Output that is not an error

This page answers: *the screen is full of red and yellow. Did something break?*

Usually nothing has. This page goes through every wall of alarming output that DD104 produces. For each one it gives the exact count, so that you can confirm you're seeing the same thing, and a one-line rule for telling it apart from a real failure.

Here are the rules in short:

Tool	It is fine when	It is real when
Icarus compile	Lines say `sorry:`, exit code 0	A line starts `error:` and compilation stops
Verilator	You passed `-Wno-fatal` and the build finished	It says `%Error:` and no binary was produced
cocotb	`TESTS=n PASS=n FAIL=0`	`FAIL` is non-zero, or the results table is missing
`make check`	`VERDICT: PASS`	`VERDICT: FAIL`. The `make: *** Error 1` under it is only `make` reporting that the verdict exited non-zero

---

1. Icarus prints 28 `sorry:` lines

Compile the design the way the labs do:

```
cd labs/common/rtl
iverilog -g2012 -o /tmp/sp.vvp tt_um_ieeeuoftasic_simproc.sv simproc_system.sv tb.v
```

You should see twenty-eight lines like this, and nothing else:

```
simproc_system.sv:552: sorry: constant selects in always_* processes are not fully
supported (the process will be sensitive to all bits in 'instr_reg_out[7:0]').
```

To confirm the count, send the output to a file and count lines in it:

```
iverilog -g2012 -o /tmp/sp.vvp tt_um_ieeeuoftasic_simproc.sv simproc_system.sv tb.v > /tmp/
ic.log 2>&1
echo "exit code: $?"                       # run it UNPIPED, or you get grep's exit code
grep -c 'sorry:' /tmp/ic.log               # 28
grep -c 'error:' /tmp/ic.log               # 0
wc -l < /tmp/ic.log                        # 28 -- there is nothing else
grep -oE '^[^ ]+:[0-9]+:' /tmp/ic.log | sort -u | wc -l    # 22 distinct locations
```

> After a pipeline, `$?` holds the exit code of the last command in the pipeline, not the compiler's. That's why the commands above write `cmd > file 2>&1`, check `$?` on its own line, and only then grep the file. If you write `cmd | grep -c ...` instead, a compiler failure comes out as a count of zero that looks perfectly plausible.

`sorry:` lines	28
Distinct source locations	22 (six lines produce two each)
Total lines of output	28, so there is nothing else
`error:` lines	0
`warning:` lines	0
Exit code	0

Icarus means these as notes rather than warnings. Where the code selects `instr_reg_out[3:0]` inside an `always_comb`, Icarus makes the process sensitive to all eight bits instead of the four that were selected. That gives the block more sensitivity than it needs rather than less, so it re-evaluates more often than it strictly has to, and the simulation is still correct. All 22 locations are in the decoder, between lines 543 and 677, which is where a processor slices an instruction word into fields.

> Rule: a real Icarus failure starts a line with `error:` and stops. `sorry:` means "I am doing this a less clever way"; `error:` means "I am not doing this".

For contrast, here is a real failure. If you drop `-g2012`, the flag that turns on SystemVerilog-2012, the same three files produce:

```
simproc_system.sv:122: syntax error
simproc_system.sv:122: error: Invalid module instantiation
```

	With `-g2012`	Without it
Lines of output	28	414
`error:` lines	0	288
Exit code	0	152
`/tmp/sp.vvp`	Written	Not written

If you ever see `syntax error` on a line of perfectly good SystemVerilog, the flag you're missing is `-g2012`. You only have to remember it when you run `iverilog` by hand, because `sim.mk` already passes it.

---

2. Verilator prints 12 warnings and refuses to build

```
cd labs/common
verilator --lint-only --timescale 1ns/1ps --top-module tb \
  rtl/tt_um_ieeeuoftasic_simproc.sv rtl/simproc_system.sv rtl/tb.v
```

Verilator prints twelve warnings in 0.077 seconds and then stops with:

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

The exit code is 1. To count the warnings by kind:

```
verilator --lint-only --timescale 1ns/1ps --top-module tb \
  rtl/tt_um_ieeeuoftasic_simproc.sv rtl/simproc_system.sv rtl/tb.v > /tmp/v.log 2>&1
grep '^%Warning' /tmp/v.log | sed 's/^%Warning-\([A-Z]*\).*/\1/' | sort | uniq -c
```

```
      2 CASEINCOMPLETE
      4 WIDTHEXPAND
      6 WIDTHTRUNC
```

