Getting started

Read this page before you start Lab 01. You'll do the setup itself in Lab 01, Part 1, and in the parts after it you'll see the builds, runs and verdicts that this page describes.

Requirements

You set up the workbench in IC101, and you used the same container all through DD104. DD105-B runs in that container too. Everything in the course is simulation, so it needs no PDK and no synthesis flow.

Setting up takes three steps:

- Start the noVNC desktop from your `workspace` clone with `./scripts/start_vnc.sh` (`scripts\start_vnc.bat` on Windows). IC101: Launch the noVNC desktop gives the ports and the password, and it tells you where to look if the desktop page comes up blank.
- In a terminal on the desktop, load the environment with `. /foss/designs/common/.designinit`. It defines the workbench's `mod` command. Every new terminal starts without it, so you load it in each one.
- Run `mod add dd105-b`, the first time only.

`mod add dd105-b` clones the course into `/foss/designs/modules/dd105-b` and prints:

```
Cloning https://github.com/uoftasic/dd105-b.git
     into /foss/designs/modules/dd105-b
Cloning into '/foss/designs/modules/dd105-b'...
OK  module ready: /foss/designs/modules/dd105-b
    run: mod dd105-b
cwd: /foss/designs/modules/dd105-b
```

From then on, `mod dd105-b` takes you to that folder and prints the same `cwd:` line. Your `workspace` folder is bind-mounted into the container at `/foss/designs`, so the course folder and your files in it are on your own disk, and deleting the container doesn't delete your work.

Labs 01 and 02 run on the image alone. Verilator, cocotb, pyuvm and Surfer are all in it. From Lab 03 on, the labs also use pyvsc, for constrained-random stimulus, and cocotb-coverage, for functional coverage. Neither of those is in the image. In Lab 03, Part 1 you install them once, with `make venv`, into a Python virtual environment (a *venv*) that also sees the image's own packages. pip downloads them, so that step needs the network. The venv goes in your workspace, at `/foss/designs/.venvs/dd105-b`, so it survives a restart of the container. Ours came to 37 MB. Activating a venv in your shell doesn't change the Python that the simulator starts, so whenever the venv exists, the labs' shared makefile tells cocotb to start the venv's Python, in every run.

Each lab's makefile works out its own paths, so you can also run any lab with `make` alone, from a plain `git clone` of the course inside the image, without `.designinit` or `mod`. If a setup step fails late at night, you can still run `make` in a lab folder to find out whether the tools themselves work.

Versions

The course is pinned to the workbench image `hpretl/iic-osic-tools:2026.08`. These are the versions its tools report, and every exact number in the labs was recorded with them:

Tool	What its version check prints
Verilator	`Verilator 5.050 2026-07-01 rev v5.050`
cocotb	`2.0.1`
pyuvm	`4.0.1`

Tool	What its version check prints
Python	`Python 3.12.3`
Surfer	`surfer 0.7.0 (git: v0.7.0)`
pyvsc and cocotb-coverage, in the venv	`pyvsc 1.1.0` and `cocotb-coverage 2.0`, as `make venv` lists them

You'll check the first five rows yourself in Lab 01, Part 1, and `make venv` prints the last one in Lab 03. Keep pyuvm's version in mind when you read about pyuvm elsewhere, because pyuvm's online documentation describes a newer release than 4.0.1. Its example test calls `self.raise_objection("Description")`, but in 4.0.1 `raise_objection()` takes no argument. It also describes logging through `self.uvm_report`, which 4.0.1 doesn't have. This course goes by the pyuvm source that is installed in the image, in `/usr/local/lib/python3.12/dist-packages/pyuvm`.

How long things take

Verilator compiles only the design, not the testbench. Every testbench in this course is Python, which cocotb runs inside the simulator each time the simulator starts, so editing a testbench never rebuilds anything. These are the times we measured for Lab 01 in the course image, with 2 CPUs and 3 GB:

Job	Wall time
The first build, with an empty compiler cache	7.3 to 7.6 s
The second build of the same ALU, for the pyuvm testbench	1.6 to 1.8 s
A build after you change the testbench	none: `Vtop is up to date.`, in 0.07 s
One run of the pyuvm test, 2000 vectors	0.54 s
`make check-all`: the test against the correct ALU and three bugs	2.0 s, 0.5 s a run
The cocotb test, all 393,216 vectors	11.0 to 11.7 s

The first build compiles seven C++ files, and only one of them, `Vtop__ALL.o`, is the ALU. The other six are Verilator's and cocotb's support code, and the compiler cache keeps all seven. The second build of the same ALU found six of them in the cache and compiled one, so it took under 2 s. Lab 02's first build of the router took 2.2 s and compiled two files: the router, and cocotb's `main()` for the simulator. It compiled `main()` again because `main()` includes a header that Verilator writes for each design. Build times vary with the machine and with whatever else it's doing, so use the table to judge how long to wait, and don't expect your times to match it.

