit code, and none of them is the absence of the word `Error`.

Everything on this page comes from the harness in `labs/common`, documented call by call in The SimProc harness.