Five of the twelve say `Bit extraction`, and they're the only ones that are findings rather than complaints about tidiness:

Line	Module	What it says
`simproc_system.sv:848`	`dp_ram`	`Bit extraction of array[63:0] requires 6 bit index, not 8 bits`
`simproc_system.sv:851`	`dp_ram`	Same, the other write port
`simproc_system.sv:856`	`dp_ram`	Same, read port A
`simproc_system.sv:857`	`dp_ram`	Same, read port B
`simproc_system.sv:1022`	`UART_RX`	`Bit extraction of var[7:0] requires 3 bit index, not 4 bits`

Four of the five are the same defect, reported at four lines: the RAM has 64 entries and the address bus is 8 bits wide, so addresses 64–255 are off the end of the array. Verilator found it just by reading the file, before it simulated anything. Icarus never mentions it, so under Icarus you only see it at run time, as eight X bits on a pin, and only if your stimulus happens to read, say, address 100. Two simulators measures that run-time side, and you run it yourself in Lab 02.

Verilator treats warnings as fatal by default. That is a reasonable policy for code you own and an impossible one for code you have been handed to test, so `sim.mk` passes `-Wno-fatal` when `SIM=verilator`. You will see the same twelve warnings scroll past during a normal `make sim SIM=verilator`, and then the build finishes.

> Rule: with `-Wno-fatal`, a `%Error: Exiting due to N warning(s)` line at the end of a build that still produced a binary is just noise. Without `-Wno-fatal`, the same line means no binary was produced and nothing ran. Either way, look for the results table rather than for the word `Error`.
>
> Don't take `-Wno-fatal` to mean that the warnings don't matter, though. It means this design doesn't lint clean, and that is itself a finding.

`-Wall`: 19 warnings, or 27 with the testbench

```
verilator --lint-only -Wall --timescale 1ns/1ps \
  --top-module tt_um_ieeeuoftasic_simproc \
  rtl/tt_um_ieeeuoftasic_simproc.sv rtl/simproc_system.sv
```

With `-Wall`, the design on its own gives 19 warnings:

Kind	Count	Is it a finding?
`WIDTHTRUNC`	6	Yes: the five `Bit extraction` warnings, plus one slip in a packet width
`WIDTHEXPAND`	4	No: `index <= 1'b0` assigning 1 bit to a 4-bit register
`UNUSEDSIGNAL`	4	No
`CASEINCOMPLETE`	2	Worth a look, because an incomplete `case` is how latches and X get into a design
`UNUSEDPARAM`	1	No
`PINCONNECTEMPTY`	1	No
`DECLFILENAME`	1	No: it only says the module isn't named after its file

6 + 4 + 4 + 2 + 1 + 1 + 1 = 19.

If you add the course's `rtl/tb.v` and make `tb` the top module, as the twelve-warning command in section 2 does, `-Wall` reports 27:

```
verilator --lint-only -Wall --timescale 1ns/1ps --top-module tb \
  rtl/tt_um_ieeeuoftasic_simproc.sv rtl/simproc_system.sv rtl/tb.v
```

The extra eight are 5 `UNDRIVEN` and 3 `UNUSEDSIGNAL`, all inside `tb.v`, which is a testbench wrapper with pins it deliberately doesn't drive. 19 + 8 = 27, and checking that the counts add up like this is how you know you haven't found something new by accident. Adding `tb.v` to the first command and keeping `--top-module tt_um_ieeeuoftasic_simproc` still gives 19, because `tb.v` sits above that top module rather than under it.

> Rule for `-Wall` output: sort it by warning kind and look at the width and case warnings first, because `WIDTHTRUNC` and `CASEINCOMPLETE` describe the design doing something you didn't write. `UNUSED*`, `DECLFILENAME` and `PINCONNECTEMPTY` only describe a design that isn't tidy.

Ten more you can cause yourself

If you leave `--timescale 1ns/1ps` off the lint command, you get ten extra `TIMESCALEMOD` warnings, one per module, because `tb.v` has a timescale and the RTL files don't. cocotb's own makefiles pass `--timescale`, so you only see these when you run `verilator` by hand, and they're caused by your command line rather than by anything in the design.

