Getting started

Read this page before you start Lab 01. You'll do the setup itself in Lab 01, Part 1, and this page says what DD105-A needs from your machine and which tool versions the labs' numbers were recorded with. It also says how long a build takes and where it leaves its files, how to read what a lab prints, and how to get help.

Requirements

You did the hard part of the setup in IC101, and you used the same container all through DD104. DD105-A runs in that container too. Like DD104, it needs no PDK and no synthesis flow, because everything in it is simulation.

The setup has three pieces:

- the noVNC desktop, started from your `workspace` clone with `./scripts/start_vnc.sh` (`scripts\start_vnc.bat` on Windows). Ports, passwords and blank-page fixes are in IC101: Launch the noVNC desktop;
- `. /foss/designs/common/.designinit`, which defines the workbench's `mod` command. You have to load it in every new terminal;
- `mod add dd105-a`, which you run once. It clones the course from `github.com/uoftasic/dd105-a` into `/foss/designs/modules/dd105-a`, and from then on `mod dd105-a` takes you there. 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.

`mod add dd105-a` prints:

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

Nothing else needs installing. Verilator, Icarus, cocotb, the Yices constraint solver and Surfer are all in the image. The image has no UVM library, so the course carries its own copy in `third_party/uvm`, and `mod add` clones it along with everything else.

Each lab's makefile works out its own paths, so every lab also runs with `make` alone, in 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`
Icarus Verilog	`Icarus Verilog version 14.0 (devel) (s20260301-391-g64f13540-dirty)`
cocotb	`2.0.1`
Yices	`Yices 2.7.0`
Surfer	`surfer 0.7.0 (git: v0.7.0)`

Tool	What its version check prints
UVM, in `third_party/uvm`	`Accellera:1800.2:UVM:2020.3.2`, at the start of every UVM run

You'll check the first five rows yourself in Lab 01, Part 1. Of the six, Verilator's version matters most, because much of the course depends on what 5.050 does and doesn't do. Where it differs from the SystemVerilog standard, the labs and these pages say so at the point where you first meet the difference.

Yices is there because Verilator needs an outside solver for every `randomize()` that has constraints. Without one, each such `randomize()` fails, nothing is randomized, and the test still passes. So the labs' makefile tells Verilator to use Yices, and if it can't find Yices it stops before it builds or runs anything.

How long things take

Every time Verilator builds a lab's simulator, it translates the whole UVM library into C++ along with your testbench, and g++ then compiles that C++, apart from any files that the compiler cache already holds. So in this course the builds are slow and the runs are quick. These are the times we measured for Lab 01 in the course image, on 2 CPUs of a shared machine:

Job	Wall time
The first build, with an empty compiler cache	126 s
A rebuild after a change inside one function	27 s
A rebuild after adding a member to a class	108 s
One run of the random test, 2000 vectors	1.2 s
`make check-all`: the test against the correct ALU and three bugs	four runs of 1.1 s to 1.2 s
The cocotb test on Icarus, for comparison	23.0 s, and 0.4 s to build

The build prints its own estimate when it starts:

```
======================================================================
 BUILD  alu_tb_top   (2 jobs; ccache: /w/.ccache)
 A cold build takes 1.5 to 2.5 minutes on 2 CPUs; a rebuild with a warm
 ccache takes from 10 s to about 2 minutes. Full log: /w/labs/lab-01-one-alu-two-testbenches/
build/build.log
======================================================================
```

We ran it with the course folder mounted at `/w`, so on the workbench your paths start with `/foss/designs/modules/dd105-a` instead. While it builds, the terminal shows a `compiling` line for each C++ file in the build.

Build times vary with the machine and with whatever else it's doing. Lab 01's own writeup records a first build of 102 s, and rebuilds of 26 s and 93 s for the same two kinds of change. So use the table to judge how long to wait, and don't expect your times to match it.

The two rebuild times differ because of what changed. A change inside a function body leaves most of the generated C++ as it was, so the compiler cache can supply the compiled copy of almost every file. A new member changes the class's layout, and most of the generated C++ changes with it, so most of it has to be compiled again. What a framework buys you says more about this.

In each lab, one build serves all of the lab's tests and all of its planted bugs. 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.

Where the builds go

Each lab builds into its own `build/` folder. Verilator's folder of generated C++ comes to almost 900 MB, and the build deletes it once the simulator is linked. So afterwards `build/` holds mainly the simulator, `build/sim`, which is about 47 MB, and the build's log. Each run gets a folder of its own under `build/runs/`, named for the test, the seed and the bug, such as `alu_rand_test-s1-A03`. It holds `run.log` and, if you asked for them, the waves.

