DD105-A — Testbenches with UVM

In this course you build testbenches for a small packet router. Each one has a driver that sends packets into the router, a monitor that turns a stream's valid/ready handshakes into packets, and a scoreboard that predicts what comes out and checks it. In Lab 02 the course gives you the driver, and you write the monitor and the scoreboard as plain SystemVerilog classes. In Lab 04 you finish a UVM version of the same testbench. Both testbenches run on Verilator.

DD105-A is part of the UofT ASIC Team education materials, published at https://uoftasic.com/dd105-a/. The labs run in the workbench: the IIC-OSIC-TOOLS container you start from the `workspace` clone you set up in IC101. You get this course's files with the workbench's `mod` command (`mod add dd105-a`), and Lab 01, Part 1 walks you through it.

Who this course is for

You have finished DD104, where you wrote cocotb tests in Python that found all thirteen bugs planted in SimProc. The testbenches you write in this course keep the stimulus apart from the checking, and the UVM ones let you choose the test when you run it, so that you can reuse and extend them. On your own runs you measure what UVM costs in lines of code and build time, and what each testbench catches.

The router was written for the course, in about 730 lines of SystemVerilog. 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 you switch one on when you run a simulation. Lab 01 starts on a design you already know, SimProc's ALU from DD104, with its three ALU bugs. By the end of Lab 07 your UVM environment catches all fifteen of the router's bugs, and you have written a verification plan and a verification report.

Track	Digital
Prerequisites	DD104, and through it IC101 and DD101 to DD103
Tools	Verilator 5.050, Yices 2.7.0 as its constraint solver, and Surfer for waveforms, all in the workbench image `hpretl/iic-osic-tools:2026.08`, and the UVM 2020-3.2 library, which ships in this course's repository. Lab 01 also runs cocotb 2.0.1 on Icarus Verilog 14.0.
Machine	The workbench you already have. Every time on these pages was measured with 2 CPUs and 3 GB: a testbench's first build takes about 2 minutes, a rebuild after an edit from about 10 s to about 2 minutes, and a run a few seconds. The longest step, Lab 07's regression, takes about 4 minutes.
Total time	About 21 hours, self-paced. The Course path rows below add up to 21 h 10 min: 15 h of labs and 6 h 10 min of reading. The lab times are the ones the lab writeups give. The reading times are our estimates, from the length of each page at about 130 words a minute, plus time for its figures and examples.

Course path

Work down this table in order. A Read row is a set of guide pages, which are for reading only and have no commands in them. A Do row is one lab writeup, and the lab writeups are where every command in the course lives. Each lab's "Before you start" section repeats which pages it expects you to have read.

#		What	Time
1	Read	Getting started · What a framework buys you · From cocotb to SystemVerilog classes	55 min
2	Do	Lab 01: One ALU, two testbenches. Test SimProc's ALU with DD104′s cocotb test and with a UVM testbench, measure what each costs, and extend the UVM scoreboard until it catches the flags bug.	1 h 30 min
3	Read	Classes for testbenches · Interfaces, clocking blocks and virtual interfaces · Threads, mailboxes and events	1 h 10 min
4	Do	Lab 02: A class testbench without UVM. Write the monitor and the scoreboard of a class testbench for the router, catch two planted bugs, and find out what each catch depends on.	2 h
5	Read	Constrained random on Verilator · Functional coverage you can read	40 min
6	Do	Lab 03: Constraints and coverage. Add a covergroup, constrain the packet until the simulator produces the lengths you ask for, and reach the two coverage holes where this lab's bugs hide.	1 h 30 min
7	Read	The UVM skeleton · Sequences, sequencers and	1 h 20 min

#		What	Time
		drivers · Monitors, analysis ports and scoreboards · The factory and config_db	
8	Do	Lab 04: The same testbench in UVM. Finish a UVM port of your Lab 03 testbench, inject errors with a factory override, and count what the structure cost in lines.	2 h 30 min
9	Read	Assertions for interface rules · Reactive agents and backpressure · Virtual sequences	1 h
10	Do	Lab 05: Protocol rules and backpressure. Bind assertions for the output rules into the router, finish a sink that answers each beat in its own time, and write a virtual sequence that catches the third bug.	2 h
11	Read	Register models · Regressions and seeds	35 min
12	Do	Lab 06: The register block. Give the testbench a register model with an adapter and a predictor, run UVM's reset and bit-bash sequences, and check the interrupt, until all five register bugs are caught.	2 h
13	Read	Verification plans and sign-off · Writing the verification report	30 min
14	Do	Lab 07: Capstone. Write a verification plan, run a seed regression, hunt mystery builds, and close the holes until all fifteen bugs are caught, then	3 h 30 min

#	What	Time
	write the verification report.	

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

- The router in one page
- The packet router, the router's full specification
- The UVM register layer
- Getting the tools outside the course image

pyuvm and DD105-B

This course uses UVM in SystemVerilog only. Its sibling course, DD105-B, covers pyuvm, the Python version of UVM, which is in the workbench image (version 4.0.1). In the research for these courses, a UVM environment for the ALU written in pyuvm built in 17 s, because Verilator then compiles only the design. The SystemVerilog version compiles the whole UVM library with it, which takes about 100 s from cold with this course's build settings.

Getting help

Ask in the team Discord; nobody expects you to work all of this out alone. 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 folder if the question is about a simulation.