In each lab, one build serves all of a testbench's tests and all of the lab's planted bugs. Every lab from Lab 02 on has one testbench, and so one build. Lab 01 has two testbenches, the cocotb test and the pyuvm testbench, and it builds the ALU once for each. You choose the test, the seed, the bug and the test's knobs (which are plusargs) when you start the simulator, so changing any of them never rebuilds anything either.

Runs take longer from Lab 03 on. Each run spends about 1.5 s importing pyvsc, and a graded test sends hundreds of packets, so a run of Lab 03's graded test took us about 3 s, where Lab 02's took 0.25 s. The longest steps are Lab 07's two regressions, which took us 8 min 27 s and 9 min 21 s.

Where the builds go

Each lab builds into its own `build/` folder. The simulator is `build/sim_build/Vtop`, and for Lab 01's ALU, `build/sim_build/` came to 1.5 MB. Lab 01's cocotb test builds its own simulator in `build/cocotb/`. Each run gets a folder

of its own under `build/runs/`, named after the test, the seed and the bug (for example `AluRandTest-s1-A03`). It holds `run.log`, cocotb's `results.xml`, the exact command (`cmd`), and the waves if you asked for them. After Lab 01's two builds and a few runs, `build/` held 3.4 MB.

The compiler cache is ccache. It lives at the top of your clone, in `.ccache`, and every lab shares it. Because it's in your workspace, it survives a container restart, so a new container doesn't have to begin with a cold build. It's capped at 1 GB, and after Lab 01's two builds it held 672 KB.

`make clean` in a lab folder removes that lab's `build/` and keeps the cache and the venv:

```
rm -rf /w/labs/lab-01-one-alu-two-frameworks/build
 removed /w/labs/lab-01-one-alu-two-frameworks/build. The ccache (/w/.ccache) and the venv
are kept.
```

We ran it with the course folder mounted at `/w`, so on the workbench your paths start with `/foss/designs/modules/dd105-b` instead. `make clean-cache` deletes the cache, and the next build after it is a cold one again. `make clean-venv` deletes the venv, and `make venv` makes it again. `labs/common/README.md` lists every make target the labs share, and `make help` in a lab folder prints the targets that lab has and the current values of its variables.

Reading a lab's output

Every run starts with cocotb's own lines, which give the Verilator version, the seed (`Seeding Python random module with supplied seed 1`) and cocotb's version. Your test's lines come next, then cocotb's table of the tests it ran, and last the course's one-line verdict on the run, between two rules. The last of your test's lines, and the wording of the verdict, depend on what kind of test it is.

Every pyuvm test in the course derives from `BaseTest`, in `labs/common/py/dd105b_base.py`. Lab 01's `AluRandTest` is a pyuvm test, and so is every test from Lab 04 on. pyuvm reports through Python's `logging`, and an error that a component logs with `self.logger.error()` doesn't fail anything by itself, so without `BaseTest` the test would still pass. `BaseTest` counts every error that any component logs, and at the end it prints one line with the count and fails the test if the count isn't 0. In a pyuvm run that passes, the last line that your test prints is this one:

```
DD105-B BaseTest: 0 errors logged, no bound assertions
```

Then come cocotb's line saying that the test passed and cocotb's table, and the run ends with the verdict:

```
-----------------------------------------------------------------------
 ok: 0 errors   0.53 s   log: build/runs/AluRandTest-s1/run.log
-----------------------------------------------------------------------
```

"No bound assertions" means that no SystemVerilog assertions are attached to the design. From Lab 05 on, assertions are bound into the router, and the line counts their failures too.

Lab 01's cocotb test and the tests in Labs 02 and 03 are plain cocotb tests. They have no `BaseTest`, because a failed check in them raises `AssertionError`, and that fails the test by itself. When one of them passes, its verdict reads `ok: passed`.

