Two simulators

This page answers: *why does this course ship two simulators, and which one should I run?*

The short answer is that you should run both, for different reasons. Every DD104 lab runs on Icarus Verilog unless you say otherwise, and every lab also accepts `SIM=verilator`. The two tools are good at different things, and by the end of this page you should be able to say which one to reach for in a given situation.

People often say "Verilator is the fast one", which is true as far as it goes, but the measurements below give you a more useful picture of when it's faster and when it isn't. We took all of them in `labs/common`, the shared harness directory, and you'll repeat the important ones in Lab 02, Part 5.

The same six tests, twice

`make sim SIM=icarus` and `make sim SIM=verilator` in `labs/common` both run `test_example.py` against the golden core, and both pass. Here is the last line of each results table:

```
icarus     ** TESTS=6 PASS=6 FAIL=0 SKIP=0     512010.01     4.38     116812.47 **
verilator  ** TESTS=6 PASS=6 FAIL=0 SKIP=0     512010.01     1.92     267302.99 **
```

The three numbers after the test counts each measure something different. The third is roughly the first divided by the second, so the first two are the ones that matter here.

The first, `512010.01`, is simulated time in nanoseconds, and it is identical on both tools, to the hundredth of a nanosecond, on every run. The design is the same state machine whichever tool runs it, so a cycle count you measure on one simulator holds on the other, which is why the course quotes timing numbers without saying which tool produced them.

The second, `4.38` against `1.92`, is how many seconds each simulator spent simulating, so Verilator did the same work in 44 % of the time. Over three runs each, Icarus measured 4.38 / 4.57 / 4.71 s and Verilator 1.90 / 1.92 / 1.97 s, which is a steady ratio of 2.3×.

The number that decides which tool to use, though, is wall-clock time. We measured it by putting `time` around the whole `make`, starting from a clean directory:

	Wall clock	Of which simulating	The rest is
Icarus	6.8 / 5.8 / 5.9 s	4.4–4.7 s	Starting `vvp`, loading Python
Verilator	16.3 / 16.4 / 16.0 s	1.9–2.0 s	Building a C++ program

So Verilator simulates 2.3× faster and still finishes about ten seconds later. The reason is that Verilator is really a compiler, not a simulator in the sense Icarus is. It reads your SystemVerilog and writes out C++, and `g++` has to turn that C++ into a binary before the first clock edge can happen. Icarus just reads your SystemVerilog into a bytecode file and starts running it.

> "Compiled is faster than interpreted" is true per unit of work, but it tells you nothing about a job so small that the compile time dominates. You already know this about C++ against Python for a ten-line script, and it's the same trade here, with simulated time as the unit of work.

The build cost

The Verilator step itself takes almost no time:

```
- Verilator: Walltime 0.035 s (elab=0.003, cvt=0.022, bld=0.000)
```

That's 35 milliseconds to read 1,178 lines of SystemVerilog and write six C++ files. Nearly all the rest of the build time is `g++`, and you can see this in the statistics from `ccache`, which caches `g++` output. Here they are for two Verilator builds in the same container, with `make clean` between them:

```
  Hits:               0 /   6 ( 0.00%)      <- first build: 16.04 s
  Hits:               6 /  12 (50.00%)      <- second build: 3.60 s
```

The first build compiled six C++ files with six cache misses and took sixteen seconds. The second found all six in the cache and took three and a half seconds. The only thing that differed between them was `g++`, and you pay that cost once per container session rather than once per run. The cache lives inside the container, so it survives `make clean` but is lost when the container stops.

A more accurate version of "Verilator is slow to start", then, is that the first build in a fresh container costs about fifteen seconds of `g++`. If you start the workbench once in the morning and work inside it all day, you pay that once. If you start a new container for every command, you pay it every time. Tool comparisons rarely mention this, and it changes which tool comes out ahead.

Where the crossover is

You're trading a fixed build cost against a cost per nanosecond of simulation, so which tool wins depends on how much simulated time is in the run. DD104 ships a deliberately long testbench, `test_values_only.py`, with 1,889,970 ns of simulated time, which is 3.7× as much as the example. Here is the wall-clock time we measured for both:

Simulated time in the run	Icarus	Verilator, cold cache	Verilator, warm cache
`test_example`: 512,010 ns	5.8–6.8 s	16.0–16.4 s	3.2–3.6 s
`test_values_only`: 1,889,970 ns	22.5 s	22.4 s	8.5 s

At half a millisecond of simulated time, Icarus wins by ten seconds. At two milliseconds the two tie on a cold cache, and Verilator is 2.6× faster on a warm one. On this design, then, the crossover is just under 2 ms of simulated time, and it moves lower every time you rerun in the same container, because more of the build is already cached.

