From cocotb tests to pyuvm components

This page answers: *I can write a cocotb test. What changes when the same checks are written as pyuvm components?*

Python is still the language and cocotb still runs the test, so `dut.A.value = a`, `await Timer(1, unit="ns")` and `int(dut.ALUout.value)` mean in a pyuvm testbench what they meant in your DD104 tests. DD104 ran its tests on Icarus, and this course runs everything on Verilator, the cocotb test and the pyuvm testbench alike. pyuvm changes how the test is organised. The sections below start from DD104's `t01` written as a single pyuvm test, and end with the pieces that Lab 01 splits it into. In Lab 01, Part 3 you'll read the whole pyuvm testbench, and in Part 5 you'll extend its scoreboard.

We ran every example on this page in the course image. The source is in `guide-examples/from-cocotb-tests-to-pyuvm-components/`, and `guide-examples/README.md` gives the command for each output, in its section for this page.

DD104's t01 as one pyuvm test

`cocotb/test_alu.py` in Lab 01 is DD104 Lab 03's reference test for SimProc's ALU. Its first test, `t01`, checks one vector for each of the six operations, with two helpers: `spec()` works out what the ALU should produce, and `alu()` drives one vector, waits 1 ns and reads the three outputs back. The test itself is a loop over the six vectors:

```
@cocotb.test()
async def t01_one_vector_per_operation(dut):
    ...
    for op, a, b in cases:
        got = await alu(dut, op, a, b)
        want = spec(op, a, b)
        assert got == want, (
            f"{OPS[op]}(0x{a:02X}, 0x{b:02X}) gave out=0x{got[0]:02X} "
            ...)
```

`T01Test`, in the example's `components_demo.py`, is the same loop, with the same `spec()`, `alu()` and six vectors, written as a pyuvm test:

```
@pyuvm.test()
class T01Test(BaseTest):
    async def run_phase(self):
        self.raise_objection()
        dut = cocotb.top
        for op, a, b in CASES:
            got = await alu(dut, op, a, b)
            want = spec(op, a, b)
            if got != want:
                self.logger.error(
                    f"{OPS[op]}(0x{a:02X}, 0x{b:02X}) gave out=0x{got[0]:02X} "
                    f"N={got[1]} Z={got[2]}, expected out=0x{want[0]:02X} "
                    f"N={want[1]} Z={want[2]}")
        self.logger.info(f"[T01] {len(CASES)} directed vectors checked")
        self.drop_objection()
```

Four things changed:

- **The test is a class.** `@pyuvm.test()` wraps the class in a cocotb test with the class's name. cocotb lists that test as `components_demo.T01Test`, and `make run TEST=T01Test` picks it. The class derives from the course's `BaseTest`, which derives from pyuvm's `uvm_test`.
- **There's no `dut` argument.** `cocotb.top` is the same handle to the design's top level, which here is the ALU. A test can use it directly, as this one does, but a component deeper in the tree gets its handle from `ConfigDB`, as From one test to components shows.
- **The body is `run_phase()`, between `raise_objection()` and `drop_objection()`.** pyuvm starts `run_phase()` as a cocotb task and ends the run phase when the last objection is dropped. The section on objections shows what happens without them.
- **The check logs an error instead of asserting.** The loop goes on after a mismatch, so one run checks every vector, and `BaseTest` fails the test at the end if anything logged an error.

On the correct ALU the test checks its six vectors and passes:

```
      6.00ns INFO      ..omponents/components_demo.py(74) [uvm_test_top]: [T01] 6 directed
vectors checked
DD105-B BaseTest: 0 errors logged, no bound assertions
```

Each of pyuvm's log lines names the file and line it came from (cocotb shortens a long path at the front, hence the `..`) and gives the full name of the component that logged it. pyuvm names every test `uvm_test_top`, so here that component is the test.

