DD105-A — Testbenches with UVM

Build a UVM testbench in SystemVerilog on Verilator, and find out what the framework costs and buys.

UofT ASIC Team

Contents

1 DD105-A — Testbenches with UVM 5

I 1. Two testbenches

2 Getting started 11

3 What a framework buys you 17

4 From cocotb to SystemVerilog classes 21

5 Lab 01: One ALU, two testbenches 27

II 2. Classes

6 Classes for testbenches 41

7 Interfaces, clocking blocks and virtual interfaces 47

8 Threads, mailboxes and events 53

9 Lab 02: A class testbench without UVM 59

III 3. Constraints and coverage

10 Constrained random on Verilator 71

11 Functional coverage you can read 77

12 Lab 03: Constraints and coverage 81

IV 4. UVM

13 The UVM skeleton 95

14 Sequences, sequencers and drivers 101

15 Monitors, analysis ports and scoreboards 107

16 The factory and config_db 113

17 Lab 04: The same testbench in UVM 117

V 5. Protocol rules

18 Assertions for interface rules 131

19 Reactive agents and backpressure 137

20 Virtual sequences 143

21 Lab 05: Protocol rules and backpressure 149

VI 6. Registers

22 Register models 163

23 Regressions and seeds 169

24 Lab 06: The register block 173

VII 7. Capstone

25 Verification plans and sign-off 187

26 Writing the verification report 191

27 Lab 07: Capstone 195

Appendices 207

A The router in one page 209

B The packet router 217

C The UVM register layer 231

D Getting the tools outside the course image 235

1. DD105-A — Testbenches with UVM

In this course you build testbenches for a small packet router. Each one has a driver that sends packets into the router, a monitor that turns a stream's valid/ready handshakes into packets, and a scoreboard that predicts what comes out and checks it. In Lab 02 the course gives you the driver, and you write the monitor and the scoreboard as plain SystemVerilog classes. In Lab 04 you finish a UVM version of the same testbench. Both testbenches run on Verilator.

DD105-A is part of the UofT ASIC Team education materials, published at https://uoftasic.com/dd105-a/. The labs run in the workbench: the IIC-OSIC-TOOLS container you start from the `workspace` clone you set up in IC101. You get this course's files with the workbench's `mod` command (`mod add dd105-a`), and Lab 01, Part 1 walks you through it.

Who this course is for

You have finished DD104, where you wrote cocotb tests in Python that found all thirteen bugs planted in SimProc. The testbenches you write in this course keep the stimulus apart from the checking, and the UVM ones let you choose the test when you run it, so that you can reuse and extend them. On your own runs you measure what UVM costs in lines of code and build time, and what each testbench catches.

The router was written for the course, in about 730 lines of SystemVerilog. It has one input stream and two output streams, each output with its own buffer. Its routing table, drop counters and interrupt enables are registers on an APB port. Fifteen one-line bugs are planted in it, and you switch one on when you run a simulation. Lab 01 starts on a design you already know, SimProc's ALU from DD104, with its three ALU bugs. By the end of Lab 07 your UVM environment catches all fifteen of the router's bugs, and you have written a verification plan and a verification report.

Track	Digital
Prerequisites	DD104, and through it IC101 and DD101 to DD103
Tools	Verilator 5.050, Yices 2.7.0 as its constraint solver, and Surfer for waveforms, all in the workbench image `hpretl/iic-osic-tools:2026.08`, and the UVM 2020-3.2 library, which ships in this course's repository. Lab 01 also runs cocotb 2.0.1 on Icarus Verilog 14.0.
Machine	The workbench you already have. Every time on these pages was measured with 2 CPUs and 3 GB: a testbench's first build takes about 2 minutes, a rebuild after an edit from about 10 s to about 2 minutes, and a run a few seconds. The longest step, Lab 07's regression, takes about 4 minutes.
Total time	About 21 hours, self-paced. The Course path rows below add up to 21 h 10 min: 15 h of labs and 6 h 10 min of reading. The lab times are the ones the lab writeups give. The reading times are our estimates, from the length of each page at about 130 words a minute, plus time for its figures and examples.

Course path

Work down this table in order. A Read row is a set of guide pages, which are for reading only and have no commands in them. A Do row is one lab writeup, and the lab writeups are where every command in the course lives. Each lab's "Before you start" section repeats which pages it expects you to have read.

#		What	Time
1	Read	Getting started · What a framework buys you · From cocotb to SystemVerilog classes	55 min
2	Do	Lab 01: One ALU, two testbenches. Test SimProc's ALU with DD104's cocotb test and with a UVM testbench, measure what each costs, and extend the UVM scoreboard until it catches the flags bug.	1 h 30 min
3	Read	Classes for testbenches · Interfaces, clocking blocks and virtual interfaces · Threads, mailboxes and events	1 h 10 min
4	Do	Lab 02: A class testbench without UVM. Write the monitor and the scoreboard of a class testbench for the router, catch two planted bugs, and find out what each catch depends on.	2 h
5	Read	Constrained random on Verilator · Functional coverage you can read	40 min
6	Do	Lab 03: Constraints and coverage. Add a covergroup, constrain the packet until the simulator produces the lengths you ask for, and reach the two coverage holes where this lab's bugs hide.	1 h 30 min
7	Read	The UVM skeleton · Sequences, sequencers and drivers · Monitors, analysis ports and scoreboards · The factory and config_db	1 h 20 min

#		What	Time
8	Do	Lab 04: The same testbench in UVM. Finish a UVM port of your Lab 03 testbench, inject errors with a factory override, and count what the structure cost in lines.	2 h 30 min
9	Read	Assertions for interface rules · Reactive agents and backpressure · Virtual sequences	1 h
10	Do	Lab 05: Protocol rules and backpressure. Bind assertions for the output rules into the router, finish a sink that answers each beat in its own time, and write a virtual sequence that catches the third bug.	2 h
11	Read	Register models · Regressions and seeds	35 min
12	Do	Lab 06: The register block. Give the testbench a register model with an adapter and a predictor, run UVM's reset and bit-bash sequences, and check the interrupt, until all five register bugs are caught.	2 h
13	Read	Verification plans and sign-off · Writing the verification report	30 min
14	Do	Lab 07: Capstone. Write a verification plan, run a seed regression, hunt mystery builds, and close the holes until all fifteen bugs are caught, then write the verification report.	3 h 30 min

Keep these reference pages open in a tab while you work. They're for looking things up rather than for reading start to finish:

- The router in one page
- The packet router, the router's full specification
- The UVM register layer
- Getting the tools outside the course image

pyuvm and DD105-B

This course uses UVM in SystemVerilog only. Its sibling course, DD105-B, covers pyuvm, the Python version of UVM, which is in the workbench image (version 4.0.1). In the research for these courses, a UVM environment for the ALU written in pyuvm built in 17 s, because Verilator then compiles only the design. The SystemVerilog version compiles the whole UVM library with it, which takes about 100 s from cold with this course's build settings.

Getting help

Ask in the team Discord; nobody expects you to work all of this out alone. Say which lab and step you're on, paste the command you ran and what it printed, and attach the `run.log` from the run folder if the question is about a simulation.

I

1. Two testbenches

2 **Getting started 11**
2 Requirements 11
2 Versions 11
2 How long things take 12
2 Where the builds go 12
2 Reading a lab's output 13
2 Getting help 14
2 Check yourself 14
2 Further reading 14
2 Answers 15

3 **What a framework buys you 17**
3 What UVM is 17
3 The two testbenches 18
3 What it costs 18
3 What it buys 19
3 Check yourself 20
3 Further reading 20
3 Answers 20

4 **From cocotb to SystemVerilog classes 21**
4 Where the testbench runs 21
4 DD104's t01 in SystemVerilog 21
4 Widths 23
4 Strings 24
4 Python and SystemVerilog side by side 24
4 Check yourself 25
4 Further reading 25
4 Answers 25

5 **Lab 01: One ALU, two testbenches . . 27**
5 Before you start 27
5 Part 1: Load the course 27
5 Part 2: The cocotb testbench 28
5 Part 3: The UVM testbench 31
5 Part 4: Both testbenches on the bugs 33
5 Part 5: Check the flags 34
5 Finishing 36
5 Going further (optional) 36
5 Answers 36

2. 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.

3. What a framework buys you

This page answers: *what is UVM for, and what does it cost on the simulator this course uses?*

A testbench framework is a library of base classes plus a set of rules for how the pieces you build from them fit together and when each one runs. UVM, the Universal Verification Methodology, is the one this course uses. In Lab 01 you'll test SimProc's ALU twice, once with the cocotb test from DD104 Lab 03 and once with a UVM testbench. You'll measure what each costs in lines, build time and run time, and see what each reports on three planted bugs. This page goes through what we measured when we ran the lab, so you know what you're looking at when your own numbers come in. Your times will differ a little from ours, but the counts (lines, vectors and errors) should match.

What UVM is

UVM is a library of SystemVerilog classes, standardised as IEEE 1800.2. It isn't part of the simulator, and the course image doesn't include a UVM library, so the course carries its own copy in `third_party/uvm`. That copy is UVM 2020-3.2 from `chipsalliance/uvm-verilator` at commit `5d72b66`, which compiles on the course's Verilator 5.050. Every lab's build compiles it along with your testbench.

The library gives you a base class for each job a testbench does, and the machinery that connects them. Lab 01's testbench uses these parts of it:

- base classes for the pieces: `uvm_sequence_item` for one transaction, `uvm_sequence` for a stream of them, and `uvm_driver`, `uvm_monitor`, `uvm_scoreboard`, `uvm_agent`, `uvm_env` and `uvm_test` for the components;
- **phases**, which build every component, connect them, run them, and then call each one's check and report functions, in that order, and **objections**, which the test raises and drops to tell UVM when the run phase is over;
- `uvm_config_db`, which carries the virtual interfaces from the top module to the driver, the monitor and the test;
- **analysis ports**, which pass each transaction the monitor writes to every component connected to the port, here the scoreboard;
- a **report server**, which counts every `` `uvm_error `` by its ID and prints the counts at the end;
- `+UVM_TESTNAME`, which picks the test when the simulator starts.

Lab 01's testbench also makes every component, sequence and transaction through UVM's factory, with `type_id::create`. Lab 04 adds a factory override, which lets a test swap one class for another without editing the code that makes it.

The two testbenches

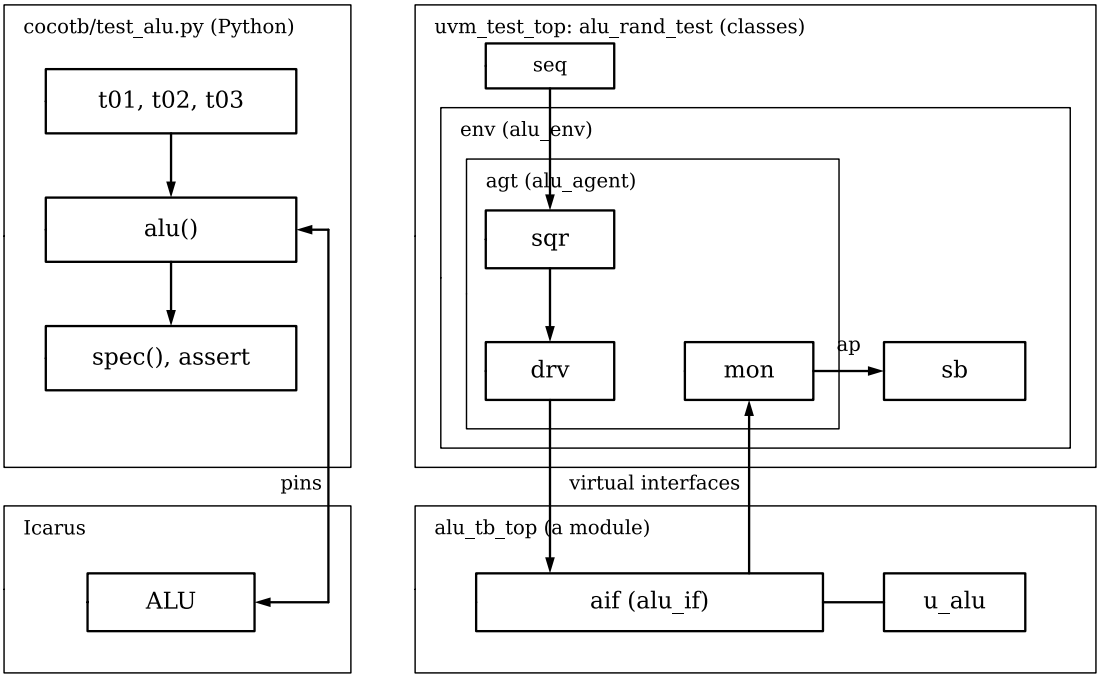


Figure 3.1: Lab 01's cocotb test beside its UVM testbench

Lab 01's two testbenches. On the left the Python test runs beside Icarus and reaches into the ALU through its pins. On the right everything is compiled into one simulator: the classes on top, the module with the interface and the ALU below.

Both testbenches do the same three things: make vectors, put them on the ALU's pins, and compare what comes back with a model. The cocotb test does all three in one file of plain functions: `alu()` drives the pins and reads them back, `spec()` is the model, and each test function loops over its vectors and asserts.

The UVM testbench gives each job its own class. A sequence makes `alu_item` objects and hands them to the sequencer, which passes them to the driver. The driver puts each one on the pins through the interface's clocking block, one per clock cycle, and sets the interface's `valid` signal in each cycle that carries a vector. In every cycle with `valid` high, the monitor reads the pins back into an `alu_item`, the ALU's answer included, and writes it to its analysis port. It doesn't know what the answer should be. The scoreboard receives each item, works out the right answer with `predict()`, and compares it with what the monitor saw. The agent and the environment create the pieces and connect them, and at the end of the run the environment also checks that the scoreboard checked every vector the driver sent. The test picks which sequence to start. Lab 01, Part 3 has a table that matches each class with the part of `test_alu.py` that does the same job.

What it costs

These are the costs Lab 01 measured, in the course image on 2 CPUs:

	cocotb, on Icarus	UVM, on Verilator
Lines of code (not counting blank lines and comments)	94	282
First build	0.6 s	102 s
Rebuild after you change the testbench	none	26 s to 99 s

	cocotb, on Icarus	UVM, on Verilator
Run	23.8 s, 17.5 s of it in `t02`′s 393,216 vectors	1.17 s for 2000 vectors
Time per vector	about 0.04 ms	about 0.6 ms

Lines. The UVM testbench has three times as many code lines, and most of the extra lines are in pieces the cocotb test doesn't need: an agent and an environment that create the other pieces and connect them, a constructor and a registration macro in every class, and a driver and a monitor that between them do what four lines of `alu()` do (its three `.value =` lines and its `return`).

Build. Verilator turns the testbench, the ALU and the whole UVM library into C++, and g++ compiles it. Lab 01′s build report says the sources came to 24 MB in 386 modules, and that g++ spent 98.5 s of the 102. The ALU's file is 94 lines of those 24 MB. The lab's optional last part runs the cocotb test on Verilator instead of Icarus. There Verilator compiles only the ALU, and the build took 13.3 s, so most of the 102 s goes to the UVM library rather than to the design.

Rebuilds. A change to a Python test costs nothing before the next run. A change to a SystemVerilog class costs a rebuild, and how long it takes depends on what you changed. The course keeps a compiler cache (ccache) in your clone, so a rebuild can reuse whatever didn't change. Changing `repeat (3)` to `repeat (4)` in `alu_base_test`'s `run_phase` rebuilt in 26 s, with 35 of 36 compiled files taken from the cache. Adding one member to the scoreboard class rebuilt in 93 s, with only 10 of 36 taken from the cache, because a new member changes the class's layout and most of the C++ that Verilator generates changes with it. So when you can, put a change inside a function or task body rather than adding a member, or make it through a plusarg such as `+N=500`, which needs no rebuild at all.

Speed. Per vector, the UVM testbench is more than ten times slower than the cocotb test, about 0.6 ms against 0.04 ms. The run is still quick, 1.17 s, because the test sends 2000 random vectors rather than sweeping all 393,216 as DD104′s `t02` does.

Running UVM on Verilator 5.050 has two more costs. A UVM run that reports errors still exits with status 0, and a constrained `randomize()` with no solver available fails without stopping the run. The course handles both for you. Every lab's makefile includes `labs/common/uvm.mk`, which sets up the solver, and every test extends `dd105_base_test`, which checks the solver at the start of the run and turns any `UVM_ERROR` into a nonzero exit status at the end.

What it buys

For that cost, Lab 01′s UVM testbench gives you three things, all of them modest on a design as small as one ALU.

- **Checking kept apart from stimulus.** In Lab 01, Part 5 you add a check of the `N` and `Z` flags to the scoreboard. Both `alu_rand_test` and `alu_directed_test` use it without being edited, because neither test does any checking of its own. In `test_alu.py`, `t01`, `t02` and `t03` each do their own comparison, so a new check would have to be written into each of them.
- **Tests picked at run time.** `alu_directed_test` is one short sequence and one short test class that reuse the driver, the monitor and the scoreboard. It's built into the same simulator as `alu_rand_test`, and `+UVM_TESTNAME` picks one of them when the simulator starts, so switching to it needs no rebuild. The whole command took 0.17 s. Every lab in the course builds once and picks tests, seeds and planted bugs this way.
- **Error counts by ID.** With bug A01 on, the shipped scoreboard reported 282 errors, and UVM's report server printed `[VALUE]  282` in its counts by ID. If you give the flag check its own ID, the summary keeps separate counts of value mismatches and flag mismatches, which DD104 Lab 03′s `t02` has to do by hand.

Deciding what to check is still up to you. Against Lab 01′s three bugs, the shipped UVM scoreboard caught A01 with 282 errors and A02 with 255, and missed A03 with no errors at all. The cocotb test caught all three, because its `t02` compares the flags and the shipped scoreboard doesn't. Once you add the flag check, the UVM test catches A03 with 131 errors in 2000 vectors.

You get more back on a bigger design, and the router you'll test from Lab 02 on is the kind of design UVM was made for. The router has a register bus, an input stream and two output streams, so a testbench for it has a driver and a monitor for each of those interfaces, and a scoreboard that predicts which output each packet comes out of. Labs 02 and 03 build that testbench from plain SystemVerilog classes first, so

that each UVM piece has an obvious job when it appears. Lab 04 then moves it into UVM and counts what moved and what was added.

Check yourself

1. In Lab 01′s scoreboard you change the body of `predict()`. About how long will the rebuild take? And if you add an `int unsigned` counter to the scoreboard class instead?

   Answer: A1, in Answers at the end.

2. Without `dd105_base_test`, what exit status would Lab 01′s UVM run have with 282 errors in its log, and why does that matter?

   Answer: A2, in Answers at the end.

3. The shipped UVM testbench misses A03 and the cocotb test catches it. Does that tell you anything about UVM?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's UVM section explains the library one class at a time. Its examples are written for commercial simulators, not for Verilator with the 1800.2 library this course uses, and the course doesn't reuse its code.

- UVM Tutorial: a map of the library's main pieces and the SystemVerilog it assumes you know.
- UVM Introduction: why UVM exists, and the difference between components and objects. Correction: its monitor fragment never constructs its analysis port, so as written its `ap.write()` would use a null handle.

Next: From cocotb to SystemVerilog classes.

Answers

A1. Check yourself

About 26 s, because after a change inside a function body the ccache can still supply almost every compiled file. A new member costs close to a full build, about 93 s in our run, because it changes the class's layout and most of the generated C++ changes with it.

A2. Check yourself

0. On Verilator 5.050 a UVM run ends with exit status 0 however many errors it reported, so anything that judges the run by its exit status, such as `make` or a regression script, would count it as a pass. `dd105_base_test` calls `$fatal` at the end of the test if UVM counted any error, and that makes the exit status nonzero.

A3. Check yourself

No. The cocotb test compares the flags and the shipped scoreboard compares only the value, so the two testbenches check different things. With the flag check added, the UVM testbench catches A03 too.

4. From cocotb to SystemVerilog classes

This page answers: *I can write a cocotb test. What changes when the testbench is written in SystemVerilog instead?*

A cocotb test and a SystemVerilog class testbench do the same work: they drive the design's inputs, read its outputs and decide whether the outputs are right. Most of the Python you wrote in DD104 has a direct SystemVerilog counterpart. The main differences are that the testbench runs inside the simulator rather than beside it, that a failed check is reported differently, and that every value has a fixed width. This page goes through them with DD104's `t01` translated line by line, and then with two short examples, one on widths and one on string literals. In Lab 01, Part 3 you'll read a whole UVM testbench, which is written in SystemVerilog classes, and in Part 5 you'll extend its scoreboard. From Lab 02 on you write classes of your own.

We ran every example on this page in the course image. The source is in `guide-examples/from-cocotb-to-systemverilog-classes/`, and `guide-examples/README.md` gives the command for each output, in its section for this page.

Where the testbench runs

In DD104 your test was a Python program running beside the simulator. Icarus or Verilator ran the design, and cocotb reached into it through VPI. The two took turns. Every `await` handed control back to the simulator, and your test carried on once the thing it was waiting for had happened.

A SystemVerilog testbench is part of the simulation. On Verilator, the testbench is translated into C++ along with the design, and the two are compiled into one program, which is the simulator. That has three consequences for how you work:

- **Every change to the testbench means a rebuild.** You can rerun a Python test as soon as you've changed it, because Python isn't compiled. A changed SystemVerilog class has to go through Verilator and g++ again, and because the UVM library is compiled in with it, a rebuild in Lab 01 took from 26 s to 99 s, depending on what was changed. What a framework buys you has Lab 01's numbers and explains why they differ.
- **Mistakes show up at two different times.** A type error, a misspelt name or a missing semicolon stops Verilator before anything runs, usually within a few seconds. Everything else shows up when the simulator runs. Every lab has `make lint`, which runs only Verilator's front end and takes seconds, so you can find the first kind of mistake without starting a full build.
- **Anything chosen when the simulator starts needs no rebuild.** The test, the seed, the planted bug and the test's own knobs are all given as plusargs, the arguments on the simulator's command line that start with `+`, such as `+UVM_TESTNAME`, `+verilator+seed+` and `+BUG`. Trying a different one takes a second or so.

DD104's t01 in SystemVerilog

`cocotb/test_alu.py` in Lab 01 is DD104 Lab 03's reference test for SimProc's ALU. Its first test, `t01`, checks one vector for each of the six operations. It uses two helpers. `spec()` works out what the ALU should produce, and `alu()` drives one vector and reads the three outputs back. Here they are, with most of `spec()`'s cases left out:

```
def spec(op, a, b):
    if op == ADD:
        out = (a + b) & 0xFF
    elif op == SUB:
        out = (a - b) & 0xFF
    ...
    return out, (out >> 7) & 1, int(out == 0)
```

```
async def alu(dut, op, a, b):
    dut.ALUop.value = op
    dut.A.value = a
    dut.B.value = b
    await Timer(1, unit="ns")
    return int(dut.ALUout.value), int(dut.N.value), int(dut.Z.value)
```

and the test calls them like this for each of its six cases:

```
got = await alu(dut, op, a, b)
want = spec(op, a, b)
assert got == want, (
    f"{OPS[op]}(0x{a:02X}, 0x{b:02X}) gave out=0x{got[0]:02X} "
    f"N={got[1]} Z={got[2]}, expected out=0x{want[0]:02X} "
    f"N={want[1]} Z={want[2]}")
```

In SystemVerilog the same test is one class, `alu_t01`. In it, `spec()` becomes a function called `predict()`, and the tuple that `spec()` returns becomes a packed struct:

```
typedef enum bit [2:0] {ADD, SUB, OR, NAND, SHL, SHR} alu_op_e;

typedef struct packed {
  bit [7:0] out;
  bit       n;
  bit       z;
} alu_result_t;

function alu_result_t predict(alu_op_e op, bit [7:0] a, bit [7:0] b);
  alu_result_t r;
  case (op)
    ADD:     r.out = a + b;
    SUB:     r.out = a - b;
    ...
  endcase
  r.n = r.out[7];
  r.z = (r.out == 8'h00);
  return r;
endfunction
```

The coroutine `alu()` becomes a task, `drive_and_read()`, because a task is the kind of routine that's allowed to wait:

```
task drive_and_read(alu_op_e op, bit [7:0] a, bit [7:0] b, output alu_result_t got);
  vif.ALUop = op;
  vif.A     = a;
  vif.B     = b;
  #1;
  got = '{vif.ALUout, vif.N, vif.Z};
endtask
```

and the `assert` becomes an `if` with an `$error`, which takes a format string the way `$display` does:

```
drive_and_read(ops[i], as[i], bs[i], got);
want = predict(ops[i], as[i], bs[i]);
if (got != want)
   $error("%s(0x%02X, 0x%02X) gave out=0x%02X N=%0d Z=%0d, expected out=0x%02X N=%0d
Z=%0d",
          ops[i].name(), as[i], bs[i], got.out, got.n, got.z, want.out, want.n, want.z);
```

Most lines translate one for one, and there are five differences:

- **Every argument and every variable has a type.** `op` is an `alu_op_e`, which is an enum. `ops[i].name()` returns the name of its value as a string, such as `"SUB"`, and that does the job of the `OPS` dictionary in the Python.

- **`predict()` has no `& 0xFF`.** `r.out` is 8 bits wide, so a sum that comes to 260 wraps to 4 as it's stored, the way the ALU's own adder wraps. The next section explains the width rule behind this, and how the same rule can make a model lose a carry it was meant to keep.
- **The class can't name the design's pins.** cocotb hands your test the design as `dut`. The classes of a testbench live in a package, and code in a package can't name anything in the module hierarchy, so the class reaches the pins through `vif` instead. `vif` is a *virtual interface*, a handle to an interface instance, and here the instance is the one that the top module connects to the ALU. `drive_and_read()` writes the pins directly and waits 1 ns, as `alu()` does, because this example copies what the cocotb test does. Lab 01′s UVM testbench gives the same ALU an interface with a clock, which the ALU itself doesn't use, and from Lab 01 on every driver writes through a clocking block. Interfaces, clocking blocks and virtual interfaces explains clocking blocks.
- **`await Timer(1, unit="ns")` becomes `#1`.** The unit of `#1` is the time unit that the code is compiled with. The example's Makefile sets it to 1 ns with `--timescale 1ns/1ps`, and so does every lab's Makefile.
- **A task can wait and a function can't.** `predict()` takes no simulated time, so it can be a function. `drive_and_read()` waits 1 ns, so it has to be a task. Classes for testbenches shows the error you get if you put a delay in a function.

On the correct ALU the class checks its six vectors and prints its summary line:

```
t01: 6 directed vectors checked
```

With bug A01 switched on, where `SUB` computes `B - A`, it stops at the second vector:

```
[2000]       %Error:       alu_t01_pkg.sv:62:       Assertion       failed       in
alu_t01_pkg.alu_t01.t01_one_vector_per_operation.unnamedblk1:  SUB(0xc8,  0x3c)  gave
out=0x74 N=0 Z=0, expected out=0x8c N=1 Z=0
%Error:        guide-examples/from-cocotb-to-systemverilog-classes/alu_t01_pkg.sv:62:
Verilog $stop
Aborting...
```

The message is the cocotb test's message, and three things about this log differ from a cocotb log:

- `[2000]` is the simulated time in units of the simulation's precision, which is 1 ps, so the error came at 2 ns. Each vector takes 1 ns, so 2 ns is the end of the second vector. Between `%Error:` and your text come the file and line of the `$error`, and then the scope it ran in, which ends with the task's name and an unnamed block.
- The hex digits are lower case. In Python `X` asks for upper-case digits, but in SystemVerilog `%X` means the same as `%x`, and Verilator printed lower case.
- The run stopped. A failed `assert` in cocotb ends that test, and cocotb moves on to the next one. Verilator's default is to stop the whole simulation at the first `$error`, which is why there's no summary line after it. In Lab 02 the Makefile adds `+verilator+error+limit+100` to every run so that a scoreboard can report more than one mismatch, and Part 3 of that lab explains it.

In a UVM testbench you report with `` `uvm_error `` instead, as Lab 01′s scoreboard does. The macro sends your message to UVM's report server, which counts errors by ID and lets the run carry on.

Widths

Python integers have no width, which is why `spec()` has to mask its `ADD`, `SUB`, `NAND` and `SHL` results with `& 0xFF`, the four that could fall outside 0 to 255. In SystemVerilog every value has a width, and the width that an expression is computed at depends on its operands and on where its result goes. The example adds `a = 200` and `b = 60`, both `bit [7:0]`, in three places:

```
sum8 = a + b;               // sum8 is bit [7:0]
sum9 = a + b;               // sum9 is bit [8:0]
$display("%0d", a + b);
```

and prints:

```
sum8 = a + b      -> 4
sum9 = a + b      -> 260
$display(a + b)   -> 4
```

The two assignments have the same expression on the right and get different answers. In an assignment the operands are widened to the width of the left-hand side before the addition, so `sum9` keeps the carry. Inside `$display` the expression has nothing to take its width from, so it's computed at the width of its widest operand, 8 bits, and the carry is lost. Both rules come from the SystemVerilog standard rather than from Verilator. Under the first rule, `predict()` gets the ALU's wrap-around without a mask, because `r.out` is 8 bits wide. The same rules are also how a model loses a carry it was meant to keep, so when a scoreboard reports mismatches only on results that overflow, check the widths in the model before you blame the design.

DD104 also taught you that Verilator has two states, 0 and 1, and no X. That still holds in a SystemVerilog testbench, whether you declare a variable `bit` or `logic`.

Strings

A string literal in SystemVerilog is a number, a packed array of 8-bit characters, until something makes it a `string`. Most of the time you won't notice, but you will in a `?:` that picks between two literals when one of them is empty or shorter than the other. The example prints the same thing two ways:

```
$display("[%0d beat%s]", n, n == 1 ? "" : "s");
$display("[%0d beat%s]", n, n == 1 ? string'("") : string'("s"));
```

```
[1 beat ]
[1 beat]
```

In the first line `?:` picks between two numbers. As a number, the empty literal `""` is one byte of zeros, as wide as `"s"`, so when `n` is 1 the result is a zero byte, and `%s` prints a zero byte as a space. A literal that's shorter than the other side of the `?:` gets zero bytes added in front, and those print as spaces too. Casting both sides to `string` keeps them strings, and an empty `string` prints as nothing. That's why Lab 02's `packet::sprint()` writes `string'("")` and `string'("s")`.

Python and SystemVerilog side by side

In cocotb and Python	In a SystemVerilog testbench
`class Packet:` with `__init__(self, ...)`	`class packet;` with `function new(...)`
`self.x`	`this.x`, or just `x`
`None`	`null`
a variable holding an object (a reference)	a handle; Classes for testbenches
`def f(...)`	`function`, which can't wait
`async def f(...)`	`task`, which can
`await Timer(1, unit="ns")`	`#1`
`await RisingEdge(dut.clk)`	`@(posedge clk)`, or `@(vif.cb)` through a clocking block
`dut.A.value = 5`	`vif.cb.A <= 5` through a clocking block, from Lab 01 on
`cocotb.start_soon(coro())`	`fork coro(); join_none`
`await First(a, b)`	`fork ... join_any`
`cocotb.triggers.Event`	`event`, with `->e` and `@e`
a `cocotb.queue.Queue` between coroutines	a `mailbox`
a list, with `append` and `pop(0)`	a queue, `int q[$]`, with `push_back` and `pop_front`

In cocotb and Python	In a SystemVerilog testbench
a dict	an associative array, `int counts[string]`
a tuple	a packed struct
`f"0x{a:02X}"`	`$sformatf("0x%02X", a)`
`assert got == want, "message"`	`if (got != want) $error("message")`, or `` `uvm_error `` in UVM
`random.randint(0, 255)`	`$urandom_range(255, 0)`, or `randomize()` with constraints, which Lab 01's `alu_item` uses and you write yourself from Lab 03 on
the `@cocotb.test()` functions in a module	UVM test classes, one chosen per run by `+UVM_TESTNAME=`

The rows about threads, events and mailboxes are the subject of Threads, mailboxes and events.

Check yourself

1. `spec()` masks its `ADD`, `SUB`, `NAND` and `SHL` results with `& 0xFF`, and `predict()` masks nothing. Why doesn't `predict()` need to, and in what kind of expression could a SystemVerilog model still lose a carry it needed?

   Answer: A1, in Answers at the end.

2. Your SystemVerilog check prints one `%Error` line, then `Verilog $stop` and `Aborting...`, and nothing after them. Is the rest of the test passing?

   Answer: A2, in Answers at the end.

3. Why is `drive_and_read()` a task, while `predict()` can be a function?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's SystemVerilog pages cover each of these language features at more length. Most of their logs come from Cadence simulators rather than Verilator, and the course doesn't reuse their code, but their explanations of the language apply here too.

- Testbench Evolution: six styles of testbench, from a linear one to UVM. Your DD104 suites sit near the end of that list.
- bit, byte and int: the 2-state integer types.
- Strings.
- Enumeration.
- Structure.
- Queue and Associative Array.
- Class vs Struct: why assigning a class variable copies a handle, while assigning a struct copies its value.
- Testbench Example Adder: a class testbench without UVM, for a design much like the ALU. Its "buggy design" listing has the same code as the correct design, so the bug that the page describes isn't in the code it shows.

Next: Lab 01: One ALU, two testbenches.

Answers

A1. Check yourself

`r.out` is 8 bits wide, so a result is cut to 8 bits as it's stored, the way the ALU's adder cuts it. A carry goes missing when an expression is computed at the width of its operands with nothing wider to take its width from, as `a + b` was inside `$display`, or when you store the result in a variable that's too narrow, such as `sum8`. In `sum9`, which is 9 bits wide, the same sum kept its carry.

A2. Check yourself

You can't tell. Verilator stops a simulation at its first `$error` unless the run raises the limit with `+verilator+error+limit+N`, so the vectors after the failing one were never checked. In cocotb only the failing test would have stopped, and the next test would still have run.

A3. Check yourself

`drive_and_read()` waits 1 ns for the ALU to answer, and only a task may contain a delay or wait for an event. `predict()` computes its answer in zero time, so a function is enough.

5. Lab 01: One ALU, two testbenches

You'll test SimProc's ALU from DD104 with two testbenches, the cocotb test from DD104 Lab 03 and a UVM testbench written in SystemVerilog. You'll measure what each one costs in lines, build time and run time, and see what each one reports on three planted bugs. Then you'll extend the UVM scoreboard to check the flags as well, so that it catches A03, a zero-flag bug that the shipped scoreboard misses.

Package: `labs/lab-01-one-alu-two-testbenches/`

Before you start

- Finish DD104, and Lab 03 in particular. You don't need its files, because this lab ships the ALU and a reference copy of the finished Lab 03 test, but you'll get more from the comparison if you wrote that test yourself.
- Read Getting started, What a framework buys you and From cocotb to SystemVerilog classes.
- Copy this table into your notes. You'll fill it in as you go, and the questions at the end ask you to compare the two columns.

	cocotb	UVM
Lines (all / code)		
First build		
Rebuild after you change the testbench		
Run		
A01		
A02		
A03		

Time: about 1 h 30 min.

Part 1: Load the course

Can you reach DD105-A's files, and are your tools the versions that this lab's outputs were recorded with? If you haven't set up the workbench yet, do IC101 Lab 03 first.

1. From your `workspace` clone, start the desktop (`./scripts/start_vnc.sh`, or `scripts\start_vnc.bat` on Windows), open http://localhost/ and open a terminal on the desktop. Type every command from here on in that terminal.
2. Load the environment and get the course files:

```
. /foss/designs/common/.designinit   # in every new terminal
mod add dd105-a                     # first time only: clones github.com/uoftasic/
dd105-a
mod dd105-a                          # every time, including the first
```

   `mod dd105-a` prints:

```
cwd: /foss/designs/modules/dd105-a
```

3. Check the tool versions:

```
verilator --version
iverilog -V 2>&1 | head -1
cocotb-config --version
yices-smt2 --version | head -1
surfer --version
```

You should see:

```
Verilator 5.050 2026-07-01 rev v5.050
Icarus Verilog version 14.0 (devel) (s20260301-391-g64f13540-dirty)
2.0.1
Yices 2.7.0
surfer 0.7.0 (git: v0.7.0)
```

The outputs in this lab were recorded with these versions. On other versions the log lines can differ, and so can the random vectors that the UVM test sends and the error counts that follow from them. Yices is the constraint solver that Verilator runs for every constrained `randomize()` in a SystemVerilog testbench, and the course's makefile tells Verilator to use it.

If you see	Do this
`bash: /foss/designs/common/.designinit: No such file or directory`	The workspace isn't mounted, so relaunch `start_vnc` from your `workspace` clone. In the meantime you can run this lab from a plain `git clone` of `github.com/uoftasic/dd105-a` inside the workbench image, because `make` needs neither `.designinit` nor `mod`.
`bash: mod: command not found`	You haven't loaded `.designinit` in this terminal (the first line of step 2). Every new terminal starts without it.
`mod: no such module: dd105-a`	You ran `mod dd105-a` before `mod add dd105-a`.
`VERILATOR_SOLVER is "yices-smt2 --incremental", and yices-smt2 is not on PATH.`	You ran `make` outside the course image. Without a solver every constrained `randomize()` fails, nothing is randomized and the test still passes, so the makefile stops with this message instead. Run it in the workbench terminal.

For anything else, see IC101's Troubleshooting page.

Check: `verilator --version` prints `5.050` and you are in `/foss/designs/modules/dd105-a`.

Part 2: The cocotb testbench

What does the cocotb test cost to build and run, and what does it report on each bug?

1. Go to the lab folder:

```
cd labs/lab-01-one-alu-two-testbenches
```

2. Open `rtl/alu.sv`. It's the `ALU` module from DD104 (lines 790 to 823 of `simproc_system.sv`) with the same name and ports, so a cocotb test written for DD104 drives it without changes. Three of its lines now choose between the correct expression and a bug, and an `initial` block at the bottom reads `+BUG` from the simulator's command line at time 0. The bugs are DD104's three ALU bugs, renamed so you can't confuse them with the router's bugs later in the course:

Bug	What's wrong	DD104 name
A01	`SUB` computes `B - A` instead of `A - B`	`B01_sub_operands_swapped`

Bug	What's wrong	DD104 name
A02	`NAND` computes NOR instead of NAND	`B02_nand_is_nor`
A03	`Z` looks only at `ALUout[3:0]`	`B03_zero_flag_low_nibble`

With no `+BUG` you get the correct ALU, and with `+BUG=A03` you get A03. Every lab in this course compiles its bugs into the design this way, because a UVM build takes long enough that building one simulator per bug would cost you minutes each time.

3. `cocotb/test_alu.py` is DD104 Lab 03's reference solution, with only its docstring changed. If you'd rather run your own, copy it in under a new name and add `TB=test_alu_mine` to every `make cocotb` and `make loc` command in this lab:

```
cp      /foss/designs/modules/dd104/labs/lab-03-three-blocks-before-the-processor/
test_alu.py cocotb/test_alu_mine.py
```

The outputs below come from the reference. Your messages may be worded differently, but the counts should match.

4. Run the test on the correct ALU:

```
make cocotb
```

Icarus runs it, as in DD104, and it takes about 25 seconds. At the end you should see:

```
393224.00ns INFO    cocotb.regression            **************************************************
                                                                                    ** TEST
STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                     ****************************************************************
                                          ** test_alu.t01_one_vector_per_operation
PASS           6.00           0.00       6710.89  **
                                                                                       **
test_alu.t02_every_vector_the_alu_can_be_given   PASS      393216.00          17.50
22472.77  **
                                              ** test_alu.t03_the_undefined_opcodes
PASS           2.00           0.00       8264.64  **
                                     ****************************************************************
                                                    ** TESTS=3 PASS=3 FAIL=0
SKIP=0                               393224.00          17.50      22468.63  **
                                     ****************************************************************
ALU: no bug switched on, this is the golden ALU
=============================================================================
 cocotb, test_alu on icarus, the golden ALU: build 0.6 s, run 23.8 s
=============================================================================
```

The line between the last two rules is the lab's makefile reporting how long the build and the run each took. The build is Icarus compiling `rtl/alu.sv`, a 94-line file, which takes well under a second (a build with nothing to do took 0.4 s, all of it `make` itself). The run is mostly `t02` driving all 393,216 vectors, and the rest is starting Python and cocotb. Write both times in your table. Times are ballpark, but the counts in this lab are exact.

5. Before you run the test against the bugs, write down which of `t01`, `t02` and `t03` you expect to fail on each one. Then run it on all three:

```
make cocotb-all
```

It takes about 85 seconds. You should see:

```
*************************************************************************
** test_alu.t01_one_vector_per_operation           PASS          6.00          0.00
8089.30  **
** test_alu.t02_every_vector_the_alu_can_be_given   PASS     393216.00         17.42
22569.25  **
** test_alu.t03_the_undefined_opcodes              PASS          2.00          0.00
11199.74  **
** TESTS=3 PASS=3 FAIL=0 SKIP=0                              393224.00         17.43
22565.68  **
ALU: no bug switched on, this is the golden ALU
 cocotb, test_alu on icarus, the golden ALU: build 0.6 s, run 23.1 s
*************************************************************************
AssertionError: SUB(0xC8, 0x3C) gave out=0x74 N=0 Z=0, expected out=0x8C N=1 Z=0
AssertionError: 65024 of 393216 vectors have the wrong VALUE: {'SUB': 65024}.
** test_alu.t01_one_vector_per_operation           FAIL          2.00          0.00
2269.65  **
** test_alu.t02_every_vector_the_alu_can_be_given   FAIL     393216.00         17.84
22044.46  **
** test_alu.t03_the_undefined_opcodes              PASS          2.00          0.00
9446.63  **
** TESTS=3 PASS=1 FAIL=2 SKIP=0                              393220.00         17.85
22033.48  **
ALU: bug A01 switched on
 cocotb, test_alu on icarus, A01: build 0.4 s, run 19.9 s
*************************************************************************
AssertionError: NAND(0xC3, 0x1D) gave out=0x20 N=0 Z=0, expected out=0xFE N=1 Z=0
AssertionError: 65280 of 393216 vectors have the wrong VALUE: {'NAND': 65280}.
** test_alu.t01_one_vector_per_operation           FAIL          4.00          0.00
4213.26  **
** test_alu.t02_every_vector_the_alu_can_be_given   FAIL     393216.00         17.54
22419.88  **
** test_alu.t03_the_undefined_opcodes              PASS          2.00          0.00
8551.08  **
** TESTS=3 PASS=1 FAIL=2 SKIP=0                              393222.00         17.55
22408.87  **
ALU: bug A02 switched on
 cocotb, test_alu on icarus, A02: build 0.4 s, run 19.6 s
*************************************************************************
AssertionError: Every value is correct, and 25726 of 393216 vectors have the wrong
FLAGS: {'ADD': 3840, 'SUB': 3840, 'OR': 255, 'NAND': 255, 'SHL': 14400, 'SHR': 3136}.
** test_alu.t01_one_vector_per_operation           PASS          6.00          0.00
8040.20  **
** test_alu.t02_every_vector_the_alu_can_be_given   FAIL     393216.00         17.73
22172.35  **
** test_alu.t03_the_undefined_opcodes              PASS          2.00          0.00
10796.15  **
** TESTS=3 PASS=2 FAIL=1 SKIP=0                              393224.00         17.74
22161.85  **
ALU: bug A03 switched on
 cocotb, test_alu on icarus, A03: build 0.4 s, run 19.7 s
*************************************************************************
```

The counts 65,024, 65,280 and 25,726 are the ones you derived by hand in DD104 Lab 03, so the copied ALU and its switch behave like DD104's three broken ALUs. Write what the test reported on each bug in your table.

6. `t01` compares the value and both flags of six vectors, and it fails on A01 and A02 but passes on A03. Why does it miss A03?

   Answer: A1, in Answers at the end.

Check: `TESTS=3 PASS=3` on the correct ALU, and `t02` failing on all three bugs with 65024, 65280 and 25726 wrong vectors.

Part 3: The UVM testbench

What does the same job cost when it's built the UVM way?

1. Start the UVM build now, because it's the slowest step in the lab:

   ```
   make build
   ```

   The build starts with a banner that estimates 1.5 to 2.5 minutes for a cold build. On the course's 2 CPUs it took us 102 s, and it ended with:

   ```
   - V e r i l a t i o n   R e p o r t: Verilator 5.050 2026-07-01 rev v5.050
   - Verilator: Built from 24.157 MB sources in 386 modules, into 22.688 MB in 2501 C+
   + files needing 112.631 MB
   - Verilator: Walltime 102.681 s (elab=0.759, cvt=2.699, bld=98.503); cpu 4.510 s on
   2 threads; allocated 307.551 MB
   ================================================================================
    built /foss/designs/modules/dd105-a/labs/lab-01-one-alu-two-testbenches/build/sim
   in 102 s (47M; the 888M of generated C++ was deleted, SLIM=0 keeps it)
    ccache:  Hits: 0 / 36 ( 0.00%)  Misses: 36 / 36 (100.0%)
   ================================================================================
    /foss/designs/modules/dd105-a/labs/lab-01-one-alu-two-testbenches/build/sim is up
   to date.
   ```

   Verilator turned the whole UVM library, the course's shared files in `labs/common/uvm/`, the three testbench files and the ALU into C++, and `bld=98.503` says that g++ spent almost all of the 102 s compiling that C++. Verilator's report counts 24 MB of source in 386 modules for all of it, nearly all from the UVM library, while the ALU's file is 94 lines. ccache keeps what g++ compiled in `.ccache` at the top of your clone, so a later build can reuse whatever didn't change. Write the build time in your table.

2. While it builds, read `tb/alu_pkg.sv`, `tb/alu_if.sv` and `tb/alu_tb_top.sv`. Most pieces of the UVM testbench do a job that the cocotb test also does:

UVM piece	Its job	In `test_alu.py`
`alu_if`	the ALU's pins, plus a clock and a `valid` signal	`dut`
`alu_item`	one vector and the ALU's answer	the `(op, a, b)` and `got` values
`alu_rand_seq`, `alu_directed_seq`	make the vectors	the loops in `t02`, the list in `t01`
`alu_driver`	puts each vector on the pins	the three `.value =` lines in `alu()`
`alu_monitor`	reads the pins back into an `alu_item`	the `return` line of `alu()`
`alu_scoreboard`	`predict()` and the comparison	`spec()`, and the comparisons in `t01` and `t02`
`alu_agent`, `alu_env`	create the pieces and connect them; `alu_env` also checks that the scoreboard saw every vector the driver sent	nothing

UVM piece	Its job	In `test_alu.py`
`alu_rand_test`, `alu_directed_test`	choose the stimulus	the `@cocotb.test()` functions
`alu_tb_top`	the clock, the ALU, the interface and `run_test()`	cocotb's makefile

The ALU is combinational, so the cocotb test sets the inputs and waits 1 ns. A UVM driver and monitor work in clock cycles instead, so the interface adds a clock and a `valid` signal, and neither of them goes to the ALU. The driver writes through the interface's clocking block, which is how this course drives every design. Two comments in `alu_item` explain why its constraints are written the way they are. On Verilator 5.050, an enum that no constraint names can come out skewed, and a `dist` on a variable that another constraint also names can make `randomize()` fail. The tests extend `dd105_base_test`, which turns any `UVM_ERROR` into a nonzero exit status, because on this simulator a UVM run that reports errors otherwise still exits with status 0.

3. When the build has finished, run the random test, which sends 2000 random vectors:

```
make run
```

The first thing it prints is the UVM library's release notes, which you can ignore. You should see:

```
UVM_INFO @ 0: reporter [RNTST] Running test alu_rand_test...
UVM_INFO   labs/lab-01-one-alu-two-testbenches/tb/alu_pkg.sv(284)   @   20025000:
uvm_test_top.env.sb [SB] 2000 vectors checked: 0 value mismatches
DD105: TEST PASSED
```

followed by UVM's report summary and, last:

```
--------------------------------------------------------------------------
 ok: 0 UVM_ERROR, 0 UVM_FATAL   1.17 s   log: build/runs/alu_rand_test-s1/run.log
--------------------------------------------------------------------------
```

UVM prints times in picoseconds here, so `@ 20025000` is 20 µs, the time that 2000 vectors take at one per 10 ns clock cycle. Write the run time in your table.

4. Run the other test, without rebuilding:

```
make run UVM_TESTNAME=alu_directed_test
```

It sends DD104's six `t01` vectors. You should see `[SB] 6 vectors checked: 0 value mismatches` and `ok: 0 UVM_ERROR, 0 UVM_FATAL   0.04 s`, and the whole command took 0.17 s when we timed it. The test is picked by `+UVM_TESTNAME` when the simulator starts, so in this lab and every later one, trying another test, seed or bug never rebuilds anything.

5. Count each testbench's lines:

```
make loc
```

You should see:

```
cocotb testbench:
  file                lines    code
  cocotb/test_alu.py    189      94
UVM testbench:
  file              lines    code
  tb/alu_if.sv         28      17
  tb/alu_pkg.sv       377     252
```

```
  tb/alu_tb_top.sv      19      13
  total                424     282
```

`code` leaves out blank lines and comments, and counts Python docstrings as comments. The UVM testbench has three times as many code lines as the cocotb test. Using the table in step 2, where do the extra lines go?

Answer: A2, in Answers at the end.

6. Record a waveform of the random test and open it:

```
make waves
surfer build/runs/alu_rand_test-s1/waves.fst &
```

Add `clk` from `alu_tb_top`, `valid` from `aif` and the six ports from `u_alu`, and look at the first 120 ns:

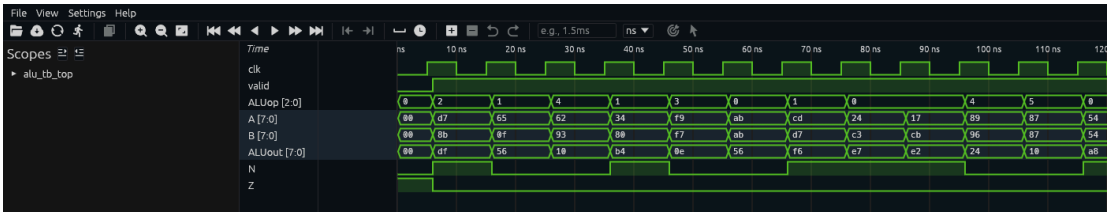


Figure 5.1: The first twelve vectors of alu_rand_test on the correct ALU, in Surfer

The correct ALU. The first vector, `ALUop 2 (OR)` with `A=d7` and `B=8b`, gives `df`.

Each vector holds the pins for one clock cycle. The driver writes it 1 ns after a rising edge, which is the clocking block's output skew, and the monitor samples it, with the ALU's answer, just before the next rising edge.

Check: `2000 vectors checked: 0 value mismatches` and `ok`, and your table has a line count, a build time and a run time in each column.

Part 4: Both testbenches on the bugs

Which bugs does the UVM testbench catch, and why?

1. Read `write()` in `alu_scoreboard`, in `tb/alu_pkg.sv`. Write down which of A01, A02 and A03 you expect it to catch.
2. Run the random test against the correct ALU and each bug:

```
make check-all
```

Each of the four runs takes about 1.1 s, because they all use the simulator you already built. You should see:

```
     ok  the golden design                    0 UVM_ERROR, 0 UVM_FATAL       1.1s
-------------------------------------------------------------------------------
  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.
```

`make` then prints an error line, because the verdict is a fail, not because the makefile broke.

3. Look at one of the catches in full:

```
make run BUG=A01
```

It prints 282 errors. The first three are:

```
UVM_ERROR    labs/lab-01-one-alu-two-testbenches/tb/alu_pkg.sv(260)    @    25000:
uvm_test_top.env.sb [VALUE] SUB A=65 B=0f -> ALUout=aa N=1 Z=0, expected ALUout=56
UVM_ERROR    labs/lab-01-one-alu-two-testbenches/tb/alu_pkg.sv(260)    @    45000:
uvm_test_top.env.sb [VALUE] SUB A=34 B=80 -> ALUout=4c N=0 Z=0, expected ALUout=b4
UVM_ERROR    labs/lab-01-one-alu-two-testbenches/tb/alu_pkg.sv(260)    @    75000:
uvm_test_top.env.sb [VALUE] SUB A=cd B=d7 -> ALUout=0a N=0 Z=0, expected ALUout=f6
```

and near the end of the log:

```
UVM_INFO   labs/lab-01-one-alu-two-testbenches/tb/alu_pkg.sv(284)   @   20025000:
uvm_test_top.env.sb [SB] 2000 vectors checked: 282 value mismatches (SUB 282)
...
** Report counts by id
[RNTST]     1
[SB]     1
[UVM/RELNOTES]     1
[VALUE]   282

DD105: TEST FAILED (282 UVM_ERROR, 0 UVM_FATAL)
[20025000]      %Fatal:      dd105_pkg.sv:89:      Assertion      failed      in
dd105_pkg.dd105_base_test.m_end_of_test: DD105: UVM counted errors, so the exit
status is nonzero
```

The `%Fatal` line is `dd105_base_test` making the exit status nonzero, as Part 3 described. UVM's report server has also counted the errors by their ID, `[VALUE]`, and in Part 5 you'll use that count by ID to keep flag errors apart from value errors.

4. The cocotb test's `t02` drove all 65,536 `SUB` vectors and found 65,024 of them wrong. The UVM test caught A01 on the second of its 2000 random vectors (`@ 25000` is 25 ns). Why does a random sample find A01 and A02 so easily, when no number of random vectors would find A03 with this scoreboard?

   Answer: A3, in Answers at the end.

5. Write what the UVM testbench reported on each bug in your table.

Check: `2 of 3 caught`, with A03 `MISSED`.

Part 5: Check the flags

What does it take to make the UVM scoreboard catch A03, and what does the change cost to rebuild?

1. See the verdict you're going to change:

   ```
   make check BUG=A03
   ```

   You should see:

   ```
        ok  the golden design                      0 UVM_ERROR, 0 UVM_FATAL       1.3s
    MISSED  A03_zero_flag_low_nibble               0 UVM_ERROR, 0 UVM_FATAL       1.2s
   ----------------------------------------------------------------------------
    VERDICT: FAIL -- your test passes A03_zero_flag_low_nibble too.
   ```

2. Extend `alu_scoreboard` in `tb/alu_pkg.sv`, following the `TODO` in `write()`. `predict()` already works out `n` and `z`, and the monitor already fills in `t.N` and `t.Z`, so you only need to compare them. Count flag mismatches separately from value mismatches, with a counter and a per-operation array like `n_bad_value` and `bad_value`, report each one with `` `uvm_error("FLAGS", ...) ``, and add the flag count

to the line that `report_phase` prints. Use a different ID from `"VALUE"`. UVM's report summary counts messages by ID, so it will then give you DD104's two separate counts without any code of your own.

3. You changed one file, and the ccache holds everything from the first build. Write down how long you expect the rebuild to take, then run it:

```
make build
```

With the reference solution's edit you should see:

```
 built /foss/designs/modules/dd105-a/labs/lab-01-one-alu-two-testbenches/build/sim
in 99 s (47M; the 888M of generated C++ was deleted, SLIM=0 keeps it)
 ccache:  Hits: 10 / 72 (13.89%)  Misses: 62 / 72 (86.11%)
```

An earlier run of the same edit took 87 s. The ccache counts add up across builds, so this build looked up 36 compiled files and found 10. Write your rebuild time in your table. Why did it cost almost as much as the first build?

Answer: A4, in Answers at the end.

4. Ask for the verdict again:

```
make check BUG=A03
```

With the reference solution you should see:

```
     ok  the golden design                      0 UVM_ERROR, 0 UVM_FATAL       1.2s
  caught  A03_zero_flag_low_nibble               131 UVM_ERROR                 1.2s
       UVM_ERROR [FLAGS] SHL A=62 B=93 -> ALUout=10 N=0 Z=1, expected N=0 Z=0
       UVM_ERROR [FLAGS] SHR A=87 B=87 -> ALUout=10 N=0 Z=1, expected N=0 Z=0
       UVM_ERROR [FLAGS] SHL A=30 B=60 -> ALUout=30 N=0 Z=1, expected N=0 Z=0
--------------------------------------------------------------------------------
 VERDICT: PASS
```

If the golden row says `FAIL` instead of `ok`, your check fires on the correct ALU, so compare it with `predict()` again. If `make build` stopped with an error, fix the line it names and build again.

5. Run all three bugs again:

```
make check-all
```

You should see `3 of 3 caught`. The reference solution reports one `[FLAGS]` error for each vector whose flags are wrong, and with it the counts are 564, 394 and 131 `UVM_ERROR`s for A01, A02 and A03. Before your change, A01 and A02 had 282 and 255. Why is A01's count exactly double, when A02's isn't?

Answer: A5, in Answers at the end.

6. Record the failure and open it:

```
make waves BUG=A03
surfer build/runs/alu_rand_test-s1-A03/waves.fst &
```

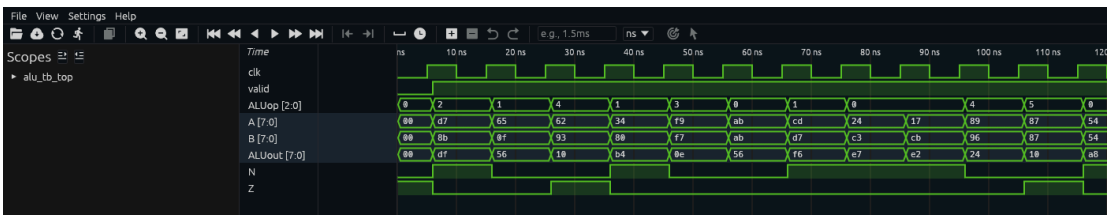


Figure 5.2: The same vectors on the ALU with A03 switched on, in Surfer

A03. The seed is the same as in Part 3's picture, so the vectors are the same too. `Z` goes high at 26 ns and again at 106 ns, where `ALUout` is `10`.

The first `[FLAGS]` error is stamped `@ 35000`, which is 35 ns. The driver wrote the `SHL A=62 B=93` vector at 26 ns, and the monitor sampled it just before the rising edge at 35 ns. Compare this with Part 3's picture, where `Z` stays low once the first vector arrives at 6 ns.

7. `alu_directed_test` sends the same six vectors as `t01`, and it now goes through your flag check, because both tests share the scoreboard. Predict whether it catches A03, then run it:

```
make run UVM_TESTNAME=alu_directed_test BUG=A03
```

You should see `[SB] 6 vectors checked: 0 value mismatches, 0 flag mismatches` and `MISSED: 0 UVM_ERROR, 0 UVM_FATAL`. It misses A03 for the same reason that `t01` does, which you worked out at the end of Part 2.

Check: `VERDICT: PASS` from `make check BUG=A03`, and `3 of 3 caught` from `make check-all`.

Finishing

You're done when `make check BUG=A03` prints `VERDICT: PASS` with the scoreboard you extended in `tb/alu_pkg.sv`.

Questions:

1. Using your table, what did the UVM structure cost you in this lab, and what did it give you that the cocotb test didn't?

   Answer: A6, in Answers at the end.

2. On A03 your scoreboard counted 131 flag mismatches in 2000 vectors, and the cocotb test counted 25,726 in 393,216. What can each number tell you?

   Answer: A7, in Answers at the end.

Submit: nothing for this lab.

Going further (optional)

Run the cocotb test on Verilator instead of Icarus:

```
make cocotb COCOTB_SIM=verilator
```

Its first build took 13.3 s and its run 17.3 s:

```
                                                 ** TESTS=3 PASS=3 FAIL=0 SKIP=0
393224.00          11.48      34242.85  **
                                    ********************************************************************
ALU: no bug switched on, this is the golden ALU
- :0: Verilog $finish
******************************************************************************
 cocotb, test_alu on verilator, the golden ALU: build 13.3 s, run 17.3 s
******************************************************************************
```

Verilator compiles only the ALU for cocotb, so this shows how much of the UVM build's 102 s goes to the UVM library rather than to the design.

Next on the Course path: the reading on classes, interfaces and threads, then Lab 02: A class testbench without UVM.

Answers

A1. Part 2: The cocotb testbench

The six results are `0x04`, `0x8C`, `0xFF`, `0xFE`, `0x84` and `0x1F`. A03 gets `Z` wrong only when the result isn't zero but its low four bits are, and none of the six results is like that, so on these vectors the broken `Z` and the correct one agree.

A2. Part 3: The UVM testbench

Mostly into pieces that the cocotb test doesn't need. The agent only creates and connects the other pieces, and the environment does the same plus one check, that the scoreboard saw every vector that the driver sent. Every class has a constructor and a `` `uvm_component_utils `` or `` `uvm_object_utils `` line. The driver and monitor take two classes to do what the five lines of `alu()` do, because cocotb reads and writes the design's signals directly from Python.

A3. Part 4: Both testbenches on the bugs

A01 and A02 change the value on almost every `SUB` and `NAND` vector (65,024 and 65,280 of 65,536 each), so any sample with a handful of those operations in it shows the bug. A03 changes only `Z`, and this scoreboard never reads `Z`. DD104 Lab 03, Part 5 showed the same thing with a sweep of all 393,216 vectors that checked only the values.

A4. Part 5: Check the flags

It cost almost a full build because the edit adds members, the new counters, to a class. We timed two one-line edits to compare. A change inside a task's body, `repeat (3)` to `repeat (4)` in the `run_phase` of `alu_base_test`, rebuilt in 26 s, with 35 of the 36 compiled files taken from the ccache. Adding one `int unsigned` member to the scoreboard rebuilt in 93 s, with 10 of 36 from the ccache. Adding a member changes the class's layout, and most of the C++ that Verilator generates changes with it. So where you can, iterate by changing plusargs or the bodies of tasks and functions, and expect a change to what a class holds to cost close to a full build. If your own edit changed only the bodies of tasks and functions, your rebuild was closer to 26 s. A cocotb test needs no build at all when you change it, because Python isn't compiled.

A5. Part 5: Check the flags

A wrong value can bring wrong flags with it, and then the vector is reported twice, once as `[VALUE]` and once as `[FLAGS]`. Under A01 the ALU returns `B - A`, the negative of the right answer, and a byte and its negative differ in bit 7 unless the byte is `0x00` or `0x80`, which are the cases A01 gets right. So every wrong `SUB` also has a wrong `N`, and A01′s 282 `[VALUE]` errors come with 282 `[FLAGS]` errors. Under A02, bit 7 of `~(A | B)` and `~(A & B)` differ only when exactly one of `A[7]` and `B[7]` is 1, so only some of the 255 wrong `NAND` values also have wrong flags.

A6. Finishing

It cost about three times the code (282 lines against 94), a first build of about 100 s against under a second, and a rebuild of 26 to 99 s every time you changed the testbench, against nothing for a Python test. It isn't faster per vector either, since 2000 vectors in about 1.2 s is about 0.6 ms each, and cocotb's 393,216 in about 17 s is about 0.04 ms each.

You saw what it gave you in Part 5. The flag check went into one class, and both tests used it without being edited, because stimulus and checking are separate pieces. In `test_alu.py`, by contrast, `t01` and `t02` each do their own comparison. `alu_directed_test` is one sequence and one short test class that reuse the driver, monitor and scoreboard, and it's chosen when you run, without a rebuild. The report server counted your two kinds of mismatch by ID for you. On one combinational block that's a modest return. The structure is meant for designs like the router that you'll test from Lab 02 on, which has a register bus, an input stream and two output streams.

A7. Finishing

131 of 2000 is 6.6%, and 25,726 of 393,216 is 6.5%, so the random run estimates the fraction of vectors that A03 breaks, from whichever 2000 vectors its seed picked. Only the sweep gives the exact count, which you can derive from the bug by hand, as you did in DD104 Lab 03, Part 6. Both runs report no value mismatches and some flag mismatches, and that combination is what tells you the bug is in the flag logic.

II 2. Classes

6 **Classes for testbenches 41**
6 Modules and classes 41
6 A class 41
6 Handles 42
6 The same object kept three times 43
6 Null handles 43
6 Functions and tasks 43
6 Static members 44
6 Inheritance and virtual methods 44
6 Check yourself 45
6 Further reading 45
6 Answers 45

7 **Interfaces, clocking blocks and virtual interfaces 47**
7 Interfaces 47
7 Virtual interfaces 48
7 Clocking blocks 48
7 Module code and nonblocking assignments . . . 50
7 Check yourself 50
7 Further reading 50
7 Answers 51

8 **Threads, mailboxes and events 53**
8 fork and its three joins 53
8 Threads forked in a loop 54
8 Mailboxes 55
8 Events 56
8 Stopping threads 56
8 Check yourself 56
8 Further reading 57
8 Answers 57

9 **Lab 02: A class testbench without UVM 59**
9 Before you start 59
9 Part 1: The starter 59
9 Part 2: The monitor 61
9 Part 3: The scoreboard 62
9 Part 4: What each catch depends on 64
9 Finishing 65
9 Going further (optional) 66
9 Answers 66

6. Classes for testbenches

This page answers: *what does a class give a testbench that a module doesn't, and what do I need to know about handles before I write a monitor and a scoreboard?*

Lab 01's UVM testbench was made of classes. You read them in Part 3, and in Part 5 you added the flag checks to one of them, `alu_scoreboard`. In Lab 02 you'll write the router's monitor and scoreboard, two plain SystemVerilog classes with no UVM in them. This page covers what those two classes need: objects that are made while the simulation runs, the handles that point at them, and the difference between functions and tasks. Static members and inheritance with virtual methods come last. The two classes you'll write use neither, but Lab 01's tests shared one `run_phase` through inheritance, and UVM calls your `build_phase` and `run_phase` through virtual methods.