`make check-all` builds and runs fourteen separate cores, which is the worst case for Verilator, since each core is a separate build:

```
  Icarus     77.18 s      10 of 13 caught
  Verilator  84.94 s      10 of 13 caught
```

Even with fourteen full C++ builds, Verilator comes within 10 % of Icarus, because `ccache` catches most of the files that don't change from one core to the next. More importantly, the two tools reach the same verdict: each catches 10 of the 13.

Lint and X

Verilator can also lint your code, that is, check it for likely mistakes without simulating it. Run with `--lint-only` on the three SimProc source files, it prints twelve warnings in 0.077 seconds, ending with:

```
%Error: Exiting due to 12 warning(s)
```

Five of the twelve say `Bit extraction`, and four of those five are the same finding, reported at four different lines of the same module:

```
%Warning-WIDTHTRUNC: rtl/simproc_system.sv:848:16: Bit extraction of array[63:0] requires
                     6 bit index, not 8 bits.
```

`dp_ram` has 64 entries but the address bus is 8 bits wide, so addresses 64–255 are off the end of the array. Verilator found that just by reading the file, before simulating anything. Icarus compiles the same file without mentioning it; it prints twenty-eight `sorry:` notes about something else and exits with status 0 (see Output that is not an error).

It's worth seeing what each tool does with that defect at run time. `labs/common/test_two_simulators.py` writes `0x5A` to address 36, then reads address 36 and address 100 back out through the chip's UART and prints the raw bits that came off the pin.

Here is what Icarus prints:

```
READ  36  (in range)      -> 01011010
READ 100  (off the end)   -> XXXXXXXX
```

And here is Verilator:

```
READ  36  (in range)      -> 01011010
READ 100  (off the end)   -> 01011010
```

Both runs pass, and both simulate exactly 60,520.00 ns, yet they disagree about what the chip did.

Icarus is a 4-state simulator, which means every bit can be `0`, `1`, `x` or `z`. Reading past the end of the array gives `x`, the `x` propagates through the read port and the UART, and eight `X` characters come out of the chip. That's an obvious and correct result, as long as your stimulus happens to read address 100.

Verilator is a 2-state simulator, so every bit is either 0 or 1. It truncated the address to the six bits the array has, `100 & 63 = 36` (the same as 100 − 64), and you got the byte at address 36, which is the same `0x5A` you wrote there. That answer looks plausible, and Verilator gives no sign that it's wrong.

Neither tool is broken; they're answering different questions.

This tells you when to trust a passing run. A design that reads uninitialised state, indexes past the end of an array or leaves a signal undriven will often work perfectly under Verilator and produce X under Icarus. If you run the same suite on both simulators, any disagreement between them is a finding, and it's one you can't get from a single passing run.

Which one to use

You are	Use	Because
Editing a test and rerunning it every ninety seconds	Icarus	There's no build, so it's six seconds from saving the file to a result
Running a long suite, or the same design many times in one session	Verilator	It's 2.3× faster on simulated time, and the build is already cached

You are	Use	Because
About to trust a passing run	Both	Any disagreement between them is a finding
Looking for width, truncation and case-coverage mistakes	`verilator --lint-only`	It takes 0.077 s and needs no test-bench
Chasing an `X` on a pin	Icarus	Verilator is 2-state, so it has no X to show you
In a CI job that runs everything once	Either	It's 77 s against 85 s here, so choose on lint rather than speed

Icarus is the default in `sim.mk` only because of the first row of that table. That doesn't make it the better simulator; it has the faster edit-and-run loop, and that loop is where you'll spend most of this course.

`-Wno-fatal`

`sim.mk` passes `-Wno-fatal` when `SIM=verilator`. Without it, Verilator refuses to build this design:

```
%Error: Exiting due to 12 warning(s)
```

Those are the same twelve lint warnings. Verilator treats warnings as fatal by default, which is a reasonable policy for code you own and an impossible one for code you've been handed to test, because you can't fix the warnings without changing the design under test. So don't read `-Wno-fatal` as "the warnings do not matter". Read it as "this design does not lint clean, and that is a finding", and you'll look at that finding in Lab 02.

Identical simulated time

cocotb writes a JUnit XML file with one `sim_time_ns` value per test. If you run both simulators, extract those values and diff them, you'll find no difference at all. These are the six values for `test_example`:

```
sim_time_ns="13680.0"
sim_time_ns="11220.0"
sim_time_ns="33370.0"
sim_time_ns="99580.0"
sim_time_ns="33820.0"
sim_time_ns="320340.0"
```

You should do the same with your own testbench. If a simulated-time number differs, stop and find out why. It means your testbench depends on something the two tools model differently, which is almost always an X, an undriven signal, or a race between two things scheduled in the same delta, and so one of the two passing results isn't telling you the truth about the design.

Check yourself

1. You will run a 10 µs test fifty times this afternoon in one container. Which simulator, and why?
2. A test passes on both simulators but its `sim_time_ns` differs between them. What does that suggest?
3. Why can Verilator never show you an `X` on a pin?

Next: SimProc in one page.