Bug A01 makes `SUB` compute `B - A`. With it switched on, the second vector fails, the loop carries on through the sixth, and `BaseTest` ends the test with an `AssertionError` that quotes the first error:

```
     2.00ns ERROR    ..omponents/components_demo.py(70) [uvm_test_top]: SUB(0xC8, 0x3C) gave
out=0x74 N=0 Z=0, expected out=0x8C N=1 Z=0
      6.00ns INFO      ..omponents/components_demo.py(74) [uvm_test_top]: [T01] 6 directed
vectors checked
DD105-B BaseTest: 1 error logged, no bound assertions
```

```
 caught: 1 error   0.28 s   log: build/runs/T01Test-s1-A01/run.log
  T01Test: AssertionError: 1 error logged, the first: SUB(0xC8, 0x3C) gave out=0x74 N=0 Z=0,
expected out=0x8C N
```

The error message is word for word the one that `t01`'s `assert` builds. In Lab 01, Part 2 the cocotb test's `t01` stopped at the same vector with the same message, because an `assert` that fails ends the test there.

Logged errors and BaseTest

Nothing in pyuvm turns a logged error into a failing test. pyuvm reports through Python's `logging`, and every component's logger has `propagate = False`, so it doesn't pass its records on to any logger that would count them. `self.logger.error()` prints a line and does nothing else. The example's `T01UvmTest` has `T01Test`'s `run_phase` line for line, in a class derived from pyuvm's `uvm_test` instead. With A01 switched on it logs the same error, and cocotb says it passed:

```
     2.00ns ERROR    ..omponents/components_demo.py(89) [uvm_test_top]: SUB(0xC8, 0x3C) gave
out=0x74 N=0 Z=0, expected out=0x8C N=1 Z=0
      6.00ns INFO      ..omponents/components_demo.py(93) [uvm_test_top]: [T01] 6 directed
```

```
vectors checked
     6.00ns INFO     cocotb.regression                  components_demo.T01UvmTest passed
```

The course's verdict grades the run `BROKEN` instead, because the log has no `BaseTest` line:

```
 BROKEN: no BaseTest line   0.27 s   log: build/runs/T01UvmTest-s1-A01/run.log
   the test passed and the log has no 'DD105-B BaseTest:' line
```

So every pyuvm test you write in this course derives from `BaseTest`, or from `RouterBaseTest` for the router. If you override `final_phase`, which is where `BaseTest` checks its error count and prints its line, call `super().final_phase()` from yours.

Phases

A cocotb test is one coroutine, and cocotb runs it from top to bottom. A pyuvm test is the root of a tree of components. pyuvm makes the test and then calls a fixed sequence of methods, the **phases**, on every component in the tree, finishing each phase everywhere before it starts the next. A component overrides the phases it needs.

