Getting the tools outside the course image

This page lists what the labs need, where each piece comes from, and what changes when you run them somewhere other than the course image.

Every number and output on this course's pages comes from a run in the course image, IIC-OSIC-TOOLS 2026.08. The workbench you set up in IC101 runs that image. Everything the labs need that the image doesn't have is in this repository, including the UVM library, so a clone builds without fetching anything. We checked every fact on this page on November 16, 2026, either in the image or against the source it names. The commands for installing tools on other systems are copied from each project's own documentation. We haven't run them, and each block of them is marked *untested by us*.

Tool versions

Tool	In the course image	What the labs use it for
Verilator	5.050 (`Verilator 5.050 2026-07-01 rev v5.050`)	builds every SystemVerilog testbench, through `labs/common/uvm.mk`
Yices	2.7.0, as `yices-smt2`	Verilator's constraint solver, called while a simulator runs
g++	13.3.0	compiles the C++ that Verilator writes; Verilator was configured with it
mold	2.30.0	the linker that the image's Verilator was configured to use (`-fuse-ld=mold`)
ccache	4.9.1	keeps compiled C++ between builds, so a rebuild after a small edit takes about 25 s instead of about 100 s
GNU make, bash, GNU coreutils	make 4.3, bash 5.2.21, coreutils 9.4	run `uvm.mk`
Python	3.12.3	runs the graders in `labs/common`, which use only the standard library
`verilator_coverage`	comes with Verilator	`make cov` merges and reads `coverage.dat` with it
`vcd2fst`	from GTKWave, in `/foss/tools/bin`	`make waves` converts the simulator's VCD to FST with it
Icarus Verilog, cocotb	14.0 (devel), 2.0.1	only Lab 01's cocotb testbench
UVM	2020-3.2	in this repository (`third_party/uvm`), not in the image

The image is Ubuntu 24.04.4. It is published on Docker Hub as `hpretl/iic-osic-tools:2026.08`, and the course pins it by digest:

```
hpretl/iic-
osic-tools@sha256:3c371645b19c6f6564dc8c7b21e39ad1c1833d274fe5b85639afe1ba9d7987e7
```

That digest names a multi-platform index, which lists a `linux/amd64` image and a `linux/arm64` image. We've run only the amd64 one. It is 3.76 GB to download (61 compressed layers) and 11.7 GB once unpacked. The course's CI (`.github/workflows/labs.yml`) runs every lab in that amd64 image through `scripts/ci/in-image.sh`, which is a single `docker run` command with the course's limits of 2 CPUs and 3 GB. From the top of a clone, with Docker and the image on your machine, this builds and runs Lab 01's UVM testbench the way the CI does:

```
export IMAGE=hpretl/iic-osic-tools@sha256:3c371645b19c6f6564dc8c7b21e39ad1c1833d274fe5b85639afe1ba9d7987e7
scripts/ci/in-image.sh make -C labs/lab-01-one-alu-two-testbenches
```

`in-image.sh` puts `--skip` in front of your command. That tells the image's entrypoint to run the command instead of starting its desktop, and the entrypoint prints `[INFO] SKIPPING UI STARTUP` when it does.

The UVM library

`third_party/uvm` is a copy of the UVM library from chipsalliance/uvm-verilator, branch `master`, at commit `5d72b6618acfddece7f09382a032ccbc05862fdc` (2026-08-14, "Update master to uvm-2020-3.2"). The library is Accellera's UVM 2020-3.2, which implements IEEE 1800.2-2020 with the deviations listed in its `DEVIATIONS.md`. The chipsalliance repository adds `src/dpi/uvm_hdl_verilator.c`, the Verilator back end for UVM's `uvm_hdl_*` functions. With it, `uvm_hdl_read` and the rest can reach a signal by its hierarchical name through Verilator's VPI, and the register layer's back door works through those functions.

Path	What it is
`src/`	the library: `uvm_pkg.sv`, the macros, and the DPI C code in `src/dpi/`
`LICENSE.txt`	the Apache License 2.0, which covers the whole library
`NOTICE.txt`	the copyright owners, from AMD to Verilab, twenty in all
`README.md`, `DEVIATIONS.md`	the library's own release notes and its list of deviations from the standard
`VENDORED.md`	where the copy came from, written by us

Only `src/`, `LICENSE.txt`, `NOTICE.txt`, `README.md` and `DEVIATIONS.md` were copied, and none of them was edited. We checked this on November 16, 2026, by cloning the repository at that commit and comparing, and the 186 files in `src/` and the four other files were identical to the clone's. To check `src/` again:

```
git clone https://github.com/chipsalliance/uvm-verilator.git
git -C uvm-verilator checkout 5d72b6618acfddece7f09382a032ccbc05862fdc
diff -r uvm-verilator/src third_party/uvm/src
```

The only part of the library that isn't SystemVerilog is its DPI C code in `src/dpi/`. `uvm.mk` compiles `src/dpi/uvm_dpi.cc` into every simulator, and that file includes the rest of `src/dpi/`, directly or through `uvm_hdl.c`. The Verilator back end is included only when `VERILATOR` is defined, and Verilator's own makefile defines it. If you copy the library into another project, keep `LICENSE.txt` and `NOTICE.txt` with it, as the Apache License asks.