We ran every example on this page in the course image. The source is in `guide-examples/classes-for-testbenches/`. The commands that produced each output are at the top of that folder's Makefile, and in the README one level up, `guide-examples/README.md`.

Modules and classes

A module behaves like hardware. Its instances are fixed when the design is elaborated, before simulated time starts, and each one exists for the whole run. That suits a design, and it suits the top module of a testbench, which holds the clock, the interfaces and the design under test.

A class is a type. The things made from it are called **objects**, and they're created while the simulation runs, as many as the testbench needs. Most of a testbench is written as classes for that reason. Lab 02's generator makes a new `packet` object for every packet the test sends, 100 of them by default, and puts each one in a mailbox for the driver. Your monitor will make one for each packet it sees on a stream. A transaction like that has no fixed place in the hardware. It exists from the moment something makes it until nothing refers to it any more.

The classes of a testbench go in a package. Lab 02's `router_tb_pkg.sv` is a list of `` `include `` lines for `packet.svh`, `generator.svh` and the rest, so any file that imports `router_tb_pkg` can use every class in those files.

A class

`handles_test`, `reuse_test` and `null_test` use this class. It's a cut-down version of Lab 02's `packet`, with a serial number added:

```
class pkt;
  static int unsigned n_made;   // one variable, shared by every pkt
  int unsigned        id;       // one per object
  bit [7:0]           data[$];

  function new();
    id = n_made;
    n_made++;
  endfunction

  function string sprint();
    string s;
    s = $sformatf("pkt %0d:", id);
    foreach (data[i]) s = {s, $sformatf(" %02h", data[i])};
    return s;
  endfunction
endclass
```

It has three **members** (`n_made`, `id` and `data`, a queue of bytes) and two **methods** (`new()` and `sprint()`). `new()` is the constructor, and it runs once each time you call `new()` to make an object. A constructor can take arguments, and Lab 02 uses them to connect its pieces. `env::new()` makes three mailboxes and passes each one to the constructors of the two classes that use it, and every class that touches the pins is given its virtual interface the same way. When an argument has the same name as a member, `this.` picks out the member, so `this.vif = vif` copies the argument `vif` into the member `vif`, as Python's `self.vif = vif` does.

Handles

`pkt a;` declares a **handle**, a variable that can point at a `pkt` object. It doesn't make a packet. Until something is assigned to it, the handle points at nothing, and its value is `null`. `new()` makes an object and returns a handle to it. After that, assigning one handle to another copies the handle and not the object. The example `handles_test` does this:

```
pkt a, b, c;
$display("a is %s", a == null ? string'("null") : string'("an object"));
a = new();
a.data = '{8'h80, 8'h11};
b = a;
b.data.push_back(8'h22);
c = new a;
c.data.push_back(8'h33);
```

and prints:

```
a is null
a: pkt 0: 80 11 22
b: pkt 0: 80 11 22
c: pkt 0: 80 11 22 33
a == b is 1, a == c is 0, pkt::n_made is 1
```

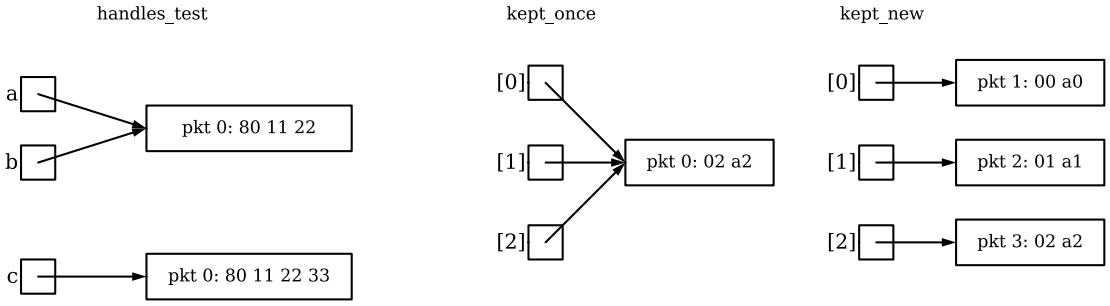


Figure 6.1: The objects that handles_test and reuse_test leave behind

Left, `handles_test`: `a` and `b` point at one object, and `c` points at a copy of it. Right, `reuse_test`: `kept_once` holds three handles to one reused object, and `kept_new` holds three handles to three objects. The values are the ones the tests printed.

`b = a` gave the one object a second name, so the `22` pushed through `b` shows up when you print `a`. `a == b` compares the handles, and it's 1 because they point at the same object. Two different objects with the same contents compare unequal, which is why Lab 02's `packet` has a `same()` method that compares the beats one by one.

`c = new a` makes a new object and copies every member of `a` into it, so `c` has its own queue, and the `33` pushed onto it doesn't appear in `a`. The copy doesn't run the constructor, so `c` kept `a`'s `id` of 0, and `n_made` stayed at 1. It's also a *shallow* copy, which means that if `pkt` had a member that was itself a handle, `c` would get a copy of that handle, and the two objects would share whatever it pointed at.

You won't often copy objects in the labs, but you'll pass handles around all the time, and every place that keeps a handle to an object shares that object with every other place that kept one. The next example shows what happens when you forget that, and it's the class bug you're most likely to write in Lab 02.

The same object kept three times

`reuse_test` has two loops, and each one stands in for a monitor. Each loop builds the same three two-byte packets and keeps them in a queue, as Lab 02's scoreboard keeps the packets it expects. The first loop makes one `pkt` before it starts and reuses it for every packet, and the second makes a new `pkt` for each packet:

```
p = new();
for (int k = 0; k < 3; k++) begin
  p.data.delete();
  p.data.push_back(8'(k));
  p.data.push_back(8'hA0 + 8'(k));
  kept_once.push_back(p);
end

for (int k = 0; k < 3; k++) begin
  p = new();
  p.data.push_back(8'(k));
  p.data.push_back(8'hA0 + 8'(k));
  kept_new.push_back(p);
end
```

```
one pkt, reused:   [0] pkt 0: 02 a2
one pkt, reused:   [1] pkt 0: 02 a2
one pkt, reused:   [2] pkt 0: 02 a2
a new pkt each:    [0] pkt 1: 00 a0
a new pkt each:    [1] pkt 2: 01 a1
a new pkt each:    [2] pkt 3: 02 a2
```

`kept_once` holds three handles to one object, and that object holds the last packet, `02 a2`, so the queue shows it three times and the first two packets are gone. A mailbox behaves the same way, because what you put into it is a handle. That's why the `TODO` in Lab 02's `monitor.svh` has you start a new packet when `p` is null, and set `p = null` once you've put a finished packet into the mailbox. The next packet then gets an object of its own, and the object the scoreboard was given keeps the beats it had.

Null handles

If you use a handle before it points at anything, the run stops. `null_test` declares `pkt p;`, prints a line, and then asks for `p.data.size()`. This is everything in its log after UVM's `Running test null_test...` line:

```
%Error: guide-examples/classes-for-testbenches/classes_demo_pkg.sv:112: Null pointer
dereferenced
Aborting...
```

The file and line in the error are where the null handle was used. Nothing after that line ran, including UVM's report summary. The labs' graders mark a run like this BROKEN rather than caught, because a null handle is an error in the testbench and not a check that fired. The line that the test printed just before the error isn't in the log either, so when a run aborts, look at the line the error names, not at the last thing that was printed. In Lab 02 you'll see this error if your monitor pushes a beat onto `p.data` before it has made `p`.

Functions and tasks

A class has two kinds of method. A **function** runs in zero simulated time and can return a value. `sprint()` and Lab 02's `packet::same()` are functions. A **task** can wait for a clock edge with `@`, for a delay with `#`, or for a mailbox to have something in it, so any method that follows the simulation through time has to be a task. In Lab 02 the `run()` methods of the monitor, the driver and the scoreboard are all tasks, and the scoreboard's `run()` waits on two mailboxes.

Verilator checks this rule before it builds anything, so a function with `#10;` in it fails `make lint` with this error:

```
%Error-FUNCTIMECTL: guide-examples/classes-for-testbenches/classes_demo_pkg.sv:120:5:
Functions cannot contain time-controlling statements (IEEE 1800-2023 13.4)
```

Methods of a class are *automatic*, which means that each call gets its own copy of the method's local variables. That's what lets two threads run the same task at the same time without disturbing each other, and Threads, mailboxes and events shows where it matters.

Static members

A `static` member belongs to the class rather than to any one object, so there's one `n_made` however many packets exist, and code outside the class reads it as `pkt::n_made`. The usual use is a counter like this one, which numbers objects as they're made. Members that aren't static exist once per object, so every `pkt` has its own `id` and `data`.

Inheritance and virtual methods

A class can **extend** another class. It inherits all of that class's members and methods, and it can override any of those methods with a version of its own. In `dispatch_test`, the class `base_test` has a `run()` task that calls `name()`, which isn't virtual, and `stimulus()`, which is declared `virtual`. The class `rand_test` extends `base_test` and overrides both:

```
class base_test;
  function string name();
    return "base_test";
  endfunction
  virtual task stimulus();
    $display("  stimulus() of base_test: no stimulus");
  endtask
  task run();
    $display("run() of %s", name());
    stimulus();
  endtask
endclass

class rand_test extends base_test;
  function string name();
    return "rand_test";
  endfunction
  virtual task stimulus();
    $display("  stimulus() of rand_test: 2000 random vectors");
  endtask
endclass
```

The test makes one `rand_test` object and calls `run()` on it twice, once through a `base_test` handle and once through a `rand_test` handle:

```
through a base_test handle:
run() of base_test
  stimulus() of rand_test: 2000 random vectors
through a rand_test handle:
run() of base_test
  stimulus() of rand_test: 2000 random vectors
```

`run()` is defined only in `base_test`, so both calls run that code. Inside it, the call to `stimulus()` went to `rand_test`'s version, because `stimulus()` is virtual, and a virtual call goes to the method of the object's own class, whatever type of handle you called it through. The call to `name()` went to `base_test`'s version both times, even though the object is a `rand_test`. A call to a method that isn't virtual is bound when the code is compiled, by the type of the handle it's made through. Inside `run()`, `name()` is called through `this`, and in code written in `base_test`, `this` is a `base_test` handle.

Lab 01′s two tests share one run phase this way. `alu_base_test::run_phase()` raises an objection, calls `stimulus()`, waits three cycles and drops the objection. `stimulus()` is a virtual task, and `alu_rand_test`

and `alu_directed_test` each override it to start their own sequence. UVM calls your code the same way. `build_phase`, `run_phase` and the other phase methods are declared virtual in the UVM library, so when UVM calls them on your components, it runs your versions.

A method that overrides another can still call the version it replaced, through `super`. Lab 01's `alu_base_test::build_phase()` starts with `super.build_phase(phase)`, which runs `dd105_base_test`'s `build_phase()`, where the course checks that the constraint solver works. If you leave the `super` call out, the check doesn't run, and no message tells you it was skipped.

Check yourself

1. A scoreboard reports that every packet it expected has the same beats as the last packet. Its monitor runs `p = new()` once, in the monitor's constructor. What's wrong, and how do you fix it?

   Answer: A1, in Answers at the end.

2. In `dispatch_test`, the call through the `rand_test` handle still printed `run() of base_test`. Why, and what would it print if `name()` were declared `virtual`?

   Answer: A2, in Answers at the end.

3. After `c = new a`, `c.id` is 0, the same as `a.id`, and `pkt::n_made` is still 1. Why didn't the copy get a new number?

   Answer: A3, in Answers at the end.

4. Which of these has to be a task: the monitor's `run()`, `packet::sprint()`, the scoreboard's `predict()`?

   Answer: A4, in Answers at the end.

Further reading

ChipVerify has a page on each of the class features below, and most of them include a complete example. The logs on its SystemVerilog pages mostly come from Cadence simulators, not from Verilator, and the course doesn't reuse its code.

- Class and Class Handle: objects, handles, `null`, and two handles to one object.
- Class Constructor: `new()`, default arguments, and `super.new()`.
- Copying Objects: shallow copies with `new`, and deep copies written by hand.
- Static Variables & Functions.
- Inheritance, Virtual Methods and super Keyword. The Virtual Methods page runs the same code with and without `virtual`, as `dispatch_test` does.
- Polymorphism: base handles to derived objects, and `$cast`. Correction: it says that a failed `$cast` returns 0 and the simulation goes on, which is true only when `$cast` is called as a function. Called as a task, a failed cast is a run-time error, as the page's own log shows. Cast Dynamic gets this right.

Next: Interfaces, clocking blocks and virtual interfaces.

Answers

A1. Check yourself

The monitor puts the same object into the mailbox for every packet and refills it each time, so every handle the scoreboard kept points at one object, and that object holds whatever was written into it last. Make a new `packet` for each packet the monitor sees, and stop using the old one once it's in the mailbox, which is what setting `p = null` after the `put()` does.

A2. Check yourself

`run()` is `base_test`'s code. Inside it, `name()` is called through `this`, and in `base_test`'s code `this` is a `base_test` handle. Because `name()` isn't virtual, the call was bound to `base_test::name()` when the class was compiled. The type of the handle you called `run()` through doesn't come into it. If `name()` were virtual, the call would go to the object's own class, and both lines would read `run() of rand_test`.

A3. Check yourself

`new a` copies the members of `a` into a new object without calling the constructor, and the constructor is the only code that numbers objects. If a copy needs its own number, give `pkt` a copy method of its own that makes the object with `new()` and then copies the data.

A4. Check yourself

Only the monitor's `run()`, because it waits for clock edges. `sprint()` builds a string, and `predict()` decides whether a packet should come out of output 0. Neither has to wait for anything, so both are functions, and that's how Lab 02 ships them.

7. Interfaces, clocking blocks and virtual interfaces

This page answers: *a class isn't part of the design. How does it drive the design's pins, and read them back, without racing the design's flip-flops?*

In Lab 01 you read a driver that wrote `vif.drv_cb.A <= req.A` and a monitor that read `vif.mon_cb.ALUout`. In Lab 02 you'll write a monitor for the router's valid/ready streams, and it will read each stream's signals through names like `vif.cb.valid`, `vif.cb.ready` and `vif.cb.data`. Each of these names has three parts. In `vif.drv_cb.A`, `A` is a signal in an **interface**, which bundles the design's signals. `vif` is a **virtual interface**, which is how a class gets hold of one particular instance of that interface. `drv_cb` is a **clocking block**, which says when the testbench samples and drives the signals. This page explains them in that order.

The example on this page drives the golden router from `labs/common/rtl`, and we ran it in the course image. Its source is in `guide-examples/interfaces-clocking-blocks-and-virtual-interfaces/`. The commands that produced each output are at the top of that folder's Makefile, and in the README one level up, `guide-examples/README.md`.

Interfaces

An interface is a named bundle of signals that you declare once and instantiate like a module. The example's interface, `rtr_pins_if`, holds the signals of the router's APB port, its input stream and both of its output streams. Shortened, it reads:

```
interface rtr_pins_if (input logic clk, input logic rst_n);
  logic        psel = 1'b0, penable = 1'b0, pwrite = 1'b0;
  logic [7:0]  paddr = 8'h00;
  logic [31:0] pwdata = 32'h0;
  logic        pready;
  logic        in_valid = 1'b0, in_last = 1'b0;
  logic [7:0]  in_data = 8'h00;
  logic        in_ready;
  logic        out0_valid;
  logic [7:0]  out0_data;
  ...
  clocking cb @(posedge clk);
    default input #1step output #1;
    output psel, penable, pwrite, paddr, pwdata, in_valid, in_data, in_last;
    input  pready, in_ready, out0_valid, out0_data, rst_n;
  endclocking
endinterface
```

The top module makes an instance of it called `pins`, with `rtr_pins_if pins (clk, rst_n);`, and connects the router's ports to the signals in that instance: `.in_valid(pins.in_valid)`, `.in_ready(pins.in_ready)` and so on. Any code that can see `pins` can reach every signal in it.

An interface can also say who drives what. Lab 01's `alu_if` does it with two modports, `DRV` and `MON`. `DRV` gives access only to the clocking block `drv_cb`, whose signals are all outputs, and `MON` only to `mon_cb`, whose signals are all inputs. The driver gets `alu_if.DRV` and the monitor gets `alu_if.MON`. Lab 02 does it for the router's streams with three interface types instead. In a `stream_src_if`, which carries the input stream, the testbench drives `valid`, `data` and `last`. In a `stream_snk_if`, which carries an output stream, the testbench drives only `ready`. Every signal of a `stream_mon_if` is an input. The top module makes one `stream_mon_if` for the input stream and another for output 0, and the monitor class holds a `stream_mon_if`, so the same code watches either stream and can't drive it by mistake.

Virtual interfaces

A class can't contain an interface instance, and the standard doesn't let code in a package name anything in the module hierarchy, so a class in a package can't reach `pins` by its name either. Instead it holds a **virtual interface**, which is a variable of type `virtual rtr_pins_if`. Its value is a handle to one particular instance of `rtr_pins_if`, and it's assigned while the simulation runs.

```
class edge_demo;
  virtual rtr_pins_if vif;

  function new(virtual rtr_pins_if vif);
    this.vif = vif;
  endfunction
  ...
endclass
```

The course hands each virtual interface to its classes in two steps. First the top module stores each instance in UVM's configuration database, before it starts the test:

```
uvm_config_db#(virtual rtr_pins_if)::set(null, "uvm_test_top", "pins_vif", pins);
run_test();
```

Then the test fetches it and passes it on to the classes that need it:

```
if (!uvm_config_db#(virtual rtr_pins_if)::get(this, "", "pins_vif", vif))
  `uvm_fatal("NOVIF", "the top module didn't set pins_vif")
d = new(vif);
```

Lab 02's `router_class_test` does this for each of its six interfaces and hands them to `env::new()`, which passes each one to the class that uses it. In Lab 01 the UVM driver and monitor each fetch their own interface from the database, in `build_phase`.

A typedef of a virtual interface type

On Verilator 5.050, a class in a package reads wrong values from a clocking block if it reaches the interface through a `typedef` of the virtual interface type. The example's `typedef_test` waits for the end of reset and then sets `CTRL.EN`, while two monitor classes in one package print what their clocking block sampled at each rising edge. One declares its handle as `virtual rtr_mon_if`, and the other as `rtr_mon_vif_t`, after `typedef virtual rtr_mon_if rtr_mon_vif_t;`. The interface has two identical clocking blocks, `cb_a` and `cb_b`. The first class reads `cb_a` and the second reads `cb_b`, so each block is only ever read through one kind of handle. Apart from that, the two classes are the same. They printed:

```
45 ns: virtual rtr_mon_if: rst_n 1 in_ready 0 out0_valid 0
45 ns: rtr_mon_vif_t:      rst_n 0 in_ready 0 out0_valid 0
55 ns: virtual rtr_mon_if: rst_n 1 in_ready 0 out0_valid 0
55 ns: rtr_mon_vif_t:      rst_n 0 in_ready 0 out0_valid 0
65 ns: virtual rtr_mon_if: rst_n 1 in_ready 1 out0_valid 0
65 ns: rtr_mon_vif_t:      rst_n 0 in_ready 0 out0_valid 0
75 ns: virtual rtr_mon_if: rst_n 1 in_ready 1 out0_valid 0
75 ns: rtr_mon_vif_t:      rst_n 0 in_ready 0 out0_valid 0
```

The reset was over at all four edges. The class that uses `virtual rtr_mon_if` read `rst_n` as 1 at each of them, and `in_ready` as 1 from 65 ns on, once `CTRL.EN` was set. The class that uses the typedef read 0 from every clocking-block input at every edge. Verilator gave no warning at build time and no error at run time. That's why every class in the labs writes its virtual interface type out in full, as in `virtual stream_mon_if`, and the comment at the top of Lab 02's `driver.svh` gives this reason.

Clocking blocks

When a rising edge comes, the design's flip-flops take their new values and your testbench, which was waiting for that edge, wakes up. Both happen at the same moment of simulated time. If the testbench then reads a flip-flop's output straight from the pin, it can get either the old value or the new one. Which one

it gets depends on when the testbench woke up within that moment, and that depends on what it was waiting for.

The example's `edge_test` shows what Verilator does. After it sets up a route, it writes 1 to `CTRL.EN`, which takes effect at the rising edge that ends the write and makes `in_ready` 1 from then on. Then it offers a two-beat packet. At every rising edge it prints `in_ready` and `out0_valid` twice, once as the clocking block sampled them (`cb`) and once read straight from the pins at the same moment (`pin`). It also prints the `in_valid` pin.

```
65 ns edge: in_ready cb 0 pin 0 | in_valid pin 0 | out0_valid cb 0 pin 0
75 ns: CTRL.EN written
75 ns edge: in_ready cb 0 pin 1 | in_valid pin 0 | out0_valid cb 0 pin 0
76 ns: in_valid pin changes to 1
85 ns: handshake on beat 00
85 ns edge: in_ready cb 1 pin 1 | in_valid pin 1 | out0_valid cb 0 pin 0
95 ns: handshake on beat 5a
95 ns edge: in_ready cb 1 pin 1 | in_valid pin 1 | out0_valid cb 0 pin 1
96 ns: in_valid pin changes to 0
105 ns edge: in_ready cb 1 pin 1 | in_valid pin 0 | out0_valid cb 1 pin 1
115 ns edge: in_ready cb 1 pin 1 | in_valid pin 0 | out0_valid cb 1 pin 0
```

At 75 ns the pin already reads 1, because the flip-flop had updated before the class read it. The clocking block reads 0, which is what `in_ready` was during the cycle that ended at 75 ns. The router's specification defines the value of a signal "in cycle n" the same way, as the value it holds just before the edge that ends cycle n. A monitor needs that value, because a handshake is a cycle in which `valid` and `ready` are both 1. A monitor that read `out0_valid` from the pin would get each value one cycle early (look at 95 ns and 105 ns), and it would pair each value with the `ready` of the wrong cycle.

The first two lines of the clocking block set this timing:

```
clocking cb @(posedge clk);
  default input #1step output #1;
```

- `@(posedge clk)` is the clocking event, and `@(vif.cb)` waits for it.
- `input #1step` makes every input in the block a sample taken just before the edge, so `vif.cb.in_ready` is its value in the cycle that the edge ends, whatever the flip-flops do at the edge itself.
- `output #1` makes every drive through the block take effect 1 ns after the edge. The class wrote `vif.cb.in_valid <= 1` right after the edge at 75 ns, and the pin changed at 76 ns. So nothing that the testbench drives through the block changes at the moment the block samples its inputs.

Here is the same run in Surfer, from 60 ns to 120 ns:



Figure 7.1: The router's APB write of CTRL.EN and a two-beat packet, driven through a clocking block, in Surfer

`in_ready` rises exactly at the 75 ns edge, because it comes from a flip-flop. `psel` and `penable` fall and `in_valid` rises at 76 ns, 1 ns after the edge, because the class drove them through the clocking block. The router takes `00` at 85 ns and `5a` at 95 ns, and `out0_valid` is high from 95 ns to 115 ns.

With both skews in place, a driver is a loop that does one clock cycle per pass. Each pass drives what the next cycle should hold, waits for the edge with `@(vif.cb)`, and then looks at what the clocking block sampled. A monitor's loop is the same without the drive. The example offers one beat with this task:

```
task send_beat(bit [7:0] data, bit last);
  vif.cb.in_valid <= 1'b1;
```

```
  vif.cb.in_data  <= data;
  vif.cb.in_last  <= last;
  do @(vif.cb); while (!vif.cb.in_ready);
endtask
```

Each pass of the `do` loop waits for an edge and then checks whether `in_ready` was 1 in the cycle that just ended. If it was, `in_valid` was 1 in that cycle too, so that cycle was the handshake and the beat moved at that edge. Lab 02's `driver::run()` offers each beat with the same loop, except that it stops with an error if `in_ready` stays 0 for 1,000 cycles.

Module code and nonblocking assignments

There's one more reason that the classes in the labs drive the design only through clocking blocks. On Verilator 5.050, a module task can lose assignments if it raises a signal for one cycle with nonblocking assignments and is called several times in a `repeat` loop. The example's top module raises one signal, `by_task`, three times by calling this task in a `repeat` loop:

```
task automatic pulse_by_task();
  by_task <= 1'b1;
  @(posedge clk);
  by_task <= 1'b0;
endtask
```

It raises a second signal three times through a clocking block, and then counts the rising edges at which each signal was high:

```
module task, <= on the signal:          high at 1 of 3 edges
module task, through a clocking block:  high at 3 of 3 edges
```

Each call's `by_task <= 1` runs after the previous call's `by_task <= 0`, in the same time step, so by the standard the signal stays high at all three edges. Verilator commits the two in source order instead, so `by_task <= 0` takes effect last, and two of the three pulses vanish. `labs/common/README.md` lists the shapes the course measured. Of those, only the module task called in a `repeat` loop lost assignments. The same task with its three calls written out kept them all, and so did every class driver that was tried. A clocking block gave three of three everywhere, and its drives don't depend on the order in which Verilator commits two nonblocking assignments in one time step.

Check yourself

1. At the 75 ns edge, `edge_test` printed `in_ready cb 0 pin 1`. What was `in_ready` in the cycle that ended at 75 ns, and why would a monitor that used the pin value be wrong?

   Answer: A1, in Answers at the end.

2. A driver executes `vif.cb.in_valid <= 1` right after the edge at 75 ns, and `in_ready` is 1 from then on. When does the pin change, and at which edge does the beat move?

   Answer: A2, in Answers at the end.

3. Lab 02's monitor holds a `virtual stream_mon_if` rather than the `stream_snk_if` that the sink holds. What does that buy?

   Answer: A3, in Answers at the end.

4. Your monitor reads 0 from every clocking-block input, though the waveform shows the signals going to 1. What's the first thing to look for in your code?

   Answer: A4, in Answers at the end.

Further reading

ChipVerify's pages on interfaces explain each construct at more length. Most of the logs on its SystemVerilog pages come from Cadence simulators rather than Verilator, and the course doesn't reuse its code.

- Interface: an overview of the whole chapter.
- Clocking Blocks Part 1 and Clocking Blocks Part 2: input and output skews, `#1step`, and the race that a clocking block removes. Part 2 has a log whose sample times match its skews.
- Modport: modports with a clocking block in them, as Lab 01's `alu_if` uses.
- Virtual Interface: how a class gets its handle, and driving through `vif.cb`.
- Scheduling Semantics: the regions of a time step, which decide what "the same moment" means at an edge. Correction: it lists seven regions and leaves out the Re-Inactive and Re-NBA regions, which clocking-block drives depend on.

Next: Threads, mailboxes and events.

Answers

A1. Check yourself

It was 0, which is what the clocking block sampled just before the edge. The pin already showed the value that the flip-flop took at the edge, and that value belongs to the next cycle. A monitor that used it would pair this cycle's `valid` with the next cycle's `ready`, and would see handshakes that didn't happen.

A2. Check yourself

The pin changes at 76 ns, one output skew after the edge. The cycle from 75 ns to 85 ns then has `in_valid` and `in_ready` both 1, so the beat moves at the 85 ns edge, which is when `edge_test` printed `handshake on beat 00`.

A3. Check yourself

Every signal in `stream_mon_if` is an input, so the monitor can't drive the stream it watches. The top module also makes a `stream_mon_if` for the input stream and another for output 0, each wired to that stream's four signals, so one `monitor` class can watch either stream.

A4. Check yourself

A typedef of the virtual interface type. On Verilator 5.050 a class that reaches the interface through `typedef virtual ... t;` reads 0 from every clocking-block input, with no warning. Write the type out in full, as `virtual stream_mon_if`.

8. Threads, mailboxes and events

This page answers: *a testbench has to drive, watch and check all at once. How do SystemVerilog classes run side by side, and how do they hand packets to each other?*

Lab 02′s environment runs seven objects at the same time: a generator, a driver, two monitors, two sinks and a scoreboard. Each object's `run()` task is a separate **thread**, a piece of code that runs alongside the others and can wait for a clock edge, a delay or a packet without holding them up. The objects pass packets to each other through three **mailboxes**. The scoreboard you'll write in Lab 02, Part 3 runs two threads of its own, one for each mailbox it reads. This page covers `fork` and its three kinds of `join`, what goes wrong when you fork threads in a loop, mailboxes, events, and how to stop threads you no longer want.

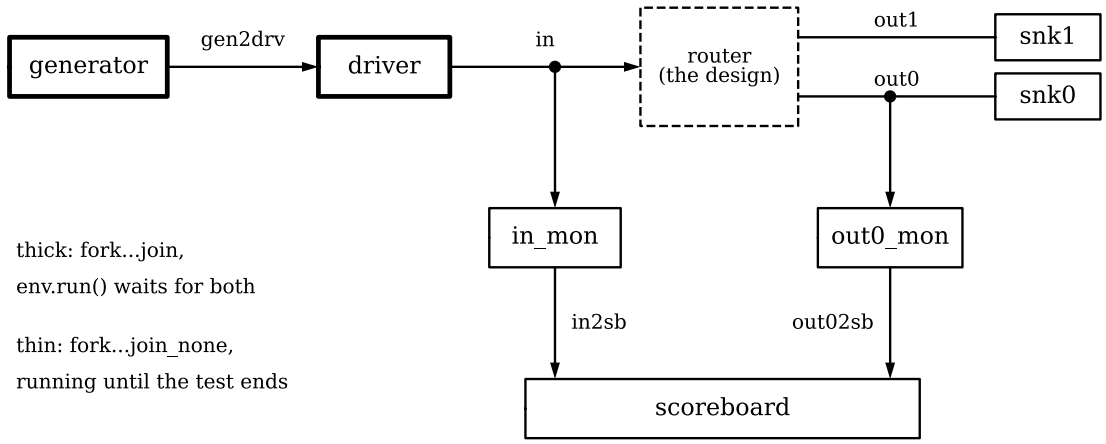


Figure 8.1: Lab 02′s environment as threads and mailboxes

From Lab 02′s `env.svh`. Each solid box is an object whose `run()` is one thread, and the dashed box is the router. The arrows labelled `gen2drv`, `in2sb` and `out02sb` are the three mailboxes, and `in`, `out0` and `out1` are the router's streams. The generator and the driver, in thick boxes, are started with `fork ... join`. The sinks, the monitors and the scoreboard are started with `fork ... join_none` and run until the test ends.

We ran every example on this page in the course image. The source is in `guide-examples/threads-mailboxes-and-events/`, and `guide-examples/README.md` gives the command for each output, in its section for this page.

fork and its three joins

`fork` starts each statement between it and its `join` as a thread of its own. The thread that ran the `fork` is called the parent, and the three kinds of `join` differ only in how long the parent waits before it goes on. The example's `fork_test` forks the same three threads three times, once with each kind of `join`. Threads A, B and C finish 10, 20 and 30 ns after they start:

```
fork
  #10 $display("%0d ns:   A finished", $time);
  #20 $display("%0d ns:   B finished", $time);
  #30 $display("%0d ns:   C finished", $time);
join            // then join_any, then join_none
```

```
0 ns: fork...join
10 ns:   A finished
```

```
20 ns:   B finished
30 ns:   C finished
30 ns: the parent goes on after join
30 ns: fork...join_any
40 ns:   A finished
40 ns: the parent goes on after join_any
50 ns:   B finished
60 ns:   C finished
60 ns: wait fork returns
60 ns: fork...join_none
60 ns: the parent goes on after join_none
70 ns:   A finished
80 ns:   B finished
90 ns:   C finished
90 ns: wait fork returns
```

- `join` waits for every thread, so the parent went on at 30 ns, when the slowest one finished.
- `join_any` waits for the first thread to finish, so the parent went on at 40 ns, when A finished. It doesn't stop the other two, and B and C kept running until 50 and 60 ns.
- `join_none` doesn't wait at all, so the parent went on at 60 ns, the moment it forked them.
- `wait fork` waits until every thread the parent has forked so far has finished. The example calls it after `join_any` and after `join_none`, so that all three threads have finished before it moves on.

Lab 02's `env::run()` uses two of these. It starts the sinks, the monitors and the scoreboard with `fork ... join_none`, because each of them runs a `forever` loop that never finishes. Then it starts the generator and the driver with `fork ... join`, because the rest of the test has to wait until the driver has sent every packet. Once they've finished, it waits 50 cycles for the outputs to drain and calls the scoreboard's `report()`. The `forever` threads are still running at that point, and they run on until the test ends.

Threads forked in a loop

A forked thread shares its variables with the code around it. That's what you want when two threads share a counter, but not when you fork a thread in each pass of a loop and each thread is meant to see the value from its own pass. `capture_test` forks three threads that each print the loop variable 1 ns later, and then three that each print a copy of it made inside the loop:

```
for (int i = 0; i < 3; i++)
  fork
    #1 $display("%0d ns:   i = %0d", $time, i);
  join_none
#5;
for (int i = 0; i < 3; i++) begin
  int j = i;
  fork
    #1 $display("%0d ns:   j = %0d", $time, j);
  join_none
end
```

```
0 ns: three threads print the loop variable i
1 ns:   i = 3
1 ns:   i = 3
1 ns:   i = 3
5 ns: three threads print j, a copy made in each pass
6 ns:   j = 0
6 ns:   j = 1
6 ns:   j = 2
```

The three threads in the first loop share one `i`, and by the time they print it, the loop has finished and left it at 3. In the second loop, `j` is declared inside the loop body. This code is in a method of a class, and as

Classes for testbenches explained, a method's local variables are automatic. Each pass through the loop body therefore makes a new `j`, and each thread keeps the `j` from its own pass.

The same loop goes wrong in a module's `initial` block, where variables are static unless you declare them `automatic`. There, `int j = i;` declares a single variable, and its initializer runs once, before the simulation starts, so all three threads print the same `j`. The example's top module runs the loop in its `initial` block twice, first with `int j = i;` and then with `automatic int j = i;`:

```
0 ns: in a module's initial block, int j = i
1 ns:   j = 0
1 ns:   j = 0
1 ns:   j = 0
5 ns: in a module's initial block, automatic int j = i
6 ns:   j = 0
6 ns:   j = 1
6 ns:   j = 2
```

Verilator warns about this (`%Warning-IMPLICITSTATIC`, "The initializer value will only be set once"), but `IMPLICITSTATIC` is one of the five warnings the labs' build turns off, so you'll only see it from `make lint`. Write `automatic` whenever you declare a variable with an initializer inside an `initial` or `always` block, or inside a task or function that isn't in a class.

Mailboxes

A mailbox carries values from threads that `put()` them to threads that `get()` them, in the order they were put. `get()` waits while the mailbox is empty, and if you gave the mailbox a size when you made it, `put()` waits while it's full. `mailbox_test` has a generator that makes four items as fast as it can and a driver that takes 10 ns over each, joined by a `mailbox #(int)`. We ran it once with no limit on the mailbox and once with room for one item:

```
mailbox bound 0 (unbounded)
0 ns: gen put 0
0 ns: gen put 1
0 ns: gen put 2
0 ns: gen put 3
0 ns: gen has made all four
0 ns:   drv got 0
10 ns:   drv got 1
20 ns:   drv got 2
30 ns:   drv got 3
```

```
mailbox bound 1
0 ns: gen put 0
0 ns:   drv got 0
0 ns: gen put 1
10 ns:   drv got 1
10 ns: gen put 2
20 ns:   drv got 2
20 ns: gen put 3
20 ns: gen has made all four
30 ns:   drv got 3
```

With the unbounded mailbox, the generator made all four items at 0 ns, and the driver worked through them, one every 10 ns. With room for one, each `put()` waited until the driver had taken the item before it, so the generator kept pace with the driver and finished at 20 ns. The driver got the same items at the same times both ways.

Lab 02's mailboxes are all unbounded (`gen2drv = new();`), so its generator makes every packet at the start and the driver works through them. The monitors' mailboxes are the other way round, with the reader waiting for the writer. A monitor puts a packet into its mailbox each time one finishes on its stream, and the scoreboard's `get()` waits until the next one arrives. The scoreboard reads both of these mailboxes, which it calls `from_in` and `from_out0`, and the two fill at their own pace. A thread waiting in `from_out0.get()` can't take a packet from `from_in` at the same time, so the scoreboard runs two threads.

One predicts each input packet as it arrives, and the other checks each packet from output 0 against the predictions made so far.

The `#(int)` in `mailbox #(int)` is the type of value the mailbox carries. Lab 02's mailboxes are `mailbox #(packet)`, so they carry handles to packets, and a `get()` hands the receiving thread the same object that the sender made. Classes for testbenches showed the same thing with a queue.

Events

An `event` lets one thread signal others without passing a value. `->done` triggers the event `done`, and every thread waiting for it goes on. There are two ways to wait for an event, and they differ in what happens when the trigger comes before the wait. `event_test` triggers `done` at 10 ns and then, in the same time step, starts one thread that waits with `@done` and another that waits with `wait (done.triggered)`:

```
10 ns: ->done
10 ns:   wait (done.triggered) returns
30 ns: end of the demo
```

`@done` waits for the next time `done` is triggered, so a thread that reaches it after the trigger, even in the same time step, misses that trigger and waits for another. `event_test` never triggers `done` again, so its `@done` thread never prints. `done.triggered` stays true for the rest of the time step in which `done` was triggered, so `wait (done.triggered)` doesn't miss it. Use `wait (e.triggered)` whenever the wait might start in the same time step as the trigger.

Stopping threads

`disable fork` ends every still-running thread that the current thread has forked, along with any threads those have forked. It's often used for a wait with a time limit. You fork the thing you're waiting for and a timer, wait with `join_any` for whichever finishes first, and then `disable fork` ends the one that's left. It also ends any threads that the same thread forked earlier for another job, such as a monitor started with `join_none`.

`disable_test` shows this with a monitor, a `forever` loop that counts a tick every 10 ns. It starts the monitor with `join_none` and then waits with a time limit, using this task:

```
task wait_with_timeout();
  fork
    #25  $display("%0d ns:   the thing we waited for happened", $time);
    #100 $display("%0d ns:   timeout", $time);
  join_any
  disable fork;
endtask
```

The test does all of this twice, with a new monitor each time. The first time, the thread that started the monitor calls the task directly. The second time, it calls the task from a thread forked just for it (`fork wait_with_timeout(); join`). Each time, 50 ns after the wait, it prints how many ticks the monitor has counted:

```
25 ns:   the thing we waited for happened
75 ns: with a bare disable fork: the monitor has counted 2 ticks
100 ns:   the thing we waited for happened
150 ns: with the wait in its own fork: the monitor has counted 7 ticks
```

The first time, `disable fork` ended the monitor along with the timer, because the same thread had forked both, and the monitor stopped at 2 ticks. The second time, the task ran in a thread of its own, which had forked only the two threads in the wait, so `disable fork` ended the timer and the monitor kept counting. In Lab 02's `env::run()`, the sinks, the monitors and the scoreboard are forked from the same thread that forks the driver, so a bare `disable fork` added there would stop them too.

Check yourself

1. `env::run()` starts the sinks, the monitors and the scoreboard with `fork ... join_none`. What would happen if it used `fork ... join` instead?

Answer: A1, in Answers at the end.

2. You copy the second loop of `capture_test`, the one with `int j = i;` in its body, into a module's `initial` block. What does it print, and what one-word change fixes it?

   Answer: A2, in Answers at the end.

3. The generator puts four items into `mailbox #(int) mb = new(1)` and the driver takes 10 ns over each. When does the generator finish, and when would it finish with `new()`?

   Answer: A3, in Answers at the end.

4. To give `env::run()` a time limit, you add a timer thread to the fork that starts the generator and the driver, change its `join` to `join_any`, and put a bare `disable fork` after it. The test now reports `[SB] 0 packets in`. What happened?

   Answer: A4, in Answers at the end.

Further reading

ChipVerify covers threads, mailboxes, events and semaphores in more detail than this page. Most of the logs on its SystemVerilog pages come from Cadence simulators rather than Verilator, and the course doesn't reuse its code.

- Processes: a short overview of the chapter.
- fork join, fork join_any and fork join_none. The join_none page covers the loop-variable problem and its `automatic` fix.
- wait fork.
- disable fork join. It warns that `disable fork` reaches further than you might expect, but it doesn't show the fix this page uses, which is to put the wait in a `fork ... join` of its own.
- Mailbox: bounded and unbounded mailboxes, `try_put`, `try_get` and `peek`.
- Event: `@e` against `e.triggered`.
- Semaphore: the third way threads coordinate, for sharing something only one of them may use at a time.

Next: Lab 02: A class testbench without UVM.

Answers

A1. Check yourself

The sinks' `run()` is a `forever` loop, and once you've written the monitor and the scoreboard, their `run()` tasks never finish either, so a `join` would wait for ever. `env::run()` would never get to the generator and the driver, no packet would be sent, and the run would go on until `make` stopped it after `RUN_TIMEOUT`, 300 s by default.

A2. Check yourself

It prints `j = 0` three times. In an `initial` block `j` is static, so there is only one `j`. Its initializer runs once, before the simulation starts, and the loop never sets it again. Declaring it `automatic int j = i;` makes a new `j` in each pass, and the threads print 0, 1 and 2.

A3. Check yourself

With room for one item, the generator finishes at 20 ns, because each `put()` waits until the driver has taken the item before it. With `new()` the mailbox is unbounded, and the generator finishes at 0 ns. `mailbox_test` printed both.

A4. Check yourself

The generator puts every packet into its unbounded mailbox at the start and returns, so `join_any` returned before the driver had sent anything. Then `disable fork` ended every thread that `env::run()` had forked. That included the driver and the timer, and also the sinks, the monitors and the scoreboard, just as it ended the monitor in `disable_test`. Start the generator outside the timed fork, and put the driver and the timer in a `fork ... join_any` inside a `fork ... join` of its own, so that `disable fork` only reaches those two threads.

9. Lab 02: A class testbench without UVM

You'll finish a class testbench for the router. Every class in it is plain SystemVerilog except the test, which is a UVM test that the top module starts with `run_test()`. You write the monitor, which turns the handshakes on a valid/ready stream into packets, and the scoreboard, which predicts the packets that should come out of output 0 and checks the ones that do. Then you'll use the testbench to catch two planted bugs, and rerun it with different plusargs to find out what each catch depends on.

Package: `labs/lab-02-a-class-testbench-without-uvm/`

Before you start

- Finish Lab 01. This lab uses the same `make` targets, the same terminal and the same course folder.
- Read Classes for testbenches, Interfaces, clocking blocks and virtual interfaces and Threads, mailboxes and events.
- Read The router in one page, and keep the specification open. You'll use its Packets, Routing and The outputs sections, and the stream rules under Valid/ready rules.

Time: about 2 h. The three builds in it took us 90 s, 89 s and 86 s.

Part 1: The starter

What does the shipped testbench do, and what does it check?

1. Open a terminal on the workbench desktop, as in Lab 01, and go to the lab folder:

```
. /foss/designs/common/.designinit   # in every new terminal
mod dd105-a
cd labs/lab-02-a-class-testbench-without-uvm
```

2. Start the build now, because it's the slowest step:

```
make build
```

   The banner estimates 1.5 to 2.5 minutes, and with an empty ccache it took us 90 s. It ends with:

```
- V e r i l a t i o n   R e p o r t: Verilator 5.050 2026-07-01 rev v5.050
- Verilator: Built from 48.403 MB sources in 393 modules, into 23.194 MB in 2538 C+
+ files needing 80.303 MB
- Verilator: Walltime 89.763 s (elab=0.694, cvt=2.551, bld=85.825); cpu 4.237 s on
2 threads; allocated 312.547 MB
=============================================================================
 built /w/labs/lab-02-a-class-testbench-without-uvm/build/sim in 90 s (48M; the 898M
of generated C++ was deleted, SLIM=0 keeps it)
 ccache:  Hits: 0 / 36 ( 0.00%)  Misses: 36 / 36 (100.0%)
=============================================================================
 /w/labs/lab-02-a-class-testbench-without-uvm/build/sim is up to date.
```

   We ran it with the course folder mounted at `/w`, so your paths start with `/foss/designs/modules/dd105-a` instead. The ccache counts add up over every build in your clone, Lab 01's included, so yours will be higher.

3. While it builds, read the files in `tb/`. Each class has one job. The generator hands its packets to the driver through a mailbox, and each monitor hands the packets it sees to the scoreboard through a mailbox of its own.

File	Class	Its job
`packet.svh`	`packet`	one packet: its beats, header first
	`router_cfg`	the test's knobs, read from plusargs
`generator.svh`	`generator`	makes the packets and puts them in a mailbox
`driver.svh`	`driver`	offers each beat on the input until the router takes it
	`sink`	holds one output's ready (`out0_ready` or `out1_ready`) at 1, and counts its beats
	`apb_master`	writes the routing table and `CTRL.EN`
`monitor.svh`	`monitor`	yours: turns a stream's handshakes into packets
`scoreboard.svh`	`scoreboard`	yours: predicts output 0 and checks it
`env.svh`	`env`	makes the classes above from `generator` to `scoreboard`, connects them, and runs the test
`router_test_pkg.sv`	`router_class_test`	the one UVM class
`router_tb_top.sv`	(module)	the clock, the reset, the interfaces, the router and `run_test()`

Read `env::run()` closely. It waits for reset and writes the configuration, then starts the sinks, the monitors and the scoreboard in a `fork ... join_none`. It sends the packets, waits 50 cycles for the outputs to drain, and asks the scoreboard for its report. Nothing in it waits for the monitors or the scoreboard, so the test ends after those 50 cycles however much of them you've written.

`router_class_test` makes the `env` and calls `run()`. It's a UVM test only so that this lab runs the way Lab 01 did, with `make` picking the test by name with `+UVM_TESTNAME` and the graders reading UVM's report summary at the end of every log. If the driver or the scoreboard counted any errors, the test reports a single `UVM_ERROR` saying how many, and the base test turns that into a failing exit status.

4. When the build has finished, run the test:

```
make run
```

After UVM's release notes you should see:

```
UVM_INFO @ 0: reporter [RNTST] Running test router_class_test...
[ENV] 100 packets, DST from 0123, routes 0110, 1 to 8 beats
[ENV] sent 100 packets (448 beats); output 0 took 252 beats and output 1 took 196
[SB] 0 packets in, 0 predicted for output 0, 0 checked, 0 errors
DD105: TEST PASSED
```

followed by UVM's report summary and, last:

```
--------------------------------------------------------------------------
 ok: 0 UVM_ERROR, 0 UVM_FATAL   0.07 s   log: build/runs/router_class_test-s1/run.log
--------------------------------------------------------------------------
```

The first `[ENV]` line gives the test's knobs, which are plusargs (the package README lists them). `routes 0110` is the routing table, one character per entry from `ROUTE0` to `ROUTE3`, so packets for `DST` 0 and 3 go to output 0, and packets for `DST` 1 and 2 go to output 1.

5. `make check-all` runs the test on the golden router and on the two bugs this lab hunts. Write down what you expect it to report, then run it:

```
make check-all
```

You should see:

```
     ok  the golden design                      0 UVM_ERROR, 0 UVM_FATAL       0.1s
--------------------------------------------------------------------------
  MISSED  tier 2  B05_dst_field_off_by_one       0 UVM_ERROR, 0 UVM_FATAL       0.1s
  MISSED  tier 3  B09_write_pointer_power_of_two  0 UVM_ERROR, 0 UVM_FATAL      0.1s
--------------------------------------------------------------------------
 0 of 2 caught.
 Still hiding from this test:
   B05_dst_field_off_by_one
      hides behind: sending to DST 0 only, or setting every route the same way
   B09_write_pointer_power_of_two
      hides behind: short tests: fewer than 16 beats through one output
```

On the golden router, output 0 delivered 252 beats and the test passed. It passed on both broken routers as well. Why?

Answer: A1, in Answers at the end.

Check: `0 of 2 caught`, and `[SB] 0 packets in` in `make run`.

Part 2: The monitor

What goes into the router, and what comes out of output 0, packet by packet?

1. Open `tb/monitor.svh` and `tb/stream_if.sv`. The monitor holds a `virtual stream_mon_if`, an interface whose signals are all inputs, so a monitor can't drive the stream it watches. The top module makes two of these interfaces, `in_mon` wired to the input's four signals and `out0_mon` wired to output 0's. Both have the same type, so one `monitor` class watches either. The clocking block samples with `#1step`, so after `@(vif.cb)` returns at a rising edge, `vif.cb.valid` is the value that `valid` had just before that edge, which the specification calls its value in the cycle that the edge ends. The type is written out in full, `virtual stream_mon_if`, and never through a `typedef`, because on Verilator 5.050, when a class in a package reaches an interface through a typedef of its virtual interface type, the class reads 0 from every clocking-block input, with no warning.
2. This is the golden router taking one packet on its input and delivering it on output 0:

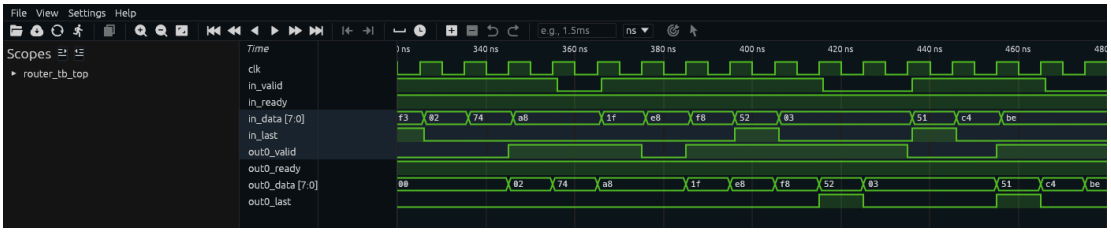


Figure 9.1: The packet 02 74 a8 1f e8 f8 52 going in and coming out of output 0, in Surfer

Seed 1, 320 to 480 ns. The input takes `02` at the rising edge at 335 ns, and `02` is on `out0_data` from 345 ns. `out0_ready` is 1 throughout.

A beat moves at a rising edge where `valid` and `ready` were both 1. Find `a8` on `in_data`. At how many rising edges is it on `in_data`, and how many beats is it?

Answer: A2, in Answers at the end.

3. Write `monitor::run()`, following the `TODO`. Each time round a `forever` loop, wait with `@(vif.cb)`; drop any half-assembled packet while `vif.cb.rst_n` is 0; and in a handshake cycle add the beat to the packet `p`, starting a new one when `p` is null. When the beat has `last` = 1, the packet is finished, so put it in `to_sb` and set `p` back to null for the next one. Keep the `$display` format that the `TODO` gives, so that your output can be compared with the one below.
4. Check the syntax, which takes a few seconds, then build:

```
make lint
make build
```

`make lint` ends with `0 warnings, by kind:` when your code is clean, and it prints any syntax error with its line number. With the reference monitor the build took 89 s, nearly as long as the first one, because filling in the empty task changed most of the C++ that Verilator generates, and the ccache could reuse only 10 of the 36 compiled files.

5. `+SHOW=5` makes each monitor print the first 5 packets it sees, with the time of the edge at which each packet's last beat moved. The clock period is 10 ns. Using the routing table `0110` and rule O8 of the specification, write down which input packets you expect to see again on output 0, and how many ns after its `[in]` line each `[out0]` line should come. Then run it:

```
make run PLUSARGS="+SHOW=5"
```

You should see, with the reference monitor:

```
[ENV] 100 packets, DST from 0123, routes 0110, 1 to 8 beats
[in] 235 ns: dst 2, 5 beats: 80 b4 e5 26 c1
[in] 325 ns: dst 2, 6 beats: 81 f3 06 8e bc f3
[in] 405 ns: dst 0, 7 beats: 02 74 a8 1f e8 f8 52
[out0] 425 ns: dst 0, 7 beats: 02 74 a8 1f e8 f8 52
[in] 445 ns: dst 0, 2 beats: 03 51
[out0] 465 ns: dst 0, 2 beats: 03 51
[in] 575 ns: dst 3, 8 beats: c4 be 9e 50 be fe e2 92
[out0] 595 ns: dst 3, 8 beats: c4 be 9e 50 be fe e2 92
[out0] 685 ns: dst 0, 2 beats: 08 df
[out0] 835 ns: dst 0, 1 beat: 0b
[ENV] sent 100 packets (448 beats); output 0 took 252 beats and output 1 took 196
```

The header's low six bits are the packet's sequence number, so `80` is packet 0, `81` packet 1 and `02` packet 2. Why are packets 0 and 1 missing from output 0, and why does each `[out0]` line come 20 ns after the `[in]` line of the same packet?

Answer: A3, in Answers at the end.

Check: your `[in]` and `[out0]` lines match the ones above. The seed is the same, so the packets are too.

Part 3: The scoreboard

Which packets should come out of output 0, and did they?

1. Read the comment at the top of `tb/scoreboard.svh`. The rule you're checking is O5, which says that an output delivers the beats of the packets routed to it in the order the router accepted them, unchanged. In this lab the sinks never stall, so no packet is dropped for a full output, and a packet goes to output 0 exactly when `ROUTE[DST]` is enabled with `PORT` 0.
2. Write the four `TODO`s:

- `run()`: a `fork ... join` of two `forever` loops, one taking packets from `from_in` and calling `predict()`, the other taking packets from `from_out0` and calling `check()`.
- `predict()`: push the packet onto `expected` if it will come out of output 0.
- `check()`: compare what came out with the oldest packet in `expected`, using `packet::same()`. A packet that comes out while `expected` is empty is an error too.
- `report()`: report an error for each packet still in `expected`, because it never came out, and one more if the test checked no packets at all.

Report each error with `$error`, which takes a format string like `$display`, and count it:

```
n_errors++;
$error("out0 delivered %s at %0d ns; expected %s", got.sprint(), got.t_last,
want.sprint());
```

Lab 01's `` `uvm_error `` won't work here, because this package doesn't import UVM. We tried it, and `make lint` stopped with two `Can't find definition of task/function` errors, one for `uvm_get_report_object` and one for `uvm_report_error`.

3. Run `make lint` and `make build`. With the reference scoreboard the build took 86 s. A small edit after that rebuilds faster. When we changed the text of one `$error`, the rebuild took 21 s.
4. Run the test on the golden router:

   ```
   make run
   ```

   You should see `[SB] 100 packets in, 51 predicted for output 0, 51 checked, 0 errors`, `DD105: TEST PASSED` and `ok`. If an error fires here, the golden router is right, so the mistake is in your scoreboard or your monitor.
5. Run it on five seeds:

   ```
   make seeds NSEEDS=5
   ```

   You should see `5 ok (5 runs)`. The coverage report under it says there are no covergroup bins, because this testbench has no covergroup yet. You'll add one in Lab 03.
6. Write down what you now expect from `make check-all`, then run it. With the reference scoreboard you should see:

   ```
        ok  the golden design                      0 UVM_ERROR, 0 UVM_FATAL       0.1s
   ------------------------------------------------------------------------------
     caught  tier 2  B05_dst_field_off_by_one       1 UVM_ERROR, 50 assertions     0.1s
     caught  tier 3  B09_write_pointer_power_of_two  1 UVM_ERROR, 46 assertions    0.1s
   ------------------------------------------------------------------------------
    2 of 2 caught.
   ```

   The grader counts each `$error` as an assertion, because Verilator prints a `$error` and a failed assertion the same way, so the 50 and 46 are the reference scoreboard's errors. The one `UVM_ERROR` is the test's summary of them. Your counts depend on how your scoreboard reports, so they may differ.
7. Look at the first error on B05:

   ```
   make run BUG=B05
   ```

   With the reference scoreboard, the first two lines after `[ENV]` are:

   ```
   [255000]      %Error:      scoreboard.svh:65:      Assertion      failed      in
   router_tb_pkg.scoreboard.check: out0 delivered dst 2, 5 beats: 80 b4 e5 26 c1 at 255
   ns, and no packet was expected
   -Info:   labs/lab-02-a-class-testbench-without-uvm/tb/scoreboard.svh:65:   Verilog
   $stop, ignored due to +verilator+error+limit
   ```

The `-Info` line appears only once, after the first error. By default Verilator stops a run at its first `$error`. When we ran `build/sim` by hand without the plusarg, it stopped at this error with `Verilog $stop`, before the scoreboard's report and UVM's summary. That's why the lab's Makefile adds `+verilator+error+limit+100` to every run, so that your runs go on past each error until the 100th.

The header is `80`, which is `1000_0000`. Which routing table entry should the router have used, which one did it use, and which bits of the header did it read?

Answer: A4, in Answers at the end.

8. Now the first error on B09:

```
make run BUG=B09
```

```
[895000]      %Error:      scoreboard.svh:72:      Assertion      failed      in
router_tb_pkg.scoreboard.check: out0 delivered dst 3, 13 beats: c4 be 9e 50 be fe
02 74 a8 1f e8 f8 52 at 895 ns; expected dst 3, 8 beats: c4 be 9e 50 be fe e2 92,
which went in at 455 ns
```

The first six beats are right. Compare the other seven with your Part 2 printout. Where have you seen them before?

Answer: A5, in Answers at the end.

Check: `2 of 2 caught` from `make check-all`.

Part 4: What each catch depends on

Which property of the stimulus does each catch depend on? The test's knobs are plusargs, so each of these runs reuses the simulator you built.

1. `+ROUTES=0000` sets all four routes to output 0. Predict whether your test still catches B05, then run:

```
make check BUG=B05 PLUSARGS="+ROUTES=0000"
```

You should see `ok` on the golden design, `MISSED` on B05 and `VERDICT: FAIL`. Why does the test miss B05 now?

Answer: A6, in Answers at the end.

2. `+DSTS=0` sends every packet to `DST` 0. Predict both results, then run:

```
make check BUG=B05 PLUSARGS="+DSTS=0 +N=32"
make check BUG=B05 PLUSARGS="+DSTS=0"
```

The first prints `MISSED`. The second, with the default 100 packets, prints

```
  caught  B05_dst_field_off_by_one              1 UVM_ERROR, 68 assertions     0.1s
```

and its first error is:

```
[4725000]      %Error:      scoreboard.svh:72:      Assertion      failed      in
router_tb_pkg.scoreboard.check: out0 delivered dst 0, 1 beat: 00 at 4725 ns; expected
dst 0, 8 beats: 20 69 85 07 03 c1 02 4b, which went in at 2435 ns
```

Why does `DST` 0 alone catch B05 with 100 packets and not with 32?

Answer: A7, in Answers at the end.

3. In Part 1, `make check-all` said that B09 hides behind short tests, with fewer than 16 beats through one output. With `+DSTS=0 +LEN=1`, every packet is one beat long and goes to output 0. Work out the smallest `+N` that should catch B09 and write it down. Then try it and the number below it:

```
make check BUG=B09 PLUSARGS="+DSTS=0 +LEN=1 +N=15"
make check BUG=B09 PLUSARGS="+DSTS=0 +LEN=1 +N=16"
```

You should see, from the two commands:

```
  MISSED  B09_write_pointer_power_of_two         0 UVM_ERROR, 0 UVM_FATAL       0.0s
```

```
  caught  B09_write_pointer_power_of_two         1 UVM_ERROR, 1 assertion       0.0s
        UVM_ERROR [TB] the testbench reported 1 error
        scoreboard.svh:72: Assertion failed in router_tb_pkg.scoreboard.check: out0
delivered dst 0, 1 beat:
```

`make run` with the same `BUG` and `PLUSARGS` prints the whole error: `out0 delivered dst 0, 1 beat: 00 at 405 ns; expected dst 0, 1 beat: 0f, which went in at 385 ns`.

4. Record the 16-beat run and open it:

```
make waves BUG=B09 PLUSARGS="+DSTS=0 +LEN=1 +N=16"
surfer build/runs/router_class_test-s1-B09/waves.fst &
```

Add `clk` from `router_tb_top`, `in_valid`, `in_ready`, `in_data`, `out0_valid` and `out0_data` from `dut`, and `wr_ptr_q` and `rd_ptr_q` from `dut.u_buf0`, output 0's buffer:

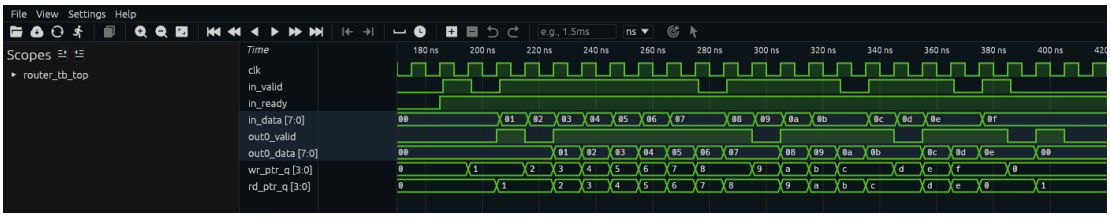


Figure 9.2: Sixteen one-beat packets through output 0 with B09 switched on, in Surfer

B09. Beats `00` to `0e` come out as they went in. `wr_ptr_q` goes on from `e` to `f` while `rd_ptr_q` wraps from `e` to `0`, and the 16th beat comes out as `00` instead of `0f`.

The buffer holds 16 beats, one in the output register that drives the pins and 15 in a small RAM behind it (`labs/common/rtl/router_fifo.sv`), so both pointers should wrap after entry `e`. Why does the 16th beat come out as `00`?

Answer: A8, in Answers at the end.

Check: `MISSED` with `+ROUTES=0000` and with `+N=15`, and `caught` with `+N=16`.

Finishing

You're done when `make check-all` prints `2 of 2 caught.` with the monitor and scoreboard you wrote in `tb/monitor.svh` and `tb/scoreboard.svh`.

Questions:

1. In Lab 01 the UVM testbench was split into the same kinds of pieces. Which jobs did UVM do there that the shipped classes, or you, did by hand here?

   Answer: A9, in Answers at the end.

2. On B09 the reference scoreboard's second error is `out0 delivered dst 0, 2 beats: 03 51 at 935 ns; expected dst 0, 2 beats: 08 df, which went in at 655 ns`. Is that a second bug, and which error should you read first when a scoreboard reports many?

   Answer: A10, in Answers at the end.

Submit: nothing for this lab.

Going further (optional)

Watch output 1 as well. The top module needs a third `stream_mon_if`, wired to `out1` and set in `config_db`, and `router_class_test` has to get it and pass it to the `env`. The `env` needs a third monitor, and a fourth mailbox to carry that monitor's packets to the scoreboard. The scoreboard needs a second queue of expected packets, which `predict()` fills for routes with `PORT` 1. Rebuild and run `make seeds NSEEDS=5` to check that the golden router still passes.

Next on the Course path: the reading on constrained random and functional coverage, then Lab 03: Constraints and coverage.

Answers

A1. Part 1: The starter

The `[SB]` line says the scoreboard saw 0 packets and checked 0, because the starter's monitor and scoreboard are empty, so nothing compared those 252 beats with anything. A pass means only that no check fired, and with no checks the test passes on the golden router and on both bugs alike.

A2. Part 2: The monitor

It's on `in_data` at two edges, 355 ns and 365 ns, and it's one beat. The driver put it there at 346 ns, the router took it at the edge at 355 ns, and the driver then left one idle cycle with `in_valid` at 0, so at 365 ns the pins still show `a8` but nothing moves. A monitor that looked only at `in_data` would record it twice. The output shows the same gap two cycles later.

A3. Part 2: The monitor

Their headers, `80` and `81`, have `DST` 2 in bits 7:6, and `ROUTE2` sends packets to output 1, which this testbench doesn't watch. The 20 ns is two clock cycles. Rule O8 puts each beat on the output from the second cycle after its input handshake, as example E2 in the specification shows, unless the beat before it is still waiting there, as in E3. Here the sink takes every beat in the first cycle it's offered, so nothing waits, and each packet's last beat leaves output 0 two cycles after it went in.

A4. Part 3: The scoreboard

`DST` is bits 7:6, `10`, so it should have used `ROUTE2`, which sends to output 1. The packet came out of output 0, which only `ROUTE0` and `ROUTE3` send to. Bits 6:5 of `80` are `00`, so the router read the field one bit too low and used `ROUTE0`. That's B05, `dst_field_off_by_one`.

A5. Part 3: The scoreboard

They're packet 2, `02 74 a8 1f e8 f8 52`, the first packet that output 0 delivered in this run. Its beats came out of the buffer a second time, and its `last` flag ended the packet, which is why this one has 13 beats. Before it went wrong, output 0 had delivered all 7 beats of packet 2, both beats of packet 3 and the first 6 of packet 4's 8, which is 15 beats, so the 16th beat (packet 4's seventh, `e2`) is the first bad one. Part 4 shows why.

A6. Part 4: What each catch depends on

B05 changes which routing table entry the router reads. With all four entries set the same way, every entry sends the packet to output 0, so whichever one the router reads, the packet still comes out of output 0 in the order it went in, and nothing at the pins shows which entry was used. A test checks the `DST` field only when each of its values, and each of its bits, has an effect of its own on where a packet goes.

A7. Part 4: What each catch depends on

The header's other six bits are the sequence number, which counts 0 to 63 and wraps. For packets 0 to 31, bits 7:5 are all 0, so bits 6:5 read `00`, the same as `DST`, and the bug has no effect. Packet 32's header is `20`, `0010_0000`, whose bits 6:5 are `01`, so the router uses `ROUTE1` and sends packets 32 to 63 to output 1. The error shows this. Output 0 was expecting packet 32 (`20`), and the next packet it delivered was packet 64, whose sequence number has wrapped to 0, so its header is `00`. With 100 packets, the sequence number's top bit happened to give header bit 5 an effect of its own.

A8. Part 4: What each catch depends on

The write pointer wrapped at 16, as if the RAM had 16 entries, so beat `0f` was written to entry `f`, which doesn't exist, and was lost. The read pointer wrapped correctly to entry 0, which still held beat `00` from the start of the test. Up to the 15th beat, each beat is read from the entry it was written to, so no test that puts fewer than 16 beats through an output can see the bug.