As in DD104, the graders accept your testbench only if it passes on the correct design and fails on a broken one. `make check` and `make check-all` run your test on the correct design and on the bugs, and mark every run with one word. On the correct design, a run is `ok` if it passed and `FAIL` if a check fired. On a bug, a run is `caught` if a check fired and `MISSED` if it passed. On either one, a run that failed for some other reason is `BROKEN`, and a `BROKEN` run never counts as a catch. The other reasons include an exception that isn't a check (a `NameError` from a typo

in your code, for example), a crash, a timeout and a failed build. A pyuvm test that passed without `BaseTest`'s line is `BROKEN` too, because its logged errors would have failed nothing. Lab 01's shipped pyuvm testbench misses one of the lab's three bugs, so your first `make check-all` there ends like this:

```
  MISSED  tier 1  A03_zero_flag_low_nibble         0 errors                        0.5s
  caught  tier 1  A01_sub_operands_swapped         291 errors                      0.5s
  caught  tier 1  A02_nand_is_nor                  278 errors                      0.5s
--------------------------------------------------------------------------------
 2 of 3 caught.
 Still hiding from this test:
   A03_zero_flag_low_nibble
      hides behind: every check that compares results and not flags
================================================================================

  The verdict above is FAIL, so this target exits non-zero and make
  prints an error line under this one. That is the verdict talking,
  not a broken Makefile.
make: *** [../common/pyuvm.mk:281: check-all] Error 1
```

You'll see that last line often. It's `make` reporting the failed verdict, and it goes away once your testbench passes on the correct design and catches every bug.

Counts, simulated times and verdicts come out the same every time you run the same code with the same seed. The seed is 1 unless you choose another, and cocotb uses it to seed Python's `random`, so your counts should match the labs' exactly. If one doesn't, something is different and we'd like to know what. Wall-clock times change from one run to the next, so treat them only as a rough guide.

Getting help

- Ask in the team Discord. No question is too basic. Much of this course is about places where the tools go wrong without telling you, such as an error that a pyuvm component logs while its test passes, so a lot of what looks like your own mistake is really the tools being unhelpful.
- Quote the exact error, the command you ran, and the test, seed and bug. "My scoreboard fails" and "my scoreboard reports 48 errors on B09 with seed 1 and none on the golden router" are different questions, and the second is much easier to answer.
- Paste the verdict block and the first error from `run.log` as text. When one bug causes many errors, the later ones are mostly its consequences, so the first is the one to read. If the build failed, its whole log is in `build/build.log`, and the makefile prints the last 30 lines of it for you.
- If a command in a lab doesn't do what the lab says, please report it. We ran every command before writing it down and copied every output from a real run, so a mismatch means something has drifted since.

Requests to avoid

`labs/common/bugs.py` lists all fifteen of the router's planted bugs, with the exact line each one changes, and every lab has a `solutions/` folder. All of it is in your clone and nothing stops you reading it, but if you do, you skip what the course is teaching you, which is how to find a bug you can't see. The graders also give `VERDICT: FAIL -- your testbench reads which bug is on.` to any testbench that reads the `+BUG` switch or the design's record of which bug is on. If you're stuck, ask in Discord for help finding a better question to put to the design, rather than for the answer.

Check yourself

1. You finish Lab 01 and run `make clean` to free space. Will Lab 02's first build be a cold one?

Answer: A1, in Answers at the end.

2. `make check-all` ends with `make: *** [...] Error 1`. Is something wrong with the makefile?

   Answer: A2, in Answers at the end.

3. Your pyuvm scoreboard logs an error, cocotb's table says `PASS`, and the verdict says `BROKEN`. What's the most likely cause?

   Answer: A3, in Answers at the end.

Further reading

- pyuvm's README at release 4.0.1, the version in the course image. Its table of the IEEE 1800.2 standard says, section by section, what pyuvm implements and what it leaves out. The same text on pyuvm.readthedocs.io describes a later release. Correction to those pages for this course: in 4.0.1, `raise_objection()` and `drop_objection()` take no argument, and there is no `self.uvm_report`.
- cocotb 2.0.1′s documentation, the version in the image. The pages for other versions differ, so check that the address has `v2.0.1` in it.
- ChipVerify's Introduction to Verilator. Correction: it says that Verilator supports only synthesizable Verilog, and it sends you to Icarus for testbenches with delays or `fork`. In this course the testbench is Python and the delays are cocotb's, so Verilator compiles only the design, which is synthesizable. DD105-A, which writes its testbenches in SystemVerilog, runs delays and `fork` on Verilator 5.050 with Verilator's `--timing` option.

Next: What a framework buys you.

Answers

A1. Check yourself

No. `make clean` removes only that lab's `build/` folder. The compiler cache in `.ccache` at the top of your clone stays, and it holds the support code that every build compiles, so Lab 02′s first build compiled two files, the router and cocotb's `main()`, and took 2.2 s. Only `make clean-cache` empties the cache.

A2. Check yourself

No. The verdict was FAIL, so the target exits with a nonzero status, and `make` reports that with its error line. The three lines just above it say so.

A3. Check yourself

The test doesn't end through `BaseTest`, so its log has no `DD105-B BaseTest:` line. Either it derives from pyuvm's `uvm_test` directly, or it overrides `final_phase` without calling `super().final_phase()`. Without `BaseTest`, the logged error failed nothing, and the verdict won't call that run a pass.