The example's `PhaseOrderTest` builds a small tree. The test has one child, an environment called `env`, and `env` has two, `drv` and `sb`. Every component prints a line in each phase. The test raises an objection, waits 35 ns and drops it, and `drv`'s `run_phase` counts a tick every 10 ns and never returns. This is everything the run prints between cocotb's `running` and `passed` lines:

```
   0 ns  build_phase               uvm_test_top
   0 ns  build_phase               uvm_test_top.env
   0 ns  build_phase               uvm_test_top.env.drv
   0 ns  build_phase               uvm_test_top.env.sb
   0 ns  connect_phase             uvm_test_top.env.drv
   0 ns  connect_phase             uvm_test_top.env.sb
   0 ns  connect_phase             uvm_test_top.env
   0 ns  connect_phase             uvm_test_top
   0 ns  end_of_elaboration_phase  uvm_test_top
   0 ns  end_of_elaboration_phase  uvm_test_top.env
   0 ns  end_of_elaboration_phase  uvm_test_top.env.drv
   0 ns  end_of_elaboration_phase  uvm_test_top.env.sb
   0 ns  start_of_simulation_phase uvm_test_top
   0 ns  start_of_simulation_phase uvm_test_top.env
   0 ns  start_of_simulation_phase uvm_test_top.env.drv
   0 ns  start_of_simulation_phase uvm_test_top.env.sb
   0 ns  run_phase starts          uvm_test_top.env.drv
   0 ns  run_phase starts          uvm_test_top.env.sb
   0 ns  run_phase starts          uvm_test_top.env
   0 ns  run_phase starts          uvm_test_top
  35 ns  drop_objection()          uvm_test_top
  35 ns  extract_phase             uvm_test_top
  35 ns  extract_phase             uvm_test_top.env
  35 ns  extract_phase             uvm_test_top.env.drv
  35 ns  extract_phase             uvm_test_top.env.sb
  35 ns  check_phase               uvm_test_top
  35 ns  check_phase               uvm_test_top.env
```

```
  35 ns  check_phase               uvm_test_top.env.drv
  35 ns  check_phase               uvm_test_top.env.sb
  35 ns  report_phase              uvm_test_top
  35 ns  report_phase              uvm_test_top.env
  35 ns  report_phase              uvm_test_top.env.drv
           uvm_test_top.env.drv counted 3 ticks
  35 ns  report_phase              uvm_test_top.env.sb
  35 ns  final_phase               uvm_test_top
DD105-B BaseTest: 0 errors logged, no bound assertions
  35 ns  final_phase               uvm_test_top.env
  35 ns  final_phase               uvm_test_top.env.drv
  35 ns  final_phase               uvm_test_top.env.sb
```

- **`build_phase`** goes from the top down, because a component's children don't exist until its own `build_phase` makes them. The test's `build_phase` makes `env`, and only then can pyuvm call `env`'s, which makes `drv` and `sb`. This is the phase where a component makes its children and reads what it needs from `ConfigDB`.
- **`connect_phase`** goes from the bottom up. By then every component exists, so it's where a component connects the ports of the components below it to each other.
- **`end_of_elaboration_phase` and `start_of_simulation_phase`** come before simulated time moves. `BaseTest` uses them to attach its error counter to every component's logger.
- **`run_phase`** is the only phase that takes simulated time. pyuvm starts every component's `run_phase` as a cocotb task, all at 0 ns, and the run phase lasts until the last objection is dropped, here at 35 ns. `drv`'s `run_phase` was still running then and was never waited for; it had counted 3 ticks, at 10, 20 and 30 ns.
- **`extract_phase`, `check_phase`, `report_phase` and `final_phase`** take no time, and run once the run phase is over. They're where a component gathers and checks its results and prints its summary, and `BaseTest`'s line comes from the test's `final_phase`.

In this run `connect_phase` went from the bottom up, the `run_phase` tasks started in the same order, and every other phase went from the top down. Those are the orders that pyuvm 4.0.1′s source sets for each phase, in `s09_phasing.py`. SystemVerilog UVM runs every phase from the bottom up except `build_phase` and `final_phase`, so a SystemVerilog UVM reference will give you the opposite order for `end_of_elaboration_phase`, `start_of_simulation_phase`, `extract_phase`, `check_phase` and `report_phase`.

pyuvm also has only these nine phases. It leaves out SystemVerilog UVM's run-time phases (reset, configure, main and shutdown), which run alongside `run_phase` there. SystemVerilog UVM can keep the run phase open for a set time after the last objection is dropped, which it calls a drain time. pyuvm has no drain time, so its run phase ends at the drop, as it did at 35 ns here.

Objections

A cocotb test ends when its function returns. In pyuvm, `run_phase` returning doesn't end anything, because every component's `run_phase` is a separate task and some of them, like `drv`'s, never return. The run phase ends instead when no component holds an objection. A test raises one before its stimulus and drops it after.