A9. Finishing

UVM connected the pieces in Lab 01. Here `env::new()` makes three mailboxes and passes each one to its two ends by hand. In Lab 01, UVM's phases started each component and its objections decided when the test was over. Here `env::run()` forks every `run()` task itself and ends the test after a fixed 50-cycle drain. UVM's report server counted every `` `uvm_error `` by its ID, while here the driver and the scoreboard

each keep their own `n_errors`, and `env::n_errors()` adds them up for the test. In Lab 01 each test was its own class, picked by name with `+UVM_TESTNAME`, and every test reused the same driver, monitor and scoreboard. Here there is one test, its variations are the plusargs that `router_cfg` reads, and anything those can't express needs a change to `env::run()`. In Lab 04 you'll finish a UVM port of this testbench as it stands after Lab 03, and compare the two piece by piece.

A10. Finishing

It's the same bug. `03 51` is packet 3, the second packet that output 0 delivered, coming out of the buffer again. After the first wrong packet the buffer keeps handing back old beats, so the packets that output 0 delivers no longer line up with the ones in `expected`, and most of the later errors follow from the first. Read the first error first. The later ones count comparisons that failed after the two fell out of step, so their number says little about how many packets the router got wrong.

III 3. Constraints and coverage

10 Constrained random on Verilator . . 71
10 rand fields, constraints and randomize() 71
10 The solver 72
10 Shapes that come out differently here 72
10 Seeds 75
10 Counting what came out 75
10 Check yourself 75
10 Further reading 75
10 Answers 76

11 Functional coverage you can read . . 77
11 Coverpoints, bins and crosses 77
11 coverage.dat and make cov 78
11 The percentage 79
11 Three kinds of empty bin 80
11 Check yourself 80
11 Further reading 80
11 Answers 80

12 Lab 03: Constraints and coverage . . . 81
12 Before you start 81
12 Part 1: The testbench you start from 81
12 Part 2: A covergroup you can read 83
12 Part 3: Constraints the simulator honours 85
12 Part 4: Close the two holes 87
12 Finishing 89
12 Going further (optional) 89
12 Answers 89

10. Constrained random on Verilator

This page answers: *how do I get the random stimulus I asked for on this simulator, and how do I know I got it?*

Lab 02's generator chose each packet's DST and length itself, with calls to `$urandom_range` written into its loop. In Lab 03 the generator works on a `rand_packet` instead, a class that adds `rand` fields and constraints to Lab 02's `packet`. It calls `randomize()`, which hands the constraints to a solver and lets the solver choose the values. That moves the decisions about what the stimulus looks like out of the generator's code and into one class, where a test can change them. It also means that what comes out depends on how the simulator chooses among the values your constraints allow, and Verilator 5.050 doesn't always choose the way the SystemVerilog standard says it should.

This page explains what the standard says `randomize()` does, where Verilator 5.050 does the same and where it doesn't, and how to check your own constraints by counting what comes out. The examples are in `guide-examples/constrained-random-on-verilator/`. They're small classes, most of them built on SimProc's ALU operations, and they build with the course's `uvm.mk` like every lab. You'll run the same kind of experiment on the router's packet in Lab 03, Part 3.

rand fields, constraints and randomize()

A class marks the fields it wants chosen with `rand`, and states the rules they must follow in named `constraint` blocks. Here is an ALU vector:

```
typedef enum bit [2:0] {ADD, SUB, OR, NAND, SHL, SHR} alu_op_e;

class alu_vec;
  rand alu_op_e  op;
  rand bit [7:0] A, B;
  constraint c_op    { op inside {[ADD:SHR]}; }
  constraint c_shift { (op inside {SHL, SHR}) -> B < 8'd4; }
endclass
```

SimProc's shifts use only `B[1:0]`, so `c_shift` keeps `B` below 4 whenever the operation is a shift, and leaves it free otherwise. A call to `v.randomize()` asks for values of `op`, `A` and `B` that satisfy every constraint block that is switched on, all at once. It returns 1 if it found such values and 0 if it didn't. A call can add constraints of its own with `with { ... }`, which apply to that call only, on top of the class's own constraints. A block can be switched off with `constraint_mode(0)`. The standard also says that every combination of values that satisfies the constraints is equally likely to be chosen.

`cr_item_test` randomizes one `alu_vec` a few times:

```
[ITEM] randomize() returned 1: SHR A=cc B=02
[ITEM] randomize() returned 1: SHR A=43 B=03
[ITEM] randomize() returned 1: SHL A=2f B=00
[ITEM] randomize() returned 1: SHL A=65 B=03
[ITEM] with { op == SUB; A == B; } returned 1: SUB A=f4 B=f4
[ITEM] with { op == SUB; A == B; } returned 1: SUB A=35 B=35
[ITEM] before: SUB A=35 B=35
[ITEM] with { op == SHL; B == 5; } returned 0: SUB A=e5 B=35
[ITEM] the same with c_shift off returned 1: SHL A=10 B=05
```

The call with `{ op == SHL; B == 5; }` can't be satisfied, because `c_shift` says a shift has `B` below 4. It returned 0, and the run printed an `UNSATCONSTR` warning naming both constraints. The standard says a failed call leaves the variables as they were, but here `A` changed from `35` to `e5`. So after the failed call the object held the old `op` and `B` with a new `A`, which is a combination that no call asked for. With `c_shift` switched off, the same call succeeds.

That's why every `randomize()` in Labs 01 to 07 and in the guide examples checks what it returned, and most of them stop the run when it's 0, as in `if (!p.randomize()) `uvm_fatal("RAND", ...)`. It's also why the course's `make run` calls a run BROKEN when any call failed, even if the test itself passed. This run's verdict was:

```
 BROKEN: randomize() failed   0.02 s   log: build/runs/cr_item_test-s1/run.log
```

The solver

Verilator doesn't solve constraints itself. It writes them out in the SMT-LIB language for a separate solver program, which the environment variable `VERILATOR_SOLVER` names. The course image has `yices-smt2`. The course's `uvm.mk` sets `VERILATOR_SOLVER` to `yices-smt2 --incremental` in the environment of every run, and it also builds that command into the simulator as the default, which the simulator uses when you start it by hand without the variable. Before it builds or runs anything, `make` checks that the program that `VERILATOR_SOLVER` names is on your PATH. Asked to use `z3`, which the image doesn't have, it stops with:

```
 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?
```

As a last check, `dd105_base_test` randomizes a small probe class in its `build_phase` and stops the run if that fails. Here is the simulator run by hand, without `make`, with `VERILATOR_SOLVER` naming `z3`:

```
VlRProcess::open: execvp(z3): No such file or directory
UVM_INFO @ 0: reporter [RNTST] Running test cr_item_test...
%Warning: Subprocess command `z3 --in' failed: exit status 127
%Warning: Unable to communicate with SAT solver, please check its installation or
specify a different one in VERILATOR_SOLVER environment variable.
 ... Tried: $ z3 --in
UVM_FATAL labs/common/uvm/dd105_pkg.sv(51) @ 0: reporter [SOLVER] a constrained
randomize() failed, so this run cannot randomize anything. Verilator needs
VERILATOR_SOLVER="yices-smt2 --incremental" in the environment. Run the test with make,
which sets it, or export it before running the simulator by hand.
```

To get random values out of the solver, Verilator's runtime (`verilated_random.cpp` in the image) picks a random target value for every bit of every `rand` variable, and asks the solver for a solution that matches all of those bits. If the targets break a constraint, it gives up one bit's target and asks again, and it keeps doing that until the remaining targets fit the constraints. A target that lands on an allowed value comes out as it is. A target that lands on a forbidden value comes out as an allowed value that differs from it in a bit or two, so the allowed values next to forbidden ones come out more often than the others. (A class with an array in it goes through a different method in the same file.)

`alu_op_e` has six names in three bits, so `op inside {[ADD:SHR]}` forbids two of its eight codes, `110` and `111`. With that constraint, 1,000 draws in `cr_enum_test` gave:

```
[ENUM] op inside {[ADD:SHR]}:            ADD  127  SUB  109  OR  148  NAND  124  SHL
234  SHR  258
```

The six counts add up to 1,000, so every draw obeyed the constraint and none came out as 6 or 7. The standard would give each operation about 167, but `SHL` (`100`) and `SHR` (`101`) got about twice the share of the others. You get those two codes from `110` and `111` by giving up the middle bit's target, so the draws that aimed at the forbidden codes landed there. A constraint decides which values can come out, and on this simulator you can't rely on it for how often each one does.

Shapes that come out differently here

Lab 03's experiments (`make traps`) and the examples here found five shapes of class and constraint for which Verilator 5.050 doesn't do what the standard says. Each section below gives a way around it.

A rand enum that no constraint names

The standard makes an unconstrained `rand` enum uniform over its names. On this simulator that holds for some enums and not others:

```
[ENUM] 3 names in 2 bits, no constraint:      317  347  336
[ENUM] 4 names in 2 bits, no constraint:      813   66   63   58
[ENUM] 6 names in 3 bits, no constraint:      160  167  179  163  146  185
[ENUM] 8 names in 3 bits, no constraint:      890   14   12   15   18   10   22   19
[ENUM] 4 names, v inside {[N4_A:N4_D]}:       245  237  247  271
```

Four names in two bits, and eight names in three, fill every code of their bits. Those two enums came out as their first name 813 and 890 times in 1,000, with no warning. Line 1 of Lab 03′s experiments gives 809 for its four-name enum. The three-name and six-name enums came out even. So the course names every `rand` enum in a constraint, except the `kind` of Lab 03′s starter packet, which you weight yourself in Part 3. For an enum whose names fill its bits, `inside` with every name is enough, as the last line above shows. For an enum whose names don't fill its bits, `inside` gives an uneven spread, as the previous section showed. A `dist` with one range and equal weights gave an even one, and that's how Lab 01′s `alu_item` writes its `op`:

```
[ENUM] op dist {[ADD:SHR] := 1}:          ADD  161  SUB  166  OR  163  NAND  173  SHL
176  SHR  161
```

No lab writes a class that takes the enum's type as a parameter, but the same test tried one (`class free_param #(type T = e3_t); rand T v;`). In that class the three-name enum took the unused code 3 in 256 draws of 1,000:

```
[ENUM] 3 names, in a parameterized class:      235  247  262   and 256 values that
aren't names
```

`randomize()` returned 1 every time, and the run printed no warning.

A dist written with enum names

A `dist` gives each value or range a weight, and under the standard it makes no difference whether you write the values as enum names or as numbers. Here are weights of 10, 20, 30 and 40 written both ways:

```
[DIST] weights 10/20/30/40 by name:   K_ZERO  229  K_ONES  256  K_SAME  267  K_ANY  248
[DIST] weights 10/20/30/40 by number:  K_ZERO   93  K_ONES  212  K_SAME  291  K_ANY  404
```

Written with the names, the weights had no effect, and each kind came out about a quarter of the time. Written as the numbers 0 to 3, the kinds came out close to 10, 20, 30 and 40 percent. In line 2 of Lab 03′s experiments, weights of 25, 50 and 25 written with names went wrong in a different way, with the 50 on the wrong name. So wherever a `dist` gives values different weights, the course writes the values as numbers, as in `kind dist {0 := w_one, 1 := w_mid, 2 := w_max}`.

A dist and another constraint on the same variable

To the standard, a `dist` is one more constraint, solved together with all the others. Verilator 5.050 behaves as if it first picks one entry of the `dist` list by weight, and only then solves the rest. If the other constraints rule out the entry it picked, the call fails:

```
[DIST] op dist {0 := 5, 1 := 1, ...}:   ADD  513  SUB   96  OR   92  NAND  102  SHL
94  SHR  103
[DIST] op dist {0 := 5, 1 := 1, ...}, then with { op == SUB; }: randomize() failed 921
times in 1000
[DIST] the same with c_op.constraint_mode(0):              randomize() failed 0 times
in 1000
[DIST] op dist {[ADD:SHR] := 1}, then with { op == SUB; }:  randomize() failed 0 times
in 1000
```

The weights came out right. But asking for `SUB` with the `dist` switched on failed 921 times in 1,000, which is about the nine draws in ten that picked an entry other than `SUB`. Lab 01′s `dist` is a single range, which counts as one entry. Whatever it picks allows `SUB`, so asking for `SUB` failed 0 times in 1,000.

The same thing happens when the call constrains a different variable, one that depends on the variable with the `dist`. This class puts the `dist` on a selector, `kind`, and uses implications to choose `A` from `kind`, which is the pattern the labs use:

```
class vec_kind;
  rand kind_e    kind;
  rand bit [7:0] A, B;
  constraint c_kind { kind dist {0 := 10, 1 := 20, 2 := 30, 3 := 40}; }
  constraint c_A    { (kind == K_ZERO) -> A == 8'h00;
                      (kind == K_ONES) -> A == 8'hFF;
                      (kind == K_SAME) -> A == B; }
endclass
```

```
[DIST] kind dist, then with { A < 8'h10; }:                    randomize() failed 199
times in 1000
[DIST]   the calls that worked: K_ZERO 114  K_ONES 0  K_SAME 305  K_ANY 382
[DIST] the same with c_kind.constraint_mode(0):                  randomize() failed 0
times in 1000
```

`A < 8'h10` rules out `K_ONES`, which has a fifth of the weight, and a fifth of the calls failed. A selector is still the better place for a `dist`, because a call then fails only when its constraint rules out a whole kind. In line 3 of Lab 03′s experiments, `len != 7` made 72 calls in 1,000 fail with the `dist` on `len` itself, and none with the `dist` on `kind` and `len` chosen from it. For a call whose constraint does rule out a kind, switch the `dist` off with `constraint_mode(0)`, or build the item without the solver. Lab 03′s `close_holes()` builds its fill packets with `rand_packet::make()`, because with the packet's weights on, `randomize() with { len == 3; }` failed 478 times in 1,000 (also line 3 of its experiments).

solve...before

`solve big before len` asks the solver to choose `big` first, which changes how often each value of `big` comes out without changing which values are legal. In Lab 03′s experiment the standard gives `big` about 500 times in 1,000 with it and about 125 without, and Verilator gave 33 and 41, as if the line weren't there. `cr_solve_test` tries a different shape, where `is_shift` chooses between the two shifts and the four other operations:

```
[SOLVE] solve is_shift before op:  is_shift in 665 of 1000;   ADD   51  SUB   77  OR
74  NAND  133  SHL  223  SHR  442
[SOLVE] without solve...before:    is_shift in 512 of 1000;   ADD   69  SUB  178  OR
58  NAND  183  SHL   64  SHR  448
```

For `is_shift`, the standard gives about 500 with `solve...before` and about 333 without, because two of the six operations are shifts. Verilator's runtime has a separate pass for `solve...before`, and in this shape it moved the count from 512 to 665, but neither experiment came out at the standard's numbers. So don't use `solve...before` to shape a distribution on this simulator. If you want `is_shift` half the time, ask for it with a `dist`, which gave 502 in 1,000:

```
[SOLVE] is_shift dist {0:=1, 1:=1}: is_shift in 502 of 1000;   ADD   56  SUB   53  OR
196  NAND  193  SHL   59  SHR  443
```

The operations within each half still aren't even (`SHR` 443 against `SHL` 59). As in "The solver" above, this simulator doesn't choose evenly among the values a constraint allows, although giving up one bit's target at a time doesn't by itself explain why `SHR` wins over `SHL` here. If the spread of the operations matters to your test, count it.

unique and sum() on arrays

Verilator accepts `unique {a}` on an array but doesn't enforce it. In line 6 of Lab 03′s experiments, four random 4-bit values repeated a value in 315 draws of 1,000, which is about the one draw in three that you'd get with no constraint at all. On four separate variables, as `unique {w, x, y, z}`, it works, and no value repeated in 1,000 draws. `lens.sum() == 8` on a dynamic array can't be solved at all and failed 20 times in 20, while the same constraint on a fixed-size array works. That's why Lab 03′s `fill_plan` has `lens[8]`, with a 0 meaning "no packet".

Seeds

Every run starts the random number generators from a seed, which `make` passes as `+verilator+seed+N` from its `SEED` variable (1 unless you set it). The same seed gives the same values. We ran `cr_item_test` twice with seed 1 and got the same nine lines both times, and seed 2 gave a different first line, `SHR A=2d B=02`. So you can rerun a failing test and watch the same failure happen again, and `make seeds` can run a test on seeds 1 to 10.

A seed reproduces a run only when the testbench is the same, because a different class asks the solver different questions. That's why Lab 03 says your own constraints may give different counts from ours with the same seed.

Counting what came out

The four-name enum that came out as its first name 813 times in 1,000 broke no constraint and printed no warning, and its run passed. In a testbench with a design in it, the scoreboard wouldn't have noticed either, because it checks the design's answers and not how often each input was sent. The tests on this page found it by counting the values in arrays of their own. In Lab 03 you count with a covergroup instead, and its first run shows about a third of the router's packets 1 beat long and a third 8 beats long (260 and 270 of 800), because the packet's length comes from a three-name enum, `kind`, that no constraint weighted. Functional coverage you can read explains how to read those counts.

Check yourself

1. A transaction has `rand bit [1:0] dst;` and `rand mode_e mode;`, where `mode_e` has four names in two bits, and no constraint mentions `mode`. What do you expect 1,000 draws to give, and what do you add?

   Answer: A1, in Answers at the end.

2. An ALU item has `constraint c_op { op dist {0 := 2, 1 := 1, 2 := 1, 3 := 1, 4 := 1, 5 := 1}; }`. A directed sequence calls `req.randomize() with { op == NAND; }`. What happens, and what are two ways to fix it?

   Answer: A2, in Answers at the end.

3. A run of your Lab 03 test passes, and the log has one `UNSATCONSTR` warning in it. Why does `make run` call it BROKEN rather than ok?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's SystemVerilog pages cover each construct here with longer examples. Their output comes from commercial simulators, so where Verilator 5.050 differs, trust the counts on this page and in Lab 03.

- SystemVerilog Randomization
- Constraint Blocks
- Inline Constraints. On Verilator 5.050, an inline constraint on a variable that has a `dist` fails whenever the `dist` picked an entry that the inline constraint rules out (see "A dist and another constraint on the same variable" above).
- Constraints. One of its `dist` examples names a variable `priority`, which is a SystemVerilog keyword, so that example doesn't compile. Another gives its weights to enum names, and on Verilator you should write those values as numbers.
- solve before explains correctly what the standard means, but Verilator 5.050 doesn't produce the distributions it shows.
- Soft Constraints. It says that the solver decides which of two conflicting soft constraints wins. The standard fixes the order: a later one beats an earlier one, a derived class beats its base, and an inline `with` beats the class.
- Disable Constraints, for `constraint_mode()`.

Next: Functional coverage you can read.

Answers

A1. Check yourself

Expect the first name about four times in five (813 here and 809 in Lab 03), with no warning. Add a constraint that names `mode`. Because its four names fill its two bits, `mode inside {...}` with all four names is enough, and that gave an even spread above. For an enum whose names don't fill their bits, use a `dist` with one range and equal weights, as Lab 01's `alu_item` does.

A2. Check yourself

About six calls in seven fail. The simulator picks an entry of the `dist` first, and only the entry for 3 (`NAND`), which has a seventh of the weight, satisfies `op == NAND`. Either switch the `dist` off for the call with `req.c_op.constraint_mode(0)`, or set `op` by hand without calling `randomize()`, as Lab 01's directed sequence does.

A3. Check yourself

A failed `randomize()` means that at least one item wasn't what the test asked for. On this simulator the failed call can also change some of the item's fields (`A` went from `35` to `e5` above), so the item may hold a mix of old and new values. A pass then says only that the router handled whatever was sent, and that isn't the stimulus the test describes.

11. Functional coverage you can read

This page answers: *how do I find out what my random stimulus actually reached, on a simulator whose coverage percentage doesn't follow the standard?*

In DD104, functional coverage was a list of questions that you kept yourself (have I run `SUB` with two different operands? have I taken a backward branch?), and you could answer each of them with a `for` loop and a set. A **covergroup** is the same kind of list, written in SystemVerilog. You name the values you care about, the simulator counts how often the stimulus produces each one, and the counts are written to a file that `verilator_coverage` can merge and report on. In Lab 03, Part 2 you write one for the router's packets. `router_rand_test` passes there and catches neither of the two bugs planted in that lab's router, and your covergroup shows you two holes in the test's stimulus, which is where those bugs are.

On Verilator 5.050 the count in each bin is right, but the percentage is worked out with a different formula from the standard's, `get_coverage()` returns 0, an array of bins on a coverpoint that's in a cross becomes a single bin, and `ignore_bins` still count as bins. This page shows the form of covergroup in a class that Verilator 5.050 supports, where its counts are written, and how to read them. The example, `guide-examples/functional-coverage-you-can-read/`, samples random ALU vectors.

Coverpoints, bins and crosses

Here is a covergroup for ALU vectors. It counts the operations, three kinds of value for `A`, and whether the two operands were equal:

```
covergroup cg_alu with function sample(int op, bit [7:0] a, bit eq);
  cp_op: coverpoint op {
    bins add = {0}; bins sub = {1}; bins or_op = {2};
    bins nand_op = {3}; bins shl = {4}; bins shr = {5};
  }
  cp_a: coverpoint a {
    bins zero  = {8'h00};
    bins ones  = {8'hFF};
    bins other = {[8'h01:8'hFE]};
  }
  cp_eq: coverpoint eq {
    bins differ = {0};
    bins equal  = {1};
  }
  x_op_eq: cross cp_op, cp_eq;
endgroup
```

Each **bin** is one question, and its count is the number of samples that answered yes. A **coverpoint** groups the bins for one value, and a **cross** makes one bin for every combination of two coverpoints' bins. `x_op_eq` therefore has 6 × 2 = 12 bins, from `add_x_differ` to `shr_x_equal`. It asks DD104's question about operands for every operation, with a `differ` bin that counts the vectors with two different operands and an `equal` bin that counts the vectors with `A == B`.

SimProc programs use equal operands on purpose, because `SUB rX, rX` is how you zero a register (nearly every program does it) and `NAND rX, rX` is how you write NOT. Equal operands also hide two of the ALU bugs from Lab 01. With `A == B`, `SUB` gives 0 whichever way round it subtracts (A01), and a `NAND` that computes NOR (A02) still gives the right answer, because `~(A & A)` and `~(A | A)` are both `~A`.

The covergroup takes its values as arguments of `sample()`, and whoever calls `sample()` passes them in. Most examples of a covergroup in a class are written another way, with coverpoints that read a member of the class, as in `coverpoint v.op`, and a call to `sample()` with no arguments. Verilator 5.050 doesn't support that form. The example's `CG_MEMBER` build writes `cg_alu` that way, and Verilator prints a warning for it and stops:

```
%Warning-COVERIGN:  guide-examples/functional-coverage-you-can-read/cov_pkg.sv:48:25:
Unsupported: 'covergroup' coverpoint referencing enclosing class member; ignoring
covergroup '__vlAnonCG_cg_alu'
%Error: Exiting due to 1 warning(s)
```

Verilator ignores the whole covergroup. The course's build stops on any warning that it hasn't been told to ignore, so this build stopped after 4 s, and you find out from the build instead of from a coverage report with no bins in it. In Lab 03, Part 2 we have you write the router's covergroup this way first, so that you see the warning. Lab 04's coverage subscriber, `router_coverage`, is written in the `with function sample` form because Verilator 5.050 ignores the other one.

coverage.dat and make cov

By default, the course's `uvm.mk` builds every simulator with Verilator's `--coverage-user` option, and every run then writes the count of every bin to `coverage.dat` in its own run folder. `make cov` merges every `coverage.dat` under `build/runs` with `verilator_coverage`. It prints how many bins of each coverpoint and cross were hit, then lists by name every bin that was never hit, and it writes every bin with its count to `build/coverage/bins.txt`. Here is what it printed after one run of `cov_plain_test`, which samples 300 vectors with `A` and `B` uniformly random:

```
 verilator_coverage: covergroup : 82.9% (34/41)
-------------------------------------------------------------------------
 41 covergroup bins, 34 hit at least once:
  __vlAnonCG_cg_alu.cp_a                          3 of 3    bins hit
  __vlAnonCG_cg_alu.cp_eq                         2 of 2    bins hit
  __vlAnonCG_cg_alu.cp_op                         6 of 6    bins hit
  __vlAnonCG_cg_alu.x_op_eq                       8 of 12   bins hit
  __vlAnonCG_cg_arr.cp_alone                      4 of 4    bins hit
  __vlAnonCG_cg_arr.cp_crossed                    1 of 1    bins hit
  __vlAnonCG_cg_arr.cp_half                       2 of 2    bins hit
  __vlAnonCG_cg_arr.x_w_half                      2 of 2    bins hit
  __vlAnonCG_cg_ign.cp_op                         6 of 9    bins hit
-------------------------------------------------------------------------
 7 bins never hit:
    __vlAnonCG_cg_alu.x_op_eq.add_x_equal
    __vlAnonCG_cg_alu.x_op_eq.or_op_x_equal
    __vlAnonCG_cg_alu.x_op_eq.shl_x_equal
    __vlAnonCG_cg_alu.x_op_eq.shr_x_equal
    __vlAnonCG_cg_ign.cp_op.op[6]
    __vlAnonCG_cg_ign.cp_op.op[7]
    __vlAnonCG_cg_ign.cp_op.unused
```

Above the `covergroup` line, the summary also lists line, toggle and four other kinds of coverage, all at `0.0% ( 0/ 0)`, because the course's builds collect only covergroups. `__vlAnonCG_cg_alu` is the name Verilator gives the covergroup `cg_alu`.

The other two covergroups in the list, `cg_arr` and `cg_ign`, are in the example to show how Verilator counts bins, and the next section comes back to them. In `cg_alu`, every operation and every kind of `A` came up, but no `ADD`, `OR`, `SHL` or `SHR` vector had equal operands, so four of the twelve cross bins are empty. Only 2 of the 300 vectors had `A == B`, which is about what uniform random operands give, because `B` matches `A` in one vector in 256. `cov_biased_test` makes `A == B` in one vector in eight, and that fills the cross:

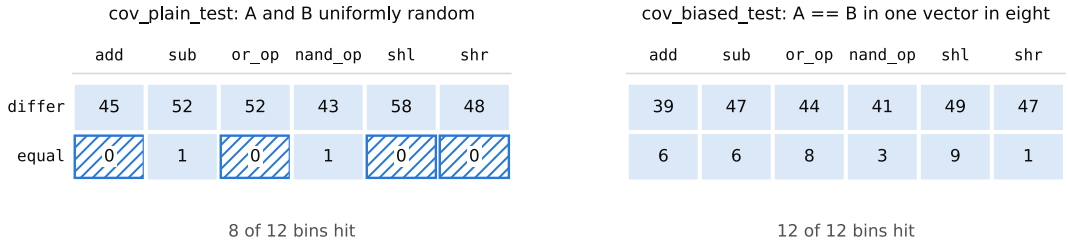


Figure 11.1: The cross x_op_eq as a grid of counts after cov_plain_test, with four empty bins in the equal row, and after cov_biased_test, with all twelve hit.

The 12 bins of `x_op_eq` after one run of each test. With uniform operands, the `equal` row has a count of 1 for `sub` and `nand_op` and 0 for the other four operations. With `A == B` in one vector in eight, every bin is hit.

The percentage

Verilator's `82.9% (34/41)` is the number of bins hit divided by the number of bins, with every bin of every covergroup counted the same, crosses included. The standard works it out another way, and gives a covergroup the average of the coverage of its coverpoints and crosses. For `cg_alu` on its own, the test's report phase prints what the covergroup's two methods return:

```
[COV] cg_alu: get_inst_coverage() = 82.61, get_coverage() = 0.00
```

`get_inst_coverage()` uses Verilator's formula again, because 82.61% is 19 bins hit out of `cg_alu`'s 23. The standard's formula would average `cp_op`, `cp_a` and `cp_eq` (100% each) with `x_op_eq` (8 of 12, 67%), which gives 92%. `get_coverage()`, which the standard defines as the coverage of the covergroup type as a whole, returned 0.

The example's other two covergroups show two more differences, and these change the bins themselves:

```
covergroup cg_arr with function sample(int v, int w);
  cp_alone:   coverpoint v { bins v[] = {[0:3]}; }
  cp_crossed: coverpoint w { bins w[] = {[0:3]}; }
  cp_half:    coverpoint v { bins lo = {[0:1]}; bins hi = {[2:3]}; }
  x_w_half:   cross cp_crossed, cp_half;
endgroup

covergroup cg_ign with function sample(int op);
  cp_op: coverpoint op { bins op[] = {[0:7]}; ignore_bins unused = {6, 7}; }
endgroup
```

- An array of bins on a crossed coverpoint becomes one bin. `bins v[] = {[0:3]}` gave four bins on `cp_alone`, but the same array on `cp_crossed` gave a single bin, `w`, which counted all 300 samples, and so the cross `x_w_half` has 2 bins where the standard gives it 8. In a coverpoint that's part of a cross, write every bin out by name, as Lab 03 asks you to.
- `ignore_bins` stay in the count. The standard removes 6 and 7 from `cp_op`, which leaves 6 bins, all of them hit, so `cp_op` would be at 100%. Verilator keeps `op[6]` and `op[7]` as bins and lists the ignored set as a bin of its own, `unused`, so `verilator_coverage` counts 6 hit of 9. `get_inst_coverage()` gave a third answer, 75.00, which is 6 of 8.

In Lab 03, Part 2 you work the same sums for the router's covergroup, and get 46.3% by Verilator's formula and 61% by the standard's. Twelve of that covergroup's 41 bins can't be hit by any test in the lab, and they count in both percentages, so neither can reach 100% however good your stimulus is. The counts in `coverage.dat` don't depend on either formula, so to find the holes, read the list of bins that were never hit at the end of `make cov`'s output, as you did above for the four `equal` bins.

Three kinds of empty bin

A bin can be empty for one of three reasons. Some bins can't be hit by any test, because the specification rules them out. `op[6]` and `op[7]` above are two of them, since SimProc's ALU has only the six operations 0 to 5. The router's `full_x_low` in Lab 03 is another, since the router drops a packet as FULL only when the output its route names holds more than 8 beats. Some bins can be hit, but not by the testbench you have, because of how it was built. Lab 03's tracker takes output 1's level as 0, for example, so `out1_x_low` stays empty until you fix that in Lab 04. The rest are holes in the stimulus, like the `equal` bins above, and those are the ones worth your time, because a bug can sit in them. Lab 03, Part 2 asks you to sort the router's 41 bins into these three groups, and both of the lab's planted bugs are in bins of the third.

Check yourself

1. A coverpoint `cp_kind` has `bins k[] = {[0:3]}` and is crossed with a coverpoint that has two bins. How many bins does the cross have on Verilator 5.050, and what do you change?

   Answer: A1, in Answers at the end.

2. A run's report prints `get_coverage() = 0.00` for its covergroup. Was nothing covered?

   Answer: A2, in Answers at the end.

3. After `cov_plain_test`, `make cov` reports `82.9% (34/41)`. Which of the 7 empty bins would more seeds of the same test fill, and which can no test fill?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's coverage pages explain the language well, but the numbers on them come from commercial simulators.

- Covergroup and Coverpoint. Correction: it says a 32-bit coverpoint with default bins would make billions of bins; the standard caps automatic bins at `auto_bin_max`, 64 by default, so it makes 64 range bins.
- Coverpoint Bins, for `bins x[]`, `bins x[N]` and `default`. It doesn't cover `ignore_bins` or `illegal_bins` on a coverpoint, or transition bins.
- Cross Coverage, for `binsof` and `intersect`. On Verilator 5.050 the percentage is computed differently (bins hit over all bins), and an array of bins on a crossed coverpoint becomes one bin.
- Functional Coverage. Correction: its `sample()` example constructs the covergroup with `CovGrp = new;`, which assigns to the type name; it needs an instance variable. It also repeats the claim that default bins make "one bin per value".
- UVM Functional Coverage, for a coverage subscriber. Its covergroup reads a member of the class, which Verilator 5.050 ignores (the COVERIGN warning above), so write yours `with function sample`, as Lab 04's `router_coverage` is written.

Next: Lab 03: Constraints and coverage.

Answers

A1. Check yourself

Two, because `k[]` becomes a single bin on a crossed coverpoint. Write the four bins out by name (`bins k0 = {0}; bins k1 = {1};` and so on), and the cross then has 4 × 2 = 8.

A2. Check yourself

Not necessarily. In the example's `cov_biased_test` run, all 23 of `cg_alu`'s bins were hit, and `get_coverage()` still returned 0. Read the counts from `make cov` instead.

A3. Check yourself

More seeds would fill the four `equal` bins, but slowly, because only about one vector in 256 has equal operands and those few are spread over six operations. `op[6]` and `op[7]` can't be hit, because SimProc's ALU has no such operations, and `unused` is the ignored set, which no sample is meant to hit.

12. Lab 03: Constraints and coverage

You'll add a covergroup to the Lab 02 testbench and use it to see what the testbench's random stimulus sends the router and what it never sends. Then you'll constrain the packet so that the simulator produces the length distribution you ask for. Last, you'll write stimulus that reaches the two coverage holes where this lab's two planted bugs are.

Package: `labs/lab-03-constraints-and-coverage/`

Before you start

- Finish Lab 02. This lab's package is the Lab 02 testbench with our solution's `monitor.svh` and `scoreboard.svh` in `tb/`. Around them it adds a packet that `randomize()` makes, a sink that can stall, an APB read, a tracker that predicts each packet's fate, and a check on `DROP_CNT`. Part 1 walks through each of them. If your own monitor and scoreboard passed Lab 02's `make check-all`, you can copy them over ours (the error counts quoted on this page come from ours, so yours may differ). Run the copy from the lab folder, which the last step of this list takes you to:

```
cp    ../lab-02-a-class-testbench-without-uvm/tb/monitor.svh    ../lab-02-a-class-
testbench-without-uvm/tb/scoreboard.svh tb/
```

- Read Constrained random on Verilator and Functional coverage you can read.
- Keep the router specification open, and read its Routing section again, because this lab is about the level of an output and what the router does with a packet at each level.
- In a terminal on the workbench desktop, load the course and go to the lab folder:

```
. /foss/designs/common/.designinit
mod dd105-a
cd labs/lab-03-constraints-and-coverage
```

Time: about 1 h 30 min. On 2 CPUs, the lab's five builds took us 107 s (the first), 4 s (the one that stops in Part 2), 100 s, 89 s and 89 s, and `make traps` took 9 s.

Part 1: The testbench you start from

What does the starting testbench check, and does it catch this lab's two bugs?

1. Start the build now, because it takes a while:

```
make build
```

With an empty ccache it took us 107 s (our other cold builds of this package took 94 to 110 s), and it ends with:

```
- V e r i l a t i o n   R e p o r t: Verilator 5.050 2026-07-01 rev v5.050
- Verilator: Built from 48.441 MB sources in 400 modules, into 23.435 MB in 2578 C+
+ files needing 82.126 MB
- Verilator: Walltime 106.553 s (elab=0.708, cvt=2.588, bld=102.550); cpu 4.308 s
on 2 threads; allocated 314.551 MB
=======================================================================
 built /w/labs/lab-03-constraints-and-coverage/build/sim in 107 s (49M; the 905M of
generated C++ was deleted, SLIM=0 keeps it)
 ccache:  Hits: 0 / 36 ( 0.00%)  Misses: 36 / 36 (100.0%)
```

```
==========================================================================
 /w/labs/lab-03-constraints-and-coverage/build/sim is up to date.
```

We ran it with the course folder mounted at `/w`; your paths start with `/foss/designs/modules/dd105-a` instead. The ccache counts add up over every build in your clone, and yours already include the builds of Labs 01 and 02, so they'll be higher than ours.

2. While it builds, read the files in `tb/`. Most of them are Lab 02's:

File	What it is
`stream_if.sv`, `apb_if.sv`, `router_tb_top.sv`	Lab 02's interfaces and top module, with the same code
`monitor.svh`, `scoreboard.svh`	Lab 02's monitor and scoreboard: our solution, or yours
`driver.svh`	Lab 02's driver, sink and APB master, with two additions: `apb_master.read()`, and `sink.stall`, which holds `out_ready` at 0 from the next rising edge on, for as long as `stall` is 1
`packet.svh`	Lab 02's `packet` and `router_cfg`, and a new `rand_packet`, whose DST (`hdr_dst`) and length kind (`kind`) come from `randomize()`. Its constraint `c_dst` keeps a packet on an enabled route. Part 3 adds the weights on `kind` and the constraint that picks the length from `kind`. `router_cfg` still reads `+N`, `+ROUTES` and `+SHOW`, but not Lab 02's `+DSTS` or `+LEN`, because the constraints now choose the DST and the length.
`generator.svh`	randomizes `rand_packet`s, where Lab 02's generator called `$urandom`
`tracker.svh`	new: follows output 0's level and decides each packet's fate
`coverage.svh`	new: the class `router_coverage`, with an empty `sample_pkt()` and no covergroup yet. Part 2 adds the covergroup.
`env.svh`	Lab 02's `env`, with `run()` split into steps a test can call one by one, and a `DROP_CNT` check
`router_test_pkg.sv`	`router_rand_test`, `router_fill_test` and Lab 02's `router_class_test`. `close_holes()` is empty: Part 4 fills it in.

The tracker watches the input and output 0 once per cycle, at the falling edge, and handles the cycle's input handshake before its output handshake, so the level a header sees counts only earlier cycles, as the Routing section defines it. From the routing table and that level it decides each packet's fate: NOROUTE, FULL, or forwarded to output 0 or 1. Nothing watches output 1, and its sink is always ready, so its level never comes near 8 and the tracker takes it as 0. It counts the drops it predicts, and `env.check_drops()` reads `DROP_CNT` over APB and reports a `$error` if the two disagree. Lab 02's scoreboard still checks the packets that come out of output 0.

`router_rand_test` sets `CTRL.EN` and runs four phases. Each phase writes a random routing table with at least one disabled route and an enabled route to each output, sends 200 random packets with both sinks ready, waits for the outputs to drain, checks `DROP_CNT` and then clears it. After the last phase it calls `close_holes()` and checks `DROP_CNT` once more.

3. Run the test:

```
make run
```

After UVM's release notes you should see:

```
UVM_INFO @ 0: reporter [RNTST] Running test router_rand_test...
[PHASE] 1 of 4: routes 011-, 200 packets
[DROP] DROP_CNT 00000000 as expected
[PHASE] 2 of 4: routes -10-, 200 packets
[DROP] DROP_CNT 00000000 as expected
[PHASE] 3 of 4: routes -01-, 200 packets
[DROP] DROP_CNT 00000000 as expected
[PHASE] 4 of 4: routes -011, 200 packets
[DROP] DROP_CNT 00000000 as expected
[DROP] DROP_CNT 00000000 as expected
[ENV] sent 800 packets (3578 beats); output 0 took 1405 beats and output 1 took 2173
[TRK] 800 packets: 321 to output 0, 479 to output 1, 0 NOROUTE, 0 FULL; the highest
level a header saw was 2
[SB] 800 packets in, 321 predicted for output 0, 321 checked, 0 errors
DD105: TEST PASSED
```

and last, `ok: 0 UVM_ERROR, 0 UVM_FATAL  0.63 s  log: build/runs/router_rand_test-s1/run.log`. The routes are written as Lab 02's `+ROUTES` was: one character per route, `0` or `1` for the output, `-` for disabled.

4. This lab's two bugs are B07 and B08. B07 counts a packet for a disabled route as both a NOROUTE drop and a FULL drop when the output its route names holds more than 8 beats. B08 drops a packet that arrives when its output holds exactly 8 beats, although an output at 8 still has room for it. B07 changes no packet that comes out, only `FULL_CNT` (and `STATUS.FULL`), so the `DROP_CNT` check is the only check in the package that could see it. Write down whether you expect `router_rand_test` to catch the two bugs, then run it against both:

   ```
   make check-all
   ```

   You should see:

   ```
        ok  the golden design                    0 UVM_ERROR, 0 UVM_FATAL       0.7s
   ----------------------------------------------------------------------------
      MISSED   tier 2   B07_full_counted_on_disabled_route  0 UVM_ERROR,  0 UVM_FATAL
   0.6s
     MISSED  tier 3  B08_admission_off_by_one       0 UVM_ERROR, 0 UVM_FATAL       0.6s
   ----------------------------------------------------------------------------
    0 of 2 caught.
    Still hiding from this test:
      B07_full_counted_on_disabled_route
         hides behind: testing each drop reason on its own
      B08_admission_off_by_one
        hides behind: keeping the outputs drained, or flooding them, and never stopping
   at exactly 8
   ```

   followed by an error line from `make`, because the verdict is a fail. Using the `[TRK]` line from step 3, explain why `router_rand_test` can't reach either bug, on any seed.

   Answer: A1, in Answers at the end.

Check: `ok` on the golden design and `0 of 2 caught.`

Part 2: A covergroup you can read

What does `router_rand_test` send the router, and which fates and levels does it never reach?

1. Read `post_randomize()` in `tb/packet.svh`, and how `kind` is declared. No constraint names `kind`. Write down how many of the 800 packets you expect to be 1 beat long, and how many 8.

2. Open `tb/coverage.svh`. The tracker calls `sample_pkt()` once for every packet, with a `pkt_fate` record, which is declared in `tracker.svh`. The record holds the packet's length, its DST, its fate (`FATE_NOROUTE`, `FATE_FULL`, `FATE_OUT0` or `FATE_OUT1`, which are 0 to 3), and the level of the output its route names in the cycle its header was accepted. Write a covergroup `cg_pkt` with these coverpoints and bins. Use these names, because the outputs below use them:

Coverpoint	Bins
`cp_len`	`len1` to `len8` for 1 to 8
`cp_dst`	`dst0` to `dst3`
`cp_fate`	`noroute`, `full`, `out0`, `out1` for 0 to 3
`cp_level`	`empty` for 0, `low` for `[1:7]`, `at_8`, `at_9`, `high` for `[10:16]`
`x_fate_level`	`cross cp_fate, cp_level`

   Write every bin out by name, as `bins low = {[1:7]};`. An array of bins such as `bins b[] = {[0:3]}` becomes a single bin on this simulator when its coverpoint is in a cross.

   Write it first the way most examples write a covergroup in a class: keep the record in a member (`pkt_fate rec;`), let the coverpoints read it (`cp_len: coverpoint rec.len { ... }`), construct the covergroup in `new()` (`cg_pkt = new();`), and in `sample_pkt()` store the record and call `cg_pkt.sample()`.

3. Build it:

```
make build
```

   It stops after about 4 s with these two lines, and then prints them again under `BUILD FAILED after 4 s`, with the source line they point at:

```
%Warning-COVERIGN:     labs/lab-03-constraints-and-coverage/tb/coverage.svh:16:24:
Unsupported: 'covergroup' coverpoint referencing enclosing class member; ignoring
covergroup '__vlAnonCG_cg_pkt'
%Error: Exiting due to 1 warning(s)
```

   with your own line and column. Verilator 5.050 doesn't support a coverpoint that reads a member of the class the covergroup is in, and, as the warning says, it ignores the whole covergroup instead. The course's build stops on any warning that it hasn't been told to ignore. If it didn't, the build would finish without the covergroup, and you'd only find out from a coverage report with no bins in it.

4. Rewrite it so the values arrive as arguments of `sample()`:

```
covergroup cg_pkt with function sample(int len, int dst, int fate, int level);
```

   with coverpoints on `len`, `dst`, `fate` and `level`, and in `sample_pkt()`:

```
cg_pkt.sample(f.len, f.dst, int'(f.fate), f.level);
```

   Delete the `rec` member, then run `make build` again. Adding a covergroup changes what the class holds, so expect a rebuild that costs close to a full build; ours took 100 s.

5. Every run writes the hit count of every bin to `coverage.dat` in its run folder, and `make cov` merges every `coverage.dat` under `build/runs`, so delete the old runs first and then run the test and read its coverage:

```
rm -rf build/runs
make run
make cov
```

The coverage summary also lists line, toggle and other kinds at `0.0% ( 0/ 0)`, because this build collects only covergroups. After them you should see:

```
 verilator_coverage: covergroup : 46.3% (19/41)
-------------------------------------------------------------------------
 41 covergroup bins, 19 hit at least once:
  __vlAnonCG_cg_pkt.cp_dst                        4 of 4    bins hit
  __vlAnonCG_cg_pkt.cp_fate                       2 of 4    bins hit
  __vlAnonCG_cg_pkt.cp_len                        8 of 8    bins hit
  __vlAnonCG_cg_pkt.cp_level                      2 of 5    bins hit
  __vlAnonCG_cg_pkt.x_fate_level                  3 of 20   bins hit
```

then the 22 empty bins by name: `cp_fate.noroute` and `full`, `cp_level.at_8`, `at_9` and `high`, and 17 of the cross's bins. `__vlAnonCG_cg_pkt` is the name Verilator gives the covergroup.

6. `build/coverage/bins.txt` has every bin with its count. Look at `cp_len`:

```
grep cp_len build/coverage/bins.txt
```

```
       260  __vlAnonCG_cg_pkt.cp_len.len1
        50  __vlAnonCG_cg_pkt.cp_len.len2
        58  __vlAnonCG_cg_pkt.cp_len.len3
        38  __vlAnonCG_cg_pkt.cp_len.len4
        51  __vlAnonCG_cg_pkt.cp_len.len5
        34  __vlAnonCG_cg_pkt.cp_len.len6
        39  __vlAnonCG_cg_pkt.cp_len.len7
       270  __vlAnonCG_cg_pkt.cp_len.len8
```

Compare 260 and 270 with what you wrote down in step 1. Nothing in the packet says how often each kind should come out, so where does a third each come from?

Answer: A2, in Answers at the end.

7. Read the hides-behind lines that `make check-all` printed in Part 1. Which empty bins would a packet hit if it met B08′s condition, and which if it met B07′s? And of the 41 bins, which can no test ever hit, and which can't this testbench hit?

Answer: A3, in Answers at the end.

Check: `covergroup : 46.3% (19/41)`, and `len1` at 260.

Part 3: Constraints the simulator honours

How do you get the length distribution you ask for on this simulator, and how do you know you got it?

1. Open `traps/traps.sv`. It has six small experiments, numbered 1 to 6, and each one randomizes two or three small classes many times. Lines 1 to 3 use the shapes of this lab's packet, and lines 4 to 6 use shapes that a test might use. For each line it prints, write down what the SystemVerilog standard says it should show.
2. Run it. It builds with plain Verilator, without UVM, in seconds:

```
make traps
```

You should see:

```
1  three names, no constraint names kind    LEN_ONE  313  LEN_MID  348  LEN_MAX  339
1  four names, no constraint names op        OP_ADD  809  OP_SUB   74  OP_AND   64
OP_OR   53
1  four names, with op inside {...}          OP_ADD  254  OP_SUB  244  OP_AND  269
OP_OR  233
2  kind dist {LEN_ONE := 25, ...}          LEN_ONE  251  LEN_MID  227  LEN_MAX  522
2  kind dist {0 := 25, ...}                LEN_ONE  262  LEN_MID  483  LEN_MAX  255
3  len dist {...} and len != 7              randomize() failed 72 times in 1000
3  kind dist {...}, len from kind, len != 7  randomize() failed 0 times in 1000
3  the same, with { len == 3; }             randomize() failed 478 times in 1000
3  the same, c_kind.constraint_mode(0)      randomize() failed 0 times in 1000
4  solve big before len                     big in 33 of 1000
4  the same without solve...before          big in 41 of 1000
5  lens.sum() == 8, dynamic array           randomize() failed 20 times in 20
5  lens.sum() == 8, fixed array of 8         randomize() failed 0 times in 20; the
sum was 8 in 20
6  unique {a}, a is bit [3:0] a[4]          a value repeated in 315 of 1000
6  unique {w, x, y, z}                      a value repeated in 0 of 1000
================================================================================
 traps: build 4.4 s, run 4.3 s, seed 1; 1618 UNSATCONSTR warnings in build/traps/
run.log
================================================================================
```

Each failed `randomize()` prints UNSATCONSTR warnings, which the target counts instead of showing. Compare each line with what you wrote.

Answer: A4, in Answers at the end.

3. Now fix the packet. In `rand_packet` in `tb/packet.svh`:

   - make `len` a `rand` field;
   - add weights `int unsigned w_one = 25, w_mid = 50, w_max = 25;` and a constraint `c_kind` with a `dist` on `kind` whose values are written as the numbers 0 to 2, as in `0 := w_one`. Weights in variables let a test change them before it calls `randomize()`;
   - add a constraint `c_len` that chooses `len` from `kind`: `(kind == LEN_ONE) -> len == 1;` and so on for 2 to 7 and for 8;
   - in `post_randomize()`, delete the `case` that set `len`, so that only `build(1)` is left.

4. Before you rebuild, write down how many of the 800 packets you expect in each `cp_len` bin. Then build, run and read the coverage:

```
make build
rm -rf build/runs
make run
make cov
grep cp_len build/coverage/bins.txt
```

The rebuild took 89 s. With the reference solution's `coverage.svh` and `packet.svh` you should see:

```
       189  __vlAnonCG_cg_pkt.cp_len.len1
       108  __vlAnonCG_cg_pkt.cp_len.len2
       110  __vlAnonCG_cg_pkt.cp_len.len3
        49  __vlAnonCG_cg_pkt.cp_len.len4
        47  __vlAnonCG_cg_pkt.cp_len.len5
        47  __vlAnonCG_cg_pkt.cp_len.len6
        50  __vlAnonCG_cg_pkt.cp_len.len7
       200  __vlAnonCG_cg_pkt.cp_len.len8
```

and the run's `[ENV]` line says `sent 800 packets (3398 beats)`. The three kinds came out 189, 411 and 200 times, close to the 200, 400 and 200 you asked for. Your own constraints may give different counts with the same seed, but they should be about as close. If `len8` came out near 400, look at line 2 of the experiments.

5. The six lengths from 2 to 7 share their kind's 411 packets, but not evenly: 108 and 110 for 2 and 3, and 47 to 50 for the rest. What does that tell you about a constraint such as `len inside {[2:7]}`?

   Answer: A5, in Answers at the end.

Check: `len1` near 200, `len8` near 200, and the lengths from 2 to 7 adding up to near 400.

Part 4: Close the two holes

What stimulus reaches B07′s and B08′s bins, and does the testbench catch the bugs once it does?

1. Work it out on paper first, from the specification's Routing section. Output 0 is empty and its sink is stalled. Which packets put it at exactly 8? What happens to the next packet for an enabled route to output 0, and then to a packet for a disabled route whose `PORT` is 0?

   Answer: A6, in Answers at the end.

2. In `tb/router_test_pkg.sv`, before `router_base_test`, write a class `fill_plan` with `rand int unsigned lens[8]`, each 0 to 8, constrained so that `lens.sum()` is 8. A 0 means "no packet". The array has a fixed size because of line 5 of the experiments.

3. Fill in `close_holes()`. The test's environment is `e`, and the steps are:

   - `e.write_routes("0---")`: `ROUTE0` enabled to output 0, the other three disabled (with `PORT` 0);
   - `e.snk0.stall = 1;` and then `e.cycles(2);`, so the stall reaches the pin before any beat can;
   - randomize a `fill_plan`, and for each length `n` that isn't 0, put `rand_packet::make(0, n)` on a queue of `rand_packet`s;
   - one random packet for DST 0: `p = e.gen.new_packet();`, then `p.randomize() with { hdr_dst == 0; }`, and onto the queue;
   - one random packet for DST 1, the same way but with `p.c_dst.constraint_mode(0);` before `randomize()`, because `c_dst` would otherwise forbid a disabled route and every call would fail;
   - `e.send()` with the queue, which returns when the last packet has gone in; then `e.snk0.stall = 0;` and `e.drain();`.

   Both tests that call `close_holes()` check `DROP_CNT` after it returns, so `close_holes()` doesn't need a check of its own. Build the fill packets with `rand_packet::make()`, not with `randomize() with { len == n; }`. With Part 3′s weights on, such a call fails whenever the `dist` draws a kind whose range doesn't hold `n`, and in line 3 of the experiments the call with `len == 3` failed 478 times in 1000. A `$display` that prints the plan's lengths will help you read the waveform later.

4. The list stops after the packet for DST 1, with output 0 still stalled and holding 9 to 16 beats. Suppose it sent one more random packet for DST 0 there. What would the golden router do with it, and what would this testbench report? Write down your answer before you open ours.

   Answer: A7, in Answers at the end.

5. Build, then run the short test, which calls `close_holes()` on its own and then checks `DROP_CNT`:

   ```
   make build
   rm -rf build/runs
   make run UVM_TESTNAME=router_fill_test
   ```

   The rebuild took 89 s. Until `close_holes()` sends something, this test fails on the golden router, because the scoreboard reports a test that checked no packet. With the reference solution you should see:

   ```
   [HOLES] output 0 stalled and filled to 8 by packets of 4 3 1 beats, then 3 beats for
   DST 0 and 1 for DST 1
   [DROP] DROP_CNT 00000001 as expected
   [ENV] sent 5 packets (12 beats); output 0 took 11 beats and output 1 took 0
   [TRK] 5 packets: 4 to output 0, 0 to output 1, 1 NOROUTE, 0 FULL; the highest level
   a header saw was 11
   [SB] 5 packets in, 4 predicted for output 0, 4 checked, 0 errors
   ```

   and `ok: 0 UVM_ERROR, 0 UVM_FATAL    0.03 s`. If the golden router doesn't pass here, read the first error in the log. The router is the golden one, so the error points at your stimulus.

6. Run the graded test too, and read the coverage of both runs:

```
make run
make cov
```

With the reference solution you should see `covergroup : 58.5% (24/41)`, with 17 bins never hit. Five of the 29 bins this testbench can reach are among them. Which are they, and what would hit each one?

Answer: A8, in Answers at the end.

7. Now ask for the verdict:

```
make check-all
```

With the reference solution you should see:

```
      ok  the golden design                        0 UVM_ERROR, 0 UVM_FATAL       1.1s
-------------------------------------------------------------------------------
   caught   tier 2   B07_full_counted_on_disabled_route  1 UVM_ERROR,  1 assertion
1.1s
  caught  tier 3  B08_admission_off_by_one        1 UVM_ERROR, 2 assertions      1.1s
-------------------------------------------------------------------------------
 2 of 2 caught.
```

The one `UVM_ERROR` is the test's summary of the testbench's errors, and the assertions are the `$error`s themselves, as in Lab 02. `make run BUG=B07` shows the one that fired on B07:

```
[54205000] %Error: env.svh:126: Assertion failed in router_tb_pkg.env.check_drops:
DROP_CNT read 00000101, expected 00000001 (FULL_CNT 0, NOROUTE_CNT 1)
```

`FULL_CNT` is bits 15:8, so the router counted the packet for DST 1 as NOROUTE and as FULL. On B08, the same `DROP_CNT` line comes first, and the scoreboard then reports the packet that was dropped at level 8 as never delivered.

8. Record the waveforms of `router_fill_test` on both routers and open them:

```
make waves UVM_TESTNAME=router_fill_test
make waves UVM_TESTNAME=router_fill_test BUG=B08
surfer build/runs/router_fill_test-s1/waves.fst &
surfer build/runs/router_fill_test-s1-B08/waves.fst &
```

Add `clk` from `router_tb_top` and, from `dut`, `in_valid`, `in_data`, `in_last`, `out0_valid`, `out0_ready`, `out0_data` and `level0`, and look at 200 to 460 ns. `level0` is the router's own count of the beats in output 0, the value `LEVEL` reads, and Surfer shows it in hex.

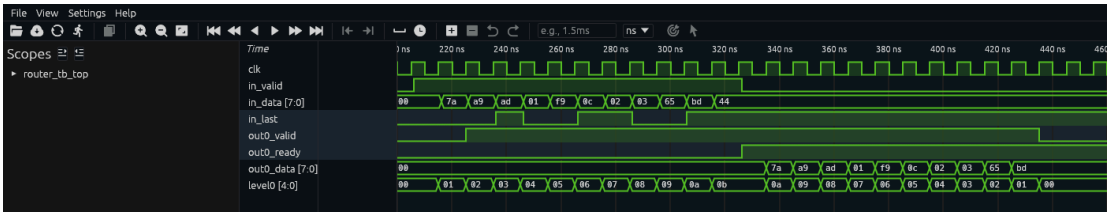


Figure 12.1: router_fill_test on the golden router, 200 to 460 ns, in Surfer

The golden router. The fill packets `00 7a a9 ad`, `01 f9 0c` and `02` go in with `out0_ready` low, and `level0` counts up to 08. The header `03` is accepted at level 8, so its packet `03 65 bd` is admitted and `level0` climbs to 0b. The packet `44`, for DST 1, is dropped. `out0_ready` rises just after 325 ns, and output 0 delivers all 11 beats.

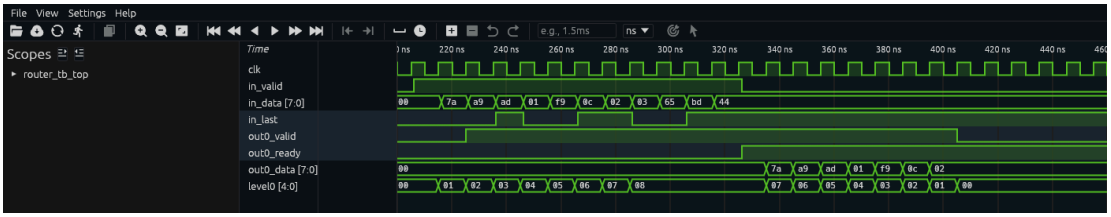


Figure 12.2: The same run with B08 switched on

B08. The same packets, and `level0` stops at 08, because the packet `03 65 bd` is dropped. Output 0 delivers 8 beats.

Which rising edge ends the cycle in which the header `03` is accepted, what is `level0` in that cycle, and what does `level0` do at that edge on each router?

Answer: A9, in Answers at the end.

Check: `2 of 2 caught.`

Finishing

You're done when `make check-all` prints `2 of 2 caught.` with your own covergroup in `tb/coverage.svh`, your constraints in `tb/packet.svh` and your `close_holes()` in `tb/router_test_pkg.sv`.

Questions:

1. The scoreboard and the `DROP_CNT` check you started with passed on both bugs, and you didn't change either of them. What did the coverage tell you that the passing test couldn't?

   Answer: A10, in Answers at the end.

2. Would running the starter on more seeds (`make seeds` runs seeds 1 to 10) have found B08?

   Answer: A11, in Answers at the end.

Submit: nothing for this lab.

Going further (optional)

- Run `make traps SEED=2` and see which lines move. Then change a class in `traps/traps.sv`, predict its line, and run it again.
- Hit the five reachable bins that Part 4 step 6 lists, and check with `make cov`.

Next on the Course path: the reading on the UVM skeleton, sequences, monitors and the factory, then Lab 04: The same testbench in UVM.

Answers

A1. Part 1: The testbench you start from

Both sinks are always ready, so every beat leaves its output within a couple of cycles of arriving, and beats don't pile up. In 800 packets the highest level a header saw was 2. B08 needs a header that arrives while its output holds exactly 8 beats. B07 needs a NOROUTE drop while the output its route names holds more than 8 beats, but `c_dst` keeps every random packet on an enabled route, so there are no NOROUTE drops at all (`0 NOROUTE`). The seed changes the packets and the routing tables, but not the sinks or `c_dst`.

A2. Part 2: A covergroup you can read

The standard makes an unconstrained `rand` enum uniform over its named values, so each of the three kinds should come out about 267 times, and here they came out 260, 270 and 270. The six lengths from 2 to 7 share `LEN_MID`'s 270, so each is about 45. The simulator did what the standard says for this enum, although line 1 of Part 3's experiments shows an enum for which it doesn't. A third of the packets are single beats and a third are 8 beats long only because the enum has three names, and in Part 3 you give the kinds weights of your own.

A3. Part 2: A covergroup you can read

B08 needs a packet admitted to output 0 at a level of exactly 8, which is `out0_x_at_8`. B07 needs a NOROUTE drop while the output its route names holds more than 8 beats, which is `noroute_x_at_9` or `noroute_x_high`.

Seven bins are impossible by the specification, because a FULL drop happens only at a level of 9 or more (`full_x_empty`, `full_x_low`, `full_x_at_8`), and a packet is admitted only at 8 or less (`out0_x_at_9`, `out0_x_high`, `out1_x_at_9`, `out1_x_high`). This testbench can't hit two more, `out1_x_low` and `out1_x_at_8`, because the tracker takes output 1's level as 0. Three more, `cp_fate.full`, `full_x_at_9` and `full_x_high`, need a FULL drop, which the router can make but no test in this lab asks for (Part 4 says why). So this testbench can reach 29 of the 41 bins.

Bins that this lab can't hit still count in the percentage, which is why you read the list of bins rather than the percentage. Verilator's 46.3% is the number of bins hit over all 41 bins, the cross's included. The standard's formula instead averages the coverage of the five items (100, 100, 50, 40 and 15%), which gives 61%.

A4. Part 3: Constraints the simulator honours

1. The standard makes an unconstrained enum uniform over its names, so about 333 each for three names and 250 each for four. The packet's three-name enum came out 313, 348 and 339. The enum whose four names fill its two bits came out as its first name 809 times in 1000, with no warning. With a constraint that names it, here an `inside` that allows all four values, it came out 254, 244, 269 and 233 times. So give every `rand` enum a constraint that names it, even one that allows every value.
2. Both should give about 250, 500 and 250. Written with the enum's names, the weight of 50 went to `LEN_MAX` (522) instead of `LEN_MID` (227); written as numbers, the weights landed where they were written.
3. None of the four should fail, because `len != 7` leaves a value in every weighted range, and with `len == 3` the solver should choose `kind` 1. Here a `dist` behaves as if it draws its value first and only then solves the other constraints, so the call fails whenever another constraint rules out the value it drew. On `len` itself, `len != 7` rules out one of the six values that share the weight of 50, and 72 calls failed. With the weights on a selector, `kind`, the length is solved inside whichever range was drawn, and none failed. But `len == 3` rules out two kinds that hold half the weight, and 478 calls failed. Turning the `dist` off with `c_kind.constraint_mode(0)` stops the failures.
4. With `solve big before len`, the standard has the solver choose `big` first, at 50/50, so about 500. Without it, each of the 8 solutions is equally likely and `big` is 1 in 8, so about 125. Here they gave 33 and 41, so in this experiment `solve...before` didn't move `big` toward 500, and the solver doesn't choose evenly among the solutions either. Constrained random on Verilator tries a different shape, in which `solve...before` did change how often the variable it names first came out, though not to the standard's numbers.
5. Both should work. `sum()` on a dynamic array can't be solved here and fails every time; on a fixed-size array it works.
6. Neither should repeat a value. Here `unique` on an array isn't enforced. A value repeated in 315 of 1000 draws, about one in three, which is how often four random 4-bit values include a repeat with no constraint at all. On separate variables `unique` works.

A5. Part 3: Constraints the simulator honours

The standard makes each of the six equally likely, about 68 each. This simulator's solver favours some of them, as line 4 of the experiments also showed. A constraint says which values are allowed, but on this simulator you can't rely on it for how often each one comes out, so you count them in the coverage.

A6. Part 4: Close the two holes

Any packets for an enabled route to output 0 whose lengths add up to 8 beats. Nothing leaves while the sink is stalled, so the level in the next header's cycle is 8, and a packet for that route is admitted (`out0_x_at_8`, B08's bin) and raises the level to between 9 and 16. A packet for a disabled route is then a NOROUTE drop at that level (`noroute_x_at_9` or `noroute_x_high`, B07's bin), and the specification says it's never counted as FULL as well.

A7. Part 4: Close the two holes

The router drops it as FULL, because output 0 holds more than 8 beats. The tracker predicts that, so the `DROP_CNT` check passes. But Lab 02's scoreboard expects every packet for an enabled route with `PORT` 0 to come out of output 0, so it reports the packet missing, even on the golden router. We tried it (a 7-beat packet, which cost an 89 s rebuild), and `router_fill_test` printed

```
[DROP] DROP_CNT 00000101 as expected
[935000]       %Error:       scoreboard.svh:83:       Assertion       failed       in
```

```
router_tb_pkg.scoreboard.report.unnamedblk3: out0 never delivered dst 0, 7 beats: 05
97 b0 de 57 f7 62, which went in at 335 ns
```

To check FULL drops, the scoreboard has to model the levels itself, which is what Lab 04′s scoreboard does. In this lab, don't send a packet for output 0 while output 0 holds more than 8 beats.

A8. Part 4: Close the two holes

The packet for DST 1 would hit `cp_level.at_9` and `noroute_x_at_9` if the random packet for DST 0 before it were 1 beat long, so that output 0 held 9 beats. A packet for a disabled route would hit `noroute_x_empty`, `noroute_x_low` or `noroute_x_at_8` if it arrived while output 0 held 0, 1 to 7 or 8 beats. Our packet for DST 1 arrived at a level of 11, so it hit `noroute_x_high`.

A9. Part 4: Close the two holes

The edge at 295 ns. In the cycle that edge ends, `level0` is 08, because the four beats of `00 7a a9 ad`, the three of `01 f9 0c` and the one of `02` have gone in, and none has left. The specification admits a packet at a level of 8 or less, so the golden router writes `03` into output 0 at that edge, and `level0` reads 09 in the next cycle. B08 drops the packet, so `level0` stays at 08. Before Part 4 no header arrived at exactly 8, so the two routers did the same thing with every packet.

