From cocotb to SystemVerilog classes

This page answers: *I can write a cocotb test. What changes when the testbench is written in SystemVerilog instead?*

A cocotb test and a SystemVerilog class testbench do the same work: they drive the design's inputs, read its outputs and decide whether the outputs are right. Most of the Python you wrote in DD104 has a direct SystemVerilog counterpart. The main differences are that the testbench runs inside the simulator rather than beside it, that a failed check is reported differently, and that every value has a fixed width. This page goes through them with DD104′s `t01` translated line by line, and then with two short examples, one on widths and one on string literals. In Lab 01, Part 3 you'll read a whole UVM testbench, which is written in SystemVerilog classes, and in Part 5 you'll extend its scoreboard. From Lab 02 on you write classes of your own.

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

Where the testbench runs

In DD104 your test was a Python program running beside the simulator. Icarus or Verilator ran the design, and cocotb reached into it through VPI. The two took turns. Every `await` handed control back to the simulator, and your test carried on once the thing it was waiting for had happened.

A SystemVerilog testbench is part of the simulation. On Verilator, the testbench is translated into C++ along with the design, and the two are compiled into one program, which is the simulator. That has three consequences for how you work:

- **Every change to the testbench means a rebuild.** You can rerun a Python test as soon as you've changed it, because Python isn't compiled. A changed SystemVerilog class has to go through Verilator and g++ again, and because the UVM library is compiled in with it, a rebuild in Lab 01 took from 26 s to 99 s, depending on what was changed. What a framework buys you has Lab 01′s numbers and explains why they differ.
- **Mistakes show up at two different times.** A type error, a misspelt name or a missing semicolon stops Verilator before anything runs, usually within a few seconds. Everything else shows up when the simulator runs. Every lab has `make lint`, which runs only Verilator's front end and takes seconds, so you can find the first kind of mistake without starting a full build.
- **Anything chosen when the simulator starts needs no rebuild.** The test, the seed, the planted bug and the test's own knobs are all given as plusargs, the arguments on the simulator's command line that start with `+`, such as `+UVM_TESTNAME`, `+verilator+seed+` and `+BUG`. Trying a different one takes a second or so.

DD104′s t01 in SystemVerilog

`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. It uses two helpers. `spec()` works out what the ALU should produce, and `alu()` drives one vector and reads the three outputs back. Here they are, with most of `spec()`'s cases left out:

```
def spec(op, a, b):
    if op == ADD:
        out = (a + b) & 0xFF
    elif op == SUB:
        out = (a - b) & 0xFF
    ...
    return out, (out >> 7) & 1, int(out == 0)

async def alu(dut, op, a, b):
    dut.ALUop.value = op
```

```
    dut.A.value = a
    dut.B.value = b
    await Timer(1, unit="ns")
    return int(dut.ALUout.value), int(dut.N.value), int(dut.Z.value)
```

and the test calls them like this for each of its six 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} "
    f"N={got[1]} Z={got[2]}, expected out=0x{want[0]:02X} "
    f"N={want[1]} Z={want[2]}")
```

In SystemVerilog the same test is one class, `alu_t01`. In it, `spec()` becomes a function called `predict()`, and the tuple that `spec()` returns becomes a packed struct:

```
typedef enum bit [2:0] {ADD, SUB, OR, NAND, SHL, SHR} alu_op_e;

typedef struct packed {
  bit [7:0] out;
  bit       n;
  bit       z;
} alu_result_t;

function alu_result_t predict(alu_op_e op, bit [7:0] a, bit [7:0] b);
  alu_result_t r;
  case (op)
    ADD:     r.out = a + b;
    SUB:     r.out = a - b;
    ...
  endcase
  r.n = r.out[7];
  r.z = (r.out == 8'h00);
  return r;
endfunction
```

The coroutine `alu()` becomes a task, `drive_and_read()`, because a task is the kind of routine that's allowed to wait:

```
task drive_and_read(alu_op_e op, bit [7:0] a, bit [7:0] b, output alu_result_t got);
  vif.ALUop = op;
  vif.A     = a;
  vif.B     = b;
  #1;
  got = '{vif.ALUout, vif.N, vif.Z};
endtask
```

and the `assert` becomes an `if` with an `$error`, which takes a format string the way `$display` does:

```
drive_and_read(ops[i], as[i], bs[i], got);
want = predict(ops[i], as[i], bs[i]);
if (got != want)
  $error("%s(0x%02X, 0x%02X) gave out=0x%02X N=%0d Z=%0d, expected out=0x%02X N=%0d Z=%0d",
         ops[i].name(), as[i], bs[i], got.out, got.n, got.z, want.out, want.n, want.z);
```

Most lines translate one for one, and there are five differences:

- **Every argument and every variable has a type.** `op` is an `alu_op_e`, which is an enum. `ops[i].name()` returns the name of its value as a string, such as `"SUB"`, and that does the job of the `OPS` dictionary in the Python.
- **`predict()` has no `& 0xFF`.** `r.out` is 8 bits wide, so a sum that comes to 260 wraps to 4 as it's stored, the way the ALU's own adder wraps. The next section explains the width rule behind this, and how the same rule can make a model lose a carry it was meant to keep.
- **The class can't name the design's pins.** cocotb hands your test the design as `dut`. The classes of a testbench live in a package, and code in a package can't name anything in the module hierarchy, so the class reaches the pins through `vif` instead. `vif` is a *virtual interface*, a handle to an interface instance, and here the instance is the one that the top module connects to the ALU. `drive_and_read()` writes the pins directly and waits 1 ns, as `alu()` does, because this example copies what the cocotb test does. Lab 01's UVM testbench gives the same ALU an interface with a clock, which the ALU itself doesn't use, and from Lab 01 on every driver writes through a clocking block. Interfaces, clocking blocks and virtual interfaces explains clocking blocks.
- **`await Timer(1, unit="ns")` becomes `#1`.** The unit of `#1` is the time unit that the code is compiled with. The example's Makefile sets it to 1 ns with `--timescale 1ns/1ps`, and so does every lab's Makefile.
- **A task can wait and a function can't.** `predict()` takes no simulated time, so it can be a function. `drive_and_read()` waits 1 ns, so it has to be a task. Classes for testbenches shows the error you get if you put a delay in a function.

On the correct ALU the class checks its six vectors and prints its summary line:

```
t01: 6 directed vectors checked
```

With bug A01 switched on, where `SUB` computes `B - A`, it stops at the second vector:

```
[2000]        %Error:        alu_t01_pkg.sv:62:        Assertion        failed        in
alu_t01_pkg.alu_t01.t01_one_vector_per_operation.unnamedblk1: SUB(0xc8, 0x3c) gave out=0x74
N=0 Z=0, expected out=0x8c N=1 Z=0
%Error: guide-examples/from-cocotb-to-systemverilog-classes/alu_t01_pkg.sv:62: Verilog $stop
Aborting...
```

The message is the cocotb test's message, and three things about this log differ from a cocotb log:

- `[2000]` is the simulated time in units of the simulation's precision, which is 1 ps, so the error came at 2 ns. Each vector takes 1 ns, so 2 ns is the end of the second vector. Between `%Error:` and your text come the file and line of the `$error`, and then the scope it ran in, which ends with the task's name and an unnamed block.
- The hex digits are lower case. In Python `X` asks for upper-case digits, but in SystemVerilog `%X` means the same as `%x`, and Verilator printed lower case.
- The run stopped. A failed `assert` in cocotb ends that test, and cocotb moves on to the next one. Verilator's default is to stop the whole simulation at the first `$error`, which is why there's no summary line after it. In Lab 02 the Makefile adds `+verilator+error+limit+100` to every run so that a scoreboard can report more than one mismatch, and Part 3 of that lab explains it.

In a UVM testbench you report with `uvm_error instead, as Lab 01's scoreboard does. The macro sends your message to UVM's report server, which counts errors by ID and lets the run carry on.

Widths

Python integers have no width, which is why spec() has to mask its ADD, SUB, NAND and SHL results with & 0xFF, the four that could fall outside 0 to 255. In SystemVerilog every value has a width, and the width that an expression is computed at depends on its operands and on where its result goes. The example adds a = 200 and b = 60, both bit [7:0], in three places:

```
sum8 = a + b;               // sum8 is bit [7:0]
sum9 = a + b;               // sum9 is bit [8:0]
$display("%0d", a + b);
```

and prints:

```
sum8 = a + b      -> 4
sum9 = a + b      -> 260
$display(a + b)   -> 4
```

The two assignments have the same expression on the right and get different answers. In an assignment the operands are widened to the width of the left-hand side before the addition, so sum9 keeps the carry. Inside $display the expression has nothing to take its width from, so it's computed at the width of its widest operand, 8 bits, and the carry is lost. Both rules come from the SystemVerilog standard rather than from Verilator. Under the first rule, predict() gets the ALU's wrap-around without a mask, because r.out is 8 bits wide. The same rules are also how a model loses a carry it was meant to keep, so when a scoreboard reports mismatches only on results that overflow, check the widths in the model before you blame the design.

DD104 also taught you that Verilator has two states, 0 and 1, and no X. That still holds in a SystemVerilog testbench, whether you declare a variable bit or logic.

Strings

A string literal in SystemVerilog is a number, a packed array of 8-bit characters, until something makes it a string. Most of the time you won't notice, but you will in a ?: that picks between two literals when one of them is empty or shorter than the other. The example prints the same thing two ways:

```
$display("[%0d beat%s]", n, n == 1 ? "" : "s");
$display("[%0d beat%s]", n, n == 1 ? string'("") : string'("s"));
```

```
[1 beat ]
[1 beat]
```

In the first line ?: picks between two numbers. As a number, the empty literal "" is one byte of zeros, as wide as "s", so when n is 1 the result is a zero byte, and %s prints a zero byte as a space. A literal that's shorter than the other side of the ?: gets zero bytes added in front, and those print as spaces too. Casting both sides to string keeps them strings, and an empty string prints as nothing. That's why Lab 02's packet::sprint() writes string'("") and string'("s").

Python and SystemVerilog side by side

In cocotb and Python	In a SystemVerilog testbench
`class Packet:` with `__init__(self, ...)`	`class packet;` with `function new(...)`
`self.x`	`this.x`, or just `x`
`None`	`null`
a variable holding an object (a reference)	a handle; Classes for testbenches
`def f(...)`	`function`, which can't wait
`async def f(...)`	`task`, which can
`await Timer(1, unit="ns")`	`#1`
`await RisingEdge(dut.clk)`	`@(posedge clk)`, or `@(vif.cb)` through a clocking block
`dut.A.value = 5`	`vif.cb.A <= 5` through a clocking block, from Lab 01 on
`cocotb.start_soon(coro())`	`fork coro(); join_none`
`await First(a, b)`	`fork ... join_any`
`cocotb.triggers.Event`	`event`, with `->e` and `@e`
a `cocotb.queue.Queue` between coroutines	a `mailbox`
a list, with `append` and `pop(0)`	a queue, `int q[$]`, with `push_back` and `pop_front`
a dict	an associative array, `int counts[string]`
a tuple	a packed struct
`f"0x{a:02X}"`	`$sformatf("0x%02X", a)`
`assert got == want, "message"`	`if (got != want) $error("message")`, or `` `uvm_error `` in UVM
`random.randint(0, 255)`	`$urandom_range(255, 0)`, or `randomize()` with constraints, which Lab 01's `alu_item` uses and you write yourself from Lab 03 on
the `@cocotb.test()` functions in a module	UVM test classes, one chosen per run by `+UVM_TESTNAME=`

The rows about threads, events and mailboxes are the subject of Threads, mailboxes and events.

Check yourself

1. `spec()` masks its `ADD`, `SUB`, `NAND` and `SHL` results with `& 0xFF`, and `predict()` masks nothing. Why doesn't `predict()` need to, and in what kind of expression could a SystemVerilog model still lose a carry it needed?

   Answer: A1, in Answers at the end.

2. Your SystemVerilog check prints one `%Error` line, then `Verilog $stop` and `Aborting...`, and nothing after them. Is the rest of the test passing?

   Answer: A2, in Answers at the end.

3. Why is `drive_and_read()` a task, while `predict()` can be a function?

Answer: A3, in Answers at the end.

Further reading

ChipVerify's SystemVerilog pages cover each of these language features at more length. Most of their logs come from Cadence simulators rather than Verilator, and the course doesn't reuse their code, but their explanations of the language apply here too.

- Testbench Evolution: six styles of testbench, from a linear one to UVM. Your DD104 suites sit near the end of that list.
- bit, byte and int: the 2-state integer types.
- Strings.
- Enumeration.
- Structure.
- Queue and Associative Array.
- Class vs Struct: why assigning a class variable copies a handle, while assigning a struct copies its value.
- Testbench Example Adder: a class testbench without UVM, for a design much like the ALU. Its "buggy design" listing has the same code as the correct design, so the bug that the page describes isn't in the code it shows.

Next: Lab 01: One ALU, two testbenches.

Answers

A1. Check yourself

`r.out` is 8 bits wide, so a result is cut to 8 bits as it's stored, the way the ALU's adder cuts it. A carry goes missing when an expression is computed at the width of its operands with nothing wider to take its width from, as `a + b` was inside `$display`, or when you store the result in a variable that's too narrow, such as `sum8`. In `sum9`, which is 9 bits wide, the same sum kept its carry.

A2. Check yourself

You can't tell. Verilator stops a simulation at its first `$error` unless the run raises the limit with `+verilator+error+limit+N`, so the vectors after the failing one were never checked. In cocotb only the failing test would have stopped, and the next test would still have run.

A3. Check yourself

`drive_and_read()` waits 1 ns for the ALU to answer, and only a task may contain a delay or wait for an event. `predict()` computes its answer in zero time, so a function is enough.