The compiler cache is ccache, and it lives at the top of your clone, in `.ccache`, where 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 2 GB, and after the Lab 01 builds in the table it held 211 MB.

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

```
 removed /w/labs/lab-01-one-alu-two-testbenches/build. The ccache (/w/.ccache) is kept: make
clean-cache removes it.
```

It's worth running `make clean` when you finish a lab, to get the space back, and every lab's README lists it. `make clean-cache` deletes the cache itself and leaves `build/` alone, and the next build after it is a cold one 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.

Reading a lab's output

Every run of a lab's simulator starts with UVM's release notes, which you can ignore. The release notes are a block of text headed `IMPORTANT RELEASE NOTES`. They give the library's version, `Accellera:1800.2:UVM:2020.3.2`, and end with `(Specify +UVM_NO_RELNOTES to turn off this notice)`. (Lab 01's cocotb test and Lab 03's `make traps` don't use UVM, so they don't print them.) Your test's own lines come next, and what comes at the end depends on whether the run passed.

A run that passes prints `DD105: TEST PASSED`, which comes from the course's base test, `dd105_base_test`. Then UVM prints its report summary, and last the labs' makefile prints a one-line verdict:

```
--------------------------------------------------------------------------
 ok: 0 UVM_ERROR, 0 UVM_FATAL   1.18 s   log: build/runs/alu_rand_test-s1/run.log
--------------------------------------------------------------------------
```

A run that reports errors prints UVM's report summary first, then a `DD105: TEST FAILED` line with the error counts, then a `%Fatal` line from `dd105_base_test` saying `DD105: UVM counted errors, so the exit status is nonzero`. We put the `%Fatal` there on purpose, because on Verilator 5.050 a UVM run that reports errors would otherwise still exit with status 0, and `make` and the graders need a failing run to exit with a nonzero status.

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, such as a crash, a timeout or a failed build, is `BROKEN`, and a `BROKEN` run never counts as a catch. Lab 01's shipped testbench misses one of its three bugs, so its first `make check-all` ends like this:

```
  MISSED  tier 1  A03_zero_flag_low_nibble         0 UVM_ERROR, 0 UVM_FATAL        1.1s
  caught  tier 1  A01_sub_operands_swapped         282 UVM_ERROR                   1.1s
  caught  tier 1  A02_nand_is_nor                  255 UVM_ERROR                   1.1s
----------------------------------------------------------------------
 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/uvm.mk:300: 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, so yours should match the labs' exactly, and if one doesn't, something is different and we'd like to know what. Wall-clock times are only a rough guide, because they depend on your machine and on what else it's doing.

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 a `randomize()` that does nothing and a UVM run that exits with status 0 after errors, so a lot of what looks like your own mistake is really the simulator being unhelpful.
- Quote the exact error, the command you ran, and the test, seed and bug. "My scoreboard fails" and "my scoreboard reports 46 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` 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. A run's log ends with `%Fatal: ... DD105: UVM counted errors, so the exit status is nonzero`. Did the simulator crash?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify has a few pages about Verilator and about getting UVM running. Each was written for a different setup from this course's, so read them with the corrections below.

- Introduction to Verilator. Correction: it says that Verilator supports only synthesizable Verilog, that it can't run `#` delays or `fork`, and that you should use Icarus for testbenches that need them. That was true before Verilator 5, which added the `--timing` and `--binary` options, and every lab here builds with both.
- Verilator Simulation: its section on two-state simulation is worth reading. Correction: it says that class-based SystemVerilog has minimal support and that UVM testbenches won't compile, and both claims are out of date. Every lab in this course builds a class-based SystemVerilog testbench, together with the whole UVM library, on Verilator 5.050. In Lab 01 and in Labs 04 to 07 it's a UVM testbench, and in Labs 02 and 03 it's plain classes that a UVM test starts.
- UVM Installation: what a UVM library is and how you'd get one. You don't need to get one, because the course carries its own. The page puts the library on the include path but never says that `uvm_pkg.sv` itself has to be compiled as well. The labs' makefile does both.

Next: What a framework buys you.

Answers

A1. Check yourself

No. `make clean` removes only that lab's `build/` folder, and the compiler cache in `.ccache` at the top of your clone stays, so Lab 02's build can reuse whatever it shares with Lab 01. Only `make clean-cache` empties it.

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

No. Your test reported at least one `UVM_ERROR` or `UVM_FATAL`, and `dd105_base_test` ended the run with `$fatal` so that its exit status would be nonzero. The `DD105: TEST FAILED` line just above it gives the counts, and the errors themselves are earlier in the log, before UVM's report summary.