A10. Finishing

A pass says that the router agreed with the scoreboard and the tracker on everything the stimulus did. The empty bins showed that no header ever arrived while its output held 8 beats or more, and that the stimulus never made a NOROUTE or FULL drop. B07 and B08 sat in two of those empty bins, so the checks never had a chance to see them, and once the stimulus reached them, the same checks caught both.

A11. Finishing

No. Every seed keeps the sinks always ready, so no seed lets output 0 reach 8. More seeds help only with holes that random stimulus can reach, and Part 1′s answer explains why it can't reach these two.

IV 4. UVM

13 **The UVM skeleton 95**
13 Objects and components 95
13 run_test() and +UVM_TESTNAME 96
13 Phases 96
13 Objections 97
13 After the run phase 98
13 Check yourself 99
13 Further reading 99
13 Answers 99

14 **Sequences, sequencers and drivers . 101**
14 The pieces 101
14 One item through the handshake 101
14 Two sequences on one sequencer 103
14 Sequences that start sequences 103
14 Driving from a clock edge 104
14 Check yourself 104
14 Further reading 104
14 Answers 105

15 **Monitors, analysis ports and scoreboards 107**
15 The monitor 107
15 Analysis ports 108
15 The scoreboard 109
15 One object per write 110
15 Coverage as a subscriber 110
15 Check yourself 110
15 Further reading 110
15 Answers 111

16 **The factory and config_db 113**
16 The factory 113
16 config_db 114
16 Check yourself 115
16 Further reading 116
16 Answers 116

17 **Lab 04: The same testbench in UVM 117**
17 Before you start 117
17 Part 1: Build the port 117
17 Part 2: Watch the second output 120
17 Part 3: Stall one output at a time 121
17 Part 4: A factory override for disabled routes . 122
17 Part 5: Count what the port cost 125
17 Finishing 125
17 Answers 126

13. The UVM skeleton

This page answers: *what are the pieces of a UVM testbench, and in what order does UVM run them?*

In Lab 01 you read a whole UVM testbench, the ALU's, from its tests down to its driver, monitor and scoreboard, and you added the flag check to its scoreboard. Labs 02 and 03 used only one piece of UVM, the test. It extended `dd105_base_test` so that `make` could pick it with `+UVM_TESTNAME` and the graders could read UVM's report summary at the end of the log. Inside its `run_phase` it fetched the virtual interfaces from `config_db`, made the environment with `new()` and ran it like any other class. In Lab 04 the router testbench becomes UVM classes too, and Part 5 of that lab counts what this costs in lines against your Lab 03 testbench. Much of what it counts is the structure that this page describes. The drivers, the monitors and the scoreboard become components in a tree, UVM builds the tree and connects it in a fixed order of phases, and objections decide when the run ends.

The example for this page, `guide-examples/the-uvm-skeleton/`, has no design at all. It has a test, an environment, two agents with one child each, and a component called `mon` that loops forever the way a monitor does. Each of them prints a line in every function phase, and the test, the two children and `mon` also print in the run phase, so you can watch the order in which UVM calls them.

Objects and components

Every UVM class in a testbench extends either `uvm_object` or `uvm_component`. (In the library, `uvm_component` is itself built on `uvm_object`, but a testbench uses the two in different ways.)

A **`uvm_object`** is data: a packet, an APB transfer, a set of knobs for an agent. You make one whenever you need it, pass it around and drop it, and many of them come and go in a run. Lab 04's `router_pkt`, `stream_beat`, `apb_item` and `out_cfg` are objects.

A **`uvm_component`** is one of the testbench's fixed parts, such as a driver, a monitor, a scoreboard or the test. It's made once, at the start, and lasts for the whole run. Every component has a name and a parent, given to its constructor as `new(name, parent)`, and the parents link the components into a tree. Lab 04's tree looks like this:

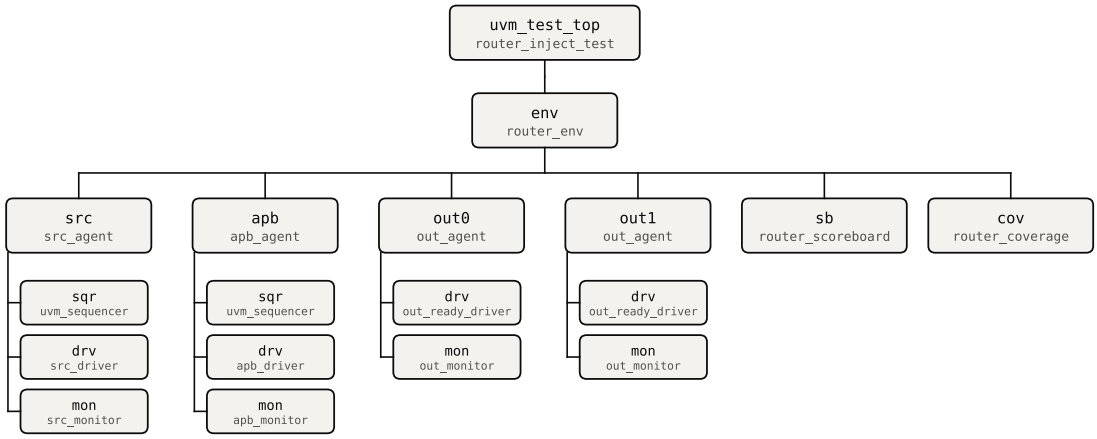


Figure 13.1: The components of Lab 04's testbench. Each box shows an instance name and, under it, the class.

The components that `router_base_test`, `router_env` and the agents build in Lab 04, once you've added `out1` in Part 2. When a component reports a message, UVM prints the component's name as a path down this tree, such as `uvm_test_top.env.out1.mon` for the monitor on output 1.

The test is at the top of everything the testbench builds, and UVM always names it `uvm_test_top`. Above it sits UVM's own root, `uvm_root`, which you'll see in a printout of the tree later on this page. The

test makes the environment, `env`, and the environment makes the agents, the scoreboard and the coverage collector. An agent holds the components for one interface: a driver that drives its pins, a monitor that watches them, and, if the agent sends items, a sequencer that hands the driver its items.

The library has a class for each of these jobs: `uvm_test`, `uvm_env`, `uvm_agent`, `uvm_driver`, `uvm_monitor`, `uvm_sequencer`, `uvm_scoreboard` and `uvm_subscriber`. Most of them add little or nothing to `uvm_component`. `uvm_driver` adds the port that it gets items through, and `uvm_subscriber` adds the port that it receives transactions on; the next two pages describe both ports. `uvm_monitor` and `uvm_scoreboard` add nothing, and each is a plain component whose name says what it's for. In Lab 04′s output agent, `out_ready_driver` takes no items, so it extends `uvm_component` rather than `uvm_driver`.

Every UVM class also has a registration line, such as `` `uvm_object_utils(router_pkt) `` or `` `uvm_component_utils(router_env) ``, which tells UVM's factory that the class exists, and the components are made with `router_env::type_id::create("env", this)` instead of `new()`. The factory page explains what the factory does with them. For now, the registration is the reason that `+UVM_TESTNAME` can name a test that no other code in the testbench refers to.

run_test() and +UVM_TESTNAME

Every lab's top module makes the clock, the design and the interfaces, puts each interface into `config_db` for the components that will need it (in Labs 02 and 03, the test), and ends by calling `run_test()`. `run_test()` reads `+UVM_TESTNAME`, asks the factory for a component of that class, names it `uvm_test_top`, and starts the phases. That's how one build serves every test in a lab.

If the name doesn't match a registered class, no test is built. `make run UVM_TESTNAME=sk_phase_tset`, with two letters of the name swapped, prints these two lines among its other output:

```
UVM_WARNING @ 0: reporter [BDTYP] Cannot create a component of type 'sk_phase_tset'
because it is not registered with the factory.
UVM_FATAL @ 0: reporter [INVTST] Requested test from command line
+UVM_TESTNAME=sk_phase_tset not found.
```

and the simulator exits with status 0, which the last section of this page comes back to.

Phases

Instead of running each component as soon as it's made, UVM calls a fixed list of methods, the **phases**, on every component in the tree, and finishes each phase everywhere before it starts the next. A component overrides the phases it needs. The ones you'll write are:

Phase	Kind	Order in the tree	What goes in it
`build_phase`	function	top down	make the children; read configuration
`connect_phase`	function	bottom up	connect ports between children
`end_of_elaboration_phase`	function	bottom up	print or check the finished tree
`start_of_simulation_phase`	function	bottom up	last settings before time moves
`run_phase`	task	all at once	everything that takes simulated time
`extract_phase`, `check_phase`, `report_phase`	function	bottom up	collect results, look for leftovers, print summaries
`final_phase`	function	top down	the very end

The function phases take no simulated time. The run phase is a task, and UVM starts every component's `run_phase` at the same moment, each in its own thread. Here is the start of `sk_phase_test`'s output:

```
0    ns  build_phase               uvm_test_top
0    ns  build_phase               uvm_test_top.env
0    ns  build_phase               uvm_test_top.env.a0
0    ns  build_phase               uvm_test_top.env.a0.child
0    ns  build_phase               uvm_test_top.env.a1
0    ns  build_phase               uvm_test_top.env.a1.child
0    ns  build_phase               uvm_test_top.env.mon
0    ns  connect_phase             uvm_test_top.env.a0.child
0    ns  connect_phase             uvm_test_top.env.a0
0    ns  connect_phase             uvm_test_top.env.a1.child
0    ns  connect_phase             uvm_test_top.env.a1
0    ns  connect_phase             uvm_test_top.env.mon
0    ns  connect_phase             uvm_test_top.env
0    ns  connect_phase             uvm_test_top
```

The build phase has to go from the top down, because a component's children don't exist until its own `build_phase` makes them, so UVM can't call `env`'s `build_phase` until the test's `build_phase` has created `env`. Within each parent, the course's UVM library visited the children in name order (`a0`, `a1`, `mon`), although `env` made them in the order `mon`, `a1`, `a0`.

The connect phase goes from the bottom up, so by the time `env` connects its agents' monitors to the scoreboard, every agent has already connected its own pieces. A component connects ports in `connect_phase` and never in `build_phase`, because the component on the other end may not exist yet during `build_phase`. UVM finishes `build_phase` in the whole tree before any `connect_phase` starts, so every component exists by the time it connects.

In `end_of_elaboration_phase` the test asks UVM to print the tree it has built:

```
Name            Type           Size  Value
-----------------------------------------
<unnamed>       uvm_root       -     @64
  uvm_test_top  sk_phase_test  -     @150
    env         sk_env         -     @220
      a0        sk_agent       -     @244
        child   sk_leaf        -     @252
      a1        sk_agent       -     @236
        child   sk_leaf        -     @260
      mon       sk_watcher     -     @228
```

Objections

A driver loops forever asking for items, and a monitor loops forever watching pins, so the run phase can't wait for every `run_phase` task to return. Instead, UVM keeps the run phase going while any component holds an **objection**. A component that is in the middle of something calls `phase.raise_objection(this)`, and calls `phase.drop_objection(this)` when it's done, and the run phase ends once every objection has been dropped. A component can also set a **drain time**, which keeps the phase open for that long after the last objection drops, so that anything still in flight can arrive.

In `sk_phase_test` the test holds an objection for 100 ns and sets a drain time of 20 ns, `env.a1.child` holds one for 150 ns, and `env.mon` loops forever, printing every 40 ns:

```
0    ns  run_phase starts          uvm_test_top.env.a0.child
0    ns  run_phase starts          uvm_test_top.env.a1.child
0    ns  run_phase: still looping  uvm_test_top.env.mon
0    ns  run_phase starts          uvm_test_top
40   ns  run_phase: still looping  uvm_test_top.env.mon
80   ns  run_phase: still looping  uvm_test_top.env.mon
100  ns  drops after 100 ns        uvm_test_top
120  ns  run_phase: still looping  uvm_test_top.env.mon
150  ns  drops after 150 ns        uvm_test_top.env.a1.child
160  ns  run_phase: still looping  uvm_test_top.env.mon
170  ns  extract_phase             uvm_test_top.env.a0.child
```

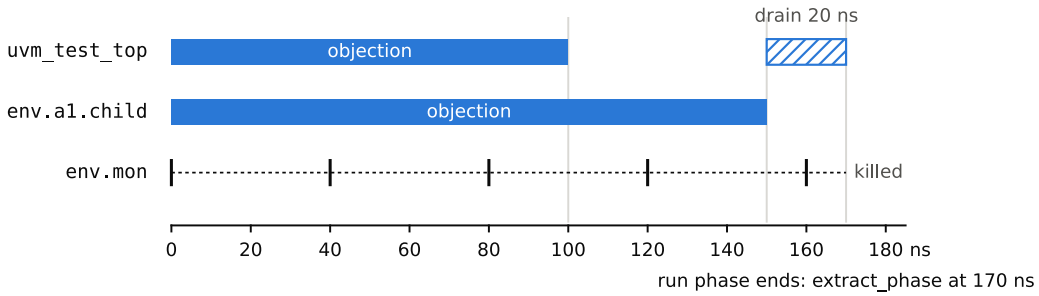


Figure 13.2: Objections in sk_phase_test: the test's from 0 to 100 ns, a1.child's from 0 to 150 ns, then 20 ns of drain time; the run phase ends at 170 ns.

The run phase of `sk_phase_test`. When the test dropped its objection at 100 ns, `a1.child` still held one. The phase ended 20 ns after that one dropped, at 170 ns, and UVM killed `mon`'s loop, which had been running the whole time without holding the phase open.

`sk_noobj_test` builds the same environment and raises no objection at all. Every `run_phase` still starts at 0 ns, and the phase ends in the same time step:

```
0    ns  run_phase starts           uvm_test_top
0    ns  extract_phase              uvm_test_top.env.a0.child
```

A new UVM test can pass this way without testing anything. If it starts its sequence in `run_phase` and forgets to raise an objection, the run ends before the first item reaches the driver. Lab 04's `router_base_test` raises its objection at the start of `run_phase`, runs the test's `body()` and drops the objection when `body()` returns. The tests built on it override `body()` and leave `run_phase` alone, so none of them has to raise an objection itself. `router_base_test` also runs a watchdog beside `body()`, because a test that holds its objection forever never ends on its own either. If the test is still running after `+MAX_CYCLES` clock cycles, the watchdog reports a `UVM_ERROR`, and `run_phase` stops `body()` and drops the objection, so the run ends.

After the run phase

Checks that need the whole run go in the three function phases that follow the run phase, `extract_phase`, `check_phase` and `report_phase`. Lab 04's scoreboard uses `check_phase` to report, for each output, how many beats it expected and never saw, and the first of them. It uses `report_phase` to print the `[SB]` lines you'll read in every run. After `final_phase`, UVM prints its report summary, which counts the run's messages by severity and by ID.

Verilator ends a UVM run with exit status 0 even when the summary counts errors, so a script that trusts the exit status would call a failing test a pass. `dd105_base_test`, which every test in the course extends, fixes that in its `final_phase`, and in `pre_abort` when a `UVM_FATAL` ends the run early. If UVM counted no error or fatal, it prints `DD105: TEST PASSED`. Otherwise it prints the report summary itself, then `DD105: TEST FAILED`, and calls `$fatal`. `sk_error_test` reports one `UVM_ERROR` at 50 ns:

```
UVM_ERROR guide-examples/the-uvm-skeleton/sk_pkg.sv(186) @ 50000: uvm_test_top [CHECK]
a check failed at 50 ns
```

and, after the report summary, prints

```
DD105: TEST FAILED (1 UVM_ERROR, 0 UVM_FATAL)
[150000]       %Fatal:       dd105_pkg.sv:89:       Assertion       failed       in
dd105_pkg.dd105_base_test.m_end_of_test: DD105: UVM counted errors, so the exit status
is nonzero
```

and the simulator exits with status 1. The `UVM_ERROR` line has the same shape as the messages that the labs' testbenches report: the severity, the file and line that reported it, `@` and the time, the full name of the component, the message's ID in brackets, and the text. Some of the UVM library's own messages leave out the file and line, as the `[BDTYP]` and `[INVTST]` lines earlier on this page do. The time is in picoseconds, the simulator's precision, so `@ 50000` is 50 ns.

The base test can't help with the misspelt test name. No test was built, so the base test's `final_phase` and `pre_abort` never ran, and that run exited with status 0. That's why the course's `make run` reads the log as well as the exit status. It reported that run as `BROKEN: UVM_FATAL [INVTST]`.

Check yourself

1. Why can't a scoreboard connect a monitor's analysis port to itself in its `build_phase`?

   Answer: A1, in Answers at the end.

2. A test's `run_phase` starts a sequence of 200 packets on the input agent and doesn't raise an objection. What happens, and what would the scoreboard's `check_phase` see?

   Answer: A2, in Answers at the end.

3. A run's log ends with a `UVM_FATAL`, and the simulator exited with status 0. Was the test built?

   Answer: A3, in Answers at the end.

Further reading

- ChipVerify's Build Your First UVM Testbench walkthrough builds one testbench for a small register file over 17 pages, one component per page, and assembles it on the last, Full UVM Testbench. As you read it, note that its testbench top module (UVM Testbench Top) never drives reset, which only the last page fixes. Its coverage collector (UVM Subscriber) has coverpoints that read a class member, so Verilator 5.050 drops the whole covergroup with a `COVERIGN` warning (see Functional coverage you can read).
- UVM Phases and UVM Objection cover every phase, including the twelve run-time sub-phases that this course doesn't use, and objections raised from sequences.
- UVM Component and UVM Object.

Next: Sequences, sequencers and drivers.

Answers

A1. Check yourself

The monitor may not exist yet. It's made in its agent's `build_phase`, and the agent's `build_phase` can run after the scoreboard's. In Lab 04, `env`'s children are built in name order, as `a0`, `a1` and `mon` were above, so `sb` is built before `src`, and `src.mon` doesn't exist yet when `sb`'s `build_phase` runs. By `connect_phase` every component has been built.

A2. Check yourself

The run phase ends at time 0, as `sk_noobj_test`'s did, before any packet reaches the pins. The scoreboard expected nothing and saw nothing, so `check_phase` finds no leftover beats. A scoreboard that also counts what it checked can catch this, as the one in Monitors, analysis ports and scoreboards does by reporting an error when it checked nothing.

A3. Check yourself

No. Every test in the course extends `dd105_base_test`, which turns a `UVM_FATAL` into a nonzero exit status in `pre_abort`, so a fatal with status 0 means that no test was built, as happens with a misspelt `UVM_TESTNAME`. `make tests` lists the names the lab knows.

14. Sequences, sequencers and drivers

This page answers: *how does a transaction get from the code that makes it to the design's pins?*

In Labs 02 and 03 the generator made packets and put them into a mailbox, and the driver took them out one at a time and drove them. In Lab 04 a **sequence** makes the items, a **sequencer** passes them on, and a **driver** puts each one on the pins. The table in Lab 04's Part 5 lists what changed. The sequence calls `start_item` and `finish_item` where the generator called the mailbox `put`, and the driver calls `get_next_item` and `item_done` where it called the mailbox `get`. Everything else the driver does is the same as before, including the idle gaps it leaves before beats and its drives through the clocking block.

This page follows one item along that path. The example for it drives SimProc's ALU from Lab 01 and prints a line, with the time, at every step. Its files are in `guide-examples/sequences-sequencers-and-drivers/`.

The pieces

The item is an object with `rand` fields, and its class extends `uvm_sequence_item`. Lab 04's item is `router_pkt`, and the example's is `alu_item`, which holds an operation and two operands.

A sequence is an object too, not a component. Its class extends `uvm_sequence #(alu_item)`, and its `body()` task makes the items and sends them. A test makes one during its run phase, when it needs it, sets its knobs and calls `seq.start(sequencer)`. `start()` returns when `body()` has finished.

The sequencer is a component in the agent. It sits between any number of sequences and one driver, and it hands the driver one item at a time. You hardly ever write a sequencer class yourself. Lab 04 has two sequencers, in the input agent and the APB agent, and it declares each of them with a `typedef`:

```
typedef uvm_sequencer #(router_pkt) src_sequencer;
```

The driver is a component too. Lab 04's `src_driver` extends `uvm_driver #(router_pkt)`, and that base class gives it a port called `seq_item_port`, which the agent connects to the sequencer in `connect_phase`:

```
drv.seq_item_port.connect(sqr.seq_item_export);
```

The port is on the driver because the driver asks for each item when it's ready for one, and the sequencer hands items over only when asked. That lets the driver set the pace. It takes the next item only when the pins can take it, and the sequence waits until then.

One item through the handshake

Here are the loops in the example's sequence and driver:

```
// alu_rand_seq::body()
req = alu_item::type_id::create("req");
start_item(req);                      // wait until the driver asks for an item
if (!req.randomize()) `uvm_fatal("RAND", "randomize() failed")
finish_item(req);                     // hand it over; wait until the driver is done

// alu_driver::run_phase()
forever begin
  seq_item_port.get_next_item(req);   // wait for an item
  @(vif.drv_cb);                      // drive from a clock edge
  vif.drv_cb.valid <= 1'b1;
  vif.drv_cb.ALUop <= req.op;
  vif.drv_cb.A     <= req.A;
  vif.drv_cb.B     <= req.B;
  @(vif.drv_cb);                      // the vector holds for one cycle
  vif.drv_cb.valid <= 1'b0;
```

```
  seq_item_port.item_done();          // let finish_item() return
end
```

(The files in the example also print a line at each step, and they tag each item with a short label for the sequence that sent it, which is `a` here.) `sq_timeline_test` runs this sequence for three items. The clock has a 10 ns period, and its first rising edge is at 5 ns. The first two items of the run print these lines:

```
0    ns  [drv]    get_next_item: waiting for an item
0    ns  [seq a]  start_item(req): waiting for the sequencer
0    ns  [seq a]  start_item returned; randomize()
0    ns  [drv]    get_next_item returned a: OR A=b0 B=b2
15   ns  [drv]    item_done, ALUout was b2
15   ns  [drv]    get_next_item: waiting for an item
15   ns  [seq a]  finish_item returned: a: OR A=b0 B=b2 is done
15   ns  [seq a]  start_item(req): waiting for the sequencer
15   ns  [seq a]  start_item returned; randomize()
15   ns  [drv]    get_next_item returned a: SHL A=c9 B=02
35   ns  [drv]    item_done, ALUout was 24
```

`start_item()` returns once the sequencer has picked this sequence and the driver is waiting for an item. At time 0 the driver was already waiting, so `start_item()` returned at once. The sequence then randomizes the item and calls `finish_item()`, which hands the item over and blocks. The driver's `get_next_item()` returns with the item in the same time step. The driver waits for the clock edge at 5 ns and drives the vector through its clocking block, and the vector reaches the pins 1 ns later, because the block's output skew is 1 ns. The driver holds the vector through the edge at 15 ns and then calls `item_done()`, and that is when `finish_item()` returns. So the sequence's loop runs at the driver's pace, which here is one item every 20 ns. This timing diagram shows the same two items:

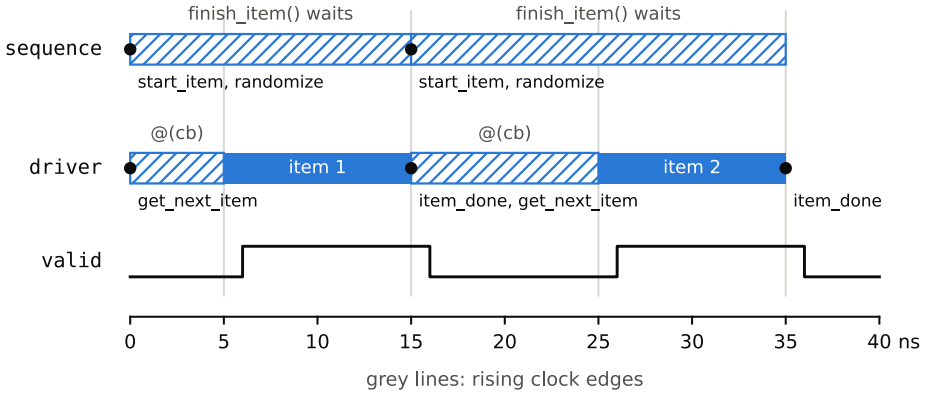


Figure 14.1: The first two items of sq_timeline_test: the sequence waits in finish_item from 0 to 15 ns while the driver waits for the edge at 5 ns and holds the vector until 15 ns.

The handshake for the first two items of `sq_timeline_test`. The sequence spends most of its time blocked in `finish_item()`, and the driver spends most of its time waiting on clock edges.

On the ALU's pins, recorded with `make waves`, the run looks like this:



Figure 14.2: sq_timeline_test on the ALU in Surfer, 0 to 55 ns: three vectors, each with valid high for one cycle.

The three vectors on the ALU's pins. Each one goes on 1 ns after a rising edge and holds for one cycle, and `ALUout` shows the ALU's answer (`b2`, `24`, `ff`). Between two vectors `valid` is low for one cycle. The driver drives `valid <= 0` right after the edge that ends a vector, before it asks for the next item, and it drives the next vector only after the edge that follows.

Because the sequence randomizes the item after `start_item()` returns, the item gets its values when the driver is ready for it rather than when the sequence made it, and the constraints can use anything that the sequence has set by then. Lab 04's `router_rand_seq` copies `route_en` into each packet before it randomizes the packet. `route_en` holds the enable bits of the routing table that the test wrote for the current phase, and the packet's `c_dst` constraint uses those bits to keep the packet on an enabled route.

Two sequences on one sequencer

A sequencer can serve several sequences at once, so one test can run background traffic and a directed burst on the same agent at the same time. `sq_two_test` starts two sequences of two items each, `a` and `b`, in a `fork`. These are the lines it prints, with some of the driver's left out:

```
0    ns  [seq a]  start_item(req): waiting for the sequencer
0    ns  [seq b]  start_item(req): waiting for the sequencer
0    ns  [seq a]  start_item returned; randomize()
0    ns  [drv]    get_next_item returned a: SUB A=05 B=80
15   ns  [seq a]  finish_item returned: a: SUB A=05 B=80 is done
15   ns  [seq a]  start_item(req): waiting for the sequencer
15   ns  [seq b]  start_item returned; randomize()
15   ns  [drv]    get_next_item returned b: OR A=cc B=ea
35   ns  [seq b]  finish_item returned: b: OR A=cc B=ea is done
35   ns  [seq b]  start_item(req): waiting for the sequencer
35   ns  [seq a]  start_item returned; randomize()
35   ns  [drv]    get_next_item returned a: SHR A=79 B=bd
```

The items alternate between the two sequences. The sequencer's default arbitration grants requests in the order they arrived, and each sequence asks again only after its last item is done. So whenever an item finishes, the other sequence's request has been waiting longer than the new one from the sequence that just finished, and the other sequence goes next. A sequence that you start on a sequencer that's already busy therefore takes turns with the one running there, item by item. Each of Lab 05's sinks has a sequence that its agent starts and that runs for the whole test, which is why Lab 05's virtual sequence steers a sink's backpressure by changing its `out_cfg` rather than by starting another sequence on the sink's sequencer.

Sequences that start sequences

A sequence's `body()` can start other sequences on the same sequencer, so you can build a larger sequence out of smaller ones. The call is `child.start(m_sequencer, this)`, where `m_sequencer` is the sequencer that the parent is running on. `sq_nested_test` runs a sequence that starts two others: first one that sends two random vectors, then one that sends two directed SUBs. From the point where the random sequence finishes, the run prints these lines (one of the driver's lines is left out):

```
35   ns  [seq r]  finish_item returned: r: SHR A=db B=14 is done
35   ns  [drv]    get_next_item returned sub: SUB A=5a B=5a
55   ns  [drv]    item_done, ALUout was 00
55   ns  [seq sub] sent sub: SUB A=5a B=5a
55   ns  [drv]    get_next_item returned sub: SUB A=5a B=21
75   ns  [drv]    item_done, ALUout was 39
```

The directed sequence never calls `randomize()`. It sets `op`, `A` and `B` by hand between `start_item()` and `finish_item()`, which is the simplest way to send exactly the item you mean. On this simulator, `randomize() with { ... }` fails when its inline constraint rules out the entry that a `dist` picked, and setting the fields by hand is also how the course avoids those failures (see Constrained random on Verilator).

A test starts sequences too. Lab 04's base test writes a register by starting a one-item sequence on the APB agent's sequencer:

```
task write_reg(bit [7:0] addr, bit [31:0] data);
  apb_rw_seq s = apb_rw_seq::type_id::create("wr");
  s.write = 1;
  s.addr  = addr;
  s.wdata = data;
  s.start(env.apb.sqr);
endtask
```

`read_reg()` does the same with `write = 0`, and then reads the result out of the sequence's item. That works because the APB driver fills in the item's `rdata` before it calls `item_done()`, and `start()` doesn't return until the driver has called it.

Driving from a clock edge

Lab 04′s drivers wait for a clock edge before their first drive of an item, as the example's driver does with `@(vif.drv_cb)` straight after `get_next_item()`. They do it because of a timing problem in this simulator that turned up while the course's reference testbench was being written (`labs/ref/README.md`, "A clocking-block drive after an NBA-region wait"). `try_next_item()` doesn't return until the nonblocking assignments of its time step have been made. If the driver then issues a drive through the clocking block in that same time step, and the time step is a clock edge, the drive takes effect at the next edge instead of this one. The same thing happens to Lab 04′s drivers, which use `get_next_item()`, when the sequencer hands one of them an item in the time step of an edge and the driver drives straight away. Waiting for an edge after getting the item puts every drive at the start of a time step. Lab 01′s driver calls `try_next_item()` instead of `get_next_item()`, and it too asks for the item before it waits for the edge.

Drive through the clocking block too, as every driver on this page does. On Verilator 5.050, a module-level task that does `x <= 1; @(posedge clk); x <= 0;` loses some of its assignments when it's called in a `repeat` loop. Class drivers kept every assignment in every shape that the course tried. Use the clocking block anyway, because it gave the right result in every shape, and it doesn't depend on the order in which Verilator commits two nonblocking assignments in one time step (`labs/common/README.md`, "Drive through a clocking block").

Check yourself

1. In `sq_timeline_test`, `finish_item()` for the first item returns at 15 ns, not at 0 ns. What is it waiting for?

   Answer: A1, in Answers at the end.

2. At what time is the second item randomized, and what has to happen first?

   Answer: A2, in Answers at the end.

3. Lab 04′s `close_holes()` needs its fill packets to arrive exactly as it built them, with set lengths. Should it use `router_rand_seq` with `randomize() with { len == n; }`?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's walkthrough builds these pieces for a small register file: Defining the Transaction Class, UVM Sequence, UVM Sequencer and UVM Driver. Two of those pages have mistakes in them. The driver page's "simplified" `get_next_item()` loop never clears the request signals, and it calls `item_done()` straight after one clock wait, which is the mistake that the page's own list of common mistakes warns about ("Driving signals with only one clock wait"). The transaction page's exercise solution adds an `idle` field that can never be 1 while the original constraint stays in.

- Using get_next_item(), for the handshake in more detail, with `try_next_item()` and responses.
- How to execute sequences via start(), for the arguments of `start()` and the hooks around `body()`. Its example test starts a sequence without raising an objection, which works there only because the sequence uses no simulated time.
- UVM Sequence Arbitration, for the other ways a sequencer can choose between sequences.

- Executing Sequences with uvm_do Macros shows what the macros expand into. This course writes out `start_item` and `finish_item` instead of using the macros. The expansions on that page are from an older version of UVM than the course's library.

Next: Monitors, analysis ports and scoreboards.

Answers

A1. Check yourself

The driver's `item_done()`. The driver got the item at 0 ns, drove it after the edge at 5 ns, held it through the edge at 15 ns and then called `item_done()`, which let `finish_item()` return.

A2. Check yourself

At 15 ns. Its `start_item()` can't return until the driver asks for another item, and the driver asks only after it has driven the first item, held it through the edge at 15 ns and called `item_done()`.

A3. Check yourself

No. `router_pkt` has a `dist` on `kind`, and on this simulator an inline constraint that rules out the entry that the `dist` picked makes the call fail. `len == n` rules out two of the three kinds. So `close_holes()` makes its fill packets by hand with `router_pkt::make()`, and it sends each one with Lab 04's `router_one_seq`, a sequence that only calls `start_item()` and `finish_item()`.

15. Monitors, analysis ports and scoreboards

This page answers: *how does what a monitor sees on the pins get to the scoreboard and the coverage, and what does the scoreboard do with it?*

Lab 01′s UVM testbench for the ALU already used an analysis port. Its monitor wrote each vector to the port, and the scoreboard checked each vector in its `write` function. Lab 02′s router testbench was built without UVM, and its classes passed packets through mailboxes. Your monitor turned the handshakes on a stream into whole packets and put them in a mailbox, and the scoreboard ran two threads of its own, one for each mailbox, that took them out. Lab 03 added a tracker, with a loop of its own, that watched the pins directly.

In Lab 04 the router testbench is rewritten in UVM, and it uses **analysis ports**, as Lab 01′s did. Each of its monitors writes every beat, or every APB transfer, to its analysis port as soon as it sees it, and the scoreboard has no thread at all. It's a class with three `write` functions, and the monitors call them. This page shows how that works on the ALU, what Lab 04′s scoreboard does when a beat arrives, and what goes wrong when a monitor reuses one object for every write.

The example for this page, `guide-examples/monitors-analysis-ports-and-scoreboards/`, tests SimProc's ALU with a complete agent, made of a sequencer, the driver from the example for *Sequences, sequencers and drivers*, and a monitor. The environment connects three subscribers to the monitor's one analysis port: a scoreboard, a coverage collector and a history that keeps every item it's given. The ALU is the one from Lab 01, and its bug switch works here too. You set it when you run the simulator, so `make run BUG=A01` runs the same test, on the same build, with the ALU whose `SUB` computes `B - A`.

The monitor

A monitor reads the pins and reports what it saw. It doesn't drive anything, and it doesn't know what the driver sent or what the answer should be, so the same monitor can feed a scoreboard, a coverage collector and anything else you add later. The example's monitor reads the pins through a clocking block that has only inputs, so it sees each signal as it was just before the clock edge, which is the value the signal held during the cycle that the edge ends:

```
virtual function alu_item next_item();
  return alu_item::type_id::create("seen");     // a new item every time
endfunction

task run_phase(uvm_phase phase);
  alu_item t;
  forever begin
    @(vif.mon_cb);
    if (vif.mon_cb.valid) begin
      t        = next_item();
      t.op     = alu_op_e'(vif.mon_cb.ALUop);
      t.A      = vif.mon_cb.A;
      t.B      = vif.mon_cb.B;
      t.ALUout = vif.mon_cb.ALUout;
      t.N      = vif.mon_cb.N;
      t.Z      = vif.mon_cb.Z;
      n_seen++;
      ap.write(t);
    end
  end
endtask
```

(`next_item()` is a separate function only so that a test can swap in a monitor that gets its items some other way, as `mon_alias_test` does in "One object per write", below.)

Lab 04′s monitors do the same for the router's streams, with one `stream_beat` per handshake, stamped with its cycle number, and for its APB port, with one `apb_item` per completed transfer.

Analysis ports

`ap` is a `uvm_analysis_port #(alu_item)`. Its `write(t)` is a function, so it can't wait, and it calls the `write` function of every subscriber connected to the port, one after another, before it returns. A port with no subscribers is allowed, and `write` then does nothing. The monitor can't tell who, if anyone, is connected to its port. The environment makes the connections, in its `connect_phase`:

```
agent.mon.ap.connect(sb.imp);
agent.mon.ap.connect(cov.analysis_export);
agent.mon.ap.connect(hist.analysis_export);
```

On the receiving end, the example's scoreboard declares an imp, `uvm_analysis_imp #(alu_item, alu_scoreboard)`, and the imp calls the scoreboard's own `write(alu_item t)`. A class that extends `uvm_subscriber #(alu_item)` gets its receiving end already built, as a member called `analysis_export`, and you write only its `write` function. The example's coverage collector and history both extend `uvm_subscriber`.

A scoreboard with several inputs needs a separate `write` function for each one, but an imp can call only a function named `write`. The macro `` `uvm_analysis_imp_decl(_in) `` declares a new imp class, `uvm_analysis_imp_in`, that calls `write_in()` instead. Lab 04′s `router_env_pkg` declares three of these classes at package scope, for the input stream, the outputs and the APB port, and its scoreboard has one imp of each. The figure shows which monitors write to each one.

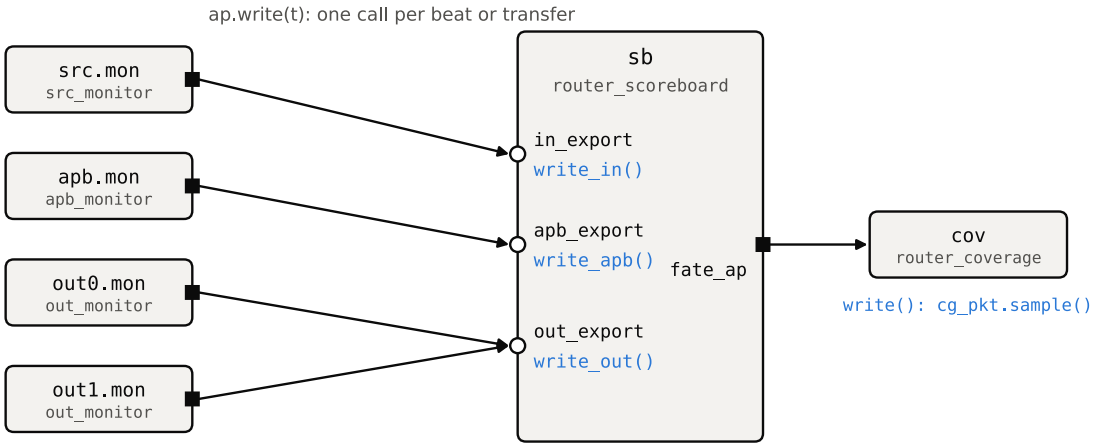


Figure 15.1: Lab 04′s analysis connections: four monitors write to three exports on the scoreboard, and the scoreboard's own analysis port feeds the coverage collector.

The connections that Lab 04′s `router_env::connect_phase` makes once you've done Part 2, which adds output 1′s agent. Before Part 2, only output 0′s monitor is connected. Both output monitors write to the same `out_export`, and each `stream_beat` carries its port number, so `write_out()` can tell which output a beat came from. The scoreboard decides what happens to each packet and writes a `pkt_fate` record for it to its own analysis port, `fate_ap`, which the coverage collector subscribes to.

Here are the report lines from one run of `mon_rand_test`, which sends 200 vectors. The history kept 200 items and the scoreboard checked 200, the same number that the monitor wrote:

```
UVM_INFO   guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(238)   @
3995000: uvm_test_top.env.cov [COV] cg: 100.0% of its bins hit
UVM_INFO   guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(260)   @
3995000: uvm_test_top.env.hist [HIST] kept 200 handles to 200 different objects
```

```
UVM_INFO   guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(217)   @
3995000: uvm_test_top.env.sb [SB] 200 vectors checked: 0 wrong
UVM_INFO   guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(313)   @
3995000: uvm_test_top [MON] the monitor wrote 200 items
```

The scoreboard

A scoreboard predicts what the design should have done and compares that with what the monitors saw. The example's scoreboard does all its work inside `write`. It gets the expected answer from `predict()`, a model of the ALU written from SimProc's instruction set, as Lab 01's scoreboard does, and compares it with the item:

```
virtual function void write(alu_item t);
  bit [7:0] want = predict(t);
  bit       bad  = (t.ALUout != want);
  if (check_flags) bad |= (t.N != want[7]) || (t.Z != (want == 8'h00));
  n_checked++;
  if (bad) begin
    n_bad++;
          `uvm_error("SB",  $sformatf("%s,  expected  ALUout=%02h  N=%0b  Z=%0b",
t.convert2string(),
              want, want[7], want == 8'h00))
  end
endfunction
```

On the ALU with bug A01 the run reports 33 errors, and this is the first one:

```
UVM_ERROR  guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(207)  @
35000: uvm_test_top.env.sb [SB] SUB A=1f B=34 -> ALUout=15 N=0 Z=0, expected ALUout=eb
N=1 Z=0
```

In `check_phase` the scoreboard also makes sure that it checked at least one vector, because a scoreboard that was never called would report no errors either.

Because `write` is a function, a scoreboard can't wait inside it for the design's answer. The ALU is combinational, so the monitor reads its inputs and its result in the same cycle, and they arrive in the same item. In the router, a packet goes in on one stream and comes out on another several cycles later, so Lab 04's scoreboard splits the work between its write functions. When a packet's header comes in, `write_in()` decides what the router must do with the packet: forward it to output 0 or 1, or drop it as NOROUTE or FULL. It decides from the routing table, as it has seen it written over APB, and from the level of each output. If the router should forward the packet, `write_in()` queues each beat that it expects to come out of that output. `write_out()` takes the next expected beat off the queue for the output that the beat came from, and compares the two. `write_apb()` keeps the scoreboard's copy of the routing table up to date, and checks every read of `DROP_CNT` against the drops the scoreboard has counted. At the end of the test, `check_phase` reports any beats still left in the queues, because those are beats the router never delivered. (As shipped, `write_in()` queues only output 0's beats, and `level()` takes output 1's level as 0. Parts 2 and 3 of Lab 04 have you change those.)

A scoreboard catches only the bugs that show up in what it compares. With `check_flags` off, which is the default, the example's scoreboard compares `ALUout` and nothing else. Against Lab 01's three bugs, `make check-all` gives:

```
  caught  tier 1  A01_sub_operands_swapped        33 UVM_ERROR                  0.2s
  caught  tier 1  A02_nand_is_nor                 26 UVM_ERROR                  0.2s
  MISSED  tier 1  A03_zero_flag_low_nibble        0 UVM_ERROR, 0 UVM_FATAL      0.2s
```

A03 makes the zero flag look only at the low four bits of `ALUout`. The monitor put every wrong `Z` into an item, but no subscriber compared `Z` with anything. The next page, The factory and config_db, turns `check_flags` on from a test, without editing the scoreboard.

One object per write

`ap.write(t)` passes the same handle to every subscriber, so they all see one object. That's cheap, and it's safe as long as no code changes the object after the write. The example's monitor makes a new `alu_item` for every vector. `mon_alias_test` swaps in a monitor that makes one item and fills it in again for every vector. (The swap is a factory override, which the next page explains.) The history subscriber keeps every handle it's given, and at the end of that test it prints this:

```
UVM_INFO   guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(260)   @
3995000: uvm_test_top.env.hist [HIST] kept 200 handles to 1 different objects
UVM_INFO   guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(262)   @
3995000: uvm_test_top.env.hist [HIST] vector 0: ADD A=c7 B=91 -> ALUout=58 N=0 Z=0
UVM_INFO   guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(262)   @
3995000: uvm_test_top.env.hist [HIST] vector 1: ADD A=c7 B=91 -> ALUout=58 N=0 Z=0
UVM_INFO   guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(262)   @
3995000: uvm_test_top.env.hist [HIST] vector 2: ADD A=c7 B=91 -> ALUout=58 N=0 Z=0
UVM_INFO   guide-examples/monitors-analysis-ports-and-scoreboards/mon_pkg.sv(217)   @
3995000: uvm_test_top.env.sb [SB] 200 vectors checked: 0 wrong
```

All 200 handles point at one object, which holds the last vector. In `mon_rand_test` the first three vectors were an `OR`, a `SUB` and a `SHL`. The scoreboard still passed, because it finishes with each item inside `write`, before the monitor fills the object in with the next vector. A subscriber that keeps the handle for later, or queues it to compare when the answer arrives, sees the wrong values. So a monitor should make a new object for every `write`. A subscriber that keeps anything should either copy the values it needs, as Lab 04′s scoreboard does when it queues `{last, data}` for each expected beat, or keep the handle and never change the object.

Coverage as a subscriber

A coverage collector is the simplest kind of subscriber, because its `write` only samples a covergroup. On Verilator 5.050 the covergroup has to take its values as arguments, as in `covergroup cg with function sample(int op, bit eq)`. Verilator leaves out a covergroup whose coverpoints read a member of the class, and prints only a warning. Functional coverage you can read shows the warning.

What a collector subscribes to depends on what it covers. The example covers what went into the ALU, so it subscribes to the monitor. Lab 04′s `router_coverage` covers each packet's fate and the level of its route's output when its header came in. Only the scoreboard works those out, so `router_coverage` subscribes to the scoreboard's `fate_ap`.

Check yourself

1. `write()` on an analysis port is a function. What does that rule out for a scoreboard's `write`, and how does Lab 04′s scoreboard live with it?

   Answer: A1, in Answers at the end.

2. In `mon_alias_test` the scoreboard passed and the history was wrong. Would a scoreboard that pushed each input item onto a queue and compared it when an output arrived have passed?

   Answer: A2, in Answers at the end.

3. Two monitors write `stream_beat`s to the same `out_export`. How does `write_out()` tell output 0′s beats from output 1′s?

   Answer: A3, in Answers at the end.

Further reading

- ChipVerify's UVM Monitor builds a monitor for a register file. Correction: it says a monitor that didn't get its virtual interface can keep running after a `uvm_error`. It can't keep running, because its `run_phase` uses the interface straight away, so use `uvm_fatal` there, as its driver page does. It also sets two knobs through `config_db` that the monitor never reads.

- UVM TLM Analysis Port, with a broadcast to three subscribers. Its log comes from UVM 1.1d, which is older than the UVM library the course uses.
- Using _decl macro in TLM builds a two-input scoreboard. Correction: it says the macro can't go inside a class because SystemVerilog doesn't allow a class inside another class. Nested classes are legal, but package scope, where Lab 04 puts the macro, is still the right place for it.
- UVM Scoreboard and UVM Scoreboard Architecture, for in-order and out-of-order comparison. The second page recommends checking at the end of the test that nothing is left waiting for a match and that the match count isn't zero. Neither of its examples checks the match count, and the in-order one doesn't check for leftovers either.
- UVM Subscriber. Its covergroup reads a member of the class, so Verilator 5.050 would leave it out. Write yours `with function sample`.

Next: The factory and config_db.

Answers

A1. Check yourself

The scoreboard's `write` can't wait for anything, so it can't use `@`, `#` or a blocking `get`. The router's answer comes out cycles after the packet goes in, so `write_in()` predicts and queues the expected beats, and `write_out()` compares each beat when an output monitor reports it. The scoreboard doesn't need a thread of its own for any of that.

A2. Check yourself

No. Every handle on its queue would point at the one object, which by the time an output arrives holds a later vector, so the scoreboard would compare the wrong inputs with each output. The example's scoreboard passed only because it finished with each item inside `write`.

A3. Check yourself

Each output monitor stamps `b.port` on every beat, from its `out_cfg`, and `write_out()` uses `b.port` to pick which queue of expected beats to compare with.

16. The factory and config_db

This page answers: *how does a test change what the environment builds, and how it's set up, without editing the environment?*

UVM gives a test two ways to do it, the factory and `config_db`, and Lab 04 has you use both. In Part 2 you make a second copy of the output agent, and `config_db` tells each copy which output it's on. In Part 3 you write a stall test that changes a knob that the output drivers read, one output at a time. In Part 4 you write a test that sends packets to disabled routes by telling the factory to build a different packet class, and the sequence that makes the packets stays as it is. Part 5 then asks what the extra lines of the UVM testbench bought you, and these two mechanisms are most of the answer.

The example in `guide-examples/the-factory-and-config-db/` compiles the files of the monitors page example unchanged, so it has that example's environment. The environment has an agent that drives SimProc's ALU, and the agent's monitor feeds a scoreboard, a coverage collector and a history. Each of the example's four tests changes what that environment does without editing a line of it.

The factory

Every UVM class in the labs and in this example registers itself with the factory through a utils macro, `` `uvm_object_utils(alu_item) `` in an object and `` `uvm_component_utils(alu_env) `` in a component. The labs make almost everything by asking the factory for it, as the example's sequence does for each vector:

```
req = alu_item::type_id::create("req");
```

With no override set, that returns an `alu_item`, as `new()` would. But a test can tell the factory to return a subclass of `alu_item` to all the code that asks for an `alu_item`, and that code doesn't have to change.

`fac_equal_test` uses an override to give every vector equal operands. The override substitutes `alu_equal_item`, a subclass of `alu_item` that replaces one of `alu_item`'s two constraints:

```
class alu_equal_item extends alu_item;
  `uvm_object_utils(alu_equal_item)
  constraint c_same { same == 1; same -> B == A; }
  ...
endclass
```

A constraint with the same name as one in the parent class replaces it, so `alu_equal_item` has the parent's `c_op` and this `c_same`. (Verilator 5.050 does this as the standard says, and Lab 04's `router_noroute_pkt` replaces `c_dst` the same way.) The test sets the override in its `build_phase`, before `super.build_phase()` builds the environment:

```
function void build_phase(uvm_phase phase);
  alu_item::type_id::set_type_override(alu_equal_item::get_type());
  super.build_phase(phase);
endfunction
```

and later, in its `end_of_elaboration_phase`, it prints the factory's overrides with `uvm_factory::get().print(0)`:

```
#### Factory Configuration (*)
No instance overrides are registered with this factory
Type Overrides:
  Requested Type  Override Type
  --------------  --------------
  alu_item        alu_equal_item
```

The sequence still calls `alu_item::type_id::create("req")` and still randomizes what it gets, so the one override line in the test gives every one of its 200 vectors `A == B`. Run against Lab 01's three ALU bugs, the test catches none of them:

```
0 of 3 caught.
Still hiding from this test:
  A01_sub_operands_swapped
     hides behind: SUB rX, rX, the idiom for clearing a register
  A02_nand_is_nor
     hides behind: NAND rX, rX, the only way to write NOT
  A03_zero_flag_low_nibble
     hides behind: every check that compares results and not flags
```

That's DD104's lesson about equal operands again. With `A == B`, `A - B` and `B - A` are both 0, and `~(A & B)` and `~(A | B)` are both `~A`, so A01 and A02 give the right answers. The coverage collector sorts the vectors by operation and by whether the two operands are equal, and in this run it reported only 65% of its bins hit, because no vector had unequal operands.

An override changes every object the factory makes of that type, and that can include objects you meant to leave alone. Lab 04's `close_holes()` needs two packets exactly as `router_pkt` defines them, so it makes them with `new()`, which doesn't go through the factory. When we made them through the factory, `router_inject_test`'s override turned them into `router_noroute_pkt`s too, and `randomize()` failed on 4 of 5 seeds (Lab 04, Part 4, step 3).

An override can also be set from the command line instead of in a test class. UVM reads `+uvm_set_type_override=alu_item,alu_equal_item` as the same override, so it works on `mon_rand_test`, which knows nothing about `alu_equal_item`. On the ALU with bug A01, the run prints:

```
UVM_INFO @ 0: reporter [UVM_CMDLINE_PROC] Applying type override from the command
line: +uvm_set_type_override=alu_item,alu_equal_item
```

and ends with `MISSED: 0 UVM_ERROR`. The same test without the plusarg catches A01 with 33 errors. A plusarg needs no rebuild, and Lab 04 uses one the same way on the router in Part 4, step 6.

The factory can also override a single instance, named by its path in the tree, instead of every object of a type. The course's labs use only type overrides. So does `mon_alias_test` on the monitors page, which replaced the monitor class in an environment that has only one monitor, so the type override reached only that monitor.

config_db

`config_db` is a table of values that one part of the testbench puts in and a component takes out. The code that sets a value names the components that may read it, by a path in the component tree. When the first argument is `this`, the path is relative to the component that makes the call:

```
uvm_config_db#(bit)::set(this, "env.sb", "check_flags", 1);     // in the test
void'(uvm_config_db#(bit)::get(this, "", "check_flags", check_flags));   // in the
scoreboard
```

The test's full name is `uvm_test_top`, so its `set` is for `uvm_test_top.env.sb`. The scoreboard's `get` has an empty path, so it looks for a value set for the scoreboard's own full name, which is also `uvm_test_top.env.sb`. A path can contain wildcards. The top module isn't a component, so it sets each virtual interface with a `null` context and a full path, such as `"uvm_test_top.env.agent.*"`, and the `*` lets every component inside the agent find it.

The example's scoreboard checks only `ALUout` unless `check_flags` is set, and then it checks N and Z as well. `fac_flags_test` sets it, and catches all three bugs:

```
      ok  the golden design                       0 UVM_ERROR, 0 UVM_FATAL       0.2s
--------------------------------------------------------------------------------
  caught  tier 1  A01_sub_operands_swapped         33 UVM_ERROR                  0.2s
  caught  tier 1  A02_nand_is_nor                  26 UVM_ERROR                  0.2s
  caught  tier 1  A03_zero_flag_low_nibble         8 UVM_ERROR                   0.2s
--------------------------------------------------------------------------------
 3 of 3 caught.
```

`fac_typo_test` makes the same `set` for `"env.scoreboard"`, and the environment has no component by that name. The run ends with 0 warnings and 0 errors. The scoreboard's `get` finds nothing, so `check_flags` keeps its default of 0, and the test misses A03, as `mon_rand_test` does. To see what went wrong, run with `+UVM_CONFIG_DB_TRACE`, which logs every `set` and `get`:

```
UVM_INFO third_party/uvm/src/base/uvm_config_db_implementation.svh(485) @ 0: reporter
[CFGDB/SET] Configuration scope='uvm_test_top.env.scoreboard' name='check_flags' (type
PARAMTYPEDTYPE 'T') set accessor=uvm_test_top = (PARAMTYPEDTYPE 'T') 1
UVM_INFO third_party/uvm/src/base/uvm_config_db_implementation.svh(485) @ 0: reporter
[CFGDB/GET]  Configuration  scope='uvm_test_top.env.sb'  name='check_flags'  (type
PARAMTYPEDTYPE 'T') read accessor=uvm_test_top.env.sb = null (failed lookup)
```

The `set` was for `uvm_test_top.env.scoreboard`, and the `get` looked in `uvm_test_top.env.sb`. A knob with a sensible default can be read the way `check_flags` is, with `void'(...)`, but a component that can't work without a value should stop the run when the `get` fails. Every driver and monitor in Lab 04 stops the run with `` `uvm_fatal("NOVIF", ...) `` if it gets no virtual interface. The output agent's driver and monitor also need an `out_cfg`, and they stop with `` `uvm_fatal("NOCFG", ...) `` if they don't get one.

Two components can set the same value for the same component. `fac_prec_test` sets `check_flags` to 1 for the scoreboard, and it also overrides the environment with a subclass, `fac_env`, that sets `check_flags` to 0 for the same scoreboard. The environment is built after the test, so its `set` comes second:

```
[TEST] uvm_test_top set check_flags = 1 for "env.sb"
[ENV] uvm_test_top.env set check_flags = 0 for "sb"
```

The scoreboard still checked the flags, and reported `200 vectors checked with N and Z: 0 wrong`. During the build phase, a `set` made higher in the component tree wins over one made lower down, even when the lower one comes later, so the test's 1 beat the environment's 0. That rule lets an environment give its components defaults that any test can override.

Objects by handle

`config_db` can hold an object as well as a number. For an object it stores the handle, so the component that gets the value and the one that set it share one object. Lab 04 uses that for the output agents. `router_base_test` makes an `out_cfg` for each output, sets output 0's for `"env.out0*"` and output 1's for `"env.out1*"`, and keeps the two handles. Both output agents are the same class. Each one gets its virtual interface and its `out_cfg` from `config_db`, because its name matches the path that they were set for. Nothing else differs between the two. Because the test kept the handles, your Part 3 stall test can change `ocfg[p].ready_pct` in the middle of a run. Each `out_ready_driver` reads `ready_pct` again every cycle, so it follows the change at the next clock edge.

Write the type of a virtual interface out in full, as in `virtual stream_src_if`, and don't reach it through a `typedef`. On Verilator 5.050, a class in a package that reached an interface through a `typedef` of the virtual interface read 0 from every clocking-block input, with no warning, so the labs' packages don't use one. `labs/ref/README.md` has the probe that showed it.

Check yourself

1. `fac_equal_test` and `mon_rand_test` both extend `mon_base_test`, and `fac_equal_test` adds a type override. It catches none of the three bugs, and `mon_rand_test` catches two of them. Did the override change the sequence?

   Answer: A1, in Answers at the end.

2. A test sets a knob for its scoreboard, and the run is no different from a run without the `set`. Nothing printed an error. What do you look at?

   Answer: A2, in Answers at the end.

3. Lab 04's `close_holes()` makes its two random packets with `new()`. What happened in `router_inject_test` when they were made through the factory instead?

   Answer: A3, in Answers at the end.

Further reading

- ChipVerify's UVM Factory Override covers type and instance overrides, and the difference in argument order between the factory's methods and the component's. Its last example calls methods on a `factory` variable that it never declares, as if UVM had a global by that name. The course's UVM library (1800.2-2020.3.2) has none, so use `uvm_factory::get()` there.
- UVM config database, for the API, the wildcards and the precedence rules.
- uvm_config_db Examples works through four cases with `+UVM_CONFIG_DB_TRACE` logs, and has a table of nine `set()` calls and the components that each one reaches. Its third case depends on a `set()` that the page never shows.
- Configure Components, on configuration objects like Lab 04's `out_cfg`.
- Passing the Virtual Interface, on how the top module's `set` reaches the driver and the monitor.

Next: Lab 04: The same testbench in UVM.

Answers

A1. Check yourself

No. The sequence's code is the same, and it still asks the factory for an `alu_item` and randomizes it. The factory hands back an `alu_equal_item` instead, and that class's `c_same` decides the operands. Every item the sequence made had `A == B`, which hides A01 and A02, and the scoreboard didn't check the flags, so it missed A03 as it does in `mon_rand_test`.

A2. Check yourself

Run it with `+UVM_CONFIG_DB_TRACE` and compare the scope in the `CFGDB/SET` line with the scope in the scoreboard's `CFGDB/GET` line. If the `get` says `failed lookup`, the `set` didn't name the component that reads the value, as in `fac_typo_test`. Without the trace, the run's log says nothing about it.

A3. Check yourself

The test's override reached them too, so both were `router_noroute_pkt`s. `close_holes()` switches `c_dst` off for the packet to DST 1 through a `router_pkt` handle, and that left the subclass's own `c_dst` on. Whenever the packet's `pre_randomize()` decided against a disabled route, that `c_dst` required an enabled one. DST 1 was disabled, so `randomize()` failed, which happened on 4 of 5 seeds (Lab 04, Part 4, step 3).

17. Lab 04: The same testbench in UVM

In this lab you'll finish our port of the Lab 03 testbench to UVM. We wrote it from our own solution to Lab 03. As shipped, it watches only output 0, as Lab 03's testbench did, and it misses B06 and B13. You'll add a second `out_agent`, so that the testbench watches output 1, and change the scoreboard so that it checks output 1's beats. Then you'll write a test that stalls one output at a time, and make the scoreboard track output 1's level, which the new test needs. After that you'll write a packet class that goes to a disabled route one time in four, and a factory override that puts it in place of `router_pkt` without any edit to the sequence that makes the packets. With those pieces in place the testbench catches both bugs. At the end you'll count the lines in the port and in your Lab 03 testbench, and work out what the extra lines bought you.

Package: `labs/lab-04-the-same-testbench-in-uvm/`

Before you start

- Finish Lab 02 and Lab 03 on the Course path. Keep your Lab 03 files, because Part 5 counts them.
- Read The UVM skeleton, Sequences, sequencers and drivers, Monitors, analysis ports and scoreboards and The factory and config_db.
- Keep the router's specification open, or the router in one page. Parts 3 and 4 use the specification's Routing section, which the one-page version covers under Packets and routing.
- Copy this table into your notes and fill it in as you go:

	Build time	`make check-all`
The port as shipped (Part 1)		
After Part 2		
After Part 3		
After Part 4		

Time: about 2 h 30 min. On the course's 2 CPUs the first build took us 116 s, and the rebuilds after each part's edits took 30 s and 31 s in Part 2, 105 s and 32 s in Part 3, and 105 s in Part 4.

Part 1: Build the port

Does the port do what the Lab 03 testbench did, and which of this lab's bugs does it miss?

1. Go to the lab folder and start the build, because it's the slowest step in the lab:

```
cd labs/lab-04-the-same-testbench-in-uvm
make build
```

The banner says a cold build takes 1.5 to 2.5 minutes on 2 CPUs, and ours took 116 s. It ends with:

```
- V e r i l a t i o n   R e p o r t: Verilator 5.050 2026-07-01 rev v5.050
- Verilator: Built from 48.745 MB sources in 415 modules, into 26.168 MB in 2960 C+
+ files needing 156.586 MB
- Verilator: Walltime 116.774 s (elab=0.822, cvt=3.289, bld=111.863); cpu 5.279 s
on 2 threads; allocated 339.555 MB
```

and then a line saying it built `build/sim` in 116 s, and a ccache line. Our ccache was empty (`Hits: 0 / 38`). Yours already holds the builds of Labs 01 to 03, so the counts on your ccache line will be higher. Write your build time in your table.

2. While it builds, open the files in `tb/`. Apart from the top module, `router_tb_top.sv`, every piece of the Lab 03 testbench is in one of these four files, and most of its classes are now UVM classes:

File	What's in it
`router_ifs.sv`	the three interfaces, with a driver's and a monitor's clocking block each
`router_agents_pkg.sv`	the packet, `router_pkt`, which is Lab 03's `rand_packet` with the same fields and the same constraints (`c_dst`, `c_kind`, `c_len`); an agent for the input, one for the APB port, and one output agent class, `out_agent`; the sequences
`router_env_pkg.sv`	the scoreboard, which is Lab 03's scoreboard and tracker in one class; the coverage, which is Lab 03's `cg_pkt` from our solution; and `router_env`, which builds and connects everything
`router_tests_pkg.sv`	`fill_plan`; the base test, with the steps Lab 03's `env` had (`write_routes`, `send_random`, `check_drops` and the rest) and `close_holes()`; and the tests, which are `router_class_test` (Lab 02's), `router_rand_test` and `router_fill_test` (Lab 03's), and the two that this lab adds, `router_stall_test` and `router_inject_test`

   Your Lab 02 and Lab 03 testbenches watched only output 0, and so does the port. `router_env` makes only one `out_agent`, the scoreboard queues only output 0's beats, and the scoreboard's `level()` takes output 1's level as 0, as Lab 03's tracker did. Find the `TODO (Part n)` comments. There are four in `router_env_pkg.sv` and three in `router_tests_pkg.sv`.

3. Run Lab 02's test, then the default test, then Lab 03's test with more packets:

```
make run UVM_TESTNAME=router_class_test
make run
make run UVM_TESTNAME=router_rand_test PLUSARGS=+N=400
```

   The default is `router_inject_test`, which for now does exactly what `router_rand_test` does. You should see, from the first run:

```
UVM_INFO labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(220) @ 7185000:
uvm_test_top.env.sb [SB] 100 packets in: 47 to output 0, 53 to output 1, 0 NOROUTE,
0 FULL; the highest level a header saw was 1
UVM_INFO labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(222) @ 7185000:
uvm_test_top.env.sb [SB] beats checked: 208 on output 0, 0 on output 1; 1 DROP_CNT
reads checked
UVM_INFO labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(224) @ 7185000:
uvm_test_top.env.sb [SB] 185 beats went to output 1, which nothing watches
DD105: TEST PASSED
```

   and from the second, Lab 03's four phases and its `close_holes()`:

```
UVM_INFO    labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)    @
185000: uvm_test_top [PHASE] 1 of 4: routes 011-, 200 packets; output 0 ready 100%,
output 1 ready 100%
```

```
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)     @
15295000: uvm_test_top.env.sb [DROP] DROP_CNT 00000000 as expected
UVM_INFO    labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)    @
15445000: uvm_test_top [PHASE] 2 of 4: routes -110, 200 packets; output 0 ready 100%,
output 1 ready 100%
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)     @
29235000: uvm_test_top.env.sb [DROP] DROP_CNT 00000000 as expected
UVM_INFO    labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)    @
29385000: uvm_test_top [PHASE] 3 of 4: routes 01-0, 200 packets; output 0 ready 100%,
output 1 ready 100%
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)     @
43715000: uvm_test_top.env.sb [DROP] DROP_CNT 00000000 as expected
UVM_INFO    labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)    @
43865000: uvm_test_top [PHASE] 4 of 4: routes 110-, 200 packets; output 0 ready 100%,
output 1 ready 100%
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)     @
57015000: uvm_test_top.env.sb [DROP] DROP_CNT 00000000 as expected
UVM_INFO    labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(243)    @
57185000: uvm_test_top [HOLES] output 0 stalled and filled to 8 by packets of 3 4 1
beats, then 1 beats for DST 0 and 8 for DST 1
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)     @
57655000: uvm_test_top.env.sb [DROP] DROP_CNT 00000001 as expected
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(220)     @
57655000: uvm_test_top.env.sb [SB] 805 packets in: 353 to output 0, 451 to output
1, 1 NOROUTE, 0 FULL; the highest level a header saw was 9
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(222)     @
57655000: uvm_test_top.env.sb [SB] beats checked: 1434 on output 0, 0 on output 1;
5 DROP_CNT reads checked
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(224)     @
57655000: uvm_test_top.env.sb [SB] 1799 beats went to output 1, which nothing watches
```

ending in `ok: 0 UVM_ERROR, 0 UVM_FATAL  1.48 s`. The routes are written as in Lab 03, with one character per route: `0` or `1` for the output, `-` for disabled. The `[DROP]` lines come from Lab 03's `DROP_CNT` check, which the scoreboard now does. It compares every read of `DROP_CNT` with the drops it predicted, and prints a line when they agree. That `make run` took us 1.6 s in all, because nothing was rebuilt. `+UVM_TESTNAME` picks the test when the simulator starts, and the simulator reads the other plusargs then too, so the third run, with `+N=400` packets in each phase, needed no rebuild either. It sent 1605 packets in 3.03 s.