`NoObjectionTest` has the same tree, and its `run_phase` waits 35 ns and then prints a line, without raising an objection first. pyuvm finds no objection raised, warns, and ends the run phase at 0 ns:

```
   0 ns  run_phase starts           uvm_test_top.env.drv
   0 ns  run_phase starts           uvm_test_top.env.sb
   0 ns  run_phase starts           uvm_test_top.env
   0 ns  run_phase starts           uvm_test_top
```

```
     0.00ns WARNING  root                           You did not call self.raise_objection()
in any run_phase
   0 ns  extract_phase             uvm_test_top
```

The rest of the phases follow at 0 ns, `drv` counts 0 ticks, the line that the test meant to print after 35 ns never appears, and the test passes, with `ok: 0 errors`, although it checked nothing. Lab 01′s environment guards against the same mistake in its `check_phase`, which logs `[COUNT] the scoreboard checked no vectors` if the scoreboard ends the run with nothing checked.

From one test to components

Lab 01′s pyuvm testbench splits `t01`′s loop into the pieces that the What a framework buys you diagram shows. Once the pieces are separate classes, a vector has to get from one piece to the next, and the driver and the monitor have to reach the pins. pyuvm has a part for each of these jobs, and its factory makes the pieces themselves.

A transaction instead of tuples. `t01` passes `(op, a, b)` into `alu()` and gets a tuple back. In Lab 01 a vector is an `AluItem`, a class derived from `uvm_sequence_item` that has fields for the stimulus and for the ALU's answer. The sequence sets the stimulus in each item it sends. The monitor makes an `AluItem` of its own for each vector it reads back, and fills in every field from the pins.

A sequence hands items to the driver. The sequence makes each item, waits for the driver to be ready for it, and hands it over. `AluRandSeq.body()` makes its 2000 random vectors in a loop:

```python
for _ in range(self.n):
    req = AluItem.create("req")
    await self.start_item(req)
    req.randomize()
    await self.finish_item(req)
```

and `AluDriver.run_phase()` takes the items at the other end, one at a time, with `await self.seq_item_port.get_next_item()`, drives the pins as `alu()` did, and calls `self.seq_item_port.item_done()` when the vector has been on the pins for its 10 ns. The sequencer between them is pyuvm's own `uvm_sequencer`, used as it comes, and the agent connects the driver to it in the agent's `connect_phase`.

An analysis port carries what the monitor sees. `AluMonitor` makes the port in its `build_phase`, with `self.ap = uvm_analysis_port("ap", self)`, and calls `self.ap.write(t)` for each vector it reads back. The scoreboard derives from `uvm_subscriber`, whose `write()` method receives every item that's written to the port it's connected to. `AluEnv.connect_phase()` makes that connection, with `self.agt.mon.ap.connect(self.sb.analysis_export)`. The monitor doesn't know the scoreboard exists, so a coverage collector could subscribe to the same port without the monitor changing.

ConfigDB carries the pins. No component is handed `dut` the way `alu()` is. `AluBaseTest.build_phase()` puts an `AluIf` holding `cocotb.top` into `ConfigDB` under the name `"vif"`, for every path (`"*"`), and the driver and the monitor fetch it in their own `build_phase` with `ConfigDB().get(self, "", "vif")`. A `get()` of a name that nobody set raises `UVMConfigItemNotFound`, which the course's verdict counts as `BROKEN`.

The factory makes the components and the items. `AluAgent.build_phase()` makes its driver with `AluDriver.create("drv", self)` rather than `AluDriver("drv", self)`, and its sequencer and monitor the same way. The two calls make the same object until a test asks the factory to make a different class in its place. Lab 04 does that with a packet class, `NoroutePkt`, in place of `RouterPkt`. A component's `create()` takes only a name and a parent, so nothing else, `dut` included, can be passed to the component when it's made.

