Getting started

This page covers what DD104 needs from your machine, which tool versions it is pinned to, how to read the labs' output, and how to get help. You'll do the actual setup in Lab 01, Part 1.

Requirements

You did the hard part of the setup in IC101, and you have been using the same container since DD102. DD104 is the lightest course in the digital track to set up, because nothing in it is ever mapped to silicon. That means there is no PDK, no liberty file, no `STD_CELL_LIBRARY` and no LibreLane, and the only tools you need are a simulator, Python and a waveform viewer.

The setup has three pieces, all of which you have used before:

- the noVNC desktop, started from your `workspace` clone with `./scripts/start_vnc.sh` (ports, passwords and blank-page fixes are in IC101: Launch the noVNC desktop);
- `. /foss/designs/common/.designinit`, which in DD104 does one thing for you: it defines the `mod` command;
- `mod add dd104`, which clones this repo into `/foss/designs/modules/dd104`. That folder is your `workspace` folder bind-mounted into the container, so your files are on your own disk and deleting the container does not delete your work.

Every DD104 lab also runs with `make` alone, in a bare container with no environment setup, because each lab pins its own paths. So if a setup step fails late at night, type `make` anyway and read what it tells you.

Versions

The course is pinned to `hpretl/iic-osic-tools:2026.08`. These are the versions it ships:

Tool	Version
Icarus Verilog	`14.0 (devel)`
Verilator	`5.050`
cocotb	`2.0.1`
Surfer	`0.6.0`

Of these, the cocotb version matters most, because cocotb 2.0 renamed things that appear in the first few lines of every 1.x example, and most of what a search engine finds for you was written for 1.x. Fortunately, the list of differences that actually break a run is shorter than you might expect:

- Most 1.x forms still run on 2.0.1 and only print a `DeprecationWarning`: `units=` instead of `unit=`, `MODULE=` in a Makefile, and `sig.value.integer`. You should still write the new spellings, but if your run is red, these aren't the reason.
- Some 1.x forms are hard errors, among them `cocotb.fork` (use `cocotb.start_soon`), `@cocotb.coroutine` (use a plain `async def`), and `dut.sig <= 1` (use `dut.sig.value = 1`). None of their error messages mentions a version number, so they won't tell you that the problem is an old example.

When a tutorial disagrees with this course, go with the course, because it describes the cocotb that is installed in your container. The full measured list is cocotb 2.x and older tutorials.

Reading a lab's output

Every lab ends with a verdict block, and a failed verdict exits non-zero on purpose so that a script or CI job can read the result without grepping text. `make` reports that non-zero exit with a line like this:

```
make: *** [Makefile:49: check] Error 1
```

That line is `make` reporting the failed verdict, and it doesn't mean the Makefile is broken; the lab prints three lines just above it to say so. You will see it often, and it goes away once the verdict passes.

Your first run is meant to fail. Lab 01 hands you a testbench that passes on both a good adder and a broken one, so `make` ends on `VERDICT: FAIL`, and the course starts from there.

Lab writeups mark each number as exact or ballpark. Simulated time, vector counts, cycle counts and verdicts are exact, so if one of those doesn't reproduce on your machine, that's a bug in the course and we want to hear about it. Wall-clock seconds are ballpark, and a difference of ±25 % from one run to the next is normal.

How long things take

The image is the same ~19 GB you pulled for IC101, and DD104 adds nothing to it. These are wall-clock times measured in the pinned container, on a machine that was busy with other work at the time:

Job	Wall time
Lab 01 end to end: two tests, against two adders	~5 s
The example testbench against the SimProc core (Icarus)	~6 s
The same six tests under Verilator, including its build	~17 s (~3 s once built)
The shipped chip test against the good core and all 13 broken ones	~30 s
Verilator toggle coverage on the Lab 01 adder, both stimulus sets	~12 s

Because runs are this cheap, you can rerun your tests every time you change a line, and the labs assume you will.

Getting help

- Ask in the team Discord. No question is too basic. In a verification course, much of what looks like your own mistake is really the tool being unhelpful, and in this course a lot of it is a design that is supposed to be broken.
- Quote the exact error and say which core you were running against. "My test fails" and "my test fails on `B07` and passes on the golden core" are different questions, and the second is much easier to answer. (`B07` is one of the thirteen broken cores, and the golden core is the correct one.) In a Python traceback, the last line is the actual error, and everything above it is the call stack, most of which is cocotb's own machinery.
- Paste the whole verdict block rather than a screenshot of one line. The block says which core, which testbench and which tests fired, and that is most of what someone needs to diagnose the problem.
- If a command in a lab doesn't do what the lab says, please report it. We ran every command before writing it down, so a mismatch means something has drifted since.

Requests to avoid

`labs/common/bugs.py` lists all thirteen bugs, in order, with line numbers. So do `labs/simproc/solutions/bug-NN.md` and `labs/simproc/variants/bug-NN/bug.patch`, the second of which is the one-line edit itself. All of them are in your clone, and nothing stops you reading them, but reading them throws away what the course teaches, which is finding a bug you cannot see. You can only practise that on bugs nobody has told you about.

If you are stuck, ask in Discord for help finding a better question to ask the design, rather than for the answer.

Next: What a passing test proves.