---

3. cocotb deprecation warnings

In a cocotb log, `WARNING` is only the log level of the message, and it says nothing about whether the test passed. These are the warnings you'll actually see:

```
0.00ns WARNING  py.warnings  .../test.py:24: DeprecationWarning: The 'units' argument
                             has been renamed to 'unit'.
0.00ns WARNING  py.warnings  DeprecationWarning: `logic_array.integer` getter is
```

```
                            deprecated. Use `logic_array.to_unsigned()` instead.
                  Using MODULE is deprecated, please use COCOTB_TEST_MODULES instead.
```

All three old spellings still work: cocotb 2.0.1 accepts `units=`, `MODULE=` and `.integer`, does the right thing with each, and prints a warning. The test runs and passes, and the only cost is a noisier log.

> Don't go looking for a failure that isn't there. If a page tells you that `MODULE=` causes a silent run with no tests, it's wrong: we measured it in this container, and it works, with a warning. Write the new spellings anyway, but if your test is failing, a `DeprecationWarning` isn't the reason.
>
> Some old forms are hard errors, though, among them `cocotb.fork`, which gives `AttributeError`, `@cocotb.coroutine`, which gives `AttributeError`, and `dut.sig <= 1`, which gives `TypeError`. Four runnable demonstrations ship in `labs/cocotb-intro/oldapi/`, and cocotb 2.x and older tutorials has the full breakdown.

Some other lines look like problems but aren't:

Line	What it is
`-.--ns INFO gpi ... VPI registered`	cocotb attaching to the simulator. `-.--ns` means the line was printed before simulation time started
`Seeding Python random module with 1787752037`	The seed for this run, so you can reproduce it
`- :0: Verilog $finish`	Verilator's normal end-of-simulation line
`sys:1: RuntimeWarning: coroutine '_do_writes' was never awaited`	Python cleaning up cocotb's writer task at shutdown

---

4. `make: *** Error 1` after a failed verdict

```
 VERDICT: FAIL — your testbench passes B03_zero_flag_low_nibble too.
   A test that passes on a broken core is not testing that core.
**************************************************************************

  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: *** [lab.mk:66: check] Error 1
```

A failed verdict exits non-zero on purpose, so the labs can be used from a script or a CI job without anyone grepping the text, and `make` reports a non-zero exit from anything it runs. The lab prints the paragraph above so that you can tell a failed verdict from a broken Makefile.

> Rule: read the `VERDICT:` line, not the `make:` line. If there is a `VERDICT:` line at all, the machinery worked, whatever the verdict says.

The same applies to a cocotb test that fails: the results table still prints, `FAIL` is non-zero, and `make` reports `Error 2` from `Makefile.icarus`. That's just the test failing, which is what you asked it to do when it finds a problem.

---

5. Green output that ran nothing

Everything above is noise that looks like failure. The runs in this section are the opposite: their output looks fine even though nothing was tested, and they're worse, because nothing in them makes you look twice.

Running `python3 -m pytest test_example.py` gives:

```
collected 0 items
============================ no tests ran in 0.10s =============================
```

The exit code is 5. It looks like the obvious command to run and nothing in the output is red, but it ran nothing, because a cocotb test isn't a pytest test.

Running `python3 test_example.py` prints nothing at all and exits with 0. A cocotb test isn't a standalone Python program either: it needs a simulator running underneath it, and here only `make` starts one.

The third case is a typo in `COCOTB_TESTCASE`:

```
make sim TB=test_example COCOTB_TESTCASE=chekcing
```

```
0.00ns WARNING  cocotb.regression   No tests left after filtering with: chekcing$
```

The exit code is 0, there's no results table, and `build/results.xml` contains six `<skipped/>` entries. The simulator started and cocotb attached, but the name you gave matches no test, so every test was filtered out.

> Rule for all three: a run that tested something prints a results table ending in `TESTS=n PASS=n FAIL=0 SKIP=0`. No table means no tests ran, whatever the exit code says. Check the table, then check that `n` is the number you expected.

If the table says `TESTS=5` when you wrote six tests, that's the same failure in a quieter form.

Checklist

Before you decide that a run was fine, check these:

1. Is there a results table? If there isn't, no tests ran.
2. Does `TESTS=n` match the number of `@cocotb.test()` functions you wrote?
3. Is `FAIL=0`?
4. If a verdict target ran, does it say `VERDICT: PASS`?
5. Did the build produce something: a `.vvp`, or a `Vtop`?

None of these checks is the exit code, and none of them is the absence of the word `Error`.

Everything on this page comes from the harness in `labs/common`, documented call by call in The SimProc harness.