cocotb and pyuvm side by side

In a cocotb test	In a pyuvm testbench
`@cocotb.test()` on an `async def` function	`@pyuvm.test()` on a class derived from `BaseTest`
`dut`, the test's argument	`cocotb.top`, or an object from `ConfigDB`
the test function's body	the test's `run_phase()`, between `raise_objection()` and `drop_objection()`
the test ends when the function returns	the run phase ends when the last objection is dropped; the other phases follow
`assert got == want, "message"`	`self.logger.error("message")`, and `BaseTest` fails the test at the end
a tuple of values	a `uvm_sequence_item`
a loop that makes vectors	a `uvm_sequence`'s `body()`
a coroutine that drives the pins	a `uvm_driver`'s `run_phase()`
reading the outputs back	a `uvm_monitor` that writes each item to its analysis port
the model and the comparison	a `uvm_subscriber`'s `write()`
`cocotb.start_soon(coro())`	the same, and pyuvm starts every `run_phase` this way
a test function, picked by its name with `COCOTB_TEST_FILTER`	a test class, picked by its class name: `make run TEST=AluDirectedTest` sets `COCOTB_TEST_FILTER` for you
`random.randrange(256)`	the same. Lab 03 adds pyvsc's constraints to a plain cocotb testbench, and the pyuvm testbenches from Lab 04 on keep them

Check yourself

1. You write a pyuvm test that logs an error on a broken ALU, and cocotb's table says `PASS`. What's the most likely mistake?

   Answer: A1, in Answers at the end.

2. A test's `run_phase` starts a sequence and awaits it, but never raises an objection. What does the run do?

   Answer: A2, in Answers at the end.

3. Why must `build_phase` go from the top down, while ports are connected in `connect_phase` rather than in `build_phase`?

   Answer: A3, in Answers at the end.

4. In `PhaseOrderTest`, `drv`'s `run_phase` never returns, yet the test ended at 35 ns. Why?

   Answer: A4, in Answers at the end.

Further reading

- pyuvm's README at release 4.0.1, "The pyuvm testbench": an ALU testbench built from the same kinds of pieces as Lab 01's, with notes on where pyuvm differs from SystemVerilog UVM and why.

- ChipVerify's UVM Phases. Its code is SystemVerilog. Correction for pyuvm 4.0.1: it says that only `build_phase` and `final_phase` run from the top down, but in pyuvm every phase except `connect_phase` and the start of `run_phase` does, as `PhaseOrderTest` shows. It also lists the run-time phases (reset, configure, main and shutdown), which pyuvm doesn't have.
- ChipVerify's UVM Objection. Its code is SystemVerilog. Its point that a run phase with no objection ends at time 0 holds in pyuvm too, and pyuvm also prints the warning shown above. Correction for pyuvm 4.0.1: `raise_objection()` is a method of the component and takes no argument, and there is no drain time.

Next: Lab 01: One ALU, two frameworks.

Answers

A1. Check yourself

The test doesn't derive from `BaseTest`, or it overrides `final_phase` without calling `super().final_phase()`. A logged error fails nothing on its own, as `T01UvmTest` showed. The course's verdict calls such a run `BROKEN: no BaseTest line`.

A2. Check yourself

pyuvm warns `You did not call self.raise_objection() in any run_phase` and ends the run phase at 0 ns, before the sequence has sent anything. The later phases run at 0 ns and the test passes, having checked nothing, unless a check such as Lab 01′s `[COUNT]` in `check_phase` notices.

A3. Check yourself

A component's children are made in its own `build_phase`, so pyuvm can't call a child's `build_phase` before its parent's has run. The two components that a parent connects might not both exist until every `build_phase` has run. `AluEnv`, for example, connects `agt.mon`'s analysis port to `sb`, and `agt.mon` is made in `agt`'s own `build_phase`, after `AluEnv`'s has finished. `connect_phase` comes after every `build_phase`, so by then both ends exist.

A4. Check yourself

The run phase ends when the last objection is dropped, which the test did at 35 ns. pyuvm doesn't wait for the components' `run_phase` tasks to return; it goes on to the extract, check, report and final phases, and cocotb ends the tasks that are still running when the test ends.