Verilator options in `uvm.mk`

Every lab's `Makefile` names its sources and includes `labs/common/uvm.mk`. The table below lists the options that `uvm.mk` adds and the reason for each, and `labs/common/README.md` has the variables that change them. Every build writes its Verilator version, its solver and its whole Verilator command line, apart from `-j`, to `build/build.flags` in the lab folder, and `uvm.mk` rebuilds the simulator whenever that file changes.

Option	Why
`--binary --timing`	`--binary` builds a complete simulator with Verilator's own `main()`, and `--timing` adds support for `#`, `@` and `wait` in classes
`+incdir+third_party/uvm/src third_party/uvm/src/uvm_pkg.sv`	the UVM library
`third_party/uvm/src/dpi/uvm_dpi.cc` and `--vpi`	UVM's DPI C code, which needs VPI for its HDL access
`-Wno-WIDTHEXPAND -Wno-WIDTHTRUNC -Wno-SYMRSVDWORD -Wno-CASTCONST -Wno-IMPLICITSTATIC`	Verilator won't build while any warning is left. A testbench that uses the UVM field macros can get about 200 width warnings from the code they expand into, and the course's own testbenches raise some too. `make lint` leaves these five options out, so it shows them: 74 width warnings in `labs/ref`, 1 in Lab 01, and none in Labs 02 to 07 as shipped
`-CFLAGS "-DVM_SOLVER_DEFAULT='\"yices-smt2 --incremental\"'"`	sets the solver that a simulator uses when you run it by hand, without `make` (see The constraint solver, below)
`--output-groups 8` and `-MAKEFLAGS OPT_FAST=-O0`	Verilator's C++ goes into eight files, which are compiled without optimization. A cold build then takes about 100 s on 2 CPUs instead of 526 s with Verilator's defaults, and the simulator runs about 1.4 times slower
`--coverage-user`	covergroups and `cover property` statements write their counts to `coverage.dat`, and `make cov` reads the covergroup bins from it
`--trace-vcd` with `labs/common/uvm/dd105_trace.vlt`	the simulator writes VCD for `make waves`, and the `.vlt` file leaves the packages out of it
`-j 2`	two g++ jobs, each of which can need up to 1.3 GB

Verilator runs in the top of the clone and names every file relative to it. Those file names end up in the generated C++, so the C++ doesn't depend on where the clone is mounted, and the part generated from the UVM library is the same in every lab. That lets ccache reuse it from one lab, container or machine to the next. `uvm.mk` works out where the clone is from its own location, and none of its commands use a `/foss` path. The CI mounts the clone at `/w`, and the labs build and pass there as they do in the workspace.

Labs 06 and 07 use the register model's back door, which reaches signals through VPI, and Verilator lets VPI see a signal only if it is marked public. Each of those labs marks its signals one at a time in `tb/router_regs.vlt`, because the option that marks every signal public breaks the UVM build on Verilator 5.050. The UVM register layer explains the back door and that file.

The constraint solver

Verilator hands every constrained `randomize()` to an external SMT solver, named by the environment variable `VERILATOR_SOLVER`. Verilator 5.050's documentation says that if the variable isn't set, it uses the solver supplied or found when Verilator was configured, or `z3 --in` if that is empty. In the course image the configured solver is empty (`verilator -V` prints `VERILATOR_SOLVER =`) and `z3` isn't installed, so `uvm.mk` puts `VERILATOR_SOLVER="yices-smt2 --incremental"` in the environment of every run. It refuses to start a build or a run if the solver that the variable names isn't on your `PATH`, because without a solver every constrained `randomize()` returns 0 and the run still passes.

The build also bakes the solver into the simulator, as its default for a run that you start by hand rather than through `make`. Verilator 5.050 gets this wrong when the value has a space in it. It writes `yices-smt2 --incremental` into the simulator as `yices-smt2\ --incremental`, which no shell can run, and a run that relies on that default fails with exit status 127. So `uvm.mk` runs Verilator with the variable unset and passes the default as a define of its own (`-DVM_SOLVER_DEFAULT`, in the table above).

You can name another solver on `make`'s command line. In the image, which has no `z3`, a build from a lab folder then stops with this message:

```
$ make build VERILATOR_SOLVER="z3 --in"
========================================================================
 VERILATOR_SOLVER is "z3 --in", and z3 is not on PATH.
========================================================================
 Without a solver every constrained randomize() returns 0, nothing is
 randomized, and the run still passes. Are you in the course image?
make: *** [../common/uvm.mk:237: solver-check] Error 1
```

Every constraint result in the course was measured with Yices, including Lab 03's experiments that compare Verilator 5.050 with what the standard says. We haven't run the labs with another solver.