4. Look at the coverage of the runs you've made so far:

   ```
   make cov
   ```

   You should see `covergroup : 58.5% (24/41)`, and `41 covergroup bins, 24 hit at least once`, followed by the 17 empty bins by name. They fall into four groups:

   - seven that no test can hit, as you found in Lab 03: `full_x_empty`, `full_x_low`, `full_x_at_8`, `out0_x_at_9`, `out0_x_high`, `out1_x_at_9` and `out1_x_high`;
   - `cp_fate.full`, `full_x_at_9` and `full_x_high`: no packet was dropped as FULL;
   - `out1_x_low` and `out1_x_at_8`: the scoreboard takes output 1's level as 0;
   - `cp_level.high`, `noroute_x_empty`, `noroute_x_low`, `noroute_x_at_8` and `noroute_x_high`: the only NOROUTE drops are `close_holes()`'s packets for DST 1, one in each of the last two runs, and they met a level of 9.

   Keep the second and third groups in mind for Part 3.

5. The port should catch what your Lab 02 and Lab 03 testbenches caught. Write down which of B05, B07, B08 and B09 you expect it to catch, then check:

   ```
   make check-all BUG_LIST="B05 B07 B08 B09"
   ```

   You should see `4 of 4 caught.`, with 50, 1, 2 and 50 `UVM_ERROR`s.

6. Now check this lab's two bugs, B06 and B13:

```
make check-all
```

You should see:

```
     ok  the golden design                      0 UVM_ERROR, 0 UVM_FATAL       1.5s
--------------------------------------------------------------------------------
 MISSED  tier 2  B06_disabled_route_forwards     0 UVM_ERROR, 0 UVM_FATAL       1.5s
 MISSED  tier 5  B13_admission_checks_other_output 0 UVM_ERROR, 0 UVM_FATAL     1.5s
--------------------------------------------------------------------------------
0 of 2 caught.
Still hiding from this test:
  B06_disabled_route_forwards
     hides behind: never disabling a route
  B13_admission_checks_other_output
     hides behind: stalling both outputs together, or never letting one fill
```

followed by a note and an error line from `make`, because the verdict is a fail. B06 forwards a packet for a disabled route when its output has room. B13 judges whether an output has room from the level of the other output. Why does this test miss each of them?

Answer: A1, in Answers at the end.

Check: `4 of 4 caught.` for Lab 02's and Lab 03's bugs, and `0 of 2 caught.` for this lab's.

Part 2: Watch the second output

What does it take to watch output 1, when an output agent is a class you can make twice?

1. Read `out_agent`, `out_monitor` and `out_ready_driver` in `router_agents_pkg.sv`. Nothing in them says which output they're on. The monitor and the driver each get their interface (`"vif"`) and their knobs (an `out_cfg`, with `port` and `ready_pct`) from `config_db`. The top module sets the interfaces for `uvm_test_top.env.out0*` and `uvm_test_top.env.out1*`, and `router_base_test` sets an `out_cfg` for each, so output 1's interface and `out_cfg` are already set for an agent named `out1`.
2. In `router_env`, do the two Part 2 TODOs: make `out[1]` with the name `"out1"`, and connect its monitor's `ap` to `sb.out_export`, as `out[0]` is. Leave the scoreboard alone for now. Before you build, write down what you expect `router_rand_test` to report on the golden router.
3. Rebuild and run:

```
make build
make run UVM_TESTNAME=router_rand_test
```

The rebuild took us 30 s. The run should fail on the golden router, with 50 errors that start:

```
UVM_ERROR labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(163) @ 225000:
uvm_test_top.env.sb [OUT] output 1: got 80 in cycle 23, and no beat was expected
UVM_ERROR labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(163) @ 265000:
uvm_test_top.env.sb [OUT] output 1: got 24 in cycle 27, and no beat was expected
UVM_ERROR labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(163) @ 275000:
uvm_test_top.env.sb [OUT] output 1: got cf in cycle 28, and no beat was expected
```

and `DD105: TEST FAILED (50 UVM_ERROR, 0 UVM_FATAL)`. The run stops at 50 because the base test sets UVM's quit count to 50. What is wrong, the router or the testbench?

Answer: A2, in Answers at the end.

4. Do the Part 2 TODO in `router_scoreboard::write_in`: queue every forwarded beat on `expected[in_port]`. Rebuild (31 s for us) and run `router_rand_test` again. You should see:

```
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(216)     @
57695000: uvm_test_top.env.sb [SB] 805 packets in: 353 to output 0, 451 to output
1, 1 NOROUTE, 0 FULL; the highest level a header saw was 9
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(218)     @
57695000: uvm_test_top.env.sb [SB] beats checked: 1434 on output 0, 1799 on output
1; 5 DROP_CNT reads checked
```

and `ok`. The 1799 beats that went unwatched in Part 1 are now checked. The number in parentheses after each file name is a line in that file, so yours can differ by a few lines if your edit kept the TODO comment or added lines.

5. Predict whether `make check-all` catches B06 or B13 now, then run it.

   Answer: A3, in Answers at the end.

Check: `router_rand_test` passes with beats checked on both outputs, and `make check-all` still says `0 of 2 caught.`

Part 3: Stall one output at a time

What does B13 need from the stimulus, and does the scoreboard know enough to judge the result?

1. Read the Routing section of the specification, and `level()` and `write_in` in the scoreboard. Then write down the situation in which B13 does something different from the golden router, given that it judges an output's room from the other output's level.

   Answer: A4, in Answers at the end.

2. Do the Part 3 TODO in `router_tests_pkg.sv`: in `router_stall_test`, override `set_sinks(k)` so that in phase `k` one output is ready in 5% of cycles and the other in 100%, output 0 stalling in phases 1 and 3 and output 1 in phases 2 and 4. `router_rand_test` calls `set_sinks(k)` at the start of each phase, with `k` from 1 to 4. Set `ocfg[p].ready_pct`. The test made those `out_cfg` objects and kept the handles, and each `out_ready_driver` reads its `ready_pct` again every cycle, so a change takes effect at the next clock edge. `ready_pct` is Lab 03's `sink.stall` grown into a percentage, where 0 is a stall and 100 is a sink that is always ready.

3. Rebuild with `make build`. It took us 105 s, against 30 s and 31 s in Part 2, because the Part 2 edits were inside existing function bodies and this one adds a function to a class. That changes most of the C++ that Verilator generates, as adding members to a class did in Lab 01, Part 5. Before you run it, look at `level()` again. Phase 2 stalls output 1. What will the scoreboard predict for the packets routed there?

4. Run the stall test on the golden router:

   ```
   make run UVM_TESTNAME=router_stall_test
   ```

   You should see phase 1 end with its `DROP_CNT` check, and phase 2 fail:

   ```
   UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(192)     @
   15395000: uvm_test_top.env.sb [DROP] DROP_CNT 00002500 as expected
   UVM_INFO    labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)    @
   15545000: uvm_test_top [PHASE] 2 of 4: routes -110, 200 packets; output 0 ready 100%,
   output 1 ready 5%
   UVM_ERROR     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(165)     @
   19275000: uvm_test_top.env.sb [OUT] output 1: got a1 in cycle 1928, expected 4d
   UVM_ERROR     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(165)     @
   19325000: uvm_test_top.env.sb [OUT] output 1: got 94 in cycle 1933, expected db
   ```

   `FULL_CNT` is bits 15:8 of `DROP_CNT`, so phase 1 dropped 0x25, or 37, packets as FULL, and the scoreboard predicted every one of them. The grader can't use this test yet, because it fails on the golden router. `make check BUG=B13 UVM_TESTNAME=router_stall_test` stops after the golden run:

```
    FAIL  the golden design                       50 UVM_ERROR                 0.8s
        UVM_ERROR [OUT] output 1: got a1 in cycle 1928, expected 4d
        UVM_ERROR [OUT] output 1: got 94 in cycle 1933, expected db
        UVM_ERROR [OUT] output 1: got 68 last in cycle 1936, expected 25
-----------------------------------------------------------------------
 VERDICT: FAIL -- your test does not pass the GOLDEN design.
```

Answer: A5, in Answers at the end.

5. Do the Part 3 TODO in `level()`, so it gives output 1's level the way it gives output 0's. Rebuild (32 s for us) and run the stall test again. You should see:

```
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(209)     @
58095000: uvm_test_top.env.sb [SB] 805 packets in: 190 to output 0, 241 to output
1, 1 NOROUTE, 373 FULL; the highest level a header saw was 16
UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(211)     @
58095000: uvm_test_top.env.sb [SB] beats checked: 753 on output 0, 988 on output 1;
5 DROP_CNT reads checked
```

and `ok`, with 373 FULL drops where the random test had none. The four phases' `DROP_CNT` reads were `00002500`, `00007700`, `00007e00` and `00005b00`, which are 37, 119, 126 and 91 FULL drops, and the scoreboard predicted each one.

6. Ask for the verdict on B13:

```
make check BUG=B13 UVM_TESTNAME=router_stall_test
```

You should see:

```
      ok  the golden design                     0 UVM_ERROR, 0 UVM_FATAL       1.5s
  caught  B13_admission_checks_other_output       50 UVM_ERROR                 0.1s
        UVM_ERROR [OUT] output 0: got e5 in cycle 142, expected 3f
        UVM_ERROR [OUT] output 0: got 90 in cycle 172, expected 73
        UVM_ERROR [OUT] output 0: got 28 in cycle 195, expected f6
-----------------------------------------------------------------------
 VERDICT: PASS
```

Phase 1 stalls output 0 and routes `DST` 1 and 2 to output 1 (`011-`). The first error is on output 0 in cycle 142, where `e5` came out and the scoreboard expected `3f`. Which of B13's two wrong decisions from step 1 puts a wrong beat on output 0?

Answer: A6, in Answers at the end.

Check: `router_stall_test` passes on the golden router with 373 FULL drops, and `make check BUG=B13 UVM_TESTNAME=router_stall_test` prints `VERDICT: PASS`.

Part 4: A factory override for disabled routes

How do you make a test send some packets to disabled routes, without editing the sequence that makes them or the driver that sends them?

1. Read `router_rand_seq` in `router_agents_pkg.sv`. It makes each packet with `router_pkt::type_id::create("pkt")`, which asks the factory for a `router_pkt`. If a test has told the factory to substitute a subclass of `router_pkt`, the sequence gets the subclass without knowing it. Then read `router_pkt`'s `c_dst` constraint, which is Lab 03's.

2. Do the first Part 4 TODO: write `router_noroute_pkt`, which extends `router_pkt` and sends one packet in four to a disabled route. Give it its own constraint named `c_dst`. A constraint with the same name as one in the parent class replaces the parent's, and Verilator 5.050 does that as the standard says. Decide which packets go to a disabled route with a non-random bit that `pre_randomize()` sets with `$urandom_range`, not with a `dist`. On this simulator a `dist` draws its value first, and `randomize()` fails when another constraint rules that value out (Lab 03, Part 3, line 3 of the experiments, and Constrained

random on Verilator). Every routing table that `random_routes()` writes has at least one disabled route, so the constraint can always be met.

3. Do the second Part 4 TODO: in `router_inject_test`, add a `build_phase` that calls `router_pkt::type_id::set_type_override(router_noroute_pkt::get_type())` and then `super.build_phase(phase)`. Nothing else in the test changes, so it's still the stall test, with different packets.

   The override reaches every packet the factory makes, wherever it's made. `close_holes()` picks the `DST` of its two random packets itself, and switches `c_dst` off for the one it sends to `DST` 1, so it needs both to be plain `router_pkt`s. It makes them with `new()`, in `new_packet()`, where the override can't reach them. When we made them through the factory instead, `router_inject_test` failed on 4 of 5 seeds with `UVM_FATAL [RAND]`. The packet for `DST` 1 was then a `router_noroute_pkt`, and when `close_holes()` switched `c_dst` off through a `router_pkt` handle, the subclass's `c_dst` stayed on.

4. Write down what you expect the golden run's `[SB]` line to say about NOROUTE drops. Then rebuild (105 s for us) and run the default test:

   ```
   make build
   make run
   ```

   With the reference solution you should see:

   ```
   UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(209)     @
   61465000: uvm_test_top.env.sb [SB] 805 packets in: 144 to output 0, 190 to output
   1, 194 NOROUTE, 277 FULL; the highest level a header saw was 16
   UVM_INFO     labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(211)     @
   61465000: uvm_test_top.env.sb [SB] beats checked: 648 on output 0, 799 on output 1;
   5 DROP_CNT reads checked
   ```

   and `ok`. You didn't have to change the scoreboard for this. It was written from the specification's Routing table, so it already predicts a NOROUTE drop for a packet to a disabled route; until now, no test had sent one while its output had room. Which packets go to disabled routes depends on how your `pre_randomize()` draws its bit. A version of ours that wrote the constraint differently but drew the bit the same way gave the same counts, so if yours differ, compare your draw with the reference solution's once step 5 says `2 of 2 caught.`

5. Ask for the lab's verdict:

   ```
   make check-all
   ```

   You should see:

   ```
        ok  the golden design                       0 UVM_ERROR, 0 UVM_FATAL       1.6s
   ---------------------------------------------------------------------------
     caught  tier 2  B06_disabled_route_forwards     50 UVM_ERROR                  0.4s
     caught  tier 5  B13_admission_checks_other_output 50 UVM_ERROR                0.1s
   ---------------------------------------------------------------------------
    2 of 2 caught.
   ```

6. You can also apply a factory override from the command line, without a test class. Apply yours to Lab 03's test that way, on the golden router and then on B06, and time the first run:

   ```
   time              make              run              UVM_TESTNAME=router_rand_test
   PLUSARGS=+uvm_set_type_override=router_pkt,router_noroute_pkt
   make                        run                        UVM_TESTNAME=router_rand_test
   PLUSARGS=+uvm_set_type_override=router_pkt,router_noroute_pkt BUG=B06
   ```

   The first prints `[UVM_CMDLINE_PROC] Applying type override from the command line: +uvm_set_type_override=router_pkt,router_noroute_pkt`, then `194 NOROUTE, 0 FULL` and `ok`, in 1.70 s with no rebuild. The second fails, and its first error is:

```
UVM_ERROR labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(152) @ 705000:
uvm_test_top.env.sb [OUT] output 0: got c6 in cycle 71, and no beat was expected
```

Without the plusarg, the same test on B06 says `MISSED`.

7. Record both runs from step 6 and look at the first packet that B06 forwards:

```
make                    waves                   UVM_TESTNAME=router_rand_test
PLUSARGS=+uvm_set_type_override=router_pkt,router_noroute_pkt
make                    waves                   UVM_TESTNAME=router_rand_test
PLUSARGS=+uvm_set_type_override=router_pkt,router_noroute_pkt BUG=B06
surfer build/runs/router_rand_test-s1/waves.fst &
surfer build/runs/router_rand_test-s1-B06/waves.fst &
```

Add `valid`, `ready`, `data` and `last` from `in_s` and `out0`, and `route_en_q`, `status_q` and `cnt_noroute_q` from `dut.u_regs`, and look at 650 to 850 ns:

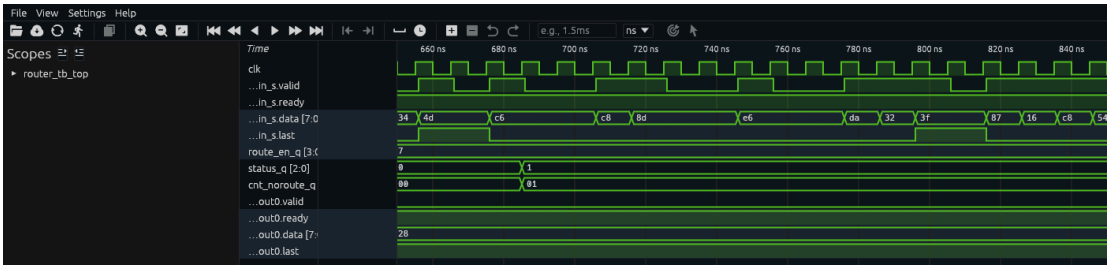


Figure 17.1: Packets for a disabled route on the golden router, in Surfer

The golden router. The header `c6` (`DST` 3) is accepted at the edge at 685 ns, while `route_en_q` is `7` (routes 0, 1 and 2 enabled). `STATUS.NOROUTE` and `NOROUTE_CNT` change at that edge, and `out0_valid` stays low.

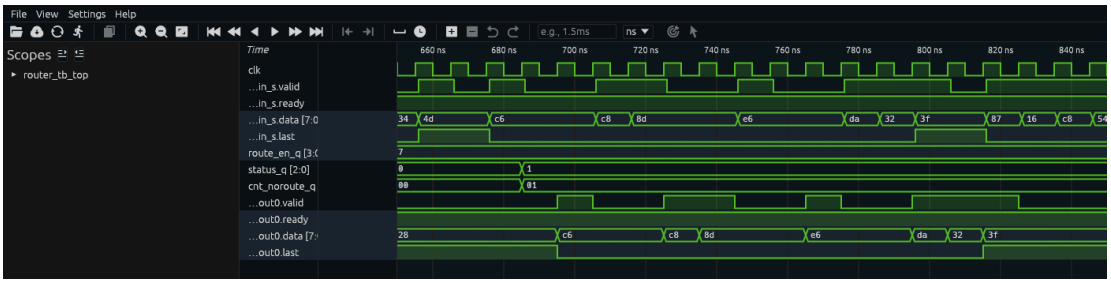


Figure 17.2: The same packet with B06 on, in Surfer

B06. The router counts the same NOROUTE drop, and then puts `c6 c8 8d e6 da 32 3f` on output 0, starting in the cycle that ends at 705 ns. That's two cycles after the header's cycle, which is when rule O8 says a forwarded beat first appears on the output.

Both pictures have the same input and the same register values, because the seed is the same and this is the first packet for a disabled route in the run, so B06 has changed nothing before it.

8. Run every test on five seeds and read the coverage:

```
make regress SEEDS="1 2 3 4 5"
```

You should see `25 ok  (25 runs)` and `covergroup : 82.9% (34/41)`. The stall test's FULL drops hit the three bins of Part 1's second group: `cp_fate.full`, `full_x_at_9` and `full_x_high`. The two bins of the third group, `out1_x_low` and `out1_x_at_8`, are hit because `level()` now gives output 1's real level. The five bins of the fourth group are hit too, now that `router_inject_test` sends packets to disabled routes while one output is stalled. The seven bins left empty are the seven that no test can hit, so this

is as far as the percentage goes. Read the coverage that `make regress` prints, not `make cov`'s. `make regress` reports only its own runs, while `make cov` merges every run in `build/runs`, and the runs you made before an edit moved the covergroup's lines show each bin a second time.

Check: `2 of 2 caught.` from `make check-all`, and `cp_fate.full` hit.

Part 5: Count what the port cost

Where did the lines go, compared with your Lab 03 testbench, and what did they buy you?

1. Count both testbenches. `FILES` takes the paths of your Lab 03 files, which are in Lab 03's `tb/` with the covergroup, the constraints and the `close_holes()` you wrote:

   ```
   make loc FILES="../lab-03-constraints-and-coverage/tb/*.sv ../lab-03-constraints-and-coverage/tb/*.svh"
   ```

   With the reference solution's edits, the UVM part prints:

   ```
   This lab's UVM testbench:
     file                       lines    code
     tb/router_ifs.sv              85      54
     tb/router_agents_pkg.sv      495     359
     tb/router_env_pkg.sv         305     214
     tb/router_tests_pkg.sv       396     266
     tb/router_tb_top.sv           64      49
     total                       1345     942
   ```

   As shipped it was 1330 lines and 921 code lines, so the reference solution's edits added 21 code lines. With our Lab 03 solution in Lab 03's `tb/`, the `FILES:` part totals 1206 lines and 789 code lines. Write your two totals, for this lab's testbench and for your Lab 03 testbench, in your notes.

2. For each piece of your Lab 03 testbench, find the UVM piece that does its job, and write down whether its code moved across nearly unchanged or changed shape. Then list the UVM pieces that have no Lab 03 counterpart at all.

   Answer: A7, in Answers at the end.

3. Using Parts 2 to 4, what did the extra lines buy you in this lab?

   Answer: A8, in Answers at the end.

Check: your notes have a line count for both testbenches, and your table has a build time and a `make check-all` result for each part.

Finishing

You're done when `make check-all` prints `2 of 2 caught.` with the `router_stall_test`, `router_noroute_pkt` and `router_inject_test` you wrote.

Questions:

1. The first B13 error came in phase 1, where output 1 is always ready and the old `level()` shortcut gave the right answer. Why did you still have to fix `level()` before `make check` would give a verdict on B13?

   Answer: A9, in Answers at the end.

2. In Part 4, step 7, why does the B06 run's `NOROUTE_CNT` agree with the golden run's?

   Answer: A10, in Answers at the end.

Submit: nothing for this lab.

Next on the Course path: the reading on assertions, reactive agents and virtual sequences, then Lab 05: Protocol rules and backpressure.

Answers

A1. Part 1: Build the port

`router_pkt`'s `c_dst` constraint sends random packets only to enabled routes. The only packet the test sends to a disabled route is `close_holes()`'s packet for DST 1. It arrives when output 0 already holds 9 to 16 beats (9 on this seed: the fill of 3, 4 and 1 beats, then a 1-beat packet for DST 0), so there's no room, and B06 drops it just as the golden router does.

B13 changes a packet's fate only when one output holds more than 8 beats and the other 8 or fewer. In the random phases both outputs are always ready, so neither holds more than a beat or two (Lab 02's test reports a highest level of 1). In `close_holes()` output 0 fills, but the packet for DST 0 arrives at exactly 8, which B13 admits too because output 1 is empty, and the last packet goes to a disabled route, which is a NOROUTE drop either way. No packet for an enabled route met an output holding more than 8 beats, which is also why `cp_fate.full` was never hit.

A2. Part 2: Watch the second output

The testbench. The new monitor reports every beat on output 1, but `write_in` still queues only output 0's beats (that's the scoreboard's Part 2 TODO), so the scoreboard has nothing to compare them with. When a check fires on the golden router, look for the bug in the check first.

A3. Part 2: Watch the second output

Neither. `make check-all` says `0 of 2 caught.` again. The scoreboard now sees both outputs, but the stimulus hasn't changed, so the router never gets a packet that either bug would treat differently.

A4. Part 3: Stall one output at a time

When a header arrives, one output has to hold more than 8 beats and the other 8 or fewer. A packet for the output with room is then dropped as FULL, because B13 looks at the full one, and a packet for the full output is admitted, because B13 looks at the one with room.

A5. Part 3: Stall one output at a time

`level()` returns 0 for output 1, a shortcut that was safe while output 1 was always ready, as it was in every test before this one. With output 1 ready in only 5% of cycles it fills past 8, and the router drops its next packets as FULL, as the specification says, but the scoreboard still expected them. The packet starting `4d` (`DST` 1, routed to output 1 by `-110`) was dropped, so the next header to come out of output 1, `a1` (`DST` 2, also routed to output 1), was compared with `4d`. Output 0's level was right all along, which is why phase 1 passed.

A6. Part 3: Stall one output at a time

B13 admits a packet for output 0 while output 0 holds more than 8 beats, because it judges output 0's room from output 1, which is nearly empty. In the waves of this run, the header `0d` (`DST` 0) arrived in cycle 114, when output 0 held 15 beats and output 1 none. The golden router dropped it as FULL. B13 admitted it and wrote its six beats into a buffer that holds 16 and already held 15, so they overran the buffer and wrote over beats that were still waiting in it. `e5` is the second beat of that packet.

B13 had made the other wrong decision first. From cycle 100 on it dropped packets for output 1 as FULL (the first was `8a`, `DST` 2), because output 0 was above 8. Those packets never came out, and output 1 sent nothing else before cycle 142, so the first beat to disagree with the scoreboard was on output 0.

A7. Part 5: Count what the port cost

Lab 03	Lab 04	What happened
`packet` and `rand_packet`	`router_pkt`	moved: the fields and the constraints `c_dst`, `c_kind` and `c_len` are the same; it gained `` `uvm_object_utils `` and is made through the factory
`generator` and the mailbox to the driver	`router_rand_seq` on the input agent's sequencer	changed shape: `start_item`/`finish_item` instead of a mailbox `put`
`driver`	`src_driver`	moved: `get_next_item`/`item_done` replace the mailbox `get`, and the idle gaps and the clocking-block drives are the same

Lab 03	Lab 04	What happened
`sink` and its `stall` knob	`out_ready_driver` and `out_cfg.ready_pct`	moved: the stall grew into a percentage, which the driver reads every cycle
`monitor` and the mailboxes to the scoreboard	`src_monitor`, `out_monitor` and their analysis ports	changed shape: each beat goes to `ap.write()` as it moves, instead of a whole packet into a mailbox
`scoreboard` and `tracker`, each with its own `run()` loop	`write_in`, `write_out`, `write_apb`, `level()`	changed shape: one class, which the monitors call, with no thread; it predicts FULL drops as well, from the tracker's levels
`router_coverage` and `cg_pkt`	`router_coverage` as a subscriber	moved: the same covergroup, sampled from the scoreboard's `fate_ap`
`apb_master` with `write()` and `read()`	`apb_agent`, `apb_rw_seq`, `write_reg`, `read_reg`	grew: a driver, a monitor and a sequence, and the monitor reports every transfer to the scoreboard
`env` and its steps (`write_routes`, `send_random`, `check_drops`, ...)	`router_env`, and the steps as tasks of `router_base_test`	changed shape: `build_phase` and `connect_phase` instead of a constructor that makes the mailboxes
`fill_plan` and `close_holes()`	the same, in `router_tests_pkg.sv`	moved: the same steps in the same order, now sent through the agents' sequences
`router_cfg` and its plusargs	plusargs read in the tests	moved, except `+SHOW`, which is gone
`$error` and `n_errors`	`` `uvm_error ``	changed: UVM counts the errors and stops the run at 50
`router_base_test` and its three tests	`router_base_test` and five tests	moved: the three tests' `body()`s call the same steps, and two tests are new; `+UVM_TESTNAME` picks one at run time, as it did in Lab 03
nothing	the agent classes, `out_cfg` and `config_db`, the factory registration lines	new

The code that touches the router (the drives, the sampling, the prediction) does the same work in both testbenches. UVM changed how the pieces find each other and pass data between them: through `config_db` and the factory instead of the environment's constructor, and through sequencers and analysis ports instead of mailboxes.

A8. Part 5: Count what the port cost

Watching output 1 took two lines in `router_env` and one in the scoreboard, which Lab 03 had written for output 0 alone. Two lines were enough in `router_env` because the output agent's driver and monitor get from `config_db` everything that differs between the two outputs. Stalling one output took a short function in a new test, which you pick at run time, and it works because the test holds the `out_cfg` handles that the drivers read. The factory override changed every packet that the sequence made, in every phase, without an edit to the sequence or the driver. You could apply it from a test class, or from the command line, where it took 1.70 s with no rebuild; any edit to the testbench cost 30 to 105 s of rebuilding. Lab 03's testbench could have done each of these, but each would have meant editing the classes that already worked.

A9. Finishing

`make check` runs the golden router first and stops if your test fails it (Part 3, step 4), because a test that fails on a correct design tells you nothing about a broken one. The shortcut predicted the wrong fate for output 1's packets in phases 2 and 4, where output 1 stalls, so the scoreboard reported errors on the golden router.

A10. Finishing

B06 changes only whether the packet is written to its output, and the router still counts it as a NOROUTE drop. So at this point the counters agree, and the bug shows as beats on an output that should have stayed idle, which the scoreboard checks.



5. Protocol rules

18 Assertions for interface rules 131
18 Three rules 131
18 A rule as a property 131
18 Binding the checker into the design 132
18 Failure messages on Verilator 132
18 When the antecedent never holds 134
18 Which rules suit an assertion 134
18 Check yourself 134
18 Further reading 135
18 Answers 135

19 Reactive agents and backpressure . 137
19 Who starts a transfer 137
19 A sequence that never ends 137
19 A driver that can't wait 138
19 Measured waits for each sink 139
19 Check the distribution you got 141
19 The sink's configuration object 141
19 Check yourself 141
19 Further reading 141
19 Answers 141

20 Virtual sequences 143
20 A sequencer that holds handles 143
20 Doing two things at once 144
20 A loop that ends with the traffic 145
20 Two sequences on one sequencer 146
20 Configuration changes and the scoreboard . . . 146
20 Check yourself 146
20 Further reading 146
20 Answers 147

21 Lab 05: Protocol rules and backpressure 149
21 Before you start 149
21 Part 1: The Lab 04 testbench against this lab's bugs 150
21 Part 2: Assertions for the output rules 151
21 Part 3: A sink that answers each beat 154
21 Part 4: A virtual sequence 155
21 Finishing 157
21 Answers 158

18. Assertions for interface rules

This page answers: *the scoreboard says a beat came out wrong. How do you get a check that names the rule the design broke, and the cycle it broke it in?*

You'll need the answer as soon as you start Lab 05. Your Lab 04 testbench already catches two of that lab's three bugs, B10 and B12, but on B10 the first thing it prints is `[OUT] output 0: got 04 in cycle 56, expected 03`. The scoreboard sees a beat only when a sink takes it, so that line is about the first wrong beat that output 0's sink took. While a beat waits on a stalled output, the scoreboard has no way to notice it going wrong. Part 1 of the lab has you work out how many cycles before that line the router broke its rule. A check that watches the pins themselves fires in the cycle in which the rule is broken, and its message can say which rule it was. A check of that kind is a SystemVerilog **concurrent assertion**. In Lab 05, Part 2 you'll write three of them for the router's outputs, in a checker that the lab's package already binds into the router.

So that this page doesn't write those three for you, its examples use a smaller design of our own, a one-beat valid/ready stage. The stage takes a beat on its input and holds it on its output until the sink takes it. Its source, its checker and the test that drives it are in `guide-examples/assertions-for-interface-rules/`, and every output quoted below comes from a run of it in the course image.

Three rules

The stage has three rules, written in the same style as the router specification's Valid/ready rules:

- **S1**. Once `out_valid` is 1, it stays 1 until the handshake.
- **S2**. While `out_valid` is 1 and `out_ready` is 0, `out_data` doesn't change.
- **S3**. While the stage is empty (`out_valid` is 0), `in_ready` is 1.

S1 and S2 are the router's O1 and O2, written for a design with one output and no `out_last`. We planted two bugs in the stage, and `+BUG=` switches one of them on at run time. With `drop`, the stage throws away a beat that has waited four cycles, which breaks S1. With `bubble`, it holds `in_ready` at 0 for a cycle after each beat leaves, which breaks S3 whenever the stage is empty in that cycle.

A rule as a property

Here is S1 as the checker writes it:

```
a_S1: assert property (@(posedge clk) disable iff (!rst_n)
                       out_valid && !out_ready |=> out_valid)
  else begin n_S1++; $error("S1: out_valid fell without a handshake"); end
```

Read it from the left. `@(posedge clk)` says the property is evaluated at every rising edge, and the values it uses are the ones the signals had just before that edge. That's the same value that a clocking block's `#1step` input gives you, and it's what the specification means by the value of a signal "in a cycle", so the property sees each cycle the way your monitor does.

The expression left of `|=>`, `out_valid && !out_ready`, is the **antecedent**. It picks the cycles that the rule talks about, here the cycles in which a beat is on offer and isn't taken. `|=>` means "then, in the next cycle". The expression after it is the **consequent**, which must hold in that next cycle. In a cycle where the antecedent is false, the property checks nothing and passes. The action block after `else` runs only on a failure. Ours counts the failure and calls `$error` with the rule's name, so the message tells you which rule broke without your having to look up a line number.

S2 has the same antecedent and a different consequent, `$stable(out_data)`, which is 1 when `out_data` has the same value as it had in the previous cycle. S3 is about a single cycle, so it uses the other implication:

```
a_S3: assert property (@(posedge clk) disable iff (!rst_n)
                       !out_valid |-> in_ready)
```

`|->` checks the consequent in the same cycle as the antecedent. A rule of the form "whenever this is true, that is true too" uses `|->`; a rule about what happens next uses `|=>`. Lab 05's O6 is of the first kind, because it's about the beat in the cycle of its own handshake.

`disable iff (!rst_n)` switches the property off while `rst_n` is 0, and abandons any attempt that was waiting for its next cycle when the reset came. The router specification explains why every rule about normal operation needs it. A reset ends whatever was in progress, and the router lowers `out_valid` at once, with no handshake. O1 allows that, but without `disable iff` the assertion for O1 (or for S1) would count it as a failure. The rules about what holds *during* a reset are checked without `disable iff`, and you'll write some of them in Lab 07.

Binding the checker into the design

The assertions are in a module of their own, `stage_sva`, whose inputs are the stage's pins. A `bind` statement at the end of the same file, `stage_sva.sv`, puts a copy of that module inside the stage:

```
bind stage stage_sva u_sva (
  .clk(clk), .rst_n(rst_n),
  .in_valid(in_valid), .in_ready(in_ready),
  .out_valid(out_valid), .out_ready(out_ready), .out_data(out_data)
);
```

`bind` instantiates `stage_sva` as `u_sva` in every instance of `stage`. The names on the right of each connection are signals inside `stage`, so the checker sees the design's own ports, and the stage's RTL doesn't change. The top module reads the failure count back out through the design instance, as `dut.u_sva.n_fail`. Lab 05 does the same for the router, with the `bind` in a file of its own, `tb/router_sva_bind.sv`, and Part 2 asks you why the testbench attaches its checker with `bind` instead of instantiating it in the top module.

Failure messages on Verilator

The checker is plain SystemVerilog with no UVM in it, so the simulator reports its failures itself, and they don't look like the `UVM_ERROR` lines you're used to. Here is the `drop` bug with Verilator's default error limit. The command is `make run BUG=drop PLUSARGS=`, in which the empty `PLUSARGS=` replaces the `+verilator+error+limit+1000` that our Makefile passes:

```
UVM_INFO @ 0: reporter [RNTST] Running test stage_test...
[245000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1: S1:
out_valid fell without a handshake
%Error: guide-examples/assertions-for-interface-rules/stage_sva.sv:37: Verilog $stop
Aborting...
```

The number in brackets is the simulation time in picoseconds, so the failure was at 245 ns. Our clock has a 10 ns period with its first rising edge at 5 ns, so the edge that ends cycle n is at $10n - 5$ ns. The failure was therefore found at the edge that ends cycle 25. A `|=>` failure is reported in the cycle of the consequent, so the beat was still waiting in cycle 24, and `out_valid` was 0 in cycle 25. The waveform of the same run shows both cycles:

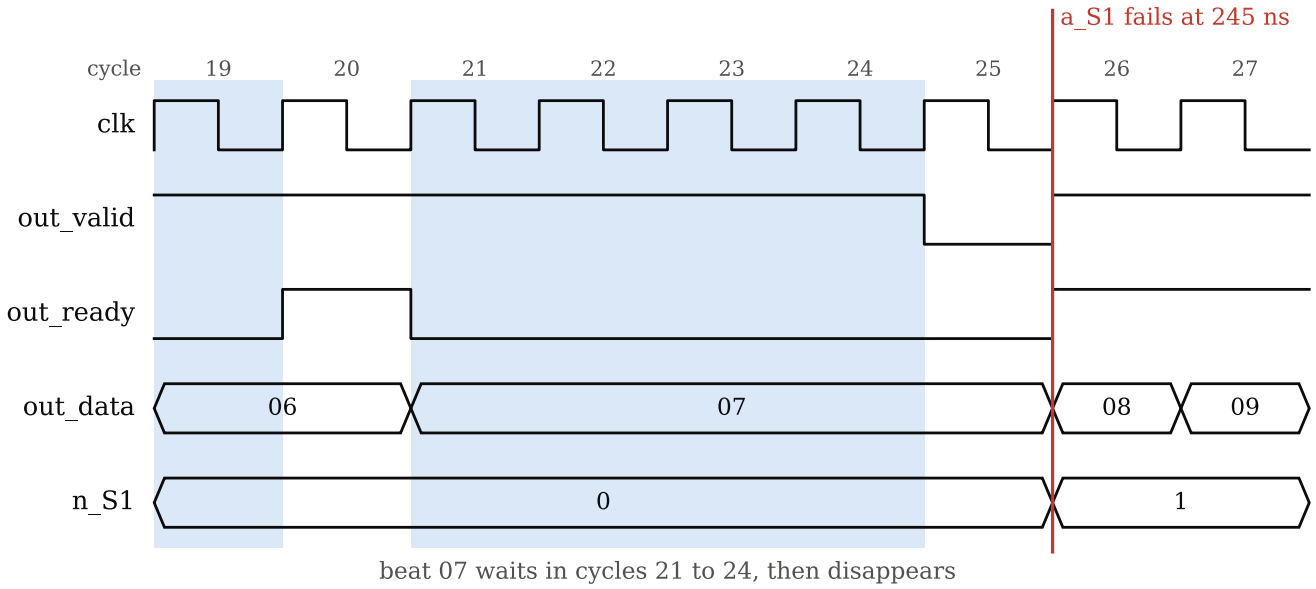


Figure 18.1: The stage with the drop bug, cycles 19 to 27

The shaded cycles are the ones in which a beat is on offer and isn't taken, so the antecedent of S1 holds in them. Beat `07` waits in cycles 21 to 24, and in cycle 25 `out_valid` is 0 although the sink never took `07`. At the edge that ends cycle 25 the assertion fails and `n_S1` counts it. (The stage takes `08` in cycle 25, so the output is busy again from cycle 26.)

The message names the assertion by its full path, `stage_tb_top.dut.u_sva.a_S1`, so it also tells you which instance of the checker fired. In Lab 05 the path tells you the output as well, because the router's checker has one instance of `router_sva_out` for each output, `u_out0` and `u_out1`.

The run stopped at that failure. Verilator's default error limit is 1, so the first `$error` ends the simulation with exit status 1, before the test's report phase and before UVM's summary. The course's grader reads the log as well as the exit status. It found the assertion's `%Error` line in this run's log and called the run `caught: 1 assertion`.

The run-time option `+verilator+error+limit+<value>` raises the limit (`verilator --help` lists it as "Set error limit"). Every lab from Lab 05 on passes `+verilator+error+limit+100000`, and our example's Makefile passes 1000. With that limit, `make run BUG=drop` prints every failure of the same bug:

```
[245000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1: S1:
out_valid fell without a handshake
-Info: guide-examples/assertions-for-interface-rules/stage_sva.sv:37: Verilog $stop,
ignored due to +verilator+error+limit
[375000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1: S1:
out_valid fell without a handshake
[605000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1: S1:
out_valid fell without a handshake
[1165000] %Error: stage_sva.sv:37: Assertion failed in stage_tb_top.dut.u_sva.a_S1:
S1: out_valid fell without a handshake
UVM_ERROR guide-examples/assertions-for-interface-rules/stage_tb.sv(79) @ 2025000:
uvm_test_top [SVA] the bound assertions counted 4 failures
```

Verilator prints the `-Info` line once, after the first failure. A raised limit still stops the run at the error that reaches it, again before UVM's summary, so the labs use a number that no run will reach.

Our test prints the `UVM_ERROR` on the last line itself. In its check phase it reads the checker's `n_fail` and reports an `[SVA]` error if the count isn't 0. Without that check the test would pass, because `dd105_base_test` passes a test when UVM's report server has counted no `UVM_ERROR` and no `UVM_FATAL`, and the server counts only UVM's own reports, not an assertion's `$error`. With `+IGNORE_SVA`, which skips that check, the same run printed the same four `%Error` lines and then `DD105: TEST PASSED`, and its exit

status was 0. So any testbench that relies on assertions needs something that turns their failures into a `UVM_ERROR` before the test ends. In the labs, the top module passes `dut.u_sva.n_fail` to the scoreboard through an interface, `router_sva_if`, and the scoreboard reports an `[SVA]` error at the end of the test if the count isn't 0.

When the antecedent never holds

Here is the same buggy stage again, with the sink ready in every cycle (`+READY_PCT=100`):

```
UVM_INFO  guide-examples/assertions-for-interface-rules/stage_tb.sv(85)  @  2025000:
uvm_test_top [STAGE] 132 beats in, 131 out; a beat waited in 0 cycles (cover property:
0); 0 assertion failures
```

and the grader's verdict on it was `MISSED`. The bug is still in the stage, but it acts only on a beat that has waited four cycles, and with the sink always ready no beat waited at all. The antecedent of S1 was false in every cycle, so the assertion passed in every cycle without checking anything. A passing assertion tells you only that its rule held in the cycles where its antecedent was true, and not whether there were any such cycles. Lab 05's B11 hides from your O6 assertion for a related reason. O6's antecedent holds on the last beat of every 8-beat packet, but B11 acts only on a packet longer than 8 beats, so O6 can fail only on the 8th beat of a packet that has a 9th. None of Lab 04's tests sends such a packet, and your virtual sequence in Part 4 is the first stimulus that does.

The checker therefore also counts the cycles in which the antecedent of S1 and S2 holds, in two ways. `n_tried` is a counter in an `always` block beside the assertions, and `c_S1` is a cover property that counts the same cycles:

```
c_S1: cover property (@(posedge clk) disable iff (!rst_n) out_valid && !out_ready)
  n_covered++;
```

On the golden stage, with the sink ready half the time, both counted 90. In the `drop` run above, with the sink always ready, both counted 0. Verilator also writes the cover property's count into `coverage.dat`, as a `user` point named `stage_tb_top.dut.u_sva.c_S1` with the count 90, and `verilator_coverage` reports that point as `user : 100.0% (1/1)`. The course's `make cov` lists only covergroup bins, though, so in the labs the evidence that a rule was exercised comes from covergroups. Lab 05's `cg_wait` is one of them, and it records, for each packet an output delivers, the longest that any of its beats waited.

Which rules suit an assertion

A rule about one or two cycles at a time becomes a short property like S1. I1, I2, O1 and O2 are rules of that kind, and the specification says that most of its other rules are too. Assertions for them name the rule and fire in the cycle in which it's broken. They need no model of the design, so they keep working in any testbench that binds the file. The specification's Valid/ready rules section lists the rules that need more than a short property:

- O5, O6 and O8 need state kept for each output (the order of the beats, how many beats the current packet has had, and the cycles in which each beat went in and was taken). They belong in the scoreboard, or in an assertion with tracking code beside it. Your O6 in Lab 05 uses a small counter, `beats`, which counts the beats of the current packet that have already been taken. It's written for you in `router_sva_out`.
- I4, O3 and O7 say what a signal does *not* depend on, which no single trace can show. A model that predicts the outputs checks them, and an assertion can catch only their consequences.
- I3 is a rule for your own source's stimulus, so it belongs in the driver or the sequence, not in an assertion.

The scoreboard knows what every beat should be, so it catches far more kinds of bug than the assertions do, but its message says only that a beat was wrong. An assertion names the rule and the cycle, which tells you where and when to look, so a testbench needs both kinds of check. In Lab 05 you'll see both fire on the same bugs, often many cycles apart.

Check yourself

1. The `drop` run's first failure printed `[245000]`. In which cycle was `out_valid` 0 without a handshake, and why is the failure reported at the edge that ends that cycle rather than one cycle earlier?

   Answer: A1, in Answers at the end.

2. A test of yours prints three `%Error: ... Assertion failed` lines and then `DD105: TEST PASSED`, with exit status 0. What is it missing?

   Answer: A2, in Answers at the end.

3. With the sink always ready, the `drop` bug passed. Which number in that run's `[STAGE]` line tells you that S1 was never tried?

   Answer: A3, in Answers at the end.

4. Which implication would you use for the router's O6, "a beat taken when the packet has already had 7 beats has `out_last` = 1", and why?

   Answer: A4, in Answers at the end.

Further reading

ChipVerify's assertion pages are a good reference for the syntax. Their example output comes from commercial simulators, so the failure messages look different from Verilator's.

- Property Implication: `|->` against `|=>`, and vacuous success, with the same trace read both ways.
- disable iff: where it goes, and why gating the antecedent instead isn't the same thing.
- $rose, $fell, $stable.
- Handshake Properties, in the formal section: the valid/ready rules, which side's rules are assertions and which are assumptions, and why a sink that is always ready hides a sender bug.
- assert, assume & cover.
- SystemVerilog Assertions, the overview. Correction: it writes an immediate assertion as `$assert(...)`, which isn't SystemVerilog (the keyword is `assert`), and its practice solution asserts a bare sequence with no implication, so it fails in every cycle where `req` is 0.
- Assertion Based Verification. Correction: it says `assume` has no effect in simulation; the standard says a simulator checks an assumed property the same way as an assertion.
- SystemVerilog Assertions in a UVM Testbench: which checks belong in assertions, which in the monitor and which in the scoreboard.

Next: Reactive agents and backpressure.

Answers

A1. Check yourself

245 ns is the edge that ends cycle 25 ($10 \times 25 - 5 = 245$), so `out_valid` was 0 in cycle 25. The antecedent held in cycle 24, when the beat was on offer and not taken, and `|=>` checks the consequent one cycle later, so the failure can't be known until the edge that ends cycle 25.

A2. Check yourself

Something that turns the assertions' failure count into a `UVM_ERROR`. UVM counts only its own reports, so `dd105_base_test` saw no errors and passed the test. The labs pass the count from the bound checker to the scoreboard, which reports it at the end of the test.

A3. Check yourself

`a beat waited in 0 cycles` is `n_tried`, and the cover property's count beside it is 0 as well. Either one tells you. Give every assertion you rely on a count like these, or a coverage bin that shows its antecedent held.

A4. Check yourself

`|->`. The rule is about the beat in the cycle of its own handshake, so the consequent has to be checked in the same cycle as the antecedent. With `|=>` the check would look at `out_last` in the cycle after, when it belongs to a different beat or to no beat at all.

19. Reactive agents and backpressure

This page answers: *the design decides when each beat appears on its output. How does a testbench answer each one, and how do you choose how long each beat waits?*

Your Lab 04 sink flipped a coin in every cycle to set `out_ready`, which was 1 with probability `ready_pct` percent whether or not a beat was on offer. That is backpressure, but `ready_pct` was the only thing about it you could set. In Lab 05, Part 1, the covergroup `cg_wait` takes each packet of the graded test and records the longest that any of its beats waited for `ready`. Its bins are `none` for 0 cycles, `short_wait` for 1 to 3, `long_wait` for 4 to 20 and `stalled` for more than 20, and with Lab 04's sinks it reports:

```
          0  __vlAnonCG_cg_pkt.cp_len.oversize
        257  __vlAnonCG_cg_wait.cp_wait.none
          2  __vlAnonCG_cg_wait.cp_wait.short_wait
         18  __vlAnonCG_cg_wait.cp_wait.long_wait
         57  __vlAnonCG_cg_wait.cp_wait.stalled
```

Only two of the 334 packets had a longest wait of 1 to 3 cycles. Every other packet either went through without waiting at all or had at least one beat that waited 4 cycles or more. (The first line counts packets of 9 beats or more, and it is 0 because no Lab 04 test sends one.) In Part 3 you'll finish a sink driver that answers each beat separately, with an item from a sequence. This page shows how an agent like that is put together, using a small design of our own in `guide-examples/reactive-agents-and-backpressure/`, and measures how long the beats wait with each kind of sink.

Who starts a transfer

In the agents of Labs 01 and 04 that take items from a sequence, the testbench decides when each transfer happens. Lab 04's source driver, for example, calls `get_next_item()`, waits until its sequence hands over a packet, and then puts the packet on the input. Nothing happens on the input until the driver has a packet, so the driver can wait as long as it likes.

A **reactive** agent works the other way round. The design decides when something happens, and the testbench answers. A sink is the simplest example. The router decides when a beat appears on an output, and the sink decides how long that beat waits before it's taken. Other examples are a memory model that answers a processor's read requests, and a bus slave. A reactive agent has the usual pieces (an item, a sequence, a sequencer, a driver and a monitor), but each of its items describes an answer to something the design does, rather than a transfer for the driver to start. Its driver also has to drive its side of the interface in every cycle, whether or not it has an item.

Our example's design, `beat_src`, numbers its beats 0, 1, 2 and so on. After each handshake it leaves its output idle for 1 to 4 cycles, a number it takes from an LFSR, and then puts the next beat on the output, where the beat stays until the sink takes it. The sink agent has two modes. In `coin` mode its driver does what Lab 04's did. In `beat` mode the driver answers each beat with one `snk_item`, and the item says how many cycles the beat waits after it first appears:

```
class snk_item extends uvm_sequence_item;
  rand bit [4:0] delay;
  int unsigned   max_delay = 12;
  constraint c_delay { int'(delay) <= max_delay; }
  ...
```

A delay of 0 means `ready` is already 1 when the beat appears.

A sequence that never ends

The sink's sequence makes one item per beat, for as long as the test runs:

```
task body();
  forever begin
    it = snk_item::type_id::create("it");
    start_item(it);
    it.max_delay = cfg.max_delay;
    ok = it.randomize();
    if (ok == 0) `uvm_fatal("RAND", "snk_item randomize() failed")
    finish_item(it);
  end
endtask
```

The agent starts this sequence itself, from its own `run_phase`, on its own sequencer at time 0, and the sequence never returns. Lab 05's `out_agent` starts its `out_ready_seq` the same way. The sequence also raises no objection. The test's objection decides when the test ends, and a sequence that never finishes would keep the test running forever if it raised one.

Each call to `finish_item()` blocks until the driver has taken the item and called `item_done()`, so the sequence is never more than one item ahead of the beats. The item the driver holds is the answer to the beat on offer, or, if no beat is on offer yet, to the next one.

A driver that can't wait

The driver's loop has two halves, one for each half of the clock cycle:

```
forever begin
  @(vif.cb);                                  // a rising edge
  vif.cb.ready <= rdy;                        // ready for the cycle that starts here
  @(negedge vif.clk);                         // the middle of the cycle
  if (vif.valid && rdy) begin                 // a handshake at the next edge
    seq_item_port.item_done();
    it = null;
  end else if (...)
    ...                                       // one more cycle of this beat's wait
  if (it == null)
     seq_item_port.try_next_item(it);           // returns at once, with or without
an item
  rdy = ...;                                  // ready in the next cycle
end
```

It drives `ready` just after the rising edge, through the clocking block, and it looks at `valid` at the falling edge, when `valid` and `ready` have settled. At that point it knows whether a beat moves at the next rising edge. If one does, that beat has had its answer, so the driver calls `item_done()`.

The driver asks for items with `try_next_item()`, not with `get_next_item()` as Lab 04's source driver does. `get_next_item()` blocks until the sequence hands over an item. `try_next_item()` returns straight away, and leaves the handle `null` if the sequence has nothing ready. The driver then drives `ready` to 0 for the next cycle, and asks again in the middle of that cycle. Lab 05's `out_ready_driver` has the same shape, and Part 3 asks you why it uses `try_next_item()`.

On Verilator 5.050, a drive issued in the same time step as a clock edge, but after the sequencer has handed over an item, lands a cycle late. The labs found this, and the comment at the top of `tb/router_agents_pkg.sv` in Labs 04 to 07 records it. This is the second reason the driver's loop is split between the edge and the middle of the cycle; the first is to look at `valid` once it has settled. When the driver asks for items in the middle of the cycle and drives only from an edge, the handover and the drive happen in different time steps.

The rest of the loop counts the cycles the beat has been on offer, starting from the first cycle it appears, and sets `rdy` once the item's delay has gone by. You'll write that countdown for the router in Part 3, so it's left out above.

Measured waits for each sink

We ran 2,000 beats through each sink and counted, for each beat, the cycles it waited between appearing and being taken. The table sorts the waits into the same ranges as `cg_wait`'s bins, but it counts every beat, where `cg_wait` counts each packet once, by its longest wait.

Sink	Waited 0	1 to 3	4 to 20	More than 20	Mean wait
coin flip, 5%	99	237	1014	650	18.27
coin flip, 50%	1017	854	129	0	0.98
coin flip, 95%	1907	93	0	0	0.05
per beat, delays 0 to 12 from `randomize()`	116	390	1494	0	6.81
per beat, delays 0 to 12 from `$urandom_range`	169	466	1365	0	6.01

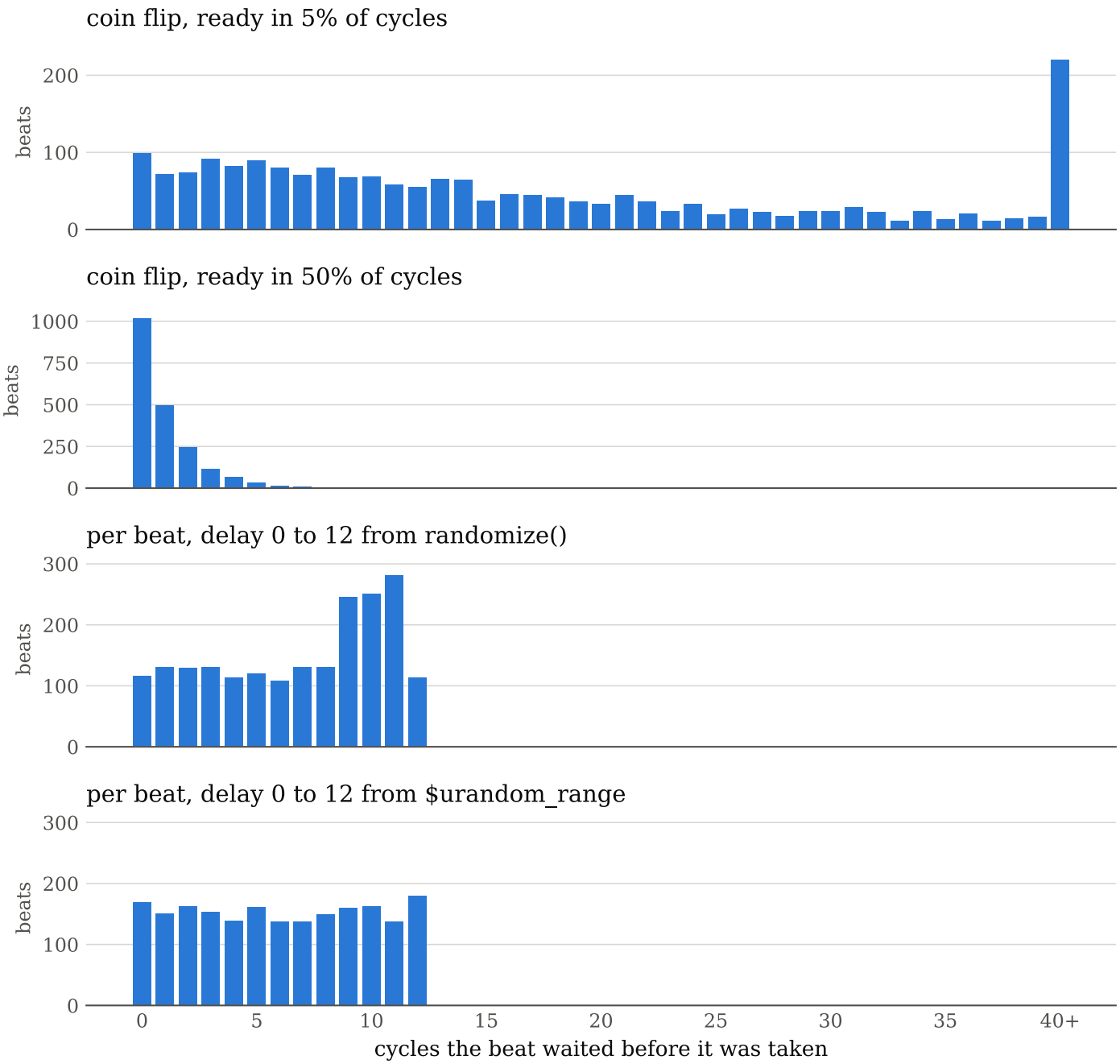


Figure 19.1: How long each of 2,000 beats waited, for four of the sinks in the table

A coin-flip sink gives each beat the same chance of being taken in every cycle, so its waits fall off geometrically. If p is `ready_pct` as a fraction, a beat waits w cycles with probability $p\,(1-p)^w$, and the mean wait is $(1-p)/p$, which is 19 cycles at $p = 5\%$ and 1 cycle at 50%, against our measured 18.27 and 0.98. Whatever the setting, a wait of 0 cycles is the most likely, and each wait is $1-p$ times as likely as a wait one cycle shorter, so `ready_pct` sets only how quickly the waits fall off. At 5%, most beats waited 4 cycles or more, and only 237 of the 2,000 waited 1 to 3. At 50%, no beat waited more than 11 cycles. At 95%, 1,907 of the 2,000 beats were taken in the cycle they appeared.

The test you inherit from Lab 04 runs each sink at 100% or 5% in its four phases, and stalls output 0 completely while `close_holes()` runs. At 100% a beat never waits. At 5% a packet of more than one beat nearly always has a beat that waits 4 cycles or more, so a packet whose longest wait is 1 to 3 cycles is rare, and that test produced only two.

A per-beat sink makes each beat wait for the delay in its item, so the spread of the waits is the spread of the delays that the sequence produced. With delays of 0 to 12, between 1,365 and 1,494 beats waited 4 to 20 cycles, and none waited more than 20. Lab 05′s `out_cfg.mode` offers both kinds of sink. `SNK_PCT` is Lab 04′s coin flip, and in `SNK_FAST`, `SNK_SLOW` and `SNK_RANDOM` the driver answers each beat with an item.

Check the distribution you got

The delays from `randomize()` weren't uniform, as the fourth row of the table and the third chart of the figure show. The values 9, 10 and 11 came up about twice as often as the others (245, 251 and 281 times out of 2,000 on seed 1, and 234, 235 and 260 on seed 2, where a uniform choice would give about 154 each). With `$urandom_range(12)` in place of `randomize()`, every value from 0 to 12 came up between 137 and 180 times. The standard says that each value the constraint `delay <= 12` allows is equally likely, but Verilator 5.050′s solver didn't pick them equally often. As in Lab 03, you can't rely on a constraint for how often each value comes out, so count what the stimulus actually produced. Lab 05′s `out_ready_item` uses `randcase` in `pre_randomize()` to pick the range a delay comes from (0, 1 to 3, or 4 to 12), and leaves only the value inside that range to the solver. `cg_wait` is where you check the result.

The sink's configuration object

The sink's driver reads its mode and `ready_pct` from a configuration object, `snk_cfg`, in every cycle, and its sequence reads `max_delay` for every item. So a test, or a virtual sequence, can change the backpressure in the middle of a run just by writing a field of that object, without starting anything on the sink's sequencer. In Lab 05, Part 4, `backpressure()` changes both sinks' modes this way, every 30 to 250 cycles while the packets go in. Virtual sequences explains how a sequence gets hold of the object.

Check yourself

1. A coin-flip sink has `ready_pct` set to 20. What fraction of the beats are taken in the cycle they appear, and what is the mean wait?

   Answer: A1, in Answers at the end.

2. Why does the sink's sequence raise no objection?

   Answer: A2, in Answers at the end.

3. An item with a delay of 0 reaches the driver while the design's output is idle. What does the driver drive, and how long does the beat wait when it appears?

   Answer: A3, in Answers at the end.

Further reading

- UVM Reactive Sequence: the responder pattern for a slave agent, with a sequence that keeps its own memory model and answers the design's reads and writes from it. The test starts that sequence in a background thread, with no objection.
- Using get_next_item(): the driver-sequencer handshake, which call blocks and what releases it, and `try_next_item()`.
- UVM Driver. Correction: its "simplified" loop calls `item_done()` after one edge and never clears the request, which is the mistake the same page warns against further down.
- Using get() and put(). Its log comes from a different version of the code: the driver's path in the log has an environment the test never creates.
- Handshake Properties, in the formal section, on why a sink that is always ready hides bugs in the sender.

Next: Virtual sequences.

Answers

A1. Check yourself
20% of beats wait 0 cycles, because the coin comes up 1 in the beat's first cycle with probability 0.2. The mean wait is $(1 - p)/p = 0.8/0.2 = 4$ cycles.

A2. Check yourself

It never finishes. It makes one item per beat for as long as the test runs, so if it raised an objection, the run phase could never end. The test's own objection decides when the test is over. Our example's test holds it until the monitor has counted 2,000 beats, and Lab 05's tests hold theirs while the source sends its packets and the sinks drain them.

A3. Check yourself

From the next edge on it drives `ready` = 1, because a delay of 0 leaves nothing to wait for. When the beat appears, `ready` is already 1, so the handshake happens in the beat's first cycle and the beat waits 0 cycles. A coin-flip sink can promise that for a particular beat only when its `ready_pct` is 100.

20. Virtual sequences

This page answers: *the routing table, the packets and the backpressure are each driven by a different agent. How do you make them change together, from one place?*

Lab 04′s tests did one thing at a time, in the test's `body()`. A test wrote the routing table, set the sinks, sent packets, drained, read `DROP_CNT` and then started again, and each step waited for the one before it to finish. In Lab 05, Part 4 you'll want routes that move to the other output while packets are still going in, and both sinks changing their behaviour at the same time. So the APB agent, the source agent and both sinks' agents all have work to do at the same time, in an order that you choose. In UVM you do this with a **virtual sequence**, which is a sequence that starts other sequences on several agents' sequencers and changes those agents' knobs, but makes no items of its own.

This page builds one for a small design of our own, a FIFO that holds four beats and has a valid/ready stream on each side. The files are in `guide-examples/virtual-sequences/`. A source agent drives the FIFO's input, and its sequences send one beat per item. A sink driver drives `out_ready` on the output side, and how often it's ready depends on `ready_pct`, a knob in a configuration object. Neither of the two knows about the other. The sink has no sequencer at all. Lab 05′s sinks do have one, but a virtual sequence steers them the way you steer this sink, by changing their configuration rather than by sending them items, so this sink is a good stand-in for them.

A sequencer that holds handles

An ordinary sequence runs on its agent's sequencer, and in the labs each virtual sequence runs on a sequencer too. A **virtual sequencer** is a sequencer with no driver connected to it. It's there to hold handles to everything that the virtual sequence needs to reach, and ours holds four:

```
class vs_vsequencer extends uvm_sequencer;
  `uvm_component_utils(vs_vsequencer)
  uvm_sequencer #(src_item) src_sqr;   // the source agent's sequencer
  snk_cfg                   snk;       // the sink's knob
  vs_monitor                mon;      // what has come out, for the sequence to report
  virtual vs_if             vif;       // a clock to count cycles with
  ...
```

The environment makes it in `build_phase`, like any other component, and fills in its handles in `connect_phase`, when everything that they point to exists. Lab 05′s `router_vsequencer` holds the same kinds of thing for the router. It has the sequencers of the source agent and the APB agent, the sequencer and the `out_cfg` object of each sink, the scoreboard, and the APB interface.

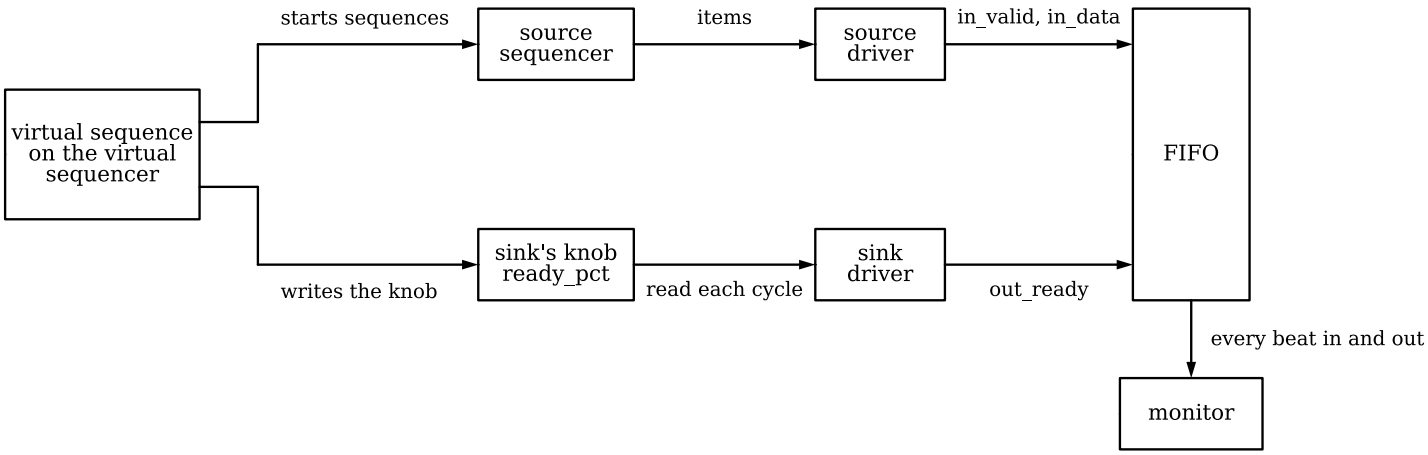


Figure 20.1: The example's testbench: one virtual sequence steering two drivers

Our example's testbench. The virtual sequence reaches the source through the source's sequencer, and the sink through the sink's configuration object, using handles that the virtual sequencer holds. The monitor checks every beat, whatever caused it.

