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`

File	Result
`oldapi/test_1x_fork.py`	`AttributeError: module 'cocotb' has no attribute 'fork'`, `TESTS=1 PASS=0 FAIL=1`
`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.