Yices's README gives these commands for installing it on other systems (*untested by us*):

```
# Ubuntu or Debian
sudo add-apt-repository ppa:sri-csl/formal-methods
sudo apt-get update
sudo apt-get install yices2

# macOS, with Homebrew
brew install SRI-CSL/sri-csl/yices2
```

ccache

Verilator's generated makefile runs every compile as `$(OBJCACHE) $(CXX)`. Verilator's documentation says that if `OBJCACHE` isn't set and ccache was present when Verilator was configured, ccache is used by default. The image's Verilator was configured that way, and its `verilated.mk` has `OBJCACHE ?= ccache`. A Verilator configured on a machine without ccache doesn't use it unless you set `OBJCACHE=ccache` in the environment. Without ccache every build is a cold one, because `uvm.mk` deletes the generated C++ after each build (`SLIM=1`). Verilator's install guide also marks ccache and mold "If present at build, needed for run", which means that if they were installed when Verilator was configured, they have to be installed wherever you use it.

`uvm.mk` keeps the cache in `.ccache` at the top of the clone, capped at 2 GB. When the clone is in your workspace, as it is in the course, the cache survives a container restart. In the course's measurements on a small UVM testbench,

a cold build took 102 to 103 s and a rebuild after a one-line edit took 24 to 26 s (Build times). `make cache` shows how full the cache is, and `make clean-cache` deletes it.

Waves

Verilator can write FST directly with `--trace-fst`, but its FST writer needs the header `lz4.h`, and the course image has no `/usr/include/lz4.h`, so that option doesn't work there. Instead, `uvm.mk` builds with `--trace-vcd`, and `make waves` converts the VCD file to FST with `vcd2fst`, which comes with GTKWave. Elsewhere, `make waves` needs `vcd2fst` on your `PATH`, and `make cov` needs `verilator_coverage`, which is installed with Verilator.

Verilator 5.050

Every expected output in the labs comes from Verilator 5.050, apart from the outputs of Lab 01's cocotb testbench, which runs on Icarus Verilog. The simulator's limits that the course teaches you to work around are also 5.050's: what `solve...before` and `dist` do, how coverage is counted, and which ways of driving a signal lose a nonblocking assignment. Another version may print different numbers, and it may fix a limit or add one. Verilator's install guide warns that distribution packages "almost never have the most recent Verilator version" and recommends building from Git. If you use another version, run `make FULL=1` in `labs/common/selftest` first. It builds a small UVM testbench, runs every grader on it end to end, and fails if any of the nonblocking-assignment results that the course describes has changed.

Verilator 5.050's install guide (`docs/guide/install.rst` at tag `v5.050`) gives these steps for building it from Git on Ubuntu (*untested by us*):

```
# Prerequisites:
#sudo apt-get install git help2man perl python3 make autoconf g++ flex bison ccache
#sudo apt-get install libgoogle-perftools-dev libjemalloc-dev numactl perl-doc
#sudo apt-get install libfl2  # Ubuntu only (ignore if gives error)
#sudo apt-get install libfl-dev  # Ubuntu only (ignore if gives error)
#sudo apt-get install zlibc zlib1g zlib1g-dev liblz4 liblz4-dev  # Ubuntu only (ignore if
gives error)

git clone https://github.com/verilator/verilator   # Only first time

# Every time you need to build:
unsetenv VERILATOR_ROOT  # For csh; ignore error if on bash
unset VERILATOR_ROOT  # For bash
cd verilator
git pull         # Make sure git repository is up-to-date
git tag          # See what versions exist
#git checkout master      # Use development branch (e.g. recent bug fixes)
#git checkout stable      # Use most recent stable release
#git checkout v{version}  # Switch to specified release version

autoconf         # Create ./configure script
./configure      # Configure and create Makefile
make -j `nproc`  # Build Verilator itself (if error, try just 'make')
sudo make install
```

For this course, the last of the three checkout lines becomes `git checkout v5.050`. The guide lists mold as optional (`sudo apt-get install mold`) and z3 as an optional solver for constraints (`sudo apt-get install z3`). The course uses Yices instead of z3 (see The constraint solver, above). It also notes that Verilator compiles in, as defaults, the

values that `VERILATOR_ROOT`, `VERILATOR_SOLVER`, `SYSTEMC_INCLUDE` and `SYSTEMC_LIBDIR` have when you run `./configure`, so set them to what you want before you run it.

Other differences outside the image

What	In the image	Elsewhere
Memory	the course's runs use a 3 GB limit	each g++ job can need 1.3 GB, so keep `JOBS=2` (the default) unless you have more memory
Shell tools	GNU make 4.3, bash, GNU coreutils	`uvm.mk` uses `realpath -m`, `stdbuf`, `timeout`, `seq`, `awk` and `date +%s.%N`; we haven't run it where those differ, as on macOS
Python	3.12.3	the graders use only the standard library; we haven't run them on other versions
The solver	`yices-smt2` on `PATH`	install Yices, or name another solver with `VERILATOR_SOLVER`
ccache	used, because the image's Verilator was configured with it	install it before you build Verilator, or set `OBJCACHE=ccache`
Waves	VCD converted with `vcd2fst`	`vcd2fst` on `PATH` for `make waves`
Lab 01's cocotb half	cocotb 2.0.1 and Icarus Verilog 14.0 (devel)	other versions untested