The virtual sequence reaches the virtual sequencer through `p_sequencer`. That's a handle of the virtual sequencer's own type, and the `uvm_declare_p_sequencer` macro declares it and fills it in when the sequence starts:

```
class vs_base_vseq extends uvm_sequence;
  `uvm_object_utils(vs_base_vseq)
  `uvm_declare_p_sequencer(vs_vsequencer)
  ...
  task send(int unsigned n, bit [7:0] base);   // n beats, through the source agent
    src_seq s = src_seq::type_id::create($sformatf("src_%h", base));
    s.n    = n;
    s.base = base;
    s.start(p_sequencer.src_sqr, this);
  endtask
  function void sink(int unsigned pct);        // the sink's readiness from now on
    p_sequencer.snk.ready_pct = pct;
    ...
```

`s.start(p_sequencer.src_sqr, this)` runs the source agent's own sequence on the source agent's sequencer, as a test would, with the virtual sequence as its parent. `sink()` only writes the knob, without starting anything, and the sink's driver reads the new value at its next clock edge. Lab 05's `router_base_vseq` gives you eleven helpers like these, among them `route()`, `ctrl()`, `burst()`, `send_rand()`, `stall()` and `drain()`. Most of them are the base test's steps from Lab 04, rewritten as tasks of a sequence.

You can also do without a virtual sequencer. The test then hands the virtual sequence its handles itself and starts it with `start(null)`, and that works too. The labs use a virtual sequencer because then the environment fills in the handles once, for every virtual sequence and every test.

Doing two things at once

The test starts our virtual sequence, `vs_demo_vseq`, on the virtual sequencer. The sequence's first step stalls the sink and then, in a `fork`, sends six beats in one branch while a second branch waits 15 cycles and lets the sink go:

```
sink(0);
fork
  begin
```

```
    send(6, 8'h10);
    `uvm_info("VSEQ", "step 1: all 6 beats have gone in", UVM_LOW)
  end
  begin
    cycles(15);
          `uvm_info("VSEQ",  $sformatf("step  1:  the  FIFO  holds  %0d  beats",
p_sequencer.vif.level), UVM_LOW)
    sink(100);
  end
join
```

It printed:

```
UVM_INFO       guide-examples/virtual-sequences/vseq_tb.sv(218)       @       26000:
uvm_test_top.env.vsqr@@v [VSEQ] sink ready 0% of cycles
UVM_INFO       guide-examples/virtual-sequences/vseq_tb.sv(241)       @      175000:
uvm_test_top.env.vsqr@@v [VSEQ] step 1: the FIFO holds 4 beats
UVM_INFO       guide-examples/virtual-sequences/vseq_tb.sv(218)       @      175000:
uvm_test_top.env.vsqr@@v [VSEQ] sink ready 100% of cycles
UVM_INFO       guide-examples/virtual-sequences/vseq_tb.sv(237)       @      235000:
uvm_test_top.env.vsqr@@v [VSEQ] step 1: all 6 beats have gone in
```

By 175 ns the FIFO held four beats and was full, and `send()` still had two of its six beats to send. It could only return once the sink had taken some beats, and the sink would only take beats once something set `ready_pct` again. If you wrote the two as statements one after the other, `send()` and then `sink(100)`, they would deadlock, because `send()` waits for the sink and the line that releases the sink waits for `send()` to return. With `fork ... join` the two branches run at the same time, and `join` waits for both, so the step ended at 235 ns, when the last beat had gone in.

Every message carries the path `uvm_test_top.env.vsqr@@v`, which means that the report came from the sequence `v` running on `env.vsqr`. The path is how you tell a virtual sequence's messages apart from an agent's.

A loop that ends with the traffic

The second step sends 30 beats while another branch keeps changing how often the sink is ready. That branch can't know how long the traffic will take, so it loops until the traffic branch sets a flag, `traffic_done`:

```
traffic_done = 0;
fork
  begin
    send(30, 8'h40);
    traffic_done = 1;
  end
  while (!traffic_done) begin
    sink(pick[$urandom_range(2)]);     // 0, 30 or 100 percent
    cycles($urandom_range(12, 4));
  end
join
```

The loop set the sink twelve times between 335 ns and 1415 ns, and the last beat went in at 1445 ns. Step 4 of the sequence you write in Lab 05, Part 4 has the same shape, with `backpressure()` as the looping branch and a third branch that moves routes. If the traffic branch didn't set `traffic_done`, the loop would go on changing the sink for ever, and `join` would never return.

None of these sequences checks anything. In this example the monitor does the checking, and in the labs the scoreboard and the assertions do. The monitor watches both sides of the FIFO in every cycle and checks every beat that comes out, whatever is driving the FIFO, and at the end of this run it said `44 beats in, 44 out, 0 out of order`.

Two sequences on one sequencer

The third step starts two source sequences at once on the same sequencer. Each sends four beats. The first numbers its beats from `80`, and the second from `a0`:

```
fork
  send(4, 8'h80);
  send(4, 8'hA0);
join
```

and the beats came out of the FIFO as

```
step 3: the sink took 80 a0 81 a1 82 a2 83 a3
```

A sequencer takes one item at a time from the sequences that are running on it. Each `start_item()` is a request for the driver, and the sequencer grants them in turn, so the two sequences alternated item by item. Sometimes that's what you want, for example to mix two traffic streams together. It also means that a sequence you start on a sequencer that already has one running shares the driver with the first one rather than replacing it. Each of Lab 05's sinks runs a sequence that never ends, and Part 4 asks you why the virtual sequencer holds each sink's `out_cfg` and not only its sequencer.

Configuration changes and the scoreboard

With a virtual sequence it's easy to change the configuration while traffic is moving, and then the scoreboard has to apply each change from the right cycle. In Lab 04 the tests wrote the routing table only between phases, so the scoreboard could apply a write as soon as it saw it. In Lab 05 a route can change between one header and the next, and the specification says that a header accepted in the same cycle as the write completes still uses the old route. Lab 05's scoreboard stamps each write with the cycle it completed in, and Part 4 asks you why it uses the cycle rather than the order in which the two monitors report.

On Verilator 5.050, making a sequence and starting it both change the random values drawn after them, in the thread that does it and in other threads too. A sequence added at the start of a test therefore changes every random choice made after it, even if the sequence itself does nothing, so it matters where in a test you put a new one. In Lab 06, Part 1 you'll see this happen to `router_test`, and Lab 07 puts its new sequence at the end of `router_test` for this reason. Regressions and seeds has the measurements.

Check yourself

1. In step 1, the release could have gone in a branch that waits for the FIFO's level to reach 4, instead of waiting 15 cycles. Why can't it simply follow `send(6, 8'h10)` in the same branch?

   Answer: A1, in Answers at the end.

2. What does `join` wait for in step 2, and what would happen if the traffic branch forgot to set `traffic_done`?

   Answer: A2, in Answers at the end.

3. You want the four beats from `80` to go in before the four from `a0`, with both sends still started by the virtual sequence. What do you change?

   Answer: A3, in Answers at the end.

Further reading

- UVM Virtual Sequencer: declaring one, making it in the environment and filling in its handles in `connect_phase`. Its log was made from a different version of the code than the page shows, so the names in it don't match.
- UVM Virtual Sequence: both styles, `p_sequencer` and `start(null)`. Correction: two of its three example virtual sequences create their child sequences in `pre_body()` and start them without a parent, against the page's own advice.
- UVM Sequence Arbitration: how a sequencer chooses between sequences that run on it at the same time, mode by mode, with logs.

- UVM Lock & Grab: how one sequence keeps a sequencer to itself.
- How to execute sequences via start(). Its example test starts a sequence without raising an objection, which works only because that sequence takes no simulation time.

Next: Lab 05: Protocol rules and backpressure.

Answers

A1. Check yourself

`send()` returns only when all six beats have gone into the FIFO, and the FIFO holds four. The fifth beat waits for `in_ready`, which comes back only when the sink takes a beat, and the sink takes beats only after `sink(100)`. In the same branch as `send()`, `sink(100)` would never be reached.

A2. Check yourself

For both branches to finish. Without the flag the backpressure loop never ends, so the virtual sequence never returns, the test never drops its objection, and the run goes on until something stops it from outside. In the course that's `make`'s run timeout, and a run that it stops is graded BROKEN.

A3. Check yourself

Start them one after the other instead of in a `fork`: `send(4, 8'h80); send(4, 8'hA0);`. The second sequence then starts only when the first has returned.

21. Lab 05: Protocol rules and backpressure

In this lab you'll write assertions for the router's output rules and bind them into the router, finish a sink driver whose sequence chooses, for each output beat, how long the sink waits before taking it, and write a virtual sequence that changes the routing table, the traffic and the backpressure together. You start from the Lab 04 testbench, which already catches two of this lab's three planted bugs, B10 and B12. Your assertions name the rule that the router broke and the cycle it broke it in. The third bug, B11, shows only on packets of more than 8 beats, which no Lab 04 test sends, and your virtual sequence sends them.

Package: `labs/lab-05-protocol-rules-and-backpressure/`

Before you start

- Finish Lab 04.
- Read Assertions for interface rules, Reactive agents and backpressure and Virtual sequences.
- Keep the router specification open in another tab, at Valid/ready rules. The rule names on this page (I1, I2, O1, O2, O6, O7 and O8) are the ones defined there.

The package is Lab 04's testbench with our solution to Lab 04 in it: the second output agent, the scoreboard lines for output 1, the real `level()`, `router_stall_test`, `router_noroute_pkt` and the factory override. If your own Lab 04 printed `2 of 2 caught.`, your code did the same job, but use ours anyway, because your Lab 04 edits were in `router_env_pkg.sv` and `router_tests_pkg.sv`, and this lab adds to both files. What we added for you:

File	What Lab 05 adds
`tb/router_sva.sv`, `tb/router_sva_bind.sv`	the assertion module and the `bind` that puts it in the router; you write three of its assertions (Part 2)
`tb/router_ifs.sv`, `tb/router_tb_top.sv`	`router_sva_if`, which carries the assertions' failure count to the scoreboard
`tb/router_agents_pkg.sv`	an output agent whose driver can answer each beat from a sequence (you finish it in Part 3); an oversize length kind in `router_pkt`, switched off until a sequence gives it a weight; how long each output beat waited, which the output monitor records; the cycle in which each APB transfer completed; and `gap_pct` and `w_over` on `router_rand_seq`, which `send_rand` sets (Part 4)
`tb/router_env_pkg.sv`	`cg_wait`, a virtual sequencer, and three scoreboard changes: it reports the assertions' failure count, it follows route writes made while packets are coming in, and it fails a test that checked nothing
`tb/router_vseq_pkg.sv`	the virtual sequences; you write `router_lab05_vseq` (Part 4)
`tb/router_tests_pkg.sv`	`router_test` (Lab 04's `router_inject_test`, then your virtual sequence), `router_lab05_test` (the sequence alone) and `router_sink_test` (Part 3)

Time: about 2 h. On the course's 2 CPUs our first build took 106 s, and the rebuilds after the edits in Parts 2, 3 and 4 took 103 s, 32 s and 105 s.

Part 1: The Lab 04 testbench against this lab's bugs

Which of this lab's bugs does the testbench you already have catch, and how close does it get to the cycle where the router went wrong?

1. Go to the lab folder and start the build:

   ```
   cd labs/lab-05-protocol-rules-and-backpressure
   make build
   ```

   The banner says a cold build takes 1.5 to 2.5 minutes on 2 CPUs, and ours took 106 s. Near the end it prints Verilator's report:

   ```
   - V e r i l a t i o n   R e p o r t: Verilator 5.050 2026-07-01 rev v5.050
   - Verilator: Built from 59.791 MB sources in 431 modules, into 27.944 MB in 3213 C+
   + files needing 198.696 MB
   - Verilator: Walltime 106.395 s (elab=0.811, cvt=3.129, bld=101.669); cpu 5.091 s
   on 2 threads; allocated 355.551 MB
   ```

2. Run the graded test, `router_test`. It runs everything `router_inject_test` did in Lab 04 and then `router_lab05_vseq`, which is empty for now:

   ```
   make run
   ```

   You should see `ok: 0 UVM_ERROR, 0 UVM_FATAL  1.72 s`, and above it:

   ```
   UVM_INFO  labs/lab-05-protocol-rules-and-backpressure/tb/router_env_pkg.sv(257)  @
   61465000: uvm_test_top.env.sb [SB] 805 packets in: 144 to output 0, 190 to output
   1, 194 NOROUTE, 277 FULL; the highest level a header saw was 16
   UVM_INFO  labs/lab-05-protocol-rules-and-backpressure/tb/router_env_pkg.sv(259)  @
   61465000: uvm_test_top.env.sb [SB] beats checked: 648 on output 0, 799 on output 1;
   5 DROP_CNT reads checked
   UVM_INFO  labs/lab-05-protocol-rules-and-backpressure/tb/router_env_pkg.sv(263)  @
   61465000: uvm_test_top.env.sb [SVA] the bound assertions (router_sva.sv) counted
   0 failures
   ```

   The two `[SB]` lines are the ones `router_inject_test` printed at the end of Lab 04, Part 4, number for number, so the stimulus is unchanged. (The line numbers after `router_env_pkg.sv` differ, because the file has grown.) The `[SVA]` line is new.

3. Look at how long the output beats waited. `cg_wait` records, for each packet an output delivers, the longest any of its beats waited for `ready`: `none` is 0 cycles, `short_wait` 1 to 3, `long_wait` 4 to 20 and `stalled` more than 20. `cp_len.oversize` counts packets of 9 beats or more on the input, before the router cuts them.

   ```
   make cov
   grep -E "cg_wait.cp_wait|cp_len.oversize" build/coverage/bins.txt
   ```

   You should see:

   ```
             0  __vlAnonCG_cg_pkt.cp_len.oversize
           257  __vlAnonCG_cg_wait.cp_wait.none
             2  __vlAnonCG_cg_wait.cp_wait.short_wait
            18  __vlAnonCG_cg_wait.cp_wait.long_wait
            57  __vlAnonCG_cg_wait.cp_wait.stalled
   ```

   Lab 04's sink flips a coin every cycle. At 100% a beat never waits, and at 5% a beat waits about 20 cycles on average, so a packet whose longest wait is 1 to 3 cycles is rare.

4. Run the grader against this lab's three bugs:

```
make check-all
```

You should see:

```
     ok  the golden design                      0 UVM_ERROR, 0 UVM_FATAL       1.7s
-------------------------------------------------------------------------------
  MISSED  tier 4  B11_cut_packet_has_no_last      0 UVM_ERROR, 0 UVM_FATAL      1.7s
  caught  tier 4  B10_output_reloads_while_stalled 50 UVM_ERROR                 0.5s
  caught  tier 5  B12_out1_uses_out0_ready        50 UVM_ERROR                  0.1s
-------------------------------------------------------------------------------
 2 of 3 caught.
 Still hiding from this test:
   B11_cut_packet_has_no_last
      hides behind: sending only legal packets of 8 beats or fewer
```

`make check`, which grades the first bug in the list, says `VERDICT: FAIL -- your test passes B11_cut_packet_has_no_last too.` Lab 04's stall test already makes the two `out_ready` signals differ and lets beats wait on a stalled output, and that is all B12 and B10 need. B11 needs a packet of more than 8 beats, and none of Lab 04's tests sends one.

5. Look at the first errors for the two bugs the testbench catches:

```
make run BUG=B10
make run BUG=B12
```

You should see, from the first:

```
caught: 50 UVM_ERROR   0.54 s   log: build/runs/router_test-s1-B10/run.log
  UVM_ERROR [OUT] output 0: got 04 in cycle 56, expected 03
```

and from the second:

```
caught: 50 UVM_ERROR   0.08 s   log: build/runs/router_test-s1-B12/run.log
  UVM_ERROR [OUT] output 1: got 80 in cycle 24, and no beat was expected
```

6. Predict a cycle before you go on. In phase 1 output 0 is ready in 5% of cycles. The router accepts the 8-beat packet `03 a4 ab 49 70 74 f5 71` in cycles 45 to 52, into an empty output 0, and output 0's sink is first ready again in cycle 56. By rule O8, in which cycle is `03` first on `out0_data`? B10 loads the next beat into the output register even while `out0_ready` is 0. In which cycle does `out0_data` first change while `03` waits? Write that cycle down; Part 2 checks it.

   Answer: A1, in Answers at the end.

Check: `make check-all` printed `2 of 3 caught.`, and you've written down a cycle for B10.

Part 2: Assertions for the output rules

Which rule did B10 break, and in which cycle?

1. Open `tb/router_sva.sv`. The module `router_sva` has two assertions written for you, `a_I1` and `a_I2`, for the source's rules I1 and I2, so they check the testbench's own source driver. Read `a_I1` with the specification's I1 beside it. The part left of `|=>` (the antecedent) picks the cycles the rule talks about, and `|=>` means "then, in the next cycle". `disable iff (!rst_n)` switches the check off while `rst_n` is 0, because a reset ends a beat without a handshake and that doesn't break the rule. A failure adds one to `n_I1` and calls `$error`.
2. In `router_sva_out`, which is instantiated once per output, write the three assertions the TODOs describe:
   - `a_O1`: once `out_valid` is 1, it stays 1 until the handshake;
   - `a_O2`: while `out_valid` is 1 and `out_ready` is 0, `out_data` and `out_last` don't change;

- `a_O6`: a beat taken when `beats` is 7 has `out_last` = 1. (`beats` counts the beats of the current output packet already taken, and the file keeps it for you.) This one is about the cycle of the handshake itself, so it uses `|->` (the same cycle).

Count each one's failures in its own counter (`n_O1`, `n_O2`, `n_O6`) and give each one an `$error` message, as `a_I1` does. The outputs on this page show our messages. For O1 and O2 they are `a_I1`′s and `a_I2`′s with `O1` and `O2` in place of `I1` and `I2`, and `out_` in place of `in_`. For O6 it is `O6: an output packet is longer than 8 beats`.

3. Open `tb/router_sva_bind.sv`. Why does the testbench attach the assertions with `bind` instead of instantiating `router_sva` in `router_tb_top`?

   Answer: A2, in Answers at the end.

4. Rebuild and run on the golden design:

```
make build
make run
```

   The rebuild took us 103 s. You should see `ok`, the same two `[SB]` lines as in Part 1, and `[SVA] the bound assertions (router_sva.sv) counted 0 failures`. A failure here means one of your assertions is wrong, because the golden router keeps all three rules.

5. Run B10 with the simulator's own error limit. `PLUSARGS=` on the command line replaces the Makefile's `+verilator+error+limit+100000`:

```
make run BUG=B10 PLUSARGS=
cat build/runs/router_test-s1-B10/exit_status
```

   You should see the run stop at the first failure, then `1`:

```
[475000]      %Error:      router_sva.sv:50:      Assertion      failed      in
router_tb_top.dut.u_sva.u_out0.a_O2: O2: out_data or out_last changed while the
beat waited
%Error: labs/lab-05-protocol-rules-and-backpressure/tb/router_sva.sv:50: Verilog
$stop
Aborting...
-------------------------------------------------------------------------
 caught: 1 assertion   0.07 s   log: build/runs/router_test-s1-B10/run.log
    router_sva.sv:50: Assertion failed in router_tb_top.dut.u_sva.u_out0.a_O2: O2:
out_data or out_last changed wh
-------------------------------------------------------------------------
```

   The line number is the line of the `$error` call in your `a_O2`, so yours may differ, here and later. The time, 475000, is in picoseconds, so the failure was at 475 ns. The clock's period is 10 ns and its first rising edge is at 5 ns, so the edge that ends cycle n is at $10n - 5$ ns, and this failure was sampled at the edge that ends cycle 48. Compare it with your prediction from Part 1.

   Verilator's default error limit is 1, so the first failing assertion ends the run with exit status 1, and nothing after it is checked. That's why the Makefile raises the limit. With the limit raised, every failure prints, each assertion's counter keeps counting, and the scoreboard reports their sum at the end of the test.

6. Run B10 again with the Makefile's limit, and find the first check of each kind that fired:

```
make run BUG=B10
grep -m1 "%Error" build/runs/router_test-s1-B10/run.log
grep -m1 "UVM_ERROR" build/runs/router_test-s1-B10/run.log
```

   You should see `caught: 50 UVM_ERROR, 244 assertions`, then:

```
[475000]       %Error:       router_sva.sv:50:       Assertion       failed       in
router_tb_top.dut.u_sva.u_out0.a_O2:  O2:  out_data  or  out_last  changed  while  the
beat waited
UVM_ERROR  labs/lab-05-protocol-rules-and-backpressure/tb/router_env_pkg.sv(193)  @
555000: uvm_test_top.env.sb [OUT] output 0: got 04 in cycle 56, expected 03
```

The assertion fired at the edge that ends cycle 48, and the scoreboard reported its first error 8 cycles later, in cycle 56.

7. To see the rule break, record both runs and open them in Surfer:

```
make waves
make waves BUG=B10
surfer build/runs/router_test-s1/waves.fst &
surfer build/runs/router_test-s1-B10/waves.fst &
```

Add `valid`, `ready`, `data` and `last` from `in_s` and `out0`, `level0` from `dut`, and `n_O2` from `dut.u_sva.u_out0`, and look at 435 to 575 ns:

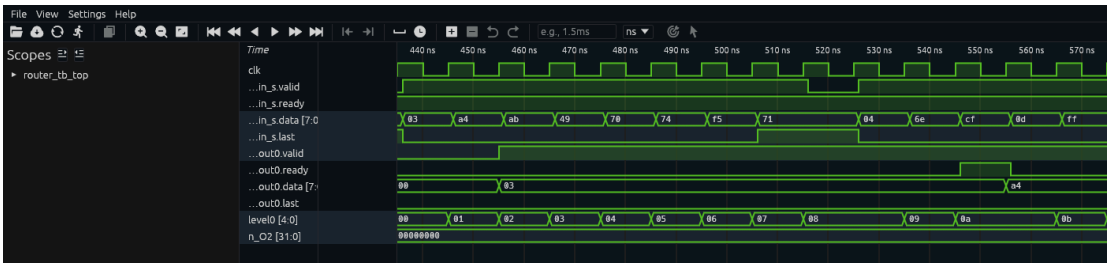


Figure 21.1: The golden router in phase 1 of router_test, in Surfer

The golden router. The packet `03 a4 ab 49 70 74 f5 71` comes in one beat per cycle from the edge at 445 ns. `03` reaches `out0_data` at the edge at 455 ns, in cycle 47, and stays there while `out0_ready` is 0; `level0` climbs as the rest of the packet arrives. `out0_ready` is 1 in cycle 56, the sink takes `03` at the edge at 555 ns, and `a4` follows. `n_O2` stays 0.

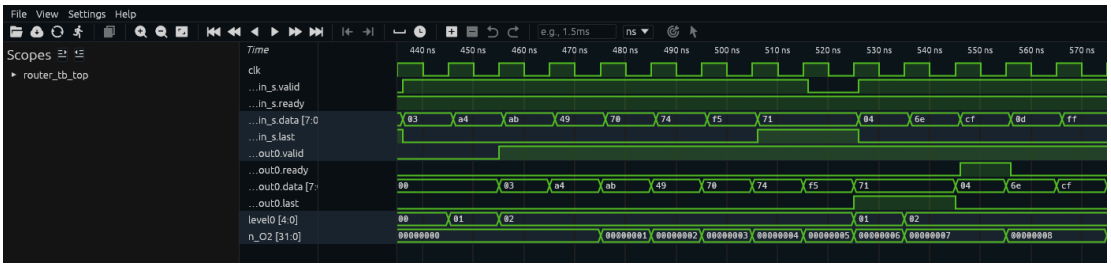


Figure 21.2: The same window with B10 on, in Surfer

B10. `out0_data` changes from `03` to `a4` at the edge at 465 ns, while `out0_ready` is 0, and then steps through the rest of the packet, one beat per cycle. `n_O2` counts its first failure at the edge at 475 ns, which ends cycle 48, and one more for nearly every cycle after. When `out0_ready` rises in cycle 56, the beat on the output is `04`.

8. Run the grader again:

```
make check-all
```

You should see B10 caught by your assertions as well, B12 with one assertion, and B11 still missed:

```
  MISSED  tier 4  B11_cut_packet_has_no_last      0 UVM_ERROR, 0 UVM_FATAL       1.7s
  caught  tier 4  B10_output_reloads_while_stalled 50 UVM_ERROR, 244 assertions   0.6s
  caught  tier 5  B12_out1_uses_out0_ready        50 UVM_ERROR, 1 assertion      0.1s
-----------------------------------------------------------------------
 2 of 3 caught.
```

The one assertion on B12 is `a_O6`, on output 1, at the edge at 635 ns (cycle 64); the scoreboard's first error was in cycle 24. Run `grep -m1 "%Error" build/grade/B12/run.log` to see it. B11 is the bug that breaks O6, but your `a_O6` passes on B11 and fails on B12. Why?

Answer: A3, in Answers at the end.

Check: `make check-all` prints `50 UVM_ERROR, 244 assertions` for B10, and your answer to Part 1's prediction matches cycle 48.

Part 3: A sink that answers each beat

What changes when each sink takes its beats at its own pace, instead of flipping a coin every cycle?

1. Read the output agent in `tb/router_agents_pkg.sv`: `out_cfg`, `out_ready_item`, `out_ready_seq`, `out_ready_driver` and `out_agent`. `out_cfg.mode` picks how the sink answers. In `SNK_PCT`, the default, the driver is Lab 04's: every cycle `out_ready` is 1 with probability `ready_pct` percent. In the other modes it is a reactive driver. The router decides when a beat appears, and the driver answers each beat with an item from `out_ready_seq`: with `pre_ready` set, `ready` is already 1 when the beat appears; otherwise the sink waits `delay` cycles after the beat's first cycle on offer, then raises `ready`. `SNK_FAST` makes every item `pre_ready`, `SNK_SLOW` gives every beat a delay of 4 to 20, and `SNK_RANDOM` randomizes each item. The driver looks at the output at the falling edge in the middle of each cycle and drives the next cycle's `ready` just after the rising edge; that loop is written for you. Why does it ask for items with `try_next_item()` and not `get_next_item()`, as Lab 04's source driver does?

   Answer: A4, in Answers at the end.

2. `router_sink_test` is Lab 03's `router_rand_test` with both sinks in `SNK_RANDOM`. Run it as shipped, on its own, and look at its waits:

```
rm -rf build/runs
make run UVM_TESTNAME=router_sink_test
make cov
grep -E "cg_wait.cp_wait" build/coverage/bins.txt
```

   You should see `ok` in about 4 s, and:

```
UVM_INFO  labs/lab-05-protocol-rules-and-backpressure/tb/router_env_pkg.sv(257)  @
57695000: uvm_test_top.env.sb [SB] 805 packets in: 353 to output 0, 451 to output
1, 1 NOROUTE, 0 FULL; the highest level a header saw was 9
```

```
       803  __vlAnonCG_cg_wait.cp_wait.none
         0  __vlAnonCG_cg_wait.cp_wait.short_wait
         0  __vlAnonCG_cg_wait.cp_wait.long_wait
         1  __vlAnonCG_cg_wait.cp_wait.stalled
```

   The `[SB]` line is the one `router_rand_test` printed in Lab 04, Part 2, and 803 of the 804 packets that came out never waited, because as shipped the driver is always ready in the per-beat modes, whatever the item says. The one `stalled` packet is from `close_holes()`, which stalls output 0 in Lab 04's mode, `SNK_PCT`.

3. Fill in the two TODOs in `out_ready_driver::run_phase`: the countdown while a beat waits, and the expression for `rdy`, the next cycle's `ready`. `rdy` must be 0 while the driver has no item. The countdown has to stop at 0. A beat whose wait is over can still be on offer if a virtual sequence then stalls its output (Part 4), and the countdown goes on running in every cycle the beat waits. `wait_left` is unsigned, so

one more count would wrap it round to about four billion, and once the sink was back in a per-beat mode it would never raise `ready` for that beat.

4. Before you rebuild, predict what happens to the `[SB]` line's `FULL` count and to the four `cp_wait` bins.

   Answer: A5, in Answers at the end.

5. Rebuild, clear the runs and look again:

```
make build
rm -rf build/runs
make run UVM_TESTNAME=router_sink_test
make cov
grep -E "cg_wait.cp_wait" build/coverage/bins.txt
```

   The rebuild took us 32 s. If your driver does what the TODOs say, you should see `ok` and:

```
UVM_INFO  labs/lab-05-protocol-rules-and-backpressure/tb/router_env_pkg.sv(257)  @
58285000: uvm_test_top.env.sb [SB] 805 packets in: 317 to output 0, 401 to output
1, 1 NOROUTE, 86 FULL; the highest level a header saw was 13
```

```
       129  __vlAnonCG_cg_wait.cp_wait.none
       305  __vlAnonCG_cg_wait.cp_wait.short_wait
       283  __vlAnonCG_cg_wait.cp_wait.long_wait
         1  __vlAnonCG_cg_wait.cp_wait.stalled
```

   Lab 04's sinks put 2 packets in `short_wait` in the whole of `router_test`, and your driver puts 305 there in one run of `router_sink_test`. The run passed, so every phase's `DROP_CNT` read matched the scoreboard's count, which means the scoreboard predicted all 86 FULL drops.

6. Run the grader:

```
make check-all
```

   You should see `2 of 3 caught.` again, with the same counts as in Part 2. `router_test` keeps its sinks in `SNK_PCT`, so your driver changes none of its stimulus, and B11 still needs a packet longer than 8 beats.

Check: `router_sink_test` passes with `305` in `short_wait`, and `make check-all` says `2 of 3 caught.`

Part 4: A virtual sequence

How do you make the configuration, the traffic and the backpressure change together, from one place?

1. Read `router_vsequencer` in `tb/router_env_pkg.sv`, then `router_base_vseq` and `router_lab05_vseq` in `tb/router_vseq_pkg.sv`. The helpers you'll use are `route(n, en, port)`, `ctrl(en)`, `sink(p, mode)`, `stall(p)`, `burst(dst, n, len)`, `send_rand(n, gap_pct, w_over)`, `backpressure()`, `cycles(n)`, `drain()` and `rd(A_...)`. Most of them are the base test's steps from Lab 04, rewritten as tasks of a sequence, and they start the same Lab 04 sequences that the steps started. Why does the virtual sequencer hold each sink's `out_cfg`, and not only its sequencer?

   Answer: A6, in Answers at the end.

2. Run the test that runs your sequence alone, as it is now:

```
make run UVM_TESTNAME=router_lab05_test
```

   You should see:

```
 FAIL: 1 UVM_ERROR   0.07 s   log: build/runs/router_lab05_test-s1/run.log
   UVM_ERROR [SB] no packet came in and no DROP_CNT read was checked, so nothing
was checked
```

The scoreboard fails a test that checked nothing. `make regress`, which runs every test in the Makefile, fails for the same reason until you finish this part.

3. Read `write_apb` and the start of `write_in` in the scoreboard. In Lab 04 a route write changed the scoreboard's routing table at once, which was safe because the tests wrote routes only between phases. Now `write_apb` puts the write in `pending` with the cycle it completed, and `write_in` applies it at the first header of a later cycle. Why the cycle, and not the order in which the two calls arrive?

   Answer: A7, in Answers at the end.

4. Write `router_lab05_vseq::body()`. The TODO lists four steps:

   a. Configure the router: every route enabled, two DSTs to each output, and `CTRL.EN` set.
   b. Stall output 1 and send packets to both outputs, then stall output 0 instead, with a `drain()` after each.
   c. Send oversize packets (9 beats or more) to every DST with both sinks in `SNK_RANDOM`, then drain.
   d. In one `fork`, run random packets with oversize ones among them (`send_rand`'s `w_over`), `backpressure()`, and a branch that moves two routes to the other output and back while the packets are in flight. Then drain, and read `DROP_CNT`.

   Step 4 has this shape. `backpressure()` loops until `traffic_done` is set:

```
traffic_done = 0;
fork
  begin
    send_rand(...);          // the packets
    traffic_done = 1;
  end
  backpressure();            // each sink changes its mode on its own
  begin
    ...                      // route changes, with cycles() between them
  end
join
```

5. Predict how each oversize packet should come out, and which bug your sequence now reaches.

   Answer: A8, in Answers at the end.

6. Rebuild, then run both tests and look at the coverage:

```
make build
rm -rf build/runs
make run UVM_TESTNAME=router_lab05_test
make run
make cov
grep -E "cg_wait.cp_wait|cp_len.oversize" build/coverage/bins.txt
```

   The rebuild took us 105 s. You should see `ok` from both runs and `oversize` above 0. The counts depend on your sequence. Ours were 77 for `oversize`, over the two runs, and `router_test` said:

```
UVM_INFO  labs/lab-05-protocol-rules-and-backpressure/tb/router_env_pkg.sv(257)  @
87215000: uvm_test_top.env.sb [SB] 1122 packets in: 240 to output 0, 270 to output
1, 194 NOROUTE, 418 FULL; the highest level a header saw was 16
UVM_INFO  labs/lab-05-protocol-rules-and-backpressure/tb/router_env_pkg.sv(259)  @
87215000: uvm_test_top.env.sb [SB] beats checked: 1108 on output 0, 1219 on output
1; 6 DROP_CNT reads checked
```

   The first 805 packets are Lab 04's, as in Part 1, and our sequence sent the other 317.

7. Run the grader:

```
make check-all
```

   You should see `3 of 3 caught.` Ours:

```
     ok  the golden design                      0 UVM_ERROR, 0 UVM_FATAL       2.8s
-------------------------------------------------------------------------------
  caught  tier 4  B11_cut_packet_has_no_last     33 UVM_ERROR, 26 assertions    2.8s
  caught  tier 4  B10_output_reloads_while_stalled 50 UVM_ERROR, 244 assertions   0.6s
  caught  tier 5  B12_out1_uses_out0_ready       50 UVM_ERROR, 1 assertion      0.1s
-------------------------------------------------------------------------------
 3 of 3 caught.
 Every bug in the list, each one caught by a check that fired.
```

On B11 our `a_O6` failed at the edge that ends cycle 6265, and the scoreboard reported the same beat in the same cycle (`output 0: got a7 in cycle 6265, expected a7 last`). At the end of the run the scoreboard added `[SVA] the bound assertions (router_sva.sv) counted 26 failures`. What did O6 add that the scoreboard's error didn't already give you?

Answer: A9, in Answers at the end.

8. Now run B12 against your sequence alone, which stalls output 1 first, and compare the first check of each kind:

```
make run UVM_TESTNAME=router_lab05_test BUG=B12
grep -m1 "%Error" build/runs/router_lab05_test-s1-B12/run.log
grep -m1 "UVM_ERROR" build/runs/router_lab05_test-s1-B12/run.log
```

Ours said `caught: 50 UVM_ERROR, 34 assertions`, and:

```
[395000]      %Error:      router_sva.sv:50:      Assertion      failed      in
router_tb_top.dut.u_sva.u_out1.a_O2: O2: out_data or out_last changed while the
beat waited
UVM_ERROR labs/lab-05-protocol-rules-and-backpressure/tb/router_env_pkg.sv(193) @
3605000: uvm_test_top.env.sb [OUT] output 1: got 46 in cycle 361, expected 44
```

Our O2 failed in cycle 40 and the scoreboard complained in cycle 361. In `router_test` the scoreboard was first. Why is it so much later here?

Answer: A10, in Answers at the end.

9. Check that your testbench still passes the golden design on other seeds:

```
make seeds SEEDS="1 2 3 4 5"
```

You should see `5 ok  (5 runs)`.

Finishing

You're done when `make check-all` prints `3 of 3 caught.` with your assertions, your sink driver and your virtual sequence in `tb/`, and `make seeds SEEDS="1 2 3 4 5"` prints `5 ok  (5 runs)`.

Questions:

1. B12 breaks rule O7, because it makes output 1 depend on `out0_ready`. Your file has no `a_O7`. Which checks caught B12, and why is there no assertion for O7?

   Answer: A11, in Answers at the end.

2. Your O1 and O2 are disabled while `rst_n` is 0. What would O1 do on the golden design, without `disable iff`, in a test that resets the router while a beat waits on a stalled output?

   Answer: A12, in Answers at the end.

Submit: nothing for this lab.

Next on the Course path: read Register models and Regressions and seeds, then do Lab 06: The register block.

Answers

A1. Part 1: The Lab 04 testbench against this lab's bugs

O8 puts a beat into an empty output two cycles after its input handshake, so `03` is on the output from cycle 47. B10 loads `a4` over it at the next edge, so `out0_data` first changes in cycle 48. The scoreboard sees only beats that are taken, and output 0's sink took nothing until cycle 56, by which time the router had moved through the whole packet and on to `04`, the next header. The router went wrong 8 cycles before the scoreboard could see anything.

A2. Part 2: Assertions for the output rules

`bind` puts an instance of `router_sva` inside every `router`, wired to the router's own ports, without editing the router's RTL or the top module. The same file works on the golden design and with any planted bug switched on, and the checker sees the same pins that the rules are written about. The top module reads the failure count back out through the router instance, `dut.u_sva.n_fail`, and hands it to the scoreboard through `router_sva_if`.

A3. Part 2: Assertions for the output rules

An assertion can only fail on something that the stimulus makes the router do. Under B11, O6 can fail only on the 8th beat of a packet that has a 9th, and nothing in `router_test` sends a packet longer than 8 beats yet, so on B11 the router never has a packet to cut.

B12 moves output 1 on `out0_ready`. In phase 1 output 0 is ready in 5% of cycles and output 1 in all of them, so in most cycles output 1's sink takes the beat on its pins, but the buffer behind them doesn't move on to the next beat. The pins show a handshake in each of those cycles with the same data, which is what the scoreboard saw in cycle 24 (`got 80 ... and no beat was expected`). Each repeat is one more beat of the packet on output 1's pins, so once eight beats in a row have had no `out_last`, O6 fails. O1 and O2 are about cycles where `out_ready` is 0, and `out1_ready` is 1 all through phase 1; the run stops at 50 errors in cycle 73, long before phase 2 stalls output 1.

A4. Part 3: A sink that answers each beat

`get_next_item()` blocks until the sequence hands over an item. The source driver can wait, because nothing happens on the input until it has a packet. The sink driver can't wait like that, because it has to drive `out_ready` in every cycle, and the router may put a beat on the output whether or not an item is ready. `try_next_item()` returns in the same time step, with no item if the sequence hasn't handed one over, and the driver then keeps `ready` at 0 for the next cycle.

A5. Part 3: A sink that answers each beat

Each beat now waits 0 cycles when its item is `pre_ready` and otherwise 1 to 13, so most packets land in `short_wait` or `long_wait`. A sink that waits a cycle or more on most beats takes them more slowly than the input can bring them, so an output sometimes fills past 8 beats, and the packets that arrive for it then are FULL drops.

A6. Part 4: A virtual sequence

Each sink's `out_ready_seq` is started by its agent and runs for the whole test, one item per beat. A virtual sequence steers the backpressure by changing `mode` and `ready_pct` in `out_cfg`. The sequence reads `mode` for every item, and the driver reads `mode` every cycle, and `ready_pct` too while the mode is `SNK_PCT`. Starting another sequence on a sink's sequencer would compete with the one already running there.

A7. Part 4: A virtual sequence

The specification's Routing section says a header accepted in the cycle a register write completes still uses the old value. The APB monitor and the input monitor both report at the same clock edge, and SystemVerilog leaves the order of processes woken by the same edge to the simulator, so a header and a route write in the same cycle could reach the scoreboard in either order. The cycle stamps don't depend on that order.

A8. Part 4: A virtual sequence

Cut to 8 beats, with `out_last` on the 8th; the router takes the rest from the input and throws them away (example E6 in the specification). With B11, the 8th beat has no `out_last`, so O6 fails on it, and the scoreboard, which already expected the cut, sees the wrong `out_last` on the same beat.

A9. Part 4: A virtual sequence

O6 names the rule, and it needs no model of the router. It checks one line of the specification against the pins, so it would catch B11 in any testbench that binds the file and sends an oversize packet, including one whose scoreboard didn't know that the router cuts long packets. The scoreboard knows what every beat should be, so it catches far more, but its message gives only the beat that was wrong and what it expected instead.

A10. Part 4: A virtual sequence

Step 2 of the sequence stalls output 1 while output 0 takes its beats. Under B12 output 1′s buffer moves on `out0_ready`, so its beats leave while `out1_ready` is 0. None of them has a handshake on output 1′s pins, so its monitor reports nothing. By the time the stall ends, every beat routed to output 1 has left its buffer, so output 1′s sink has nothing to take either, and `drain()` waits its full 300 cycles, because the scoreboard still expects those beats. Output 1′s first handshake of the run is in cycle 361, when the second half of step 2 sends it new packets, and that is the scoreboard's first chance to compare a beat on output 1. O2 watches the pins, so it fails in the first cycle the beat on output 1 changes while it waits. In `router_test`'s phase 1 the readies differed the other way round, output 1 ready and output 0 stalled, and then the scoreboard sees the extra beats at once.

A11. Finishing

The scoreboard in `router_test`, from cycle 24, and O6 on output 1 in cycle 64; in `router_lab05_test`, O2 and O1 on output 1 first and the scoreboard much later (Part 4, step 8). O7 says what an output does not depend on, and the specification points out that no single trace can show that. You can only see its consequences, when the two `out_ready` signals differ and output 1 then does something that O1, O2, O6 or the scoreboard's model forbids. Lab 04′s stall test was the first test to make them differ while both outputs carried traffic, which is why your Lab 04 testbench already caught B12.

A12. Finishing

It would fail. A reset lowers `out_valid` at once, with no handshake (the specification's Valid/ready rules and example E8), and that is what O1′s consequent forbids. Rules about normal operation don't hold across a reset, so their assertions are switched off during one.

VI 6. Registers

22 **Register models 163**
22 The example block 163
22 Fields, registers and the map 163
22 From a register operation to a transfer 164
22 Auto-predict and the predictor 164
22 Volatile fields 165
22 Desired and mirrored values 165
22 The built-in sequences 166
22 The back door 166
22 Check yourself 167
22 Further reading 167
22 Answers 167

23 **Regressions and seeds 169**
23 What a seed fixes 169
23 One more random call 169
23 Running many seeds 170
23 Corners no seed reaches 171
23 Check yourself 171
23 Further reading 172
23 Answers 172

24 **Lab 06: The register block 173**
24 Before you start 173
24 Part 1: The Lab 05 testbench against the registers 174
24 Part 2: The register model 175
24 Part 3: The predictor and the bit bash 176
24 Part 4: The addresses outside the map 177
24 Part 5: The interrupt 178
24 Part 6: The back door 179
24 Finishing 181
24 Going further (optional) 181
24 Answers 181

22. Register models

This page answers: *what does a UVM register model know about a design's registers, and what does it do for you that raw bus transfers don't?*

Up to the end of Lab 05, your testbench reached the router's registers with raw APB transfers, such as `wr(A_ROUTE0, ...)` to set up a route and `rd(A_DROP_CNT)` to read the two drop counters. The scoreboard checked reads of one register, `DROP_CNT`, and nothing in the testbench read any of the other nine. So the testbench had no way to notice a register that reset to the wrong value, or one that read back with its fields swapped. Lab 06 plants one bug of each of those kinds among its five, and your Lab 05 testbench misses all five. In that lab you'll give the testbench a **register model**, which is a set of UVM classes that describes every register and field as the specification's Register map does. The model can also turn an operation such as "write 1 to `CTRL.EN`" into a bus transfer.

We wrote a small register block for this page, `regblk`, and a testbench for it with a register model, an adapter and a predictor. Both are in `guide-examples/register-models/`, and the sections below use them to show how those parts work together. The UVM register layer reference page has the arguments, policies and messages you'll look up while you work.

The example block

`regblk` has four 16-bit registers on a simple bus. Each transfer takes one cycle, in which `sel` is 1. The block answers in that same cycle, and a write takes effect at the rising edge that ends it.

Address	Register	Access	Fields
`0x0`	`ID`	RO	[15:0], reads `105A`
`0x4`	`CTRL`	RW	[0] `EN`, [3:1] `MODE`; resets to 0
`0x8`	`FLAGS`	W1C	[0] `OVF`, [1] `ERR`; set by the design when an event happens
`0xC`	`COUNT`	RO	[7:0], the number of events since reset

Apart from a reset, only a write changes `CTRL`, so it holds whatever software last wrote to it, which here means the testbench. `FLAGS` and `COUNT` change on their own, whenever the design sees an event. A register model has to treat the two kinds of register differently, and the sections on volatile fields and on the built-in sequences show how.

Fields, registers and the map

Each register is a class, and its `build()` creates the register's fields and configures each one. This is `CTRL`'s:

```
virtual function void build();
  en   = uvm_reg_field::type_id::create("en");
  mode = uvm_reg_field::type_id::create("mode");
  en.configure  (this, 1, 0, "RW", 0, 0, 1, 1, 0);
  mode.configure(this, 3, 1, "RW", 0, 0, 1, 1, 0);
endfunction
```

The arguments after `this` are, in order, the field's width, its lowest bit, its access policy, whether it's **volatile** (changed by the design on its own), its reset value, whether it has a reset value, whether `randomize()` may choose it, and whether it's alone in its register. Both fields of `FLAGS` are configured as

"W1C" with volatile set to 1, and the field in COUNT as "RO", also with volatile set to 1. The register block creates the four registers and puts each one at its address in a map:

```
default_map = create_map("map", 0, 2, UVM_LITTLE_ENDIAN, 1);   // 2 bytes per register
default_map.add_reg(id,    'h0, "RO");
default_map.add_reg(ctrl,  'h4, "RW");
default_map.add_reg(flags, 'h8, "RW");
default_map.add_reg(count, 'hC, "RO");
```

For every field, the model keeps two values. The **mirror** is what the model believes the design holds right now. The **desired** value is what the testbench last asked for. A read with a check, such as `mirror(UVM_CHECK)`, compares what the design returns with the mirror, and `update()` compares the desired value with the mirror to decide whether to write the register. The sections below show both.

From a register operation to a transfer

A test calls `regs.ctrl.write(status, 'h0005)`. The model looks up `CTRL` in the map, finds its address, and hands an abstract operation (a write of `0005` to address `4`) to an **adapter**. The adapter's `reg2bus()` turns that operation into the sequence item that your bus agent uses, and the map starts that item on the agent's sequencer, so it goes through the same driver as every other transfer. When the driver has finished, the adapter's `bus2reg()` turns the item back into an operation, with the read data and a status, which is `UVM_IS_OK`, or `UVM_NOT_OK` if the bus reported an error. Those two functions are the only code in the register layer that knows what your bus looks like, and in Lab 06, Part 2 you'll write them for the router's APB agent.

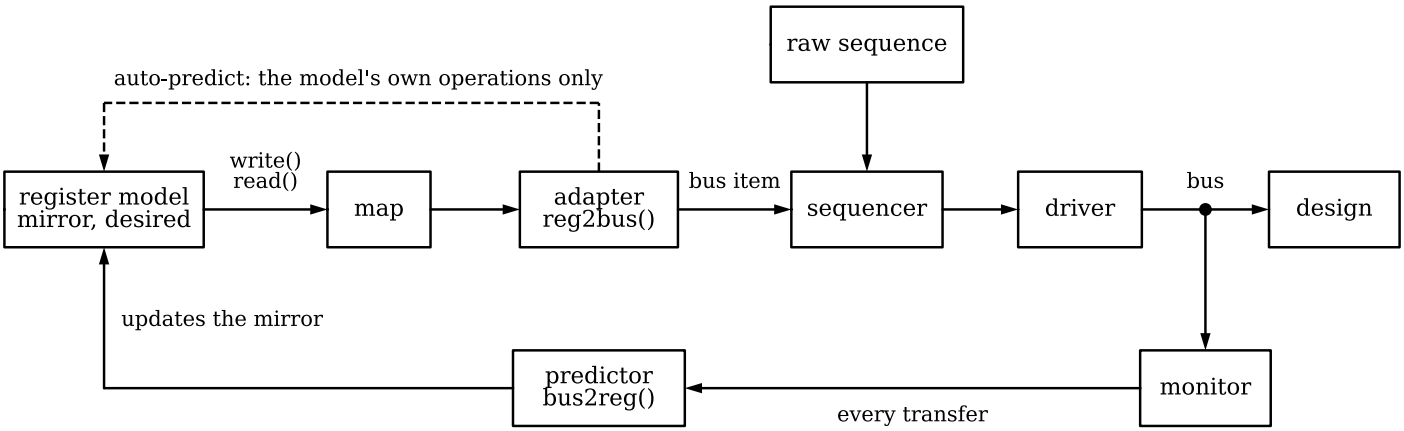


Figure 22.1: The register model and the two ways its mirror can follow the design

Auto-predict and the predictor

The model's mirror follows the bus only if something tells the model about each transfer, and there are two ways to do that. With **auto-predict** on (`set_auto_predict(1)` on the map), the model updates the mirror after each operation that it sends itself. The other way is a **predictor** (`uvm_reg_predictor`), which you connect to the bus monitor's analysis port. It updates the mirror after every transfer the monitor sees, whoever started it.

Our example's `reg_mirror_test` writes `0005` to `CTRL` through the model, then writes `000b` with a raw bus sequence that goes around the model, the way Lab 05's `wr()` does. Then it calls `regs.ctrl.mirror(status, UVM_CHECK)`, which reads `CTRL` and compares the value with the mirror. We ran it three times, with `+PREDICT=` choosing a different way of keeping the mirror each time:

`+PREDICT=`	Mirror after the model's write	After the raw write	`mirror(UVM_CHECK)`
`predictor`	`0005`	`000b`	passes

+PREDICT=	Mirror after the model's write	After the raw write	mirror(UVM_CHECK)
auto	0005	0005	UVM_ERROR
none	0000	0000	UVM_ERROR

With auto-predict, the error was:

```
UVM_ERROR third_party/uvm/src/reg/uvm_reg.svh(2812) @ 95000: reporter [RegModel]
Register "regs.CTRL" value read from DUT (0x000000000000000b) does not match mirrored
value (0x0000000000000005) (valid bit mask = 0x000000000000000f)
UVM_INFO third_party/uvm/src/reg/uvm_reg.svh(2833) @ 95000: reporter [RegModel] Field
mode (regs.CTRL[3:1]) mismatch read=3'h5 mirrored=3'h2
```

In the first line, the valid bit mask has a 1 for each bit that was compared, which here is all four bits of `CTRL`'s two fields. The second line names the one field that differs, `MODE`, whose bits 3 to 1 are `101` (5) in `000b` and `010` (2) in `0005`. The design returned `000b`, the value that the raw write put there, and the mirror still held `0005` because auto-predict never saw that write. `regblk` ran as the golden design, with no bug switched on, and a check that fails on the golden design means the testbench is wrong. This testbench needs a predictor. In Lab 06, Part 3 you connect one to the APB monitor, which the scoreboard already listens to.

Volatile fields

`FLAGS` is set by the design, so the model can't know its value until it reads it. The test pulses the design's `ev_ovf` input, which sets `FLAGS.OVF`, and prints the mirror of `FLAGS` before and after a `mirror(UVM_CHECK)` of it. These lines are from the run with the predictor:

```
UVM_INFO guide-examples/register-models/regblk_tb.sv(366) @ 125000: uvm_test_top [RAL]
after an overflow event, the mirror of FLAGS is 0000
UVM_INFO guide-examples/register-models/regblk_tb.sv(368) @ 145000: uvm_test_top [RAL]
after mirror(UVM_CHECK) reads FLAGS, the mirror is 0001
```

The mirror said `0000` while the design held `0001`, and `mirror(UVM_CHECK)` reported no error, because the register layer doesn't compare a volatile field when it reads one. The predictor saw the read and set the mirror to the value read, which is why the second line says `0001`. UVM also warns you whenever you ask for the mirror of a volatile field. Asking for the mirror of `FLAGS` printed one warning for each of its two fields, such as `[UVM/FLD/GET_MIRRORED_VAL/VOL] Mirrored value returned for volatile field "regs.FLAGS.ovf" may not reflect the actual current value.` So the register model can't check a register like `FLAGS` for you. The router's `STATUS`, `DROP_CNT` and `LEVEL` are all volatile. In Lab 06 the scoreboard keeps its own copies of `STATUS` and `INT_EN`, which it updates cycle by cycle the way the router does, and checks `STATUS` reads and `irq` against them.

Desired and mirrored values

`set()` changes only the desired value. `update()` then writes the register if the desired value differs from the mirror. With the predictor connected, the test printed:

```
UVM_INFO guide-examples/register-models/regblk_tb.sv(373) @ 145000: uvm_test_top [RAL]
after set(3) on CTRL.MODE: desired 0007, mirror 000b, needs_update 1
UVM_INFO guide-examples/register-models/regblk_tb.sv(376) @ 165000: uvm_test_top [RAL]
after update(): desired 0007, mirror 0007, needs_update 0
```

`MODE` is bits 3 to 1, so setting it to 3, with `EN` still 1, makes the desired value `0111` in binary, which is `0007`. The mirror held `000b` from the raw write, which the predictor saw. With `+PREDICT=none` the last line read `desired 0007, mirror 0000, needs_update 1`. The write happened, but nothing told the mirror about it, so the model would write the register again on every `update()`.

The built-in sequences

A register model also gives you stimulus without your writing any, because it knows every register's access policy and reset value. UVM has built-in sequences that work from the model alone, and Lab 06 uses two of them. Neither of the two runs on a sequencer of its own. You set its `model` to the register block and start it with `start(null)`, and it sends its transfers through the model.

`uvm_reg_hw_reset_seq` resets the model to its reset values, then reads every register it can check and compares each read with the reset value. On `regblk` it printed

```
... [uvm_reg_hw_reset_seq] Verifying reset value of register regs.ID in map
"regs.map"...
... [uvm_reg_hw_reset_seq] Verifying reset value of register regs.CTRL in map
"regs.map"...
```

and nothing for `FLAGS` or `COUNT`. Every field in those two registers is volatile, so a read of either would compare nothing, and the sequence skips a register like that. With the planted bug `en_reset_1`, which makes `CTRL.EN` reset to 1, it reported

```
UVM_ERROR third_party/uvm/src/reg/uvm_reg.svh(2812) @ 75000: reporter [RegModel]
Register "regs.CTRL" value read from DUT (0x0000000000000001) does not match mirrored
value (0x0000000000000000) (valid bit mask = 0x000000000000000f)
UVM_INFO third_party/uvm/src/reg/uvm_reg.svh(2833) @ 75000: reporter [RegModel] Field
en (regs.CTRL[0:0]) mismatch read=1'h1 mirrored=1'h0
```

The reset sequence has to run while the design still holds its reset values, which means before anything has written a register, or straight after a reset.

`uvm_reg_bit_bash_seq` tests every bit it can check, one at a time. For each bit it writes the register with that bit the opposite of its desired value, reads the register back, and compares the read with the desired value, and then it does all of that a second time. On a bit you can write, the second write puts the bit back where it was. The sequence skips the bits of volatile fields, and it treats a bit with no field as read-only, so it still checks that bits 2 to 15 of `FLAGS` and 8 to 15 of `COUNT` read 0. On `regblk` it printed `Verifying bits in register` for all four registers. With the planted bug `mode_stuck`, which makes bit 3 of `CTRL` impossible to set, it reported the same line once for each of its two writes:

```
UVM_ERROR third_party/uvm/src/reg/sequences/uvm_reg_bit_bash_seq.svh(211) @ 1595000:
reporter@@s.reg_single_bit_bash_seq [uvm_reg_bit_bash_seq] Writing a 1 in bit #3 of
register "regs.CTRL" with initial value 'h0000000000000000 yielded 'h0000000000000000
instead of 'h0000000000000008
```

Neither sequence knows anything about `regblk` beyond what the register model describes. In Lab 06 the same two sequences catch two of the five planted bugs, B14 and B03, without a test step written for either bug.

The model can't send a transfer to an address that has no register, because there's no register to name, so the router's 246 bad addresses still need raw transfers (Lab 06, Part 4). The model also knows nothing about what the registers *do* to the rest of the design. The router's `irq` output depends on `STATUS` and `INT_EN`, and the scoreboard checks it (Part 5).

The back door

A **back-door** access reads (`peek()`) or writes (`poke()`) a register through the name of the signal that holds it, with no bus transfer and no simulation time. UVM finds the signal through VPI, and Verilator lets VPI see a signal only if it's marked public. Lab 06's `tb/router_regs.vlt` marks the five signals that hold `CTRL`, `STATUS`, `INT_EN` and `DROP_CNT` (`DROP_CNT`'s two counters are separate signals), one line each. It names them one at a time because the flag that makes every signal public, `--public-flat-rw`, breaks the UVM build on Verilator 5.050.

Lab 06's `router_backdoor_test` measures whether a poke sticks on the router. It pokes `CTRL`, `INT_EN` and `DROP_CNT`, and peeks each one at once, one rising edge later and two rising edges later. Every peek returned the poked value. These are the lines for `DROP_CNT`:

```
165000 ps  poke DROP_CNT with                    505   (UVM_IS_OK)
165000 ps    peek at once:                       505   (UVM_IS_OK)
```

```
175000 ps    peek one rising edge later:          505   (UVM_IS_OK)
185000 ps    peek two rising edges later:         505   (UVM_IS_OK)
```

That's a measurement of three registers in one design, so before you rely on a poke anywhere else, peek the register after a rising edge, as that test does. A poke also sets the register model's mirror to the value poked, without applying the field's access policy. Any write clears `DROP_CNT`, so when auto-predict or a predictor updates the mirror after a write of `505`, the mirror becomes 0, but after the poke it holds `505`. A poke puts nothing on the bus, so a scoreboard that keeps its own copy of the register never hears about it. Lab 06's scoreboard learns `DROP_CNT` only from the transfers it sees on the bus, so after a poke its copy and the router's disagree. In Part 6 you read the poked value through the front door to see what the scoreboard makes of it. A peek changes nothing in the design, so it can't leave a copy out of date, and that makes `peek()` the safer of the two.

Check yourself

1. A testbench has auto-predict on and no predictor. A test sets up the routing table with raw transfers, then runs `mirror(UVM_CHECK)` on `ROUTE0`. What happens, and is the router at fault?

   Answer: A1, in Answers at the end.

2. Why did `uvm_reg_hw_reset_seq` read only `ID` and `CTRL` on `regblk`, and what does the bit bash still test in `FLAGS`?

   Answer: A2, in Answers at the end.

3. After `set(3)` on `CTRL.MODE`, `needs_update` was 1. Which two values differed, and what made them equal again?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's register-layer pages (section 12) build one model for one small APB block, across several pages. Read 12.1 to 12.6 in order.

- UVM Register Abstraction Layer: registers, fields, maps, and the path of a model write down to the bus.
- UVM Register Model Classes, the clearest account of the desired and mirrored values. Correction: its block maps the status register with `"RO"` rights, but its write-1-to-clear bit can only be cleared through a map that allows writes.
- UVM Register Environment: the adapter, and auto-predict against a predictor.
- Connecting register env: the map's sequencer and the predictor's connection, and why each goes in the phase it does.
- UVM Register Access Methods. Correction: it says a front-door `write()` or `read()` always updates the mirror. That's true only with auto-predict on or a predictor connected, which the same page says further down.
- UVM Register Model Example, the whole model in one place. It has no top module, so it doesn't run as shown, and its bus item's address constraint leaves out the status register the test reads.
- UVM Register Sequences: the built-in sequences, why the reset test goes before the bit bash, and how to leave a register out.
- UVM Register Backdoor Access. On Verilator a back door needs UVM's DPI code, `--vpi` and the signals listed in a `.vlt` file, as Lab 06 does.

Next: Regressions and seeds.

Answers

A1. Check yourself
The read returns what the raw transfer wrote, but the mirror still holds the reset value, so the register model reports a `[RegModel] ... does not match mirrored value` error, unless the raw transfer happened to write the reset value. The router is right. Auto-predict updates the mirror only after the model's own operations, so the testbench needs a predictor on the monitor, which sees every transfer.

A2. Check yourself

Every field of `FLAGS` and `COUNT` is volatile, and the register layer doesn't compare a volatile field, so a read of either register would check nothing and the reset sequence skips them. The bit bash skips the bits of volatile fields too, but bits 2 to 15 of `FLAGS` hold no field, so it treats them as read-only and checks that they still read 0 after it writes 1 to each.

A3. Check yourself

The desired value, `0007`, and the mirror, `000b`. `update()` wrote the desired value to the register, and the predictor saw the write and set the mirror to `0007`.

23. Regressions and seeds

This page answers: *what does a seed fix, what can still change a run on the same seed, and what does a regression over many seeds tell you that one run doesn't?*

Every run in this course has a seed. `make run` passes `+verilator+seed+1` unless you give it `SEED=`, so every "You should see" in the labs is from seed 1, except in the steps that run several seeds. Labs 05 and 06 both end by asking for `make seeds SEEDS="1 2 3 4 5"`, which runs the graded test on five seeds from the same build and prints `5 ok  (5 runs)` when all five pass. In Lab 06, Part 1 every packet count that `router_test` prints changes, on the same seed, although none of your code has changed. In Lab 07, Part 3 you'll run eleven tests on ten seeds each and be asked whether 110 passing runs mean that the environment is finished. This page explains both, with measurements from a small example of our own in `guide-examples/regressions-and-seeds/`.

What a seed fixes

The example's `random_test` forks two threads in its run phase. Thread A draws four values with `$urandom` at time 0, and thread B draws four more a nanosecond later. On seed 1 the test printed

```
UVM_INFO guide-examples/regressions-and-seeds/seeds_tb.sv(143) @ 1000: uvm_test_top
[RANDOM] thread A: 0bbddfb0 daff8fa1 bfb2d373 4a0f49df
UVM_INFO guide-examples/regressions-and-seeds/seeds_tb.sv(144) @ 1000: uvm_test_top
[RANDOM] thread B: b0011fed 80a529cd c8e18eaf d407ae8b
```

On seed 1 again it printed the same eight values, and on seed 2 it printed eight different ones:

```
UVM_INFO guide-examples/regressions-and-seeds/seeds_tb.sv(143) @ 1000: uvm_test_top
[RANDOM] thread A: 6d5ed04a 79212f85 cac574d1 3eae46dd
UVM_INFO guide-examples/regressions-and-seeds/seeds_tb.sv(144) @ 1000: uvm_test_top
[RANDOM] thread B: b1403e12 ca4b19dc 91ea6ece 9a0aa6d2
```

Given the same simulator, the same plusargs and the same seed, every random choice in every thread comes out the same, so you can repeat a run that failed exactly. `make run SEED=7` reruns seed 7 and puts the run in a folder named after the seed, `build/runs/<test>-s7`. `SEED=random` picks a seed for you and names the folder after it in the same way. When we ran `make run UVM_TESTNAME=count_test SEED=random`, the run went into `build/runs/count_test-s16616597`, so you can run that seed again by giving its number to `SEED=`.

One more random call

By "the same simulator" we mean the same build of the same code, and the runs in this table show why. They are all on seed 1, and in each one a plusarg makes thread A do one thing more before it draws:

Plusarg	What thread A does first	Thread A's values	Thread B's values
none	nothing	`0bbddfb0 daff8fa1 bfb2d373 4a0f49df`	`b0011fed 80a529cd c8e18eaf d407ae8b`
`+EXTRA_DRAW`	one more `$urandom`	`daff8fa1 bfb2d373 4a0f49df 5f2fa507`	`2be22380 36bdaeeb b33e045a 78e0e61e`
`+EXTRA_NEW`	`new()` of a plain class with a `rand` member	`0bbddfb0 daff8fa1 bfb2d373 4a0f49df`	`b0011fed 80a529cd c8e18eaf d407ae8b`

Plusarg	What thread A does first	Thread A's values	Thread B's values
`+EXTRA_SEQ`	`type_id::create()` of an empty UVM sequence	`03c400f3 7d776947 f2c8eda9 d78d5d6e`	`cca11dc6 fee8bd5a b5a1ea57 fd5d592f`
`+EXTRA_START`	`type_id::create()` of that sequence, then `start()`	`e1b55258 15eacca0 ec923ff4 eb3f888a`	`291de991 9e00215f 4e3d822b 6575033c`

The extra draw moved thread A along by one value, as you'd expect, so A's first three values are the original's second to fourth. It also changed all four of thread B's values, although B's code is the same and B draws a nanosecond after A has finished. Making a UVM sequence changed every value in both threads, and starting it as well changed them again. A plain `new()` changed nothing. (The SystemVerilog standard says otherwise on two points. Under the standard each thread has its own generator, so B's values shouldn't have moved, and `new()` seeds the new object's generator from its thread's, so A's should have. The table shows what Verilator 5.050 does, and that is the simulator your runs use.)

On this simulator, then, one more random call or one more sequence anywhere in a test can change the random choices made after it, in other threads as well as in its own.

You'll be asked about this in Lab 06, Part 1. It is also why Lab 07 starts its new sequence, `router_lab07_vseq`, last in `router_test`. When we left that sequence out, Lab 07's `router_test` printed Lab 06's `[SB]` lines number for number.

Three habits follow from this:

- Compare two runs on the same seed only when the code is the same. A seed names a run of one build; on a different build the same seed is a different run, and its values can all change.
- When a seed fails, write down the commit it failed on as well as the seed. After you change the testbench, that seed is a different run, and the failure may move or disappear without the bug being fixed.
- If a failure matters, write a directed step in a sequence that reaches the bug whatever the seed. Labs 06 and 07 have you write one for each bug you need to reach.

Running many seeds

A **regression** runs a set of tests on a set of seeds, all from one build, and reports every run and then the coverage of all the runs together. `make seeds` runs one test this way, and `make regress` runs every test in the lab's `TESTS`. Both run `labs/common/regress.py`, which gives each run the same verdict that `make run` would, so a run that crashed or stopped on a `UVM_FATAL` shows as BROKEN and isn't counted as a pass.

