DD105-B — Testbenches with UVM

In this course you build testbenches in Python for a small packet router and run them on Verilator. Each one has a driver that sends packets into the router, a monitor that turns the valid/ready handshakes on one of the router's streams into packets, and a scoreboard that predicts which packets should come out and checks the ones that do. In Lab 02 we give you the driver, and you write the monitor and the scoreboard as plain Python classes on cocotb. In Lab 03 you add functional coverage and constraints on the packet lengths to that testbench, and in Lab 04 you finish our port of it to pyuvm, a Python library of UVM's classes that runs on cocotb.

DD105-B is part of the UofT ASIC Team education materials, published at https://uoftasic.com/dd105-b/. The labs run in the workbench, which is the IIC-OSIC-TOOLS container that you start from the `workspace` clone you set up in IC101. In the workbench, the command `mod add dd105-b` clones this course's repository, and everything the labs run is in that repository's `labs/` folder. Lab 01, Part 1 walks you through `mod add dd105-b`.

Who this course is for

You have finished DD104, where you wrote cocotb tests in Python that caught all thirteen bugs planted in SimProc. Lab 01 starts on SimProc's ALU, which you tested in DD104 Lab 03, with the three bugs that DD104 planted in it.

From Lab 02 on you test the router. The router is about 730 lines of SystemVerilog. It was written for DD105-A, and this course has its own copy of it. It has one input stream and two output streams, each output with its own buffer. Its routing table, drop counters and interrupt enables are registers on an APB port. Fifteen one-line bugs are planted in it, and when you run a simulation you can switch one of them on. By the end of Lab 07 your pyuvm testbench catches all fifteen, and you have written a verification plan and a verification report. Everything you write for the testbenches is Python, apart from the assertions in Labs 05 and 07, which you write in SystemVerilog and the testbench binds into the router.

Track	Digital
Prerequisites	DD104, and through it IC101 and DD101 to DD103
Tools	Verilator 5.050, cocotb 2.0.1, pyuvm 4.0.1, Python 3.12.3 and Surfer 0.7.0, all in the workbench image `hpretl/iic-osic-tools:2026.08`. From Lab 03 on, the labs also use pyvsc 1.1.0 for constrained random and cocotb-coverage 2.0 for functional coverage. Neither is in the image, so in Lab 03 you install both into a Python venv with `make venv`.
Machine	The workbench you already have. Every time on these pages was measured with 2 CPUs and 3 GB. The router and its assertions build in about 8 s with an empty compiler cache, and an edit to a Python file rebuilds nothing. One run of a lab's graded test takes about 8 s in Lab 06 and about 12 s in Lab 07. The longest steps are the two regressions in Lab 07, which take about 9 minutes each.
Total time	21 h 10 min, which is the sum of the row times in the Course path below. The 14 h for the seven labs is the sum of the times that the lab writeups give. The 7 h 10 min for the eighteen guide pages is our estimate, at a minute for every 130 words, plus 2 minutes for each

figure and 1 minute for each 10 lines of code or output, with each Read row rounded to the nearest 5 minutes.

Course path

Work down this table in order. A Read row is a set of guide pages, and a Do row is one lab writeup. The guide pages are for reading only. Some of them show a command we ran and what it printed, but the lab writeups are the only pages that ask you to run anything.

#		What	Time
1	Read	Getting started · What a framework buys you · From cocotb tests to pyuvm components	1 h
2	Do	Lab 01: One ALU, two frameworks. Test SimProc's ALU with DD104′s cocotb test and with a pyuvm testbench, compare their lines, build times and run times with DD105-A's numbers for a SystemVerilog UVM testbench, and extend the pyuvm scoreboard until it catches A03, a bug in the zero flag.	1 h 15 min
3	Read	Python classes and cocotb concurrency · Driving and sampling on the clock	1 h 5 min
4	Do	Lab 02: A layered testbench without pyuvm. Write the monitor and the scoreboard of a plain-cocotb testbench for the router, catch B05 and B09 with it, and find out which part of the stimulus each catch depends on.	1 h 30 min
5	Read	Constrained random with pyvsc · Functional coverage you can read	50 min
6	Do	Lab 03: Constraints and coverage. Add functional	1 h 30 min

#		What	Time
		coverage to the Lab 02 testbench, weight its packet lengths with pyvsc constraints, and write the stimulus that reaches the two coverage holes where B07 and B08 are.	
7	Read	The pyuvm skeleton · Sequences, sequencers and drivers · Monitors, analysis ports and scoreboards · The factory and ConfigDB	1 h 35 min
8	Do	Lab 04: The same testbench in pyuvm. Finish our pyuvm port of the Lab 03 testbench by adding an agent for output 1, a test that stalls one output at a time, and a factory override that replaces each random packet with one that goes to a disabled route one time in four, so that the port catches B06 and B13.	2 h
9	Read	Assertions for interface rules · Reactive agents and backpressure · Virtual sequences	1 h 15 min
10	Do	Lab 05: Protocol rules and backpressure. Write three SystemVerilog assertions for the router's output rules, finish a sink driver that answers each output beat, and write a virtual sequence whose oversize packets reach B11.	2 h
11	Read	Register models in pyuvm · Regressions and seeds	50 min

#		What	Time
12	Do	Lab 06: The register block. Write the fields of seven registers, the adapter, the predictor, and the reset and bit-bash sequences for the router's pyuvm register model, then test the 246 addresses outside the register map and check `irq` in every cycle, so that the testbench catches B01 to B04 and B14.	2 h 15 min
13	Read	Verification plans and sign-off · Writing the verification report	35 min
14	Do	Lab 07: Capstone. Hunt three mystery bugs, catch B15 by writing the assertions for the rules about reset and a sequence that resets the router with packets in flight, and keep your verification plan, your report, your testbench and its results in a GitHub repository of your own.	3 h 30 min

Keep these reference pages open in a tab while you work. They're for looking things up rather than for reading from start to finish:

- The router in one page
- The packet router, the router's full specification
- The pyuvm register API, for Lab 06
- Getting the tools outside the course image

DD105-A

DD105-A has the same router, the same fifteen bugs and the same seven labs in the same order as this course, but its testbenches are written in SystemVerilog, with the UVM library, on Verilator 5.050. You take one of the two courses, not both. In DD105-A's Lab 01, a SystemVerilog UVM testbench for SimProc's ALU took 102 s to build, because Verilator compiled the whole UVM library along with the ALU and the testbench, and an edit to its scoreboard took 99 s to rebuild. In this course's Lab 01, the simulator for the pyuvm testbench of the same ALU builds in 7.3 s with an empty compiler cache, because Verilator compiles only the ALU. The testbench is Python, which cocotb imports each time the simulator starts, so an edit to its scoreboard rebuilds nothing.

Getting help

Ask in the team Discord. Say which lab and step you're on, paste the command you ran and what it printed, and attach the `run.log` from the run's folder in `build/runs/` if the question is about a simulation.