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.