The example's other test, `count_test`, drives a 4-bit counter that counts up on `inc`, goes back to 0 on `clr` and should stop at 15. For 300 cycles the test sets `inc` at random in 60% of cycles and `clr` in 15%, and it checks the count against a model in every cycle. `make seeds UVM_TESTNAME=count_test SEEDS="1 2 3 4 5 6 7 8 9 10"` printed

```
      ok  count_test                 seed 1      0 UVM_ERROR, 0 UVM_FATAL       0.0s
      ok  count_test                 seed 2      0 UVM_ERROR, 0 UVM_FATAL       0.0s
      ok  count_test                 seed 3      0 UVM_ERROR, 0 UVM_FATAL       0.0s
      ok  count_test                 seed 4      0 UVM_ERROR, 0 UVM_FATAL       0.0s
      ok  count_test                 seed 5      0 UVM_ERROR, 0 UVM_FATAL       0.0s
      ok  count_test                 seed 6      0 UVM_ERROR, 0 UVM_FATAL       0.0s
      ok  count_test                 seed 7      0 UVM_ERROR, 0 UVM_FATAL       0.0s
      ok  count_test                 seed 8      0 UVM_ERROR, 0 UVM_FATAL       0.0s
      ok  count_test                 seed 9      0 UVM_ERROR, 0 UVM_FATAL       0.0s
      ok  count_test                 seed 10     0 UVM_ERROR, 0 UVM_FATAL       0.0s
--------------------------------------------------------------------------------
  10 ok  (10 runs)
```

and then the coverage of all ten runs together, which included

```
verilator_coverage: covergroup : 100.0% (4/4)
```

All ten passed, and the merged coverage hit all four bins: the count in 0 to 7, in 8 to 14 and at 15, and an increment while it was at 15. That 100% is the union of the ten runs. The `[COUNT]` line that each run prints at the end of the test says how often its own counter was at 15, and how often it was incremented there:

Seed	1	2	3	4	5	6	7	8	9	10
Cycles at 15	0	6	20	0	0	0	0	4	3	24
Incremented at 15	0	2	13	0	0	0	0	2	2	19
`BUG=wrap`	MISSED	caught	caught	MISSED	MISSED	MISSED	MISSED	caught	caught	caught

The last row is the same test with the planted bug `wrap` switched on, which makes the counter wrap from 15 to 0. We ran it one seed at a time, with `make run SEED=n BUG=wrap`, because `make seeds` doesn't pass `BUG` on to its runs. The test caught the wrap on exactly the five seeds that incremented the counter at 15, each time with `[COUNT] count is 0, expected 15`, and passed on the other five. To reach 15 the counter needs fifteen increments with no clear between them, and with a clear in 15% of cycles that doesn't happen often, so whether a seed gets there is a matter of luck. Seed 1, the default, is one of the five unlucky ones.

Each extra seed in a regression is another chance at a corner that the stimulus reaches only by luck. When a corner matters, it's better not to leave it to luck at all, which is why Lab 07, Part 6 has you write a sequence that drives both drop counters to 255 on purpose.

Corners no seed reaches

More seeds help only with corners the stimulus can reach. In Lab 07, Part 3, the starter's regression of eleven tests on ten seeds printed

```
  110 ok  (110 runs)
verilator_coverage: covergroup : 81.7% (49/60)
```

and listed eleven coverage bins that no run hit. Four of them are situations that a correct router can be put in: a reset while a buffer holds beats, a reset in the middle of an input packet, and a drop of each kind, NOROUTE and FULL, while its counter is already at 255. No test in the starter resets the router after time 0, so no number of seeds would hit the two reset bins. Nothing in it sets out to drive a counter to 255 either, and every phase of `router_inject_test` clears the counters. The 110 passes mean that no check fired in 110 runs, which tells you nothing about a situation that none of those runs produced. One of the fifteen planted bugs shows only when the router is reset with a beat waiting on an output. Verification plans and sign-off is about the verification plan, which gives each rule in the specification a row saying what checks it and what makes it happen. A situation that no test produces then has nothing in its row to make it happen, and you can see that in the plan before you trust a regression's passes.

All 110 runs used the one simulator that `make build` had made. On 2 CPUs the whole regression took 3 min 57 s, about two seconds a run, and a build takes about two minutes. That is why the course chooses the test, the seed and the bug with plusargs when you run, rather than with anything that needs a rebuild.

Check yourself

1. You ran `make run SEED=4` yesterday and it failed. Today, after adding one `$urandom_range` call at the start of your sequence, `SEED=4` passes. Is the bug fixed?

   Answer: A1, in Answers at the end.

2. The counter's ten-seed regression reported `covergroup : 100.0% (4/4)`. Did any one seed hit all four bins, and does it matter?

   Answer: A2, in Answers at the end.

3. Lab 07's starter passed 110 runs and left `reset_with_beats` empty. Would 1,000 seeds have filled it?

   Answer: A3, in Answers at the end.

Further reading

- Regression Testing & Seed Management: regression tiers, why many seeds widen coverage, and keeping the seeds that found bugs. Correction: it gives `+ntb_random_seed`, which is VCS's option, for Xcelium too. On Verilator the run-time option is `+verilator+seed+N`.
- Linear Random Testbench: logging a seed and running many of them from a shell loop. Correction: `$get_initial_random_seed()` is a VCS function, not standard SystemVerilog, and the page's claim that `$random`'s sign bit skews its low four bits is wrong, because truncating to four bits keeps bits 3 to 0 and drops the sign bit with the rest.
- SystemVerilog Command Line Arguments: `$test$plusargs` and `$value$plusargs`, which every test in the course uses to read its knobs.
- Coverage Driven Verification: merging coverage across seeds and deciding what an empty bin means.

Next: Lab 06: The register block.

Answers

A1. Check yourself

You can't tell. On Verilator 5.050 one extra random call changes the values drawn after it, so seed 4 is now a different run with different stimulus, and it may simply no longer reach the bug. Run yesterday's commit on seed 4 to confirm the failure, then turn it into a directed step that fails whatever the seed.

A2. Check yourself

Yes. The five seeds that incremented the counter at 15 passed through 0 to 7 and 8 to 14 on the way, so each of them hit all four bins. The other five seeds matter more. They never got the counter to 15, and on those five the wrap bug passed. Merged coverage tells you that some run hit each bin, not that every run did.

A3. Check yourself

No. Nothing in the starter resets the router after time 0, so no seed can produce a reset with beats in a buffer. A seed changes the values a test picks, but which choices the test makes at all is fixed by its code, so filling the bin takes a new sequence, which you write in Lab 07, Part 5.

24. Lab 06: The register block

In this lab you'll give the Lab 05 testbench a register model. You write the fields of four of the router's registers and the adapter that turns register operations into APB transfers, and run UVM's built-in reset sequence through the model and your adapter. Then you run UVM's bit-bash sequence and connect a predictor, which updates the model from every transfer on the bus. After that you test the 246 addresses that the model can't reach, and make the scoreboard check `irq` against `STATUS` and `INT_EN` in every cycle. The Lab 05 testbench misses all five of this lab's planted bugs (B01, B02, B03, B04 and B14), and with your additions it catches all five by the end of Part 5. Part 6 shows what peek and poke do on this router.

Package: `labs/lab-06-the-register-block/`

Before you start

- Finish Lab 05.
- Read Register models and Regressions and seeds.
- Keep the router specification open at its Register map, APB, INT_EN and Reset sections, and the UVM register layer page beside it, which lists the arguments of a field's `configure()`, the access policies, and the messages the register layer prints.

The package is the Lab 05 testbench with our solution to Lab 05 filled in: the three assertions, the sink driver's answer to each beat, and the body of `router_lab05_vseq`. If your own Lab 05 printed `3 of 3 caught.`, your code did the same job, but keep ours, because this lab's additions are in the same files you edited. Here is what we added for you:

File	What Lab 06 adds
`tb/router_ral_pkg.sv`	the register model and its adapter; four registers' fields and the adapter are yours (Part 2)
`tb/router_ifs.sv`, `tb/router_tb_top.sv`, `tb/router_agents_pkg.sv`	`router_sys_if` and `sys_monitor`, which report `rst_n` and `irq` every cycle
`tb/router_env_pkg.sv`	the environment makes the register model, the adapter and the predictor, and you write the predictor's connections (Part 3); the scoreboard models `STATUS` and `INT_EN`, checks `STATUS` reads, `PSLVERR` and reads of bad addresses, and has an `irq` check for you to write (Part 5)
`tb/router_vseq_pkg.sv`	helpers that go through the model (`wr_reg`, `rd_reg`, `check_reg`, `rd_all_regs`), `router_lab06_vseq` with one `TODO` per part, and `router_backdoor_vseq`
`tb/router_tests_pkg.sv`	`router_test` runs `router_lab06_vseq` first, then everything Lab 05's `router_test` ran; `router_lab06_test` and `router_backdoor_test`
`tb/router_regs.vlt`	the five register signals the back door reaches (Part 6)

Time: about 2 h. You'll build the simulator seven times. On the course's 2 CPUs our first build took 127 s, and the rebuilds after each part's edits took 107 s (Part 2), 108 s and 113 s (Part 3), 35 s (Part 4), and 105 s and 107 s (Part 5), which comes to about 12 minutes of building. Each `make check-all` runs six simulations, each taking about 5 s or less.

Part 1: The Lab 05 testbench against the registers

What does your Lab 05 testbench check about the registers?

1. Go to the lab folder and start the build:

   ```
   cd labs/lab-06-the-register-block
   make build
   ```

   Ours took 127 s. Near the end it prints Verilator's report:

   ```
   - V e r i l a t i o n   R e p o r t: Verilator 5.050 2026-07-01 rev v5.050
   - Verilator: Built from 63.624 MB sources in 449 modules, into 29.574 MB in 3440 C+
   + files needing 213.243 MB
   - Verilator: Walltime 126.831 s (elab=0.857, cvt=3.307, bld=121.710); cpu 5.400 s
   on 2 threads; allocated 369.230 MB
   ```

2. Run the graded test, `router_test`, and pull the scoreboard's summary out of the log:

   ```
   make run
   grep -o '\[SB\] [0-9a-z].*' build/runs/router_test-s1/run.log
   ```

   You should see `ok: 0 UVM_ERROR, 0 UVM_FATAL  3.44 s` from `make run`, and from `grep`:

   ```
   [SB] 1123 packets in: 287 to output 0, 355 to output 1, 192 NOROUTE, 289 FULL; the
   highest level a header saw was 16
   [SB] beats checked: 1281 on output 0, 1552 on output 1; 6 DROP_CNT reads checked
   [SB] registers: 6 reads, 0 of STATUS checked; 0 transfers to bad addresses; irq
   checked in 0 cycles, wrong in 0
   ```

   The third line is new in this lab. The six reads are `router_inject_test`'s five `DROP_CNT` reads and the one at the end of your Lab 05 sequence. Nothing checks `irq` yet, so the line says it was checked in 0 cycles.

3. At the end of Lab 05 (Part 4, step 6), `router_test` printed `1122 packets in: 240 to output 0, 270 to output 1, 194 NOROUTE, 418 FULL` on the same seed. Your Lab 05 code hasn't changed, and `router_lab06_vseq`, which `router_test` now starts first, does nothing yet. Why are the numbers different?

   Answer: A1, in Answers at the end.

4. `make check-all` runs `router_test` on the golden router and on the five bugs. Run `python3 ../common/bugs.py`, which lists all 15 bugs, find this lab's five in the list, and read their "hides behind" hints. Write down how many you expect the starter to catch, then run it:

   ```
   make check-all
   ```

   You should see:

   ```
        ok  the golden design                     0 UVM_ERROR, 0 UVM_FATAL       3.6s
   ------------------------------------------------------------------------
     MISSED  tier 1  B01_status_write_clears_all     0 UVM_ERROR, 0 UVM_FATAL      3.6s
     MISSED  tier 1  B02_irq_ignores_int_en          0 UVM_ERROR, 0 UVM_FATAL      3.6s
     MISSED  tier 1  B03_route3_readback_swapped     0 UVM_ERROR, 0 UVM_FATAL      3.6s
     MISSED  tier 1  B04_misaligned_address_accepted  0 UVM_ERROR, 0 UVM_FATAL     3.6s
     MISSED  tier 6  B14_enable_resets_to_one        0 UVM_ERROR, 0 UVM_FATAL      3.6s
   ------------------------------------------------------------------------
    0 of 5 caught.
   ```

   `make check`, which grades the first bug in the list, says `VERDICT: FAIL -- your test passes B01_status_write_clears_all too.`

B14 makes `CTRL.EN` reset to 1 instead of 0. Why doesn't your testbench see it?

Answer: A2, in Answers at the end.

Check: `0 of 5 caught.`, and `irq checked in 0 cycles`.

Part 2: The register model

What does UVM need to know about each register, and what does the reset sequence do with it?

1. Open `tb/router_ral_pkg.sv` and read `router_reg_ctrl`, `router_reg_int_en` and `router_reg_block::build()`. Each register's `build()` creates its fields and configures each one with a width, a position, an access policy, whether it's volatile (changed by the router on its own), and a reset value. The block's `build()` adds each register to a map at its address. The specification's Register map section gives each field's policy and volatility, and the register layer page explains each argument of `configure()`.
2. Write `build()` for `router_reg_status`, `router_reg_drop_cnt`, `router_reg_level` and `router_reg_route`, as their `TODO`s describe. Pass `is_rand` as 1 only for an `RW` field.
3. Write the adapter's two functions at the bottom of the file. `reg2bus()` fills in an `apb_item` from a register operation, and `bus2reg()` does the reverse for a finished transfer. Give a transfer that ended with `PSLVERR` the status `UVM_NOT_OK`.
4. In `tb/router_vseq_pkg.sv`, write the Part 2 step of `router_lab06_vseq`: create a `uvm_reg_hw_reset_seq`, set its `model` to `regs`, and start it with `start(null, this)`. It reads every register it can check, through the model and so through your adapter, and compares each with its reset value. Run `make lint`, which takes seconds and prints any syntax error with its line; ours printed `0 warnings`. Then run `make build` (107 s for us).
5. Write down which of the ten registers you expect the reset sequence to read. Then run the test and list the ones it did:

```
make run
grep -o 'Verifying reset value of register [^ ]*' build/runs/router_test-s1/run.log
```

You should see `ok` and seven lines:

```
Verifying reset value of register regs.ID
Verifying reset value of register regs.CTRL
Verifying reset value of register regs.INT_EN
Verifying reset value of register regs.ROUTE0
Verifying reset value of register regs.ROUTE1
Verifying reset value of register regs.ROUTE2
Verifying reset value of register regs.ROUTE3
```

Why are `STATUS`, `DROP_CNT` and `LEVEL` missing?

Answer: A3, in Answers at the end.

6. Run `make check-all`. B14's row becomes

```
  caught  tier 6  B14_enable_resets_to_one        1 UVM_ERROR                  3.4s
```

and the other four stay `MISSED`, for `1 of 5 caught.` That error comes from the register model, and this prints it:

```
grep '^UVM_ERROR' build/grade/B14/run.log | grep -o '\[[A-Za-z_]*\] .*'
```

```
[RegModel] Register "regs.CTRL" value read from DUT (0x0000000000000001) does not
match mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000001)
```

`router_test` runs `router_lab06_vseq` before Lab 04's steps. What would the reset sequence report on the golden router if it ran after them?

Answer: A4, in Answers at the end.

Check: B14 is `caught`, and the other four are `MISSED`.

Part 3: The predictor and the bit bash

How does the register model keep its mirror in step with the router, and what does the bit-bash sequence test?

1. Write the first half of the Part 3 step: run a `uvm_reg_bit_bash_seq` the way you ran the reset sequence. The register layer page says what it writes and reads for each bit, and what it skips.
2. Write the second half. Lab 04 and Lab 05 set up the router with raw APB transfers, which go around the model, and the `route()` and `ctrl()` you use here send raw transfers too. After the five writes that the `TODO` lists, read `CTRL` and the four `ROUTEn` with `check_reg()`, which compares each read with the model's mirror.
3. The predictor isn't connected yet, and the model's auto-predict is off (`set_auto_predict(0)` in `router_env::connect_phase`). Predict what `router_test` does on the golden router, then build (108 s for us) and run it:

```
make build
make run
grep '^UVM_ERROR' build/runs/router_test-s1/run.log | grep -o '\[[A-Za-z_]*\] .*'
```

You should see `FAIL: 5 UVM_ERROR  4.33 s` from `make run`, and from `grep`:

```
[RegModel] Register "regs.CTRL" value read from DUT (0x0000000000000001) does not
match mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000001)
[RegModel] Register "regs.ROUTE0" value read from DUT (0x0000000000000001) does not
match mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000003)
[RegModel] Register "regs.ROUTE1" value read from DUT (0x0000000000000003) does not
match mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000003)
[RegModel] Register "regs.ROUTE2" value read from DUT (0x0000000000000002) does not
match mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000003)
[RegModel] Register "regs.ROUTE3" value read from DUT (0x0000000000000003) does not
match mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000003)
```

The test fails on the golden router. Which side is wrong, and would turning auto-predict on fix it?

Answer: A5, in Answers at the end.

4. Write the Part 3 `TODO` in `router_env::connect_phase`: give the predictor the map and the adapter, connect the APB monitor's analysis port, `apb.mon.ap`, to its `bus_in`, and turn on the map's check-on-read, so the predictor compares every read it sees with the mirror. A predictor is usually connected to the bus monitor, because the monitor sees every transfer on the bus, whoever started it. The scoreboard gets its APB transfers from the same port.
5. Build (113 s for us) and run:

```
make build
make run
grep -o 'Verifying bits in register [^ ]*' build/runs/router_test-s1/run.log
```

You should see `ok: 0 UVM_ERROR, 0 UVM_FATAL  4.45 s`, and ten lines, from `Verifying bits in register regs.ID` to `Verifying bits in register regs.ROUTE3`. The list includes `STATUS`, although every field of `STATUS` is volatile and the bit bash skips the bits of volatile fields. What does it test in `STATUS`?

Answer: A6, in Answers at the end.

6. Run `make check-all`. You should see `caught` on B03 (4 `UVM_ERROR`s) and B14 (2), and the other three `MISSED`, for `2 of 5 caught.` B14 has one error more than in Part 2, because the predictor now checks the reset sequence's read of `CTRL` as well. Look at B03's first two errors:

```
grep '^UVM_ERROR' build/grade/B03/run.log | grep -o '\[[A-Za-z_]*\] .*' | head -2
```

```
[RegModel] Register "regs.ROUTE3" value read from DUT (0x0000000000000002) does not
match mirrored value (0x0000000000000001) (valid bit mask = 0x0000000000000003)
[uvm_reg_bit_bash_seq] Writing a 1 in bit #0 of register "regs.ROUTE3" with initial
value 'h0000000000000000 yielded 'h0000000000000002 instead of 'h0000000000000001
```

   Where did the 1 written to `EN` come back, and why did your Lab 05 testbench never notice B03?

   Answer: A7, in Answers at the end.

Check: B03 and B14 are `caught`, and B01, B02 and B04 are `MISSED`.

Part 4: The addresses outside the map

Which addresses can the register model never test?

1. The APB address has 8 bits, so there are 256 addresses, and the map has a register at ten of them. A transfer to any of the other 246 completes with `PSLVERR`; a bad read returns 0 and a bad write changes nothing. The model can't send a transfer to an address it has no register for, so you have to reach these with raw transfers. Write the Part 4 step: read every bad address with `rd(a)`, write each with `wr(a, 32'hFFFF_FFFF)`, then read every register with `rd_all_regs()`. `good_addr(a)` is 1 for the ten good addresses. The scoreboard already checks `PSLVERR` on every transfer and the data of every bad read.
2. Write down what the golden router should return for a read of `0x01`, and what `pslverr` should be. Then build (35 s for us), run `make check-all`, and look at the scoreboard's summary in the golden run, whose log `make check-all` keeps in `build/grade/golden/`:

   ```
   make build
   make check-all
   grep -o '\[SB\] [0-9a-z].*' build/grade/golden/run.log
   ```

   You should see B04 `caught` with `50 UVM_ERROR` (the run stops at 50) and `3 of 5 caught.`, and the last of the three `[SB]` lines should be:

   ```
   [SB] registers: 856 reads, 59 of STATUS checked; 492 transfers to bad addresses; irq
   checked in 0 cycles, wrong in 0
   ```

3. Look at B04's first two errors:

   ```
   grep '^UVM_ERROR' build/grade/B04/run.log | grep -o '\[[A-Za-z_]*\] .*' | head -2
   ```

   ```
   [APB] a read of 01, completing in cycle 4145: pslverr is 0, but 01 is a bad address
   [APB] a read of 01, completing in cycle 4145 returned d1050001; a read of a bad
   address returns 0
   ```

   `d1050001` is `ID`'s value. Why did a read of `0x01` return it?

   Answer: A8, in Answers at the end.

4. To see these reads in a waveform, record `router_lab06_test`, which runs your sequence alone, on the golden router and on B04. It runs cycle for cycle like the first part of `router_test`, so the read of `01` completes in cycle 4145 there too:

```
make waves UVM_TESTNAME=router_lab06_test
make waves UVM_TESTNAME=router_lab06_test BUG=B04
surfer build/runs/router_lab06_test-s1-B04/waves.fst &
```

Add `clk` from `router_tb_top`, `psel`, `penable`, `pwrite`, `paddr`, `pready`, `pslverr` and `prdata` from `apb`, and `bad_addr` from `dut.u_regs`. Cycle n ends at the rising edge at 10n − 5 ns, so the read of `01` takes cycles 4143 to 4145, from 41415 to 41445 ns. The first error in the B04 run's log is printed at 41445000 ps, the edge at which that read completes.

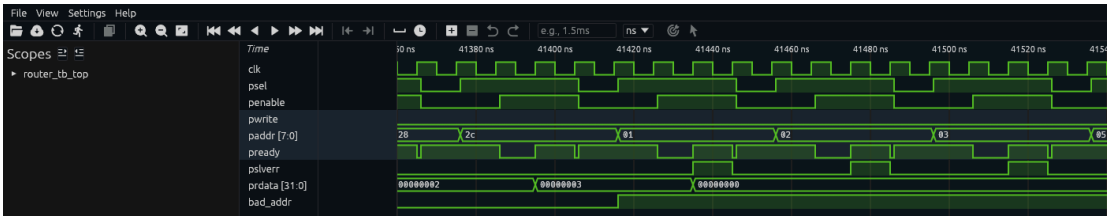


Figure 24.1: Reads of 2c, 01, 02 and 03 on the golden router, in Surfer

The golden router, 41360 to 41540 ns. `pslverr` is 1 in the last cycle of the reads of `01`, `02` and `03`, and each of those reads returns 0.

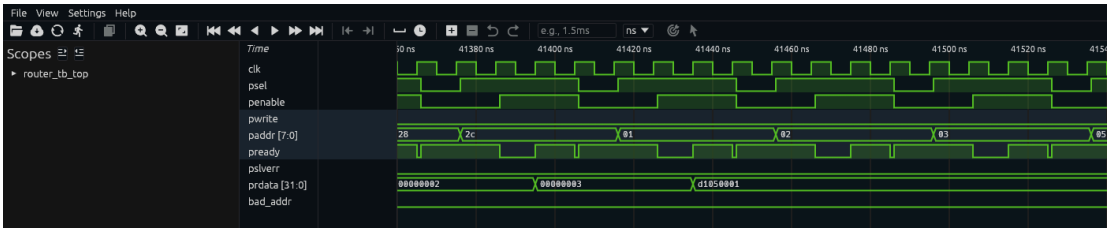


Figure 24.2: The same reads with B04 switched on, in Surfer

With B04, `bad_addr` stays 0, `pslverr` never rises, and the reads of `01`, `02` and `03` all return `d1050001`.

How many cycles does each read take, and in which of them does the router decide what `prdata` will be?

Answer: A9, in Answers at the end.

Check: B03, B04 and B14 are `caught`, and B01 and B02 are `MISSED`.

Part 5: The interrupt

Does `irq` follow `STATUS` and `INT_EN` in every cycle?

1. The specification's INT_EN section defines `irq` as the OR, over the three bits, of `STATUS[i] & INT_EN[i]`, with no delay. The scoreboard already models both registers cycle by cycle. Each drop, oversize packet, or write to `STATUS` or `INT_EN` waits in `changes` until `advance()` reaches the cycle in which it takes effect, as route writes waited in `pending` in Lab 05, and the function `irq()` combines the two registers. Write the two Part 5 `TODO`s in `router_scoreboard::write_sys()`, which the sys monitor calls every cycle: in a reset cycle `irq` must be 0, and in every other cycle it must equal `irq()`.
2. You haven't written any stimulus for `irq` yet. Write down which of B01 and B02 the check will catch with the stimulus you already have, then build (105 s for us), run `make check-all` and look at B02's error:

```
make build
make check-all
grep '^UVM_ERROR' build/grade/B02/run.log | grep -o '\[[A-Za-z_]*\] .*'
```

You should see `4 of 5 caught.`, with B01 still `MISSED` and B02 caught by one `UVM_ERROR`. You chose the wording of your message; in the reference code's wording it is:

```
[IRQ] irq is 1 from cycle 5921, but STATUS = 001 and INT_EN = 000 make it 0
```

In the golden run, the scoreboard's summary (the `grep` in Part 4, step 2) now says `irq checked in 14681 cycles, wrong in 0`. Why does B02 show up in Lab 04′s traffic, and why does B01 still hide?

Answer: A10, in Answers at the end.

3. Write the Part 5 step of `router_lab06_vseq`:

   - disable `DST` 1 with `route(1, 0, 0)` and make both sinks fast with `sink(p, SNK_FAST)`;
   - with `INT_EN` still 0, set all three `STATUS` bits, sending the packets with `burst()`: one 1-beat packet to `DST` 1, which is a NOROUTE drop, and one 9-beat packet to `DST` 0, which is OVERSIZE; then `drain()`, stall output 0 with `stall(0)`, and send ten 1-beat packets to `DST` 0, the last of which is dropped as FULL;
   - read `STATUS`;
   - turn each `INT_EN` bit on by itself, waiting two cycles after each write;
   - set `INT_EN` to `'b111`, then clear the `STATUS` bits one at a time by writing `'b001`, `'b100` and `'b010`, and read `STATUS` after each write;
   - `drain()`, then `rd_all_regs()`.

   Write `INT_EN` and `STATUS` with `wr_reg()` and read them with `rd_reg()`. The model doesn't check a read of `STATUS`, because its fields are volatile, but the scoreboard checks every one.

4. Build (107 s for us) and run `make check-all`. You should see:

```
     ok  the golden design                      0 UVM_ERROR, 0 UVM_FATAL       4.8s
-------------------------------------------------------------------------------
  caught  tier 1  B01_status_write_clears_all     3 UVM_ERROR                  4.8s
  caught  tier 1  B02_irq_ignores_int_en          1 UVM_ERROR                  4.7s
  caught  tier 1  B03_route3_readback_swapped     4 UVM_ERROR                  4.7s
  caught  tier 1  B04_misaligned_address_accepted  50 UVM_ERROR                1.4s
  caught  tier 6  B14_enable_resets_to_one        2 UVM_ERROR                  4.7s
-------------------------------------------------------------------------------
 5 of 5 caught.
 Every bug in the list, each one caught by a check that fired.
```

   B01′s first two errors, with the reference code, are:

```
[IRQ] irq is 0 from cycle 5969, but STATUS = 110 and INT_EN = 111 make it 1
[STATUS] a read of 08, completing in cycle 5972 returned 00000000; it should return
00000006, STATUS in cycle 5971
```

   The first write of `'b001` cleared `NOROUTE` and, with B01, `FULL` and `OVERSIZE` too. B02 is still caught by one error, but the error is now `irq is 1 from cycle 5909`, in your own step. Why doesn't Lab 04′s traffic catch B02 any more?

   Answer: A11, in Answers at the end.

5. The register model has a mirror of `INT_EN` and `STATUS` as well. Why does the scoreboard keep its own copy of both, instead of reading the mirror?

   Answer: A12, in Answers at the end.

Check: `5 of 5 caught.`, and `irq checked in 14840 cycles, wrong in 0` in the golden run.

Part 6: The back door

What do peek and poke do on this router?

1. A back-door access reads (`peek`) or writes (`poke`) a register through the signal that holds it, with no bus transfer and no simulation time. Read the HDL paths at the end of `router_reg_block::build()`,

`tb/router_regs.vlt` (the register layer page says how they fit together), and `router_backdoor_vseq` in `tb/router_vseq_pkg.sv`.

2. The sequence writes `CTRL` with a raw transfer and peeks it twice, then peeks `STATUS` and `DROP_CNT` after a NOROUTE drop. Then it pokes `CTRL`, `INT_EN` and `DROP_CNT` in turn, and peeks each one at once, one rising edge later and two rising edges later. Write down what you expect each peek to return, then run the test and pull out its lines:

```
make run UVM_TESTNAME=router_backdoor_test
sed -n 's/.*@ \([0-9]*\): .*\[BACKDOOR\] /\1 ps  /p' build/runs/router_backdoor_test-s1/run.log
```

You should see `ok: 0 UVM_ERROR, 0 UVM_FATAL   0.09 s` and:

```
65000 ps  peek CTRL as the write of 1 returns:   1     (UVM_IS_OK)
75000 ps  peek CTRL one rising edge later:       1     (UVM_IS_OK)
125000 ps  peek STATUS after a NOROUTE drop:      1     (UVM_IS_OK)
125000 ps  peek DROP_CNT:                         1     (UVM_IS_OK)
125000 ps  poke CTRL with                         0     (UVM_IS_OK)
125000 ps    peek at once:                        0     (UVM_IS_OK)
135000 ps    peek one rising edge later:          0     (UVM_IS_OK)
145000 ps    peek two rising edges later:         0     (UVM_IS_OK)
145000 ps  poke INT_EN with                       6     (UVM_IS_OK)
145000 ps    peek at once:                        6     (UVM_IS_OK)
155000 ps    peek one rising edge later:          6     (UVM_IS_OK)
165000 ps    peek two rising edges later:         6     (UVM_IS_OK)
165000 ps  poke DROP_CNT with                     505   (UVM_IS_OK)
165000 ps    peek at once:                        505   (UVM_IS_OK)
175000 ps    peek one rising edge later:          505   (UVM_IS_OK)
185000 ps    peek two rising edges later:         505   (UVM_IS_OK)
```

Each poked value is still there two rising edges later. We measured that only on these three registers of this router, so before you rely on a poke in another design, peek the register after a rising edge, as this test does. `CTRL` and `INT_EN` change only on a write or a reset, but the router assigns both `DROP_CNT` counters at every rising edge. Why does the poked `505` survive?

Answer: A13, in Answers at the end.

3. Before the test reads anything through the front door, it pokes each register back to the value it had. `+BACKDOOR_READBACK` adds one front-door read of `DROP_CNT` before that, while `DROP_CNT` still holds the poked `505`:

```
make run UVM_TESTNAME=router_backdoor_test PLUSARGS="+verilator+error+limit+100000 +BACKDOOR_READBACK"
```

The `sed` command now prints one more line,

```
225000 ps  front-door read of DROP_CNT:           505   (UVM_IS_OK)
```

and `make run` prints:

```
 FAIL: 1 UVM_ERROR   0.09 s   log: build/runs/router_backdoor_test-s1/run.log
   UVM_ERROR [DROP_CNT] read 00000505, expected 00000001 (FULL_CNT 0, NOROUTE_CNT 1)
```

The router returned what was poked. Why does the scoreboard call it wrong?

Answer: A14, in Answers at the end.

Check: the sixteen lines above and `ok` from the shipped test, and the one `[DROP_CNT]` error with `+BACKDOOR_READBACK`.

Finishing

You're done when `make check-all` prints `5 of 5 caught.` with the register fields, adapter, predictor connections, `irq` check and `router_lab06_vseq` steps you wrote in `tb/`, and `make seeds SEEDS="1 2 3 4 5"` prints `5 ok (5 runs)`.

Questions:

1. Before this lab, the scoreboard checked one register, `DROP_CNT`. What did the register model add, and what did it leave for the scoreboard and for raw transfers?

   Answer: A15, in Answers at the end.

Submit: nothing for this lab.

Going further (optional)

Run your testbench against every bug in the catalogue:

```
make check-all BUG_LIST="B01 B02 B03 B04 B05 B06 B07 B08 B09 B10 B11 B12 B13 B14 B15"
```

The reference solution catches 14 of the 15, and its output ends with:

```
  caught  tier 6  B14_enable_resets_to_one         2 UVM_ERROR                    4.8s
  MISSED  tier 6  B15_output_valid_survives_reset  0 UVM_ERROR, 0 UVM_FATAL       4.9s
-----------------------------------------------------------------------
 14 of 15 caught.
```

It misses only B15, which hides behind "resetting only at time 0, when every buffer is already empty". You'll meet B15 in Lab 07.

Next on the Course path: the reading on verification plans and sign-off and on writing the verification report, then Lab 07: Capstone.

Answers

A1. Part 1: The Lab 05 testbench against the registers

Making a sequence and starting it both draw from the random state of the thread that does it. In `router_test` that is the test's own thread, which then goes on to pick Lab 04's routing tables and to start the sequences that randomize Lab 04's packets. So all of those random choices come out differently, even though `router_lab06_vseq` does nothing. That is why Lab 05's `router_test` made and started its own sequence only after Lab 04's steps. We also ran this lab's `router_test` with `router_lab06_vseq` left out, and it printed Lab 05's two lines number for number, so nothing else this lab added changes the stimulus, and the checks are the same either way.

A2. Part 1: The Lab 05 testbench against the registers

A wrong reset value shows only between the reset and the first write to the register. A read of `CTRL` in that window would show it, and so would a packet offered then, because `in_ready` stays 0 after reset until software sets `CTRL.EN`. Lab 04's steps write `CTRL` before the first packet and never read it, and the scoreboard wouldn't check a read of `CTRL` anyway, since the only register reads it checks are those of `DROP_CNT` and `STATUS`. B02 hides because nothing checks `irq` yet, as the third `[SB]` line says.

A3. Part 2: The register model

Every field in them is volatile. UVM doesn't compare a volatile field when it reads one, since the design may have changed it since the model last looked, and the reset sequence skips a register in which no field would be compared. If your run printed more than seven lines, one of your fields in those three registers isn't marked volatile.

A4. Part 2: The register model

It would report errors, although the router is right. The sequence resets the model to the reset values and then compares every read with them, but the router is reset only once, at time 0, and Lab 04's steps and your Lab 05 sequence leave `CTRL.EN` at 1 and the routing table set. So `CTRL` and at least some of the `ROUTEn` would read back what those steps wrote, and the sequence would call each of them wrong. A reset sequence has to run while the registers still hold their reset values, which in this testbench means first.

A5. Part 3: The predictor and the bit bash

The mirror is wrong; the router returned what was written. The mirror changes only when something tells the model about a transfer. Auto-predict updates it after each operation the model itself sends, but

`route()` and `ctrl()` sent raw transfers, so with auto-predict on the mirror would still hold the reset values and the five errors would stay. Since the golden router is right, the bug is in the testbench, which has no predictor to update the mirror from every transfer on the bus, whoever started it.

A6. Part 3: The predictor and the bit bash

It tests bits 3 to 31, which belong to no field. The bit bash treats a bit with no field as read-only. The model expects it to read 0, so both writes set it to 1, and after each one the bit bash checks that it still reads 0. That way it tests the reserved bits of every register that has them. In `STATUS`, `DROP_CNT` and `LEVEL`, whose fields are all volatile, the reserved bits are the only ones it tests.

A7. Part 3: The predictor and the bit bash

It came back in bit 1, `PORT`, because B03 swaps `ROUTE3`'s two fields when it's read. The route itself works, because B03 is in the read path only, so every packet still went where `ROUTE3` sent it. Lab 04's steps and your Lab 05 sequence never read a `ROUTEn`, so no read ever came back with the fields swapped. A test that never reads a register can't see a bug in its read path. The bit bash reads the register back after every write, and each write changes one bit.

A8. Part 4: The addresses outside the map

The registers are 4 bytes apart, so the router finds the register from `paddr[7:2]`, and for `0x01` that's 0, the word where `ID` lives. The router calls an address bad if no register is at its word or if `paddr[1:0]` isn't `00`. B04 drops the second test, so a misaligned address reaches the register below it, for reads and writes alike.

A9. Part 4: The addresses outside the map

Each read takes three cycles: a setup cycle (`psel` 1, `penable` 0), a first access cycle with `pready` 0, and a second with `pready` 1, in which the read completes. The router copies the register into `prdata` at the edge that ends the first access cycle (rule A8), so `prdata` changes in the middle of each read. `pslverr` is 1 only in the cycle in which a bad transfer completes (rule A7). The idle cycle with `psel` 0 between reads comes from the APB driver, which waits for a clock edge before it starts each transfer.

A10. Part 5: The interrupt

Your `router_lab06_vseq` sends no packets yet, so the drop that set `STATUS.NOROUTE` was in Lab 04's traffic. Every phase of `router_inject_test` disables a route and sends packets to it (in the first phase the routing table is `010-`). `INT_EN` stays 0 all through that traffic, because Lab 04's steps and your Lab 05 sequence never write it, and the bit bash leaves each of its bits the way it found it. So `irq` must stay 0, but B02, which ignores `INT_EN`, raises it at the first NOROUTE drop. Nothing after that drop clears `STATUS`, so `irq` stays wrong and the check reports it once. B01 shows only when something writes `STATUS` while a bit that the write shouldn't clear is set. The only writes to `STATUS` so far are the bit bash's, right after reset, when every bit is 0.

A11. Part 5: The interrupt

Your step ends with `INT_EN` at `'b111`, and nothing writes it again, so Lab 04's traffic and your Lab 05 sequence run with every enable on. With every enable on, `|(STATUS & INT_EN)` equals `|STATUS`, so B02 makes no difference to `irq`. That is the situation `bugs.py` means when it says that B02 hides behind "turning on every interrupt enable before looking at irq". Your step catches B02 earlier, after the NOROUTE drop and while `INT_EN` is still 0.

A12. Part 5: The interrupt

The mirror of `STATUS` is out of date most of the time, because the router sets its bits on its own and the model learns their value only when something reads the register. The mirror of `INT_EN` changes at the wrong moment for this check. The predictor updates it when the monitor reports the write, at the edge that ends the cycle in which the write completes, and the router uses the new value from the next cycle. The scoreboard checks `irq` for a cycle at that same edge, so it would get the old or the new `INT_EN` depending on which monitor the simulator woke first. That is the race that your Lab 05 scoreboard avoided by stamping route writes with their cycle. To check `irq` exactly in every cycle, the scoreboard has to apply each change to the registers in the cycle in which the router applies it, so it keeps its own copy of both.

A13. Part 6: The back door

`router_regs.sv` computes each counter's next value from its present one, in the function `bump()`, which clears the counter only on a write to `DROP_CNT` and adds one only on a drop. With neither, the next value is the present value, and after a poke the present value is the poked one.

A14. Part 6: The back door

The scoreboard works out what `DROP_CNT` should hold from the drops it sees on the input and the writes it sees on the bus, and a poke is neither. It has seen one NOROUTE drop and no write to `DROP_CNT`, so as far as it knows `DROP_CNT` holds `00000001`. A testbench that sets registers through the back door has to tell every model that keeps its own copy of them. `peek` is the safer of the two. It shows you `STATUS` and `DROP_CNT` without a bus transfer, and since it doesn't change the router, no model needs to hear about it.

A15. Finishing

It added stimulus made from the register description. The reset sequence read every register it could check, and the bit bash wrote and read back every bit it could check, which caught B14 and B03 without a step written for either bug. It also added access by name (`wr_reg(regs.int_en, ...)`), a mirror that the model compares with every read of a register that doesn't change on its own, and the back door. It left the 246 bad addresses (B04), the volatile registers, and what the registers do to the rest of the router, such as `irq` (B01 and B02), to raw transfers and the scoreboard.

VII 7. Capstone

25 Verification plans and sign-off 187
25 One row for each rule 187
25 Bugs that hid behind an empty column 188
25 Coverage against the plan 188
25 Sign-off criteria 189
25 A quiet hunt 189
25 Check yourself 189
25 Further reading 189
25 Answers 190

26 Writing the verification report 191
26 A short report for the counter 191
26 Numbers and the commands behind them . . . 192
26 The sign-off sentence 192
26 Section 7: the limits 192
26 The hunt table 193
26 Check yourself 193
26 Further reading 193
26 Answers 193

27 Lab 07: Capstone 195
27 Before you start 195
27 Part 1: Your repository and the first build 196
27 Part 2: The verification plan 197
27 Part 3: A seed regression 198
27 Part 4: Mystery hunts 198
27 Part 5: Reset with traffic in flight 199
27 Part 6: Counters at 255 201
27 Part 7: Sign-off 201
27 Part 8: The report 203
27 Finishing 203
27 Going further (optional) 204
27 Answers 204

25. Verification plans and sign-off

This page answers: *your regression passes on every seed. Is the testbench finished, and how would anyone know?*

You start Lab 07 with our solution to Lab 06, to which Lab 07 adds a reset driver, a scoreboard that follows a reset with traffic in flight, and four corners in the coverage. On that package, `make regress` runs eleven tests, each on seeds 1 to 10, and prints

```
  110 ok  (110 runs)
```

`make check-all` runs the graded test, `router_test`, against each of the fifteen planted bugs. On the same package, its row for B15 is

```
  MISSED  tier 6  B15_output_valid_survives_reset  0 UVM_ERROR, 0 UVM_FATAL       8.6s
```

and after the other fourteen rows it prints

```
14 of 15 caught.
Still hiding from this test:
  B15_output_valid_survives_reset
     hides behind: resetting only at time 0, when every buffer is already empty
```

The regression ran the golden router, and every check in all 110 runs passed. `make check-all` shows that the same graded test also passes on a router with B15 switched on. That router breaks rule O4 of the specification, because a beat that was waiting on a stalled output when `rst_n` fell is still on the output afterwards. The line `110 ok` can't tell you which of the specification's rules those checks cover, or which situations the stimulus ever produced. A **verification plan** answers those two questions rule by rule, and you'll write one for the router in Lab 07, Part 2, before you run the regression and the mystery hunts.

One row for each rule

The plan template in `labs/lab-07-capstone/PLAN-template.md` has one row for every rule in the router specification. The input, output and APB rules keep the specification's own numbers (I1 to I7, O1 to O8, A1 to A9), and the rules that the specification states in prose get labels of their own: RT1 for the table that decides whether a packet is dropped or written to an output, DR2 for a counter that stops at 255, RS2 for a reset with beats in a buffer, and so on. Because the rows come from the specification rather than from the tests you already have, a rule that you haven't thought to test still gets a row, and you can see that its row is empty.

Each row has four columns after the rule itself:

- **Checked by**: the check that fires if the router breaks the rule. In this environment that's an assertion, one of the scoreboard's `write_` functions, the register model's mirror, or "nothing".
- **Made to happen by**: the sequence, and the step in it, that creates the situation the rule describes.
- **Seen in coverage**: the coverage bin whose count shows that the situation happened in a run, or "no bin".
- **Status**: `done`, `open` (you mean to close it), or `gap` (you won't, and you say why).

The template has three rows filled in as examples, O1, RT1 and A7. Here is O1's:

Id	The rule	Checked by	Made to happen by	Seen in coverage	Status
O1	`out_valid` holds until the handshake	`a_O1` on each output	every stalled or slow sink: `router_inject_test`'s phases, and `backpressure_test` in `router_lab05_test`	`cg_wait.x_port_wait`: `long_wait` and `stalled` on both outputs	done

Bugs that hid behind an empty column

The three middle columns ask whether a check would fire if the router broke the rule, whether the stimulus ever creates the situation that the rule needs, and whether a run showed that it did. A rule is verified only when all three have an answer. Each of these three bugs got past a testbench in this course because its rule's *Checked by* or *Made to happen by* was empty:

Bug	Lab	Hides behind	What was missing
B11	05	sending only legal packets of 8 beats or fewer	*Made to happen by*: the scoreboard already expected a cut packet, but no test sent a packet of more than 8 beats, and `cp_len.oversize` was 0
B02	06	turning on every interrupt enable before looking at irq	*Checked by*: nothing checked `irq` until Lab 06, Part 5
B15	07	resetting only at time 0, when every buffer is already empty	*Made to happen by*: the only reset came at time 0, and the assertions for the reset rules were still to be written

The "hides behind" strings are the catalogue's own, from `labs/common/bugs.py`, and each one describes a way of testing that leaves one of those columns empty. A check with no stimulus behind it is the easiest gap to miss, because the check is there and it passes. On the small stage in Assertions for interface rules, a bug that throws away a beat once it has waited four cycles passes the assertion for the rule it breaks when the sink is always ready, because then no beat ever waits.

B02 shows a less obvious version, in which the check is there and so is the traffic, but the traffic runs in a state where the bug changes nothing. Lab 06, Part 5 adds the `irq` check, and the step you write in that part catches B02 in the cycles before it turns the enables on. The step ends with `INT_EN` at `'b111`, though, and in Lab 06 nothing after it in `router_test` writes `INT_EN` again. With every enable on, `|(STATUS & INT_EN)` equals `|STATUS`, so a router that ignores the enables raises `irq` exactly when a correct one does, and none of the traffic after the step can catch B02. So in *Made to happen by*, write the situation that the rule needs (for IRQ1, a `STATUS` bit set while its enable is 0) and the step that creates it. A test's name on its own doesn't tell you whether the situation ever happened in it.

Coverage against the plan

You fill in *Seen in coverage* from the regression's coverage. The starter's regression ended with

```
verilator_coverage: covergroup : 81.7% (49/60)
```

and a list of the eleven bins it never hit. Seven of them are bins of `x_fate_level`, the cross of a packet's fate with its output's level, and the specification rules all seven out. A packet is FULL only at a level of 9 or more, and written to an output only at 8 or less, so no test can hit `full_x_low` or `out0_x_high`. The other four are the bins of `cg_corner`. They count a reset with beats in a buffer, a reset in the middle of an input packet, a NOROUTE drop while `NOROUTE_CNT` is already 255, and a FULL drop while `FULL_CNT` is already 255. A test could hit any of the four, and none of the eleven tests did.

Those seven bins put 100% out of reach, so the target is that every empty bin is explained. Either the specification rules the bin out, and your plan says so, or it's a hole that the stimulus has to close. With Parts 5 and 6 done, the reference solution's regression, which runs thirteen tests, printed

```
verilator_coverage: covergroup : 88.3% (53/60)
```

with only the seven bins of `x_fate_level` still empty, which meets that target. (Verilator 5.050 also works out the percentage differently from the standard. It divides the bins hit by all the bins, crosses included,

where the standard's formula averages the coverpoints and crosses, as Lab 03, Part 2 shows. That's one more reason to read the bins rather than the total.)

You can check the plan and the coverage against each other. In Lab 07 you'll find the reset and saturation rows empty twice, once in Part 2 by reading the sequences and once in Part 3 by counting bins. If your plan says a row is `done` and the coverage shows its bin empty, one of the two is wrong, and it's worth finding out which.

Sign-off criteria

Sign-off is the point at which you say the design does what the plan says, under the conditions the plan lists. You'll write the criteria for it in Lab 07, Part 3. The template asks for each one as a command and the output it has to print, and gives this example:

```
make regress prints ... ok (... runs) with no other class, on seeds ...
```

Anyone can check a criterion written this way, including you a week later, by running the command and comparing one line. Writing it also makes you say which seeds, which tests and which build you mean, none of which "the regression passes" says.

A good set of criteria covers each column of the plan. The regression's summary line shows that the checks passed, and its coverage line, with each empty bin named, shows which situations happened. `make check-all` on more than one seed shows that the checks can fail. A statement that every row is `done`, or has its gap written down, covers the Status column.

`make check-all` is in that list because a regression on the golden router can show only that your checks stay quiet when they should, and the planted bugs show that they fire when they should. Fifteen catches tell you about those fifteen bugs, though. A bug unlike any of them, in a rule with no checker, would pass every criterion on the list, which is why the plan's open items go into the report.

A quiet hunt

In Part 4 you'll deal mystery bugs with `make new` and run the graded test against each with `make hunt`. When the starter was dealt B15, the hunt printed

```
   quiet  the mystery bug                          0 UVM_ERROR, 0 UVM_FATAL       7.3s
-------------------------------------------------------------------------------
 Nothing your test drives tells this design from a correct one. The
 bug is in there; your stimulus or your checks have not reached it.
 `make cov` shows what the stimulus covered.
```

Your plan is where you follow up a quiet hunt. The rows that could hide a bug from `router_test` are the ones with an empty column, and Part 4 has you list them in the quiet hunt's row of your report. Parts 5 and 6 close the reset and saturation rows among them, and the others stay open through the lab.

Check yourself

1. A row of your plan says *Checked by*: `a_06`; *Made to happen by*: `router_test`; *Seen in coverage*: no bin; *Status*: done. What could still be wrong with it?

   Answer: A1, in Answers at the end.

2. The starter's regression hit 49 of 60 bins. Why is 60 of 60 the wrong target, and what is the right one?

   Answer: A2, in Answers at the end.

3. Why does a sign-off criterion name a command and the exact line it must print, rather than say "the regression passes"?

   Answer: A3, in Answers at the end.

Further reading

- Verification Plan: what a plan contains, from scope and features to sign-off criteria, with a worked entry.
- Identifying Features & Corner Cases: turning a specification's wording into rows, and finding the corners between them.

- Pass Fail Criteria & Scoreboards: what has to be true for a test to count as a pass.
- Coverage Driven Verification: the loop of running seeds, merging coverage and closing holes, and how to tell a hole from a bin that can't be hit.
- Sign-Off Criteria: the gates an industrial project clears before sign-off, of which Lab 07's criteria are a small version. Its first gate is code coverage, which this course's builds don't collect (they collect covergroup coverage only), and its functional-coverage gate is a percentage, so on Verilator 5.050 count the bins in `coverage.dat` instead, from the list that `make cov` prints.

Next: Writing the verification report.

Answers

A1. Check yourself

Nothing shows that `router_test` ever sent a packet whose 8th beat wasn't its last, and that's the only kind of packet the router has to cut. B11 breaks O6 only on a cut packet, which is how it hid from Lab 04's tests. A bin such as `cp_len.oversize` above 0 is the evidence the row needs.

A2. Check yourself

Seven of the bins are crosses that the specification rules out, so no test can hit them, and 60 of 60 is out of reach however good the stimulus is. The right target is that every empty bin is either ruled out by the specification, with the reason written in the plan, or closed by new stimulus.

A3. Check yourself

So that someone else can check it without your help, by running the command and comparing the line. It also pins down which tests, which seeds and which build the claim is about, which "the regression passes" leaves open.

26. Writing the verification report

This page answers: *you've run the regression and closed the holes your plan found. What do you hand over, so that someone else can check your claim without redoing your work?*

DD104's capstone ended with a bug report about one bug, built around a test that reproduces it and the command that runs that test. DD105's capstone ends with a **verification report**, which claims that a whole design follows its specification, as far as a written plan describes the specification. The report gives the evidence for that claim and says where the evidence stops. You'll write one for the router in Lab 07. In Part 4 you copy `labs/lab-07-capstone/REPORT-template.md` into your repository as `REPORT.md` and fill in its hunt table, and in Part 8 you fill in the rest.

A report like this is read by someone who has to rely on the design without having run your tests, such as whoever builds the design into a chip or reviews it before that happens. That reader can rely only on what they can check, so everything in the report is either a line that a tool printed, with the command that printed it beside it, or a statement with its limits written next to it.

A short report for the counter

The router's report runs to eight sections and a few pages, so it's easier to see how a report works on something small first. Regressions and seeds uses a 4-bit counter that is meant to stop at 15. With the planted bug `wrap` switched on, the counter wraps from 15 to 0 instead. The example's test, `count_test`, drives random `inc` and `clr` for 300 cycles and checks the count against a model in every cycle. Here is most of a report for it, in the template's sections, with every number taken from our runs of `guide-examples/regressions-and-seeds/`:

```
## 1. Summary

The counter in satcnt.sv was verified against three rules: clr sets it to 0, inc adds
one, and at 15 it stays at 15. count_test checks the count in every cycle against a
model. It passed on seeds 1 to 10, and it caught the planted wrap bug, but only five of
the ten seeds got the counter to 15, so the rule about 15 was tried on those five only.

## 4. Regression

make seeds UVM_TESTNAME=count_test SEEDS="1 2 3 4 5 6 7 8 9 10"

    10 ok  (10 runs)

## 5. Coverage

    verilator_coverage: covergroup : 100.0% (4/4)

The four bins are the merge of all ten runs. Run by run (the [COUNT] line at the end of
each log), the counter reached 15 on seeds 2, 3, 8, 9 and 10, and never on seeds 1, 4,
5, 6 and 7.

## 6. Bugs

make run UVM_TESTNAME=count_test SEED=n BUG=wrap, for n = 1 to 10: caught on seeds 2,
3, 8, 9 and 10, MISSED on seeds 1, 4, 5, 6 and 7. The first error on seed 2:

        UVM_ERROR  guide-examples/regressions-and-seeds/seeds_tb.sv(72)  @  1975000:
uvm_test_top [COUNT] count is 0, expected 15

## 7. What this doesn't show
```

```
- On half the seeds the counter never reached 15, so the rule that matters most was
  tried on five seeds of ten. A run on one of the other five would have passed the
  wrap bug.
- The coverage line is 100%, but it is the union of ten runs, and five of them never
  hit the bin for 15.
- Nothing here resets the counter after time 0, so a clear and a reset are never seen
  in the same cycle, and nothing checks the count while rst_n is 0.
- The simulator has two states, so nothing here says anything about X.

## 8. Sign-off

This counter holds at 15, clears on clr and counts on inc, in every cycle of count_test
on seeds 1 to 10, which got it to 15 and incremented it there on seeds 2, 3, 8, 9 and
10. To check it:

    make seeds UVM_TESTNAME=count_test SEEDS="1 2 3 4 5 6 7 8 9 10"   # 10 ok  (10 runs)
```

Sections 2 and 3, which describe the environment and the plan's status, are left out here because the counter's environment is one test, and its plan is three rows.

Numbers and the commands behind them

Every number in that report comes from the output of a command that the report names, and the lines in the indented blocks are pasted character for character, so a reader can check each one. A summary in your own words can't be checked like that. "The regression passed" could mean any number of tests on any seeds, with or without a run that crashed. The line `10 ok  (10 runs)`, under the command that printed it, says that there were ten runs, that all ten passed and that none was BROKEN, which is how `make seeds` marks a run that crashed or stopped on a `UVM_FATAL`. A reader can run the same command and compare that one line with yours.

The comment at the top of the template asks for the same thing ("paste the line the tool printed, not your summary of it"). In Lab 07 you keep that output as you go. In Part 6 you send the regression through `tee` into `results/`, in Part 7 you send the three `make check-all` runs there too, and in Part 8 you copy that folder into your repository beside the report.

The template's "Environment" line asks for the commit of your repository that the report describes, because a seed repeats a run only on the same code. (Regressions and seeds shows that one more random call, or one more sequence, anywhere in a test can change the random choices made after it, in other threads as well as its own.) The last step of Part 8 is to open the repository on GitHub and check that a reader who has only that page could find the command behind every number in your report.

The sign-off sentence

Section 8 of the template asks for one sentence that starts "This router..." and keeps going until it's true, so the sentence tends to come out long. For the counter you'd like to write "This counter saturates at 15", but your evidence is `count_test` on seeds 1 to 10, and only five of those seeds got the counter to 15, so the report's sentence names the test, the seeds and those five. Because the sentence has to be true, you have to put into it every condition that your evidence depends on, and a reader can then decide whether those conditions are good enough for what they need.

Section 7: the limits

"What this doesn't show" is the section people leave short, and it's the one a reader who has to rely on your claim needs most, because it's the only place where they can find out where your evidence stops, and without it they would have to take the claim on trust or check it from scratch. In the counter's report, section 7 says that on half the seeds the counter never reached 15, so the rule that the wrap bug breaks was never tried on them. For the router, the items come from your own work in Lab 07:

- the rows of your plan that have no checker, or whose checker no sequence exercises, which you put in the plan's Open items in Part 2;

- coverage bins that stayed empty, and why;
- the seeds you ran, and the ones you didn't;
- anything that depends on X, which a two-state simulator like Verilator doesn't have; for example, the specification warns that the router's state before its first reset is undefined, but on Verilator it happens to start at all zeros;
- the fact that catching the fifteen planted bugs shows only that your environment catches those fifteen, and says nothing about a bug unlike all of them.

The hunt table

Section 6 has a table with one row for each mystery bug that you were dealt in Part 4, and its columns are in the order in which you fill them in. You copy the first check that fired from `make hunt`, and then write your diagnosis, which is the rule that the router broke and the file in `labs/common/rtl/` that you'd look in. Only after that do you run `make verdict`, and then you fill in the bug's name and the sequence that made it show. You write the diagnosis before you see the answer because it records how well a failure message from your environment led you to the fault, and because it's a habit you'll need when there is no `make verdict` to ask.

A hunt that comes back `quiet` gets a row too. `quiet` means that nothing `router_test` drives tells that design from a correct one, so in that row you write which rows of your plan could be hiding the bug. In our runs, a hunt of B15 with the starter environment came back quiet, because the starter resets the router only at time 0, when every buffer is empty. With the assertions and the resets with traffic in flight that you write in Part 5, `make check-all` catches it.

Check yourself

1. A report's summary says "the regression passes and coverage is 100%". Name three things a reader can't tell from that sentence.

   Answer: A1, in Answers at the end.

2. Your Lab 07 report's section 7 is empty. Name two things from the lab that belong there.

   Answer: A2, in Answers at the end.

3. Why does the hunt table ask for your diagnosis before the verdict?

   Answer: A3, in Answers at the end.

Further reading

- Writing the bug report, in DD104: the other kind of report, for one bug, built around a test that reproduces it.
- Bug Lifecycle: what a useful bug report contains (the seed, the commit, expected against observed) and how a bug moves from open to closed. Its command lines are for one commercial simulator; on Verilator the seed is `+verilator+seed+N`.
- Sign-Off Criteria: the gates a report usually has to show were met.
- Verification Metrics: coverage, bug rates and escapes as measures of how far verification has got.

Next: Lab 07: Capstone.

Answers

A1. Check yourself

Which tests and seeds ran, and how many runs that was; which command produced the result, so that they could run it themselves; which commit of the code the claim is about; and whether any single run hit every bin, or which bins only some seeds reached. The counter's coverage was 100% over ten seeds while five of them never reached 15.

A2. Check yourself

Any two of: the plan's open items (rows with no checker), the seeds you didn't run, the bins that stayed empty and why, the fact that a two-state simulator says nothing about X, and the fact that catching fifteen planted bugs says nothing about a sixteenth unlike them.

A3. Check yourself

If you wrote it after the verdict, it would only repeat the bug's name, and it would say nothing about how directly your environment's first failure message led you to the rule the design broke and the block it's in.

27. Lab 07: Capstone

In this lab you take your Lab 06 testbench to sign-off. You'll write a verification plan from the specification, run a regression on ten seeds and read its coverage, and hunt three mystery bugs, each dealt from all fifteen in the catalogue. The package you start from misses one of the fifteen. You'll catch it by writing assertions for the rules about reset and a sequence that resets the router while packets are in flight, so that `make check-all` prints `15 of 15 caught.` Then you'll write a sequence that takes both drop counters to 255, which fills the last two coverage bins that a test can reach, and finish with a verification report. The plan, the report, your testbench and its results go into a GitHub repository of your own.

Package: `labs/lab-07-capstone/`

Before you start

- Finish Lab 06.
- Read Verification plans and sign-off and Writing the verification report, and reread Regressions and seeds.
- Keep the router's specification open for the whole lab, with the router in one page beside it if you like. The plan you write in Part 2 is a list of the specification's rules.
- You need a GitHub account, and a token the workbench can push with. If you made one in IC102-A Lab 01, Part 2 and it hasn't expired, use it. Otherwise make one the way that part describes. This lab needs only the Contents permission, read and write, because your repository has no workflows.
- In a terminal on the workbench desktop:

```
. /foss/designs/common/.designinit
mod dd105-a
cd labs/lab-07-capstone
```

The package is the Lab 06 testbench with our solution to Lab 06 filled in: the four registers' fields and the adapter, the predictor's connections, the `irq` check and `router_lab06_vseq`'s steps. If your own Lab 06 printed `5 of 5 caught.`, your code did the same job, but keep ours, because this lab adds to the files you edited. Until now the router has been reset only at time 0, and nothing in the testbench could reset it again. The table lists what we added for you.

File	What Lab 07 adds
`tb/router_ifs.sv`, `tb/router_tb_top.sv`	a reset the testbench can drive: `rst_req_n` in `router_sys_if`, with `rst_soon` beside it to warn the APB driver that a reset is coming, and `rst_n` in the top module, made from `rst_req_n` and the reset at time 0
`tb/router_agents_pkg.sv`	the reset driver (`sys_agent`, `router_reset_seq`); a source driver that abandons its packet at a reset (rule I3), and an APB driver that keeps `psel` at 0 during one and starts an interrupted transfer again (rule A4)
`tb/router_env_pkg.sv`	in the scoreboard, `reset_model()`, which at a reset throws away every beat it expected and puts its own models and the register model's mirror back to their reset values; four corners that the scoreboard counts (a reset with beats still to come out, a reset with a packet half accepted, and a drop

File	What Lab 07 adds
	of each kind with its counter already at 255), and the covergroup `cg_corner`, with a bin for each
`tb/router_sva.sv`, `tb/router_sva_bind.sv`	the APB port and `irq` in the bound module, and a counter for each rule about reset; the assertions are yours (Part 5)
`tb/router_vseq_pkg.sv`	`reset()` in the base sequence; `router_reset_vseq` and `router_saturate_vseq`, whose bodies are yours (Parts 5 and 6); `router_lab07_vseq`, four phases of random traffic whose resets are yours (Part 5)
`tb/router_tests_pkg.sv`	`router_test` runs `router_lab07_vseq` last; three new tests, `router_lab07_test`, `router_reset_test` and `router_saturate_test`
`Makefile`	`BUG_LIST`: all fifteen bugs, B15 first, because `make check` grades the first

Time: about 3 h 30 min. About 20 minutes of that is waiting for the machine, as we measured it on the course's 2 CPUs. The first build took 130 s, each of the three rebuilds after your edits took 114 to 118 s, each of the two regressions about 4 minutes, and each of the three runs of `make check-all` about 75 s. You'll spend most of the rest writing the plan, eight assertions, two sequences and the report.

Part 1: Your repository and the first build

1. Start the build first, so that you can make the repository in step 2 while it runs:

   ```
   make build
   ```

   The banner says a cold build takes 1.5 to 2.5 minutes. Ours, with an empty ccache, took 130 s. Near the end it printed Verilator's summary:

   ```
   - Verilator: Walltime 130.147 s (elab=0.929, cvt=3.622, bld=124.720); cpu 5.833 s
   on 2 threads; allocated 382.234 MB
   ```

   and then a line saying that it built `build/sim` in 130 s.

2. While it builds, make the repository. On GitHub, create a new repository under your own account named `dd105-capstone`, with a README so that it starts with one commit. It can be private, because nobody needs to read it yet.

3. Clone it beside the course, and tell git who you are in it:

   ```
   mkdir -p /foss/designs/projects && cd /foss/designs/projects
   git clone https://github.com/YOUR-USERNAME/dd105-capstone.git
   cd dd105-capstone
   git config user.name "Your Name"
   git config user.email "ID+USERNAME@users.noreply.github.com"
   git config credential.helper 'cache --timeout=28800'
   cd /foss/designs/modules/dd105-a/labs/lab-07-capstone
   ```

   Use your GitHub no-reply address, and leave out `--global`. A global setting goes in the container's home folder, which is thrown away when the workbench restarts. Without `--global` the settings go in `.git/config`, which is inside your repository, on your own disk, so they stay. In IC102-A Lab 01, Part 2 says where to find your no-reply address, and Part 3 explains the token cache and `--global`. You'll write your plan and your report in the repository itself. The testbench stays here in the course clone while you work on it, because it builds only from inside `labs/`, and Part 8 copies it across.

4. When the build has finished, run the graded test, `router_test`, and pull the scoreboard's summary out of the log:

   ```
   make run
   grep -o '\[SB\] [0-9a-z].*' build/runs/router_test-s1/run.log
   ```

   You should see `ok: 0 UVM_ERROR, 0 UVM_FATAL  7.38 s` from `make run`, and from `grep`:

   ```
   [SB] 1733 packets in: 479 to output 0, 381 to output 1, 260 NOROUTE, 613 FULL; the
   highest level a header saw was 16
   [SB] beats checked: 2183 on output 0, 1700 on output 1; 45 DROP_CNT reads checked
   [SB] registers: 884 reads, 65 of STATUS checked; 492 transfers to bad addresses; irq
   checked in 19951 cycles, wrong in 0
   [SB] resets: 0 after the one at time 0, 0 expected beats lost; corners hit:
   reset_with_beats=0 reset_mid_packet=0 noroute_saturated=0 full_saturated=0
   ```

5. At the end of Lab 06 your `router_test` checked `irq` in 14840 cycles. This one checks it in 19951. What ran in the other 5111 cycles, and did anything before them change?

   Answer: A1, in Answers at the end.

Check: `make run` passes, and `git -C /foss/designs/projects/dd105-capstone log --oneline` shows the one commit that GitHub made.

Part 2: The verification plan

Which of the specification's rules does this environment check, and which does it never make happen?

