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.