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.