1. Copy the template into your repository:

   ```
   cp PLAN-template.md /foss/designs/projects/dd105-capstone/PLAN.md
   ```

   It has a row for each rule in the specification. The input, output and APB rules keep the specification's numbers (I1 to I7, O1 to O8, A1 to A9), and the template gives labels of its own (RT1, DR2, RS2 and so on) to the rules that the specification states in prose. Three rows, O1, RT1 and A7, are filled in as examples.

2. Fill in the table under "How the environment checks". There are three checkers: the scoreboard (`write_in`, `write_out`, `write_apb` and `write_sys` in `router_scoreboard`, in `tb/router_env_pkg.sv`), the register model's mirror (the predictor and `set_check_on_read(1)` in `router_env::connect_phase`, which you connected in Lab 06), and the assertions in `tb/router_sva.sv`. The comment at the top of each file says what it covers.

3. Run `grep -n TODO tb/*.sv` first, because the parts that this lab leaves for you are marked `TODO`. Then, for every row, fill in "Checked by" and "Made to happen by", and write "nothing" where nothing does. Every virtual sequence in `tb/router_vseq_pkg.sv` has a comment over each step saying what it sets up, and `router_test` in `tb/router_tests_pkg.sv` lists what the graded test runs.

4. Mark the rows you think `router_test` never reaches. Then look at what the scoreboard says about resets in your Part 1 run:

   ```
   grep -o '\[SB\] resets.*' build/runs/router_test-s1/run.log
   ```

   It's the line you saw in Part 1, `resets: 0 after the one at time 0`, with every corner at 0. Which rows of your plan need more than the reset at time 0, and which have no checker?

   Answer: A2, in Answers at the end.

Check: every row of your plan has an entry in "Checked by" and "Made to happen by", and the rows that say "nothing" include the ones in the answer above.

Part 3: A seed regression

Does a regression that passes on every seed tell you the environment is finished?

1. `make regress` runs every test in `TESTS` on seeds 1 to 10. Look at `TESTS` in the Makefile, and write down how many runs you expect and how many will pass.
2. Run it, and time it:

   ```
   time make regress
   ```

   It took us 3 min 57 s. You should see `110 ok  (110 runs)` and then the coverage of those runs:

   ```
    verilator_coverage: covergroup : 81.7% (49/60)
   ```

   with, at the end of the bin list:

   ```
    11 bins never hit:
       __vlAnonCG_cg_corner.cp_corner.reset_with_beats
       __vlAnonCG_cg_corner.cp_corner.reset_mid_packet
       __vlAnonCG_cg_corner.cp_corner.noroute_saturated
       __vlAnonCG_cg_corner.cp_corner.full_saturated
       __vlAnonCG_cg_pkt.x_fate_level.full_x_at_8
       __vlAnonCG_cg_pkt.x_fate_level.full_x_empty
       __vlAnonCG_cg_pkt.x_fate_level.full_x_low
       __vlAnonCG_cg_pkt.x_fate_level.out0_x_at_9
       __vlAnonCG_cg_pkt.x_fate_level.out0_x_high
       __vlAnonCG_cg_pkt.x_fate_level.out1_x_at_9
       __vlAnonCG_cg_pkt.x_fate_level.out1_x_high
   ```

3. `cg_corner` has a bin for each of the four corners in `corner_e`, near the top of `tb/router_env_pkg.sv`, and the scoreboard counts each corner and sends it to `cg_corner` as it happens. Which of the eleven empty bins could any test ever hit, and which rows of your plan are they?

   Answer: A3, in Answers at the end.

4. Fill in "Seen in coverage" for every row, from the bin list (`build/coverage/regress/bins.txt` has every bin with its count) or from the `corners hit:` part of the `[SB] resets` line at the end of each run's log. Then write your sign-off criteria under "Sign-off criteria", each one a command and the output it must print.
5. Commit and push the plan:

   ```
   cd /foss/designs/projects/dd105-capstone
   git add PLAN.md && git commit -m "Verification plan for the router" && git push
   cd -
   ```

   git asks for your username and, as the password, the token.

Check: `110 ok  (110 runs)` and `81.7% (49/60)`, and your plan on GitHub with its sign-off criteria written.

Part 4: Mystery hunts

Can you tell which rule a design breaks from your environment's log alone?

1. Deal a mystery bug, and run `router_test` against it:

   ```
   make new
   make hunt
   ```

`make new` seals one of the fifteen bugs in `.hunt` without telling you which. `make hunt` runs the graded test with that bug switched on, which takes about 7 s, or less for a bug that piles up 50 UVM_ERRORs early, because the base test has UVM stop a run at the 50th. On one of our deals, the hunt printed:

```
  caught  the mystery bug                        1 UVM_ERROR                  7.4s
       UVM_ERROR [IRQ] irq is 1 from cycle 5909, but STATUS = 001 and INT_EN = 000
make it 0
```

2. Copy `REPORT-template.md` into your repository as `REPORT.md`:

```
cp REPORT-template.md /foss/designs/projects/dd105-capstone/REPORT.md
```

Before you ask for a verdict, fill in the first three columns of its hunt table. They hold the deal's number, the first check that fired, copied from the hunt, and your diagnosis, which is the rule the router broke and the file in `labs/common/rtl/` that you'd look in. The specification's Building the router section says what each file holds, and in the log from `make hunt VERBOSE=1` the `[VSEQ]` lines say which sequence was running when the first error came. What would you write for the hunt above?

Answer: A4, in Answers at the end.

3. Ask for the grade:

```
make verdict
```

It runs the golden design first (row 1) and then the mystery bug (row 2), about 15 s in all. For the hunt above you should see:

```
     ok  1. the golden design  must PASS         0 UVM_ERROR, 0 UVM_FATAL       7.4s
 caught  2. the mystery bug    must FAIL         1 UVM_ERROR                    7.3s
```

then the same error, and:

```
VERDICT: PASS
  The bug was B02_irq_ignores_int_en.
  It hides behind: turning on every interrupt enable before looking at irq
  It teaches: an enable is a mask, and a mask has to be tested off as well as on
```

Fill in the last two columns of the row.

4. Deal again, until the table has three rows. If a hunt prints this instead:

```
  quiet  the mystery bug                       0 UVM_ERROR, 0 UVM_FATAL       7.4s
--------------------------------------------------------------------------------
Nothing your test drives tells this design from a correct one. The
bug is in there; your stimulus or your checks have not reached it.
`make cov` shows what the stimulus covered.
```

keep that one. `make new` replaces whatever is in `.hunt`, so copy the file aside (`cp .hunt hunt-quiet.sealed`) before you deal again, and copy it back to `.hunt` when you come to its verdict in Part 7. In its row, write which rows of your plan could hide a bug from `router_test`. `make reveal` would tell you the bug's name, and then remind you that the name isn't the deliverable, because `make verdict` still needs row 2 to fire.

Check: your report's hunt table has three rows, each with its diagnosis written before its verdict.

Part 5: Reset with traffic in flight

What does the environment need before it can check a reset that happens while the router is busy?

1. Read `reset_model()` and `write_sys()` in `router_scoreboard`. Lab 06's `write_sys()` reset only its model of `STATUS` and `INT_EN`, because the router was never reset after time 0. Write down what the Lab

06 scoreboard would have reported, on the golden router, after a reset with a beat waiting on a stalled output.

Answer: A5, in Answers at the end.

2. Write the assertions marked `TODO (Part 5)` in `tb/router_sva.sv`: `a_I7`, `a_O4_rst`, `a_O4_after` and `a_A4`, and `a_R_pready`, `a_R_pslverr`, `a_R_prdata` and `a_R_irq` for the specification's Reset table. All but `a_O4_after` check rules about reset itself, so unlike Lab 05's assertions they have no `disable iff (!rst_n)`; the specification's Valid/ready rules section says why. `a_O4_after` checks the cycles after a reset, so it keeps the `disable iff`, and it needs the two flip-flops that its comment describes. As in `a_O1`, each failure should increment the assertion's counter and call `$error` with the rule's name.
3. Predict which of the eight will fail on the golden router. Then rebuild and run:

```
make build
make run
```

The rebuild took us 118 s. You should see the same `[SB]` lines as in Part 1 and `DD105: TEST PASSED`. None of the assertions fails. The golden router obeys every one of these rules that is its own, and `a_A4` checks a rule of the APB master's, which our APB driver obeys. If one of yours does fail, its `%Error` line names it, and the mistake is in that assertion. So far the only reset they have seen is the one at time 0, three cycles long, when the router holds nothing.

4. Write `router_reset_vseq::body()` in `tb/router_vseq_pkg.sv`. Its `TODO` lists the situations to reach. The first is the specification's example E8, in which output 0 is stalled with a packet in its buffer and the next packet is half accepted when `rst_n` falls. Fork the reset beside the traffic, the way `router_lab05_vseq` forks its route writes beside its random packets, and after each reset let the sinks take beats, call `setup()` and send more. `reset()` gives the APB driver a cycle's warning and then holds `rst_n` low, so a reset starts two cycles after you call it.
5. In `router_lab07_vseq`, write the line that runs your sequence (the `TODO (Parts 5 and 6)`; Part 6 adds the other line), and the two `TODO (Part 5)`s in the random phases, which set `do_reset` and add the thread that resets the router. Then add `router_reset_test` to `TESTS` in the Makefile.
6. Write down what you expect the `corners hit:` part of the `[SB] resets` line to say for your test. Then rebuild (114 s for us) and run it:

```
make build
make run UVM_TESTNAME=router_reset_test
grep -E 'reset from|abandoned|\[SB\] resets' build/runs/router_reset_test-s1/run.log
```

With the reference solution's sequence, the first lines are from the E8 reset:

```
UVM_INFO       labs/lab-07-capstone/tb/router_env_pkg.sv(397)       @       305000:
uvm_test_top.env.sb [SB] reset from cycle 31: 4 expected beats are lost, 2 beats of
the packet coming in
UVM_INFO      labs/lab-07-capstone/tb/router_agents_pkg.sv(202)      @      305000:
uvm_test_top.env.src.drv [SRC] reset: abandoned dst 0, 8 beats: 01 e5 9f 2b 57 45
55 fe after 2 beats
```

then, in the reset during register traffic,

```
UVM_INFO      labs/lab-07-capstone/tb/router_agents_pkg.sv(591)      @      1435000:
uvm_test_top.env.apb.drv [APB] reset: abandoned a read of 08; it will start again
after the reset
```

and the run ends with `[SB] resets: 4 after the one at time 0, 11 expected beats lost; corners hit: reset_with_beats=3 reset_mid_packet=1 ...` and `DD105: TEST PASSED`, in 0.35 s. If your log has no `[APB] ... abandoned` line, your reset fell between two transfers, and nothing in your test has made rule A4 happen yet. In our sequence a read and a write together take 7 cycles, idle cycles included. After `cycles(8)`, `reset(1)` raised `rst_soon` as the second read's setup cycle ended,

and `rst_n` fell after the read's first access cycle. Move your reset a cycle at a time until it lands in a transfer.

Check: `router_reset_test` passes on the golden router, with `reset_with_beats` and `reset_mid_packet` above 0, and an `[APB] ... abandoned` line in its log.

Part 6: Counters at 255

Does each drop counter stop at 255, and how will you know that the regression got there?

1. Write `router_saturate_vseq::body()`. Its `TODO` names the two corners. The specification's Sizes section says that a stream of 1-beat packets for a disabled route makes one drop per cycle, so it takes a counter to 255 in 255 cycles. Read `DROP_CNT` on the way past 255, but only between bursts. The scoreboard compares a read with the drops it has counted when the read completes, and a read returns the counter as it was a cycle earlier, so while drops are still coming in the two can differ, and the check would report an error that isn't the router's. Then clear the counters with a write and read them again.
2. Add the line that runs it to `router_lab07_vseq`, add `router_saturate_test` to `TESTS`, then rebuild (116 s for us) and run the test:

   ```
   make build
   make run UVM_TESTNAME=router_saturate_test
   ```

   With the reference solution you should see `noroute_saturated=5  full_saturated=5` in the `[SB] resets` line, and `DD105: TEST PASSED`, in 0.45 s.
3. Will closing these two bins catch a bug that your environment misses now? Write down your answer. Then run the regression again, keeping its output:

   ```
   mkdir -p results
   time make regress | tee results/regress.txt
   ```

   It took us 4 min 12 s. You should see `130 ok  (130 runs)` and:

   ```
    verilator_coverage: covergroup : 88.3% (53/60)
   ```

   with only the seven bins of `x_fate_level` never hit.

   Answer: A6, in Answers at the end.
4. Update the Status column of your plan, then commit and push it.

Check: `130 ok  (130 runs)` and `88.3% (53/60)`.

Part 7: Sign-off

Does your environment meet the plan's sign-off criteria, and what did your work in Parts 5 and 6 add?

1. If you kept a quiet mystery bug, copy its file back to `.hunt`, run `make verdict`, and fill in its row.
2. Write down how many of the fifteen you expect `router_test` to catch now. Then run the catalogue on three seeds, keeping the output:

   ```
   make check-all | tee results/check-all-s1.txt
   make check-all SEED=2 | tee results/check-all-s2.txt
   make check-all SEED=3 | tee results/check-all-s3.txt
   ```

   Each took us about 74 s. With the reference solution, seed 1 printed:

   ```
        ok  the golden design                      0 UVM_ERROR, 0 UVM_FATAL       7.6s
   ------------------------------------------------------------------------------
     caught  tier 6  B15_output_valid_survives_reset  50 UVM_ERROR, 45 assertions    6.6s
     caught  tier 1  B01_status_write_clears_all     3 UVM_ERROR                    7.6s
   ```

```
  caught  tier 1  B02_irq_ignores_int_en          6 UVM_ERROR                    7.6s
  caught  tier 1  B03_route3_readback_swapped     5 UVM_ERROR                    7.7s
  caught  tier 1  B04_misaligned_address_accepted  50 UVM_ERROR                  1.3s
  caught  tier 2  B05_dst_field_off_by_one        50 UVM_ERROR                   1.7s
  caught  tier 2  B06_disabled_route_forwards     50 UVM_ERROR                   1.8s
  caught  tier 2  B07_full_counted_on_disabled_route 6 UVM_ERROR                 7.6s
  caught  tier 3  B08_admission_off_by_one        50 UVM_ERROR                   1.8s
  caught  tier 3  B09_write_pointer_power_of_two   50 UVM_ERROR                  1.7s
  caught  tier 4  B10_output_reloads_while_stalled 50 UVM_ERROR, 256 assertions  1.8s
  caught  tier 4  B11_cut_packet_has_no_last      48 UVM_ERROR, 45 assertions    7.6s
  caught  tier 5  B12_out1_uses_out0_ready        50 UVM_ERROR, 3 assertions     1.6s
  caught  tier 5  B13_admission_checks_other_output 50 UVM_ERROR, 1 assertion    1.7s
  caught  tier 6  B14_enable_resets_to_one        4 UVM_ERROR, 23 assertions     7.6s
-------------------------------------------------------------------------------
 15 of 15 caught.
```

Seeds 2 and 3 printed `15 of 15 caught.` too, with B15 at `10 UVM_ERROR, 53 assertions` and `5 UVM_ERROR, 23 assertions`. Your error counts will differ, because your sequences aren't ours.

3. As shipped, the package caught 14 of the 15. Which one did it miss, and which check fires first on it now?

   Answer: A7, in Answers at the end.

4. B14 makes `CTRL.EN` reset to 1. The register model caught it before this lab, in Lab 06, Part 2. Which check fires first on it now?

   ```
   grep -m1 -E '%Error|UVM_ERROR [^:]' build/grade/B14/run.log
   ```

   Answer: A8, in Answers at the end.

5. Look at B15 beside the golden router, in the E8 step of the reset sequence:

   ```
   make waves UVM_TESTNAME=router_reset_test
   make waves UVM_TESTNAME=router_reset_test BUG=B15
   surfer build/runs/router_reset_test-s1/waves.fst &
   surfer build/runs/router_reset_test-s1-B15/waves.fst &
   ```

   Add `clk` from `router_tb_top`, `rst_soon` and `rst_n` from `sys`, `valid`, `ready` and `data` from `in_s` and `out0`, `level0` from `dut`, and `n_O4_rst` and `n_O4_after` from `dut.u_sva.u_out0`, and look at 230 to 380 ns:

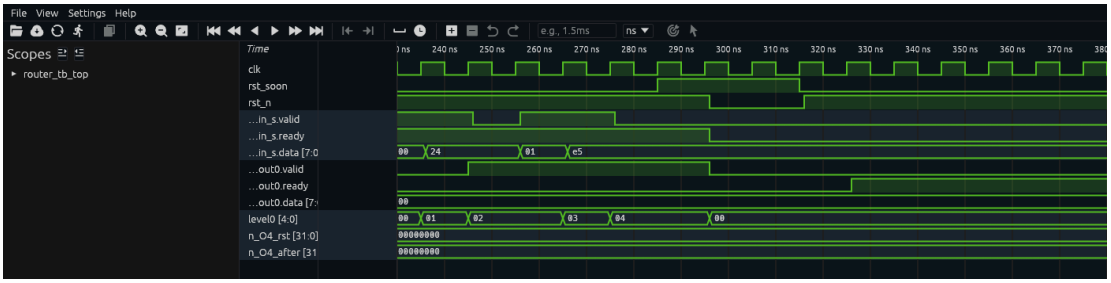


Figure 27.1: The E8 reset on the golden router, in Surfer

The golden router, with the reference solution's sequence. Output 0 has been stalled since 226 ns. The 2-beat packet `00 24` went in first, and its header `00` has been waiting on output 0 since 245 ns, when `out0_valid` rose. The next packet's first two beats, `01 e5`, went in after it, so `level0` is 4. (After 276 ns `in_valid` is 0, because that packet's third beat waits out a 3-cycle gap.) `rst_soon` rises at 285 ns, a cycle before the reset, and `rst_n` falls at 296 ns; `out0_valid`, `in_ready` and `level0` fall with it, without waiting for a clock edge. The sink is ready from 326 ns, and nothing comes out, as example E8 says.

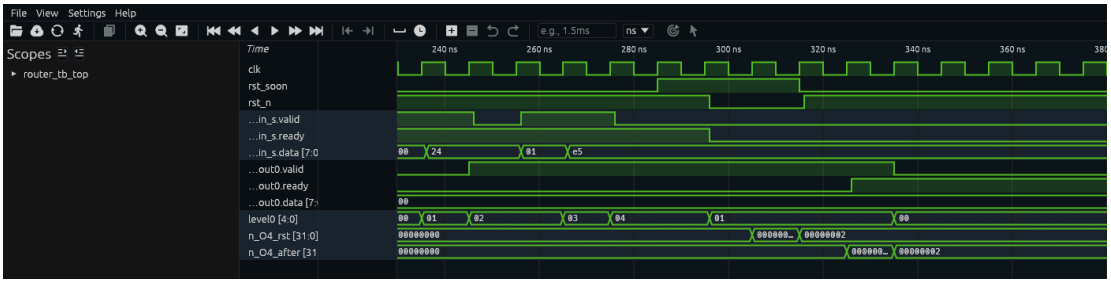


Figure 27.2: The same reset with B15 on, in Surfer

B15, from the same seed. The reset empties the buffer behind the output but not the output register, so `out0_valid` stays 1 through the reset with `00` still on `out0_data`, and `level0` reads 1, because the level counts the beat on the pins. `a_O4_rst` counts a failure at each of the two rising edges inside the reset (305 and 315 ns), and `a_O4_after` at the next two. The sink takes `00` at the edge at 335 ns, and the scoreboard reports it: `[OUT] output 0: got 00 in cycle 34, and no beat was expected`.

6. Go through your plan's sign-off criteria one at a time, against the files in `results/`.

Check: `15 of 15 caught.` on three seeds, with the reset assertions and the two sequences that you wrote.

Part 8: The report

Can someone else check your claim without redoing your work?

1. Fill in `REPORT.md` section by section. Paste each number as the tool printed it, from `results/` and from your notes. Section 7, "What this doesn't show", is the only place where a reader who has to rely on your claim can find out where your evidence stops, so don't leave it short. Your plan's Open items from Part 2 go there.
2. Copy your testbench and its outputs into the repository, and push:

```
R=/foss/designs/projects/dd105-capstone
mkdir -p $R/tb $R/results
cp Makefile $R/ && cp tb/*.sv tb/*.vlt $R/tb/ && cp results/*.txt $R/results/
cd $R
git add -A && git commit -m "The capstone environment, its results and the report"
&& git push
cd -
```

   The last line takes you back to the lab folder. The copy is safe to repeat from there, so run the block again whenever you change a file and want the repository to have it.
3. Open the repository on GitHub, and check that a reader who has only that page could find the command behind every number in your report.

Check: your repository on GitHub has `PLAN.md`, `REPORT.md`, `Makefile`, `tb/` and `results/`.

Finishing

You're done when `make check-all` prints `15 of 15 caught.` with the reset assertions and the reset and saturation sequences that you wrote, and the `REPORT.md` in your GitHub repository quotes that run.

Questions:

1. B15 was caught by an assertion and by the scoreboard. If you could keep only one of the two checks, which would you keep?

   Answer: A9, in Answers at the end.
2. Part 3's regression passed 110 runs out of 110. What would a reader who saw only that line have believed, and what did your plan say at the time?

   Answer: A10, in Answers at the end.

Submit: nothing yet. Your repository is the record of your capstone. We plan a certification process that will review capstone repositories, but it doesn't exist yet.

Going further (optional)

- Close one of the reference plan's open items (`solutions/PLAN.md`). Assertions for I5 and I6, with `CTRL.EN` as written over APB kept beside them, would give the `in_ready` rows a checker, and a check of `LEVEL` reads against the levels the scoreboard already counts would close RG4. Run `make check-all` afterwards. If your new check fires on the golden router, the mistake is in the check.
- Run more seeds, with `make regress NSEEDS=50`. The simulator is built at `-O0`, which keeps builds short but makes it run slower, so `uvm.mk` says to use `OPT_FAST=-Os` for long regressions, at the cost of a longer build.
- Deal more mystery bugs, until you've had one from every tier.

Next on the Course path: this is the last lab.

Answers

A1. Part 1: Your repository and the first build

`router_lab07_vseq`, which `router_test` now starts after everything Lab 06's `router_test` ran. Its four phases of random traffic take those cycles. Each phase sends 150 packets on a random routing table of its own and random interrupt enables, while both sinks change their modes. It doesn't run your two sequences yet. It goes last so that the random choices it makes can't move anything before it. With `router_lab07_vseq` left out, `router_test` printed Lab 06's lines number for number (`1133 packets in: 282 to output 0, 226 to output 1, 195 NOROUTE, 430 FULL`, and `irq checked in 14840 cycles`), so nothing we added changes a run that has no reset in it.

A2. Part 2: The verification plan

At time 0 the router holds nothing, so RS2 and RS3 (a reset with beats in a buffer, and a reset in the middle of an input packet), the second half of O4 and the source's rule I3 all need a reset with traffic in flight. I7, O4, the first half of A4 and RS1 have no assertion yet, because theirs are the `TODO (Part 5)` comments in `tb/router_sva.sv`. Of RS1's four values, only `irq` is checked, by the scoreboard. DR2, the rule that each counter stops at 255, needs 255 drops with no clear or reset between them, but `router_inject_test` and `router_lab07_vseq` both clear `DROP_CNT` at the end of each of their phases.

Nothing in the environment checks `in_ready` (I4, I5, I6, RT5), the cycle a beat appears (O8), that the outputs come from flip-flops (O3), what `CTRL.EN` does (EN1), the APB master's rules (A1 to A3), `pready` (A5, A6) or `LEVEL` (RG4), and nothing in this lab adds a check for them, so those rows stay "nothing". Put them in the plan's Open items, because Part 8's report has to say what the environment doesn't show.

A3. Part 3: A seed regression

Only the four in `cg_corner`. The seven in `x_fate_level` are ones the specification rules out, as they were in Labs 04 to 06, because a packet is FULL only at a level of 9 or more, and written to an output only at 8 or less. `noroute_saturated` and `full_saturated` are drops whose counter is already 255 (row DR2), and `reset_with_beats` and `reset_mid_packet` are a reset with beats in a buffer and a reset with a packet half accepted (RS2 and RS3). You found the same rows in Part 2 by reading the environment, and the regression's bin counts point at them too. So a regression that passes on every seed doesn't tell you the environment is finished. 110 passes mean that no check fired in 110 runs, and the four empty bins are situations that the stimulus never made, so no check ever saw them.

A4. Part 4: Mystery hunts

The rule is IRQ1, that `irq` is the OR of `STATUS & INT_EN`. In cycle 5909 `STATUS.NOROUTE` is set and its enable is 0, and the router raises `irq` anyway, so it isn't masking with `INT_EN`. The interrupt is in `router_regs.sv`, with the registers. In the full log from `make hunt VERBOSE=1` the error comes between `starting router_lab06_vseq` and `router_lab06_vseq is done`, so it came from the step you wrote in Lab 06, Part 5, which drops a packet while `INT_EN` is still 0.

A5. Part 5: Reset with traffic in flight

It would have reported errors on a router that is right. The beats it expected would still be queued, so the first beats out of that output after the reset would be compared with beats from before it, and any it never matched would be reported at the end as `[LEFT]`. Its routing table, its levels and its drop counts would still hold the values from before the reset, so the next packets would be predicted wrongly. Lab 06's scoreboard didn't reset the register model either, so the mirror would still say `CTRL.EN` is 1, and the register model would fail the first read of `CTRL` after the reset. `reset_model()` throws all of that away in

the first reset cycle. Lab 06's drivers needed changing too, because the source driver would go on offering the rest of its packet after the reset, and the APB driver would keep `psel` at 1 through a reset that came in the middle of a transfer.

A6. Part 6: Counters at 255

No. None of the fifteen bugs changes what a counter does at 255. We ran `make check-all` with Part 5's work in and none of Part 6's, and it printed `15 of 15 caught.` on seed 1, with B15 at `50 UVM_ERROR, 18 assertions`. The new sequence reads `DROP_CNT` past 255, and a check that you already had looks at those reads. The scoreboard compares every `DROP_CNT` read with its own counts, which stop at 255, so it would catch a counter that wraps. Until now nothing gave row DR2 of your plan any evidence either way.

A7. Part 7: Sign-off

It missed B15. We ran `make check-all` on the package before Parts 5 and 6, and it printed:

```
 MISSED  tier 6  B15_output_valid_survives_reset  0 UVM_ERROR, 0 UVM_FATAL       7.4s
```

and, after the other fourteen rows,

```
14 of 15 caught.
Still hiding from this test:
  B15_output_valid_survives_reset
     hides behind: resetting only at time 0, when every buffer is already empty
```

With B15 on, a reset doesn't clear an output's register, so a beat that was waiting on a stalled output when `rst_n` fell is still there afterwards. Until Part 5 the only reset came at time 0, with both buffers empty, so there was never a beat for B15 to keep. After your last run, `build/grade/B15/run.log` holds seed 3, and in our seed 3 run the first failure came in the E8 step of the reset sequence:

```
[146155000]      %Error:      router_sva.sv:81:      Assertion      failed      in
router_tb_top.dut.u_sva.u_out0.a_O4_rst: O4: out_valid is 1 during reset
```

Then `a_O4_rst` failed once more and `a_O4_after` twice, and three cycles after the first failure the scoreboard reported the beat that came out:

```
UVM_ERROR      labs/lab-07-capstone/tb/router_env_pkg.sv(274)      @      146185000:
uvm_test_top.env.sb [OUT] output 0: got 00 in cycle 14619, and no beat was expected
```

A8. Part 7: Sign-off

In the starter the first check was the register model's, in cycle 12, when `uvm_reg_hw_reset_seq` read `CTRL` in the Part 2 step of `router_lab06_vseq`: `[RegModel] Register "regs.CTRL" value read from DUT (0x0000000000000001) does not match mirrored value (0x0000000000000000)`. Now the first is your assertion `a_I7`, in cycle 2, during the reset at time 0: `[15000] %Error: ... a_I7: I7: in_ready is 1 during reset`. With `CTRL.EN` at 1 the router is ready even while `rst_n` is 0. The register model catches B14 ten cycles later and says which register is wrong; `a_I7` catches it at once and names the rule the router breaks.

A9. Finishing

Either can be defended. `a_O4_rst` fired in the reset cycle itself and names the rule it checks. It needs no model, so it fires whether or not the sink ever takes the stale beat. In our seed 3 log the scoreboard fired three cycles later, and only because the sink took the beat, but its message says what the bug does to whoever uses the router, which is to deliver a beat from before the reset as if it were new.

A10. Finishing

That the router worked. At the time the plan had no assertion for I7, O4, the first half of A4 or RS1, nothing that made RS2, RS3 or DR2 happen, and a list of rows with no checker at all; the coverage had four empty bins that a test could hit; and, as Part 7 showed, the graded test passed on one of the fifteen planted bugs.

Appendices

A. The router in one page

This page answers: *what does the router do, in the detail a test needs, without reading the whole specification first?*

The packet router is the specification of the router, and in Labs 02 to 07 you write every testbench from it. It describes what the router does at its pins cycle by cycle, so it's long. This page keeps only the parts that your tests depend on. It goes through the blocks, the ports, packets and routing, the timing, the register map, reset and the numbered rules, and it links each one to the section of the specification that it comes from. When you need a cycle-by-cycle answer, follow the link. The specification's timing examples, E1 to E9, are tables that `labs/common/smoke/tb_examples.sv` printed while it ran the RTL, so every value in them comes from a simulation of the router.

The blocks

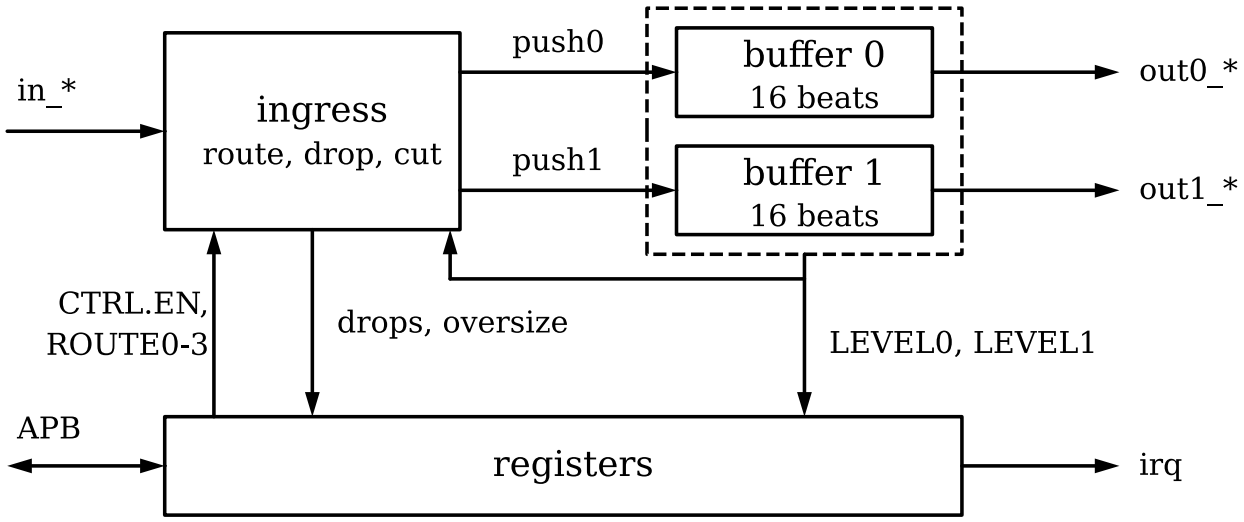


Figure A.1: The router's four blocks. Arrows show which way the data goes; each stream also has a `ready` going the other way.

The RTL is 730 lines in five files in `labs/common/rtl/`. One file is a package of sizes and register offsets, three hold one kind of block each (the two buffers are two instances of `router_fifo`), and the fifth is the top module, `router`, which connects the blocks. You don't need to read them, and you should write your tests from the specification rather than from them. From Lab 02 on, your tests run on a faulty build anyway, which is a copy of these files with the course's planted bugs added. The three kinds of block divide the work like this:

- The **ingress** takes packets from the input stream. When a packet's header arrives, the ingress looks up the header's entry in the routing table and the level of the output that the entry names. From those it decides, once for the whole packet, whether to write the packet into that output's buffer or to drop it and count it. It also cuts a packet that is longer than 8 beats down to 8.
- Each **buffer** holds up to 16 beats, 15 in a small RAM and one in the register that drives the output pins, so an output's `valid`, `data` and `last` (`out0_valid`, `out0_data` and `out0_last` for output 0) come straight from flip-flops. The buffer's **level** is the number of beats it holds, including the one on the pins.

- The **registers** are the APB slave. They hold `CTRL.EN` and the four `ROUTE` entries, which the ingress reads, and `INT_EN`, which picks the events that raise `irq`. They record drops in `STATUS` and `DROP_CNT`, and oversize packets in `STATUS` only. They also read back both levels in `LEVEL`, and drive `irq`.

A beat is 8 bits, there are 4 routes and 2 outputs, a packet is 1 to 8 beats long, each buffer holds 16 beats, and the drop counters stop at 255 (Sizes). We kept the sizes this small so that every corner of the design is a short simulation away.

Ports

Everything happens on the rising edge of `clk`, the APB port included, and the router samples its inputs at that edge (Ports, Cycles and handshakes).

Group	Inputs	Outputs	Notes
Clock and reset	`clk`, `rst_n`		`rst_n` is active low and asynchronous
APB slave	`psel`, `penable`, `pwrite`, `paddr[7:0]`, `pwdata[31:0]`	`prdata[31:0]`, `pready`, `pslverr`	APB3 with one slave select and no `PSTRB`, so a write writes the whole register
Interrupt		`irq`	active high, a level
Input stream	`in_valid`, `in_data[7:0]`, `in_last`	`in_ready`	the testbench is the source
Output 0	`out0_ready`	`out0_valid`, `out0_data[7:0]`, `out0_last`	the testbench is the sink
Output 1	`out1_ready`	`out1_valid`, `out1_data[7:0]`, `out1_last`	the same as output 0

A **handshake** happens in a cycle where `valid` and `ready` are both 1, and the beat moves at the rising edge that ends that cycle. The course's testbenches change the router's inputs 1 ns after a rising edge. The directed testbenches, such as the one that prints the timing examples, look at the router's outputs at the falling edge in the middle of the cycle. The monitors in the lab testbenches read them through clocking blocks with `#1step`, which gives the value that a signal had just before the rising edge. Either way, nothing that the testbench drives changes at the moment anything is sampled.

The router doesn't reset or clear `out0_data` and `out0_last`, and Verilator has only the values 0 and 1, so while `out0_valid` is 0 those two signals never show X. They still hold some value, usually the last beat they carried. Never compare `out0_data` or `out0_last` in a cycle where `out0_valid` is 0, because what you'd be comparing is left over from an earlier beat.

Packets and routing

A packet is 1 to 8 beats on the input stream (a longer one is oversize, which is covered at the end of this section). Its first beat is the **header**, and the beat with `in_last` = 1 ends it. The header's bits 7:6 are `DST`, which picks one of the four routing table entries, `ROUTE0` to `ROUTE3`. Bits 5:0 are a tag that the router passes through untouched, and the course's testbenches put each packet's sequence number there. A delivered packet comes out exactly as it went in, unless it was oversize and the router cut it (Packets).

The router decides what happens to a packet once, in the cycle that its header is accepted. It reads `ROUTE[DST]` and the level of the output that the entry's `PORT` names, both as they are in that cycle, and does what this table says (Routing):

`ROUTE[DST].EN`	Level of output `PORT`	The packet is	Counted in
0	anything	dropped	`NOROUTE_CNT`, `STATUS.NOROUTE`

ROUTE[DST].EN	Level of output PORT	The packet is	Counted in
1	9 to 16	dropped	FULL_CNT, STATUS.FULL
1	0 to 8	written, beat by beat, into output PORT's buffer	

The decision holds for the whole packet. If software writes ROUTE[DST] or CTRL, or either level changes, while a packet is coming in, only later packets are affected. A header accepted in the cycle after a write completes uses the new value, and a header accepted in the same cycle that the write completes uses the old one. A dropped packet is still taken from the input, with in_ready at 1 for every beat, so a full output never holds up packets for the other one.

Keep three consequences of this in mind when you write a scoreboard:

- A packet is admitted at a level of 8 or less and is at most 8 beats long, so an admitted packet always fits in the 16-beat buffer. With output 0 stalled and empty, two 8-beat packets are admitted (at levels 0 and 8) and a third is dropped. Nine 1-beat packets are admitted (at levels 0 to 8), and the tenth is dropped.
- The level that the decision uses is the number of beats written into the output in earlier cycles, minus the number taken from it by output handshakes in earlier cycles, both counted from the last reset. Neither the header itself nor an output handshake in the header's own cycle counts. The beats written into an output are the input beats of the packets that you predict are routed there, and the beats taken are the handshakes on that output's pins, so a testbench can track both levels exactly and predict every FULL drop.
- A packet for a disabled route is a NOROUTE drop whatever the level of the output that its PORT names, and it's never counted as FULL as well.

A drop adds one to its counter, unless the counter is already 255, and sets its STATUS bit. Both happen at the edge that ends the header's cycle, so both show from the next cycle (Dropped packets).

A packet is **oversize** if its 8th beat doesn't have in_last = 1. The router sets STATUS.OVERSIZE at the edge that ends the cycle in which it accepts the 8th beat, so the bit reads 1 from the next cycle. If the packet is being forwarded, the 8th beat goes out with out_last = 1. The router then takes the rest of the packet and throws it away, up to and including the beat with in_last = 1. DROP_CNT has no counter for oversize packets. An oversize packet that is also dropped is counted once, as an ordinary NOROUTE or FULL drop, and sets STATUS.OVERSIZE as well (Oversize packets).

Timing

What	When	In the specification
in_ready between packets	equals CTRL.EN, which resets to 0	Enabling the router
in_ready inside a packet	1 from its first handshake to its last, whatever CTRL.EN, the routes and the outputs do	rule I6
A beat into an empty output	on the pins 2 cycles after its input handshake (E2)	The outputs
Beat k of an output	on the pins from cycle $\max(t_k + 2, h_{k-1} + 1)$ until the cycle h_k it's taken, where t_k is its input handshake	rule O8
A stalled output	holds its beat, unchanged, until the sink takes it; the other output and the input carry on	rules O1, O2, O7

What	When	In the specification
An APB write	2 cycles (setup, one access cycle); the register changes at the edge that ends the access cycle	APB
An APB read	3 cycles (setup, two access cycles); it returns the value the register had in the first access cycle	rule A8
A drop, or an oversize packet	its `STATUS` bit (and for a drop, its counter) reads the new value from the cycle after the header, for a drop, or after the 8th beat, for an oversize packet	E4, E5, E6
`irq`	in the same cycle as the `STATUS` or `INT_EN` bit that changes it	INT_EN

Register map

Addresses are byte addresses, every register is 32 bits, and bits not listed read 0 and ignore writes (Register map).

Address	Register	Access	Reset	Fields	UVM access
`0x00`	`ID`	read-only	`0xD105_0001`	[31:0] design 105, revision 1	`RO`
`0x04`	`CTRL`	read-write	0	[0] `EN`	`RW`
`0x08`	`STATUS`	write 1 to clear	0	[0] `NOROUTE`, [1] `FULL`, [2] `OVERSIZE`	`W1C`, volatile
`0x0C`	`INT_EN`	read-write	0	[0] `NOROUTE`, [1] `FULL`, [2] `OVERSIZE`	`RW`
`0x10`	`DROP_CNT`	read-only; any write clears both	0	[7:0] `NOROUTE_CNT`, [15:8] `FULL_CNT`	`WC`, volatile
`0x14`	`LEVEL`	read-only	0	[4:0] `LEVEL0`, [12:8] `LEVEL1`	`RO`, volatile
`0x20`, `0x24`, `0x28`, `0x2C`	`ROUTE0` to `ROUTE3`	read-write	0	[0] `EN`, [1] `PORT`	`RW`

The other 246 of the 256 addresses are bad: `0x18`, `0x1C`, everything from `0x30` to `0xFF`, and every address that isn't a multiple of 4. A transfer to a bad address takes as many cycles as one to a good address and completes with `pslverr` = 1. A bad write changes nothing, and a bad read returns 0. Writing `ID`, `LEVEL` or the reserved bits of any register isn't an error. The transfer completes with `pslverr` = 0, and the router ignores the write.

Four details of the registers catch people out:

- `irq` $= \bigvee_i (\texttt{STATUS}[i] \wedge \texttt{INT_EN}[i])$, with no delay. `INT_EN` doesn't stop `STATUS` from recording events; it only decides which of them raise `irq`.
- If an event sets a `STATUS` bit in the same cycle that a write clears it, the event wins and the bit stays 1 (STATUS).
- A write to `DROP_CNT` in the cycle of a drop clears the counter first and then counts the drop, so it reads 1 afterwards (E9, DROP_CNT).

- `PORT` keeps its value while `EN` is 0, but it has no effect then, because a packet for a disabled route is a NOROUTE drop (ROUTE0 to ROUTE3).

Reset

`rst_n` is asynchronous, so when it falls the whole router resets at once, without waiting for a clock edge. The router runs normally again from the first rising edge after `rst_n` is back at 1. While `rst_n` is low, the router holds `in_ready`, both `out_valid`s, `irq`, `pslverr` and `prdata` at 0, and `pready` at 1 (with the bus idle). Every register goes back to its reset value, so every route is disabled and `CTRL.EN` is 0, and nothing gets through until software sets them up again. Both buffers are emptied and any packet in flight is lost, so nothing from before the reset ever comes out afterwards, not even a beat that was waiting on a stalled output (E8). Hold `rst_n` low through at least one rising edge at the start of every simulation, because the router's state is undefined until its first reset. (On Verilator it happens to start at all zeros, but don't rely on that.) The specification's Reset section has the details.

The rules

This table gives every rule that the specification numbers under Valid/ready rules and APB rules, one line each. The ones marked *testbench* are your stimulus's job, and the router assumes them without checking. The specification also says how to check the rules. Most are properties over one or two cycles, and each of those becomes a plain assertion. O5, O6 and O8 need state kept for each output, such as the order of its beats, so they belong in a scoreboard, or in assertions with tracking code beside them. I4, O3 and O7 say what a signal does not depend on, which no single trace can show, so you check them with a reference model, or with a test that changes an input in the middle of a cycle and confirms that the output doesn't move until the next rising edge. The rules about normal operation hold only while `rst_n` is 1. I7, the first halves of O4 and A4, and the reset values say what holds while it's 0.

Rule	Whose	What it says
I1	testbench	Once `in_valid` is 1, it stays 1 until the handshake or a reset.
I2	testbench	While `in_valid` is 1 and `in_ready` is 0, `in_data` and `in_last` don't change.
I3	testbench	After a reset, the first beat the source offers is the header of a new packet.
I4	router	`in_ready` doesn't depend on `in_valid`, `in_data` or `in_last` in the same cycle.
I5	router	Between packets, `in_ready` equals `CTRL.EN`.
I6	router	From a packet's first handshake to its last, `in_ready` is 1, unless a reset comes.
I7	router	`in_ready` is 0 while `rst_n` is 0.
O1	router	Once `out_valid` is 1, it stays 1 until the handshake or a reset.
O2	router	While `out_valid` is 1 and `out_ready` is 0, `out_data` and `out_last` don't change.
O3	router	`out_valid`, `out_data` and `out_last` don't depend on any input in the same cycle.

Rule	Whose	What it says
O4	router	`out_valid` is 0 during reset, and stays 0 after it until a beat accepted after the reset reaches the output.
O5	router	An output delivers the beats of its packets in the order that the router accepted them, unchanged except for a cut packet's `out_last`, and never interleaves two packets' beats.
O6	router	Every output packet is 1 to 8 beats and ends with the only beat that has `out_last` = 1, unless a reset cuts it short.
O7	router	Nothing about output 0 depends on `out1_ready`, or about output 1 on `out0_ready`.
O8	router	Beat k is on the output from cycle $\max(t_k + 2, h_{k-1} + 1)$ until it's taken, and `out_valid` is 0 otherwise.
A1	testbench	A transfer is one setup cycle followed at once by its access cycles.
A2	testbench	`paddr`, `pwrite` and a write's `pwdata` hold from the setup cycle to the end of the transfer.
A3	testbench	In the cycle after a transfer completes, `penable` is 0.
A4	testbench	`psel` is 0 during reset; a transfer cut by a reset has no effect.
A5	router	In a write's first access cycle, `pready` is 1.
A6	router	In a read's first access cycle `pready` is 0, and in its second it's 1; it's 0 at no other time.
A7	router	`pslverr` is 1 only in the cycle a transfer to a bad address completes.
A8	router	A completing read's `prdata` is the register's value in the first access cycle (0 for a bad address), and `prdata` changes only at the edge that ends a read's first access cycle and at reset.
A9	router	A good write takes effect at the end of its access cycle, according to the register's access type.

Check yourself

1. `ROUTE0` is `0x1` and `CTRL.EN` is 1, output 0 is stalled, and it holds seven 1-beat packets. An 8-beat packet for `DST` 0 arrives, then a 1-beat one. What happens to each?

   Answer: A1, in Answers at the end.

2. All four routes are disabled and output 1 is full. A packet for `DST` 3, whose `ROUTE3.PORT` is 1, comes in. Which counter goes up?

   Answer: A2, in Answers at the end.

3. Your test writes `0x3` to `ROUTE1`, and the write completes in the same cycle a header with `DST` 1 is accepted. Before the write, `ROUTE1` was 0. Where does the packet go?

   Answer: A3, in Answers at the end.

Answers

A1. Check yourself

The 8-beat packet's header sees a level of 7, which is 8 or less, so the packet is admitted, and once its eight beats are written the level is 15. The 1-beat packet's header then sees 15, so it's dropped as FULL. `FULL_CNT` goes up by one, and `STATUS.FULL` reads 1 from the next cycle.

A2. Check yourself

Only `NOROUTE_CNT`. A packet for a disabled route is a NOROUTE drop whatever the level of the output that its `PORT` names, so `FULL_CNT` doesn't change.

A3. Check yourself

It's dropped as NOROUTE, so it comes out of neither output. A header accepted in the same cycle that a write completes uses the register's old value, and the new value applies from the next cycle on.

B. The packet router

This page is the specification of the packet router, the design you verify in DD105. It describes everything the router does at its pins, cycle by cycle. You shouldn't need to read the RTL to write a testbench for it, and from Lab 02 on you'll be testing faulty builds of it, whose RTL you can't trust anyway. If this page and the router ever disagree, one of them has a bug, and you should tell us which.

The router takes packets from one input stream and sends each one to one of two output streams, using a routing table that software writes through an APB register port. Each output has its own buffer and its own `ready` signal, so one output can stall without stopping the other. A packet that can't be delivered (its route is disabled, or its output is too full) is dropped whole, and counted.

The RTL is in `labs/common/rtl/`. The top module is `router`.

Sizes

Quantity	Value
Data width of a beat	8 bits
Routes (routing table entries)	4, picked by a 2-bit field in the header
Outputs	2
Buffer per output	16 beats
Longest packet	8 beats, header included
Admission threshold	an output takes a new packet only if it holds 8 beats or fewer
Drop counters	two, 8 bits each, saturating at 255
APB data width	32 bits
APB address width	8 bits (byte addresses), 10 registers

The sizes are small on purpose, so that a short simulation can reach every corner case of the design:

- A beat is one byte, so you can read a packet straight off a waveform in hex, and the header fits in one beat.
- With four routes and two outputs you can cover the routing table completely, because it has only four entries and each one does one of three things: drop the packet, send it to output 0, or send it to output 1.
- Packets are 1 to 8 beats long, so packet length is a coverpoint with eight bins, and the shortest oversize packet is only one beat longer than the longest legal one.
- The buffer holds exactly two maximum-length packets. Because a packet is admitted only while its output holds 8 beats or fewer, a stalled output that keeps receiving packets starts dropping them once it holds more than 8 beats, and it takes between 9 beats of traffic (nine 1-beat packets) and 16 (two 8-beat packets) to get there.
- The counters saturate at 255. A stream of 1-beat packets to a disabled route drops one packet per cycle, so you reach saturation in 255 cycles.
- The APB address space has 256 byte addresses, of which 246 are bad. A read takes 3 cycles, so a test can read all 256 in 768 cycles.

The directed test we use to check the RTL against this page runs a reference model and 19 directed checks, and it takes 6,852 clock cycles in all.

Ports

Port	Direction	Width	What it is
`clk`	in	1	The clock for the whole design, the APB port included (APB calls it PCLK). Everything happens on its rising edge.
`rst_n`	in	1	Reset, active low and asynchronous (APB calls it PRESETn). See Reset.
`psel`	in	1	APB PSEL
`penable`	in	1	APB PENABLE
`pwrite`	in	1	APB PWRITE: 1 for a write, 0 for a read
`paddr`	in	8	APB PADDR, a byte address
`pwdata`	in	32	APB PWDATA
`prdata`	out	32	APB PRDATA
`pready`	out	1	APB PREADY
`pslverr`	out	1	APB PSLVERR
`irq`	out	1	Interrupt, active high. A level, not a pulse.
`in_valid`	in	1	Input stream: the source has a beat
`in_ready`	out	1	Input stream: the router will take it
`in_data`	in	8	Input stream: the beat
`in_last`	in	1	Input stream: this beat is the last of its packet
`out0_valid`	out	1	Output 0: the router has a beat
`out0_ready`	in	1	Output 0: the sink will take it
`out0_data`	out	8	Output 0: the beat
`out0_last`	out	1	Output 0: this beat is the last of its packet
`out1_valid`, `out1_ready`, `out1_data`, `out1_last`			Output 1, the same as output 0

The APB port has the APB3 signal set and a single slave select. There's no PSTRB (that came with APB4), so every write writes the whole register.

The course simulates the router with Verilator, which has only two states, 0 and 1. A data signal whose `valid` is low still holds some value, usually the last one it carried, so a testbench must never compare `out0_data` or `out0_last` in a cycle where `out0_valid` is low.

Cycles and handshakes

The tables and rules on this page number clock cycles. **Cycle n** is the clock period that ends at rising edge n, and the value of a signal "in cycle n" is the value it holds just before that edge. The router samples its inputs at rising edges. Most of its outputs depend only on flip-flops inside the router, so they change just after a rising edge or when `rst_n` falls (see Reset). The exceptions are `pready` and `pslverr`, which also depend on `psel`, `penable`, `pwrite` and `paddr` in the same cycle, so sample them at the end of the cycle, as you would any APB slave's.

A **handshake** on a stream happens in a cycle where `valid` and `ready` are both 1. The beat moves at the rising edge at the end of that cycle. Our testbenches change inputs 1 ns after a rising edge, and they look at outputs either at the falling edge in the middle of the cycle or through a clocking block with `#1step`, which gives the value just before the rising edge that ends the cycle. Either way, nothing changes at the moment anything is sampled.

Packets

A packet on the input stream is 1 to 8 beats long, unless it's oversize. Its first beat is the **header**, and the beat with `in_last` = 1 ends it. A 1-beat packet is a header with `in_last` = 1.

The header carries the route:

Header bits	Name	Meaning
7:6	`DST`	Which routing table entry to use: 0 to 3
5:0	(tag)	Ignored by the router and passed through unchanged. Testbenches usually put a sequence number here.

Apart from `DST`, the router doesn't look at what any beat contains; it counts the beats and watches `in_last`. The header and every other beat go to the output exactly as they arrived, header first, so a packet that's delivered comes out identical to the packet that went in. The one exception is an oversize packet, which is cut (see Oversize packets).

Routing

Each of the four routing table registers, `ROUTE0` to `ROUTE3`, has two fields: `EN`, which turns the route on, and `PORT`, which picks output 0 or 1. All four reset to 0, so every route starts disabled.

The router decides what to do with a whole packet once, in the cycle its header is accepted. It reads `ROUTE[DST]` and the level of the output that the entry's `PORT` names, both as they are in that cycle, and then:

`ROUTE[DST].EN`	Level of output `PORT`	The packet is
0	anything	dropped, and counted as NOROUTE
1	9 to 16	dropped, and counted as FULL
1	0 to 8	written, beat by beat, into the buffer of output `PORT`

The **level** of an output in cycle t is the number of beats written into it in input handshake cycles before t, minus the number taken from it in output handshake cycles before t, counting both from the last reset. It counts the beat on the output's pins as well as the beats behind it: in example E2 the level in cycle 4 is 2, because `05` is on `out0_data` and `11` is behind it. The level ranges from 0 to 16, and it's the value the `LEVEL` register reads. A testbench can track both levels exactly, and so predict every FULL drop. The beats written into an output are the input beats of the packets that the testbench predicts are routed there (only the first 8 of an oversize packet), and the beats taken from it are the handshakes on that output's pins.

The decision uses the level in the header's own cycle, so neither the header itself nor a handshake on the output in that same cycle counts.

Because a packet is admitted only at a level of 8 or less, and no packet writes more than 8 beats, an admitted packet always fits. For example, with output 0 stalled, an 8-beat packet raises its level from 0 to 8, a second 8-beat packet is still admitted (at exactly 8) and fills it to 16, and the next packet for output 0 is dropped. Nine 1-beat packets are admitted at levels 0 to 8, and the tenth is dropped.

A packet for a disabled route counts as NOROUTE whatever the level of the output its `PORT` names. It's never counted as FULL as well.

The decision holds for the whole packet. Writing `ROUTE[DST]` or `CTRL`, or a change in either output's level, while a packet is coming in affects only later packets. A register write affects a header accepted in the cycle after the write completes, and later; a header accepted in the same cycle the write completes uses the old value.

A dropped packet is still taken from the input: `in_ready` stays 1 for all of its beats, and the router throws them away. The input never waits for a full output, so a stalled output doesn't stop packets for the other one.

Dropped packets

When a packet is dropped, at the edge that ends its header's cycle the router

- adds one to `DROP_CNT.NOROUTE_CNT` or `DROP_CNT.FULL_CNT`, unless that counter is already 255, and
- sets `STATUS.NOROUTE` or `STATUS.FULL`.

Both changes are visible from the next cycle on. Example E4 below shows a NOROUTE drop and E5 a FULL drop.

Oversize packets

A packet is **oversize** if its 8th beat doesn't have `in_last` = 1. In the cycle the 8th beat is accepted, the router sets `STATUS.OVERSIZE` (visible from the next cycle), and

- if the packet is being forwarded, it writes the 8th beat with `out_last` = 1, so the output sees an 8-beat packet;
- then it takes and discards every remaining beat of the input packet, up to and including the one with `in_last` = 1, with `in_ready` held at 1.

An oversize packet sets `STATUS.OVERSIZE` once, however long it is. A dropped packet that's also oversize is counted as a drop (once) and sets `STATUS.OVERSIZE` too. Oversize packets aren't counted in `DROP_CNT`. The next packet after an oversize one is handled normally. Example E6 shows a 10-beat packet cut to 8.

Enabling the router

`CTRL.EN` starts and stops the input, and only at packet boundaries. Between packets, `in_ready` equals `CTRL.EN`. In the middle of a packet `in_ready` is 1, whatever `CTRL.EN` is, so clearing `CTRL.EN` while a packet is coming in lets that packet finish and stops the input after it. Example E7 shows this.

`CTRL.EN` doesn't affect the outputs. Beats already in the buffers keep flowing while it's 0. `CTRL.EN` resets to 0, so the router takes nothing until software sets it.

The outputs

Each output has a buffer of 16 beats and drives its stream from a register, so `out0_valid`, `out0_data` and `out0_last` come straight from flip-flops.

One rule gives the cycle in which each beat reaches the pins. Number the beats written into one output in order, and let t_k be the cycle of beat k's input handshake and h_k the cycle in which the sink takes it. Beat k is on the output (`out_valid` = 1, with beat k on `out_data` and `out_last`) from cycle

$$a_k = \max(t_k + 2, h_{k-1} + 1) \tag{2.1}$$

until cycle h_k, and `out_valid` is 0 in every cycle not covered that way. For the first beat after reset, $a_k = t_k + 2$. So:

- A beat into an empty output appears two cycles after its input handshake. In example E2 the header is accepted in cycle 2 and is on the output from cycle 4.
- Once beats are waiting, a sink that is always ready takes one per cycle with no gaps, as in E2′s cycles 4 to 6.
- A beat that has waited stays until it's taken, and the next one appears in the cycle after (E3).

An output stalls only on its own `ready`. While `out0_ready` is low, output 1 carries on at full speed, and the input keeps accepting packets for both outputs (and dropping the ones for output 0 once it's too full).

Valid/ready rules

The rules marked *source* are the testbench's job on the input stream; the router assumes them and doesn't check them. The rest are the router's promises.

A reset ends whatever was in progress: the router lowers `in_ready` and both `out_valid`s at once, with no handshake, and the source drops the beat it was offering (rule I3). So the rules about normal operation hold only while `rst_n` is 1, and an assertion of one of them should be disabled while `rst_n` is 0 (`disable iff (!rst_n)`). The rules about reset itself are the other way round: I7, the first half of O4, the first half of A4 and the reset values say what holds while `rst_n` is 0, so check them without that `disable iff`, sampled at each rising edge.

Most rules are properties over one or two cycles, and each of those becomes a plain assertion. The rest don't fit in a single plain assertion:

- O5, O6 and O8 need state kept for each output: the order of the beats, how many beats the current packet has had, and the t_k and h_k of each beat. They belong in a scoreboard, or in assertions with tracking code beside them.
- I4, O3 and O7 say what a signal does *not* depend on, which no single trace can show. A reference model that predicts `in_ready` and `out_valid` from this page checks them, and so does a test that changes an input in the middle of a cycle and confirms the output doesn't move until the next rising edge.
- I3 constrains your stimulus, so it belongs in the driver or the sequence, not in an assertion.

Input stream

- **I1** (source). Once `in_valid` is 1, it stays 1 until the cycle of the handshake, or until a reset.
- **I2** (source). While `in_valid` is 1 and `in_ready` is 0, `in_data` and `in_last` don't change.
- **I3** (source). A beat the source was offering when `rst_n` fell is abandoned: from the first cycle after the reset, the first beat the source offers is the header of a new packet. (A source that went on offering the old beat would have it accepted as a header once `CTRL.EN` is set.)
- **I4**. `in_ready` doesn't depend on `in_valid`, `in_data` or `in_last` in the same cycle. (It comes from flip-flops.) A source may therefore wait for `in_ready` before raising `in_valid`, though it doesn't have to.
- **I5**. Between packets (after reset, and after a handshake with `in_last` = 1), `in_ready` equals `CTRL.EN`.
- **I6**. In every cycle from a packet's first handshake to its last, `in_ready` is 1, whatever `CTRL.EN`, the routing table and the outputs do, unless `rst_n` falls.
- **I7**. `in_ready` is 0 while `rst_n` is 0.

The source may leave `in_valid` at 0 for any number of cycles, between packets or between the beats of one packet, and packets may also follow each other with no idle cycle at all. The router looks at `in_data` and `in_last` only in cycles with a handshake, so it doesn't notice a source that breaks I1 or I2. It handles whichever beats it accepted.

Output streams

Each rule holds for output 0 and output 1 separately.

- **O1**. Once `out_valid` is 1, it stays 1 until the cycle of the handshake, or until a reset.
- **O2**. While `out_valid` is 1 and `out_ready` is 0, `out_data` and `out_last` don't change.
- **O3**. `out_valid`, `out_data` and `out_last` don't depend on `out_ready`, or on any other input, in the same cycle. (They come from flip-flops, and the only change between edges is a falling `rst_n` clearing `out_valid`.)
- **O4**. `out_valid` is 0 while `rst_n` is 0, and stays 0 after reset until a beat accepted after the reset reaches the output.

- **O5**. An output delivers the beats of the packets routed to it in the order the router accepted them, unchanged, except that a cut packet ends at its 8th beat, which has `out_last` = 1. One packet's beats are never interleaved with another's.
- **O6**. Every packet on an output is 1 to 8 beats long, and its last beat is the only one with `out_last` = 1, unless a reset cuts it short (see Reset).
- **O7**. Nothing about output 0 depends on `out1_ready`, and nothing about output 1 depends on `out0_ready`.
- **O8**. Beat k of an output is on the output from cycle $\max(t_k + 2, h_{k-1} + 1)$ until the cycle it's taken, and `out_valid` is 0 in every other cycle, as described under The outputs.

The sink has no obligations: `out_ready` may be high before `out_valid`, may change in any cycle, and may stay low for as long as it likes.

APB

A transfer is a **setup cycle** (`psel` = 1, `penable` = 0) followed by one or more **access cycles** (`psel` = 1, `penable` = 1). It completes in the access cycle where `pready` is 1. Writes take one access cycle and reads take two, so a write is 2 cycles long and a read is 3. Example E1 shows a write, a read and a read of a bad address.

A read returns the register's value as it is in the first access cycle. In the first access cycle of a read the router copies the register into an output register, and drives `prdata` from it in the second. For registers that change on their own, like `LEVEL`, this means that a beat that enters or leaves a buffer at the edge that ends the first access cycle doesn't show in the value you read.

A write takes effect at the rising edge that ends its access cycle. From the next cycle on, the router behaves according to the new value.

A **bad address** is one that isn't in the register map, which includes every address that isn't a multiple of 4. A transfer to a bad address completes normally, with the same timing as any other, but with `pslverr` = 1. A bad write changes nothing, and a bad read returns 0. Writing `ID` or `LEVEL`, or the reserved bits of any register, is not an error: the transfer completes with `pslverr` = 0 and the write is ignored. (A write to `DROP_CNT` isn't ignored: it clears the counters.)

APB rules

The rules marked *master* are the testbench's job; the router doesn't check them.

- **A1** (master). A transfer starts with one setup cycle, and its access cycles follow immediately.
- **A2** (master). `paddr` and `pwrite`, and `pwdata` for a write, stay the same from the setup cycle to the end of the transfer.
- **A3** (master). In the cycle after a transfer completes, `penable` is 0: the bus is idle (`psel` = 0) or in the setup cycle of the next transfer. Transfers may follow each other with no idle cycle between them.
- **A4** (master). `psel` is 0 while `rst_n` is 0. A transfer that is in progress when `rst_n` falls is abandoned and has no effect. The master may start it again after the reset, from its setup cycle, or drop it; to the router a restarted transfer is just the next transfer.
- **A5**. In the first access cycle of a write, `pready` is 1.
- **A6**. In the first access cycle of a read, `pready` is 0; in the second, it's 1. This is the only time `pready` is 0.
- **A7**. `pslverr` is 1 only in the cycle in which a transfer to a bad address completes, and 0 in every other cycle.
- **A8**. In the cycle a read completes, `prdata` is the value the register had in the first access cycle, or 0 for a bad address. `prdata` changes only at the edge that ends the first access cycle of a read, when it takes the value that read returns, and when `rst_n` falls, which sets it to 0. So between reads it holds the value of the last read, and writes don't change it.
- **A9**. A write to a good address takes effect at the edge that ends its access cycle, according to the register's access type: read-write fields take the value written, `STATUS` bits written as 1 clear, any write clears `DROP_CNT`, and `ID` and `LEVEL` ignore it. A write to a bad address changes nothing.

Register map

Addresses are byte addresses. Every register is 32 bits, and bits not listed read 0 and ignore writes.

Address	Register	Access	Reset value	Fields
0x00	ID	RO	0xD105_0001	[31:0] the design's ID: design 105, revision 1
0x04	CTRL	RW	0x0000_0000	[0] EN
0x08	STATUS	W1C	0x0000_0000	[0] NOROUTE, [1] FULL, [2] OVERSIZE
0x0C	INT_EN	RW	0x0000_0000	[0] NOROUTE, [1] FULL, [2] OVERSIZE
0x10	DROP_CNT	WC: read-only, any write clears	0x0000_0000	[7:0] NOROUTE_CNT, [15:8] FULL_CNT
0x14	LEVEL	RO	0x0000_0000	[4:0] LEVEL0, [12:8] LEVEL1
0x20	ROUTE0	RW	0x0000_0000	[0] EN, [1] PORT
0x24	ROUTE1	RW	0x0000_0000	[0] EN, [1] PORT
0x28	ROUTE2	RW	0x0000_0000	[0] EN, [1] PORT
0x2C	ROUTE3	RW	0x0000_0000	[0] EN, [1] PORT

Every other address is bad and gets pslverr: 0x18, 0x1C, everything from 0x30 to 0xFF, and every address that isn't a multiple of 4. That's 246 of the 256.

If you model the registers in UVM's register layer, these are the field access policies:

Fields	UVM access	Volatile
ID	RO	no
CTRL.EN, INT_EN.*, ROUTEn.EN, ROUTEn.PORT	RW	no
STATUS.*	W1C	yes: hardware sets them
DROP_CNT.NOROUTE_CNT, DROP_CNT.FULL_CNT	WC	yes: hardware counts
LEVEL.LEVEL0, LEVEL.LEVEL1	RO	yes: they follow the buffers

ID

Reads 0xD105_0001. Reading it is a quick way to check that the APB port works at all.

CTRL

EN (bit 0) starts and stops the input at packet boundaries, as described in Enabling the router.

STATUS

Each bit records that an event has happened since software last cleared it:

Bit	Name	Set when
0	NOROUTE	a packet is dropped because its route is disabled

Bit	Name	Set when
1	`FULL`	a packet is dropped because its output holds more than 8 beats
2	`OVERSIZE`	a packet's 8th beat isn't its last

A bit is set at the edge that ends the cycle of its event and stays set until software writes a 1 to it. Writing 0 to a bit leaves it alone, so writing `0x2` clears `FULL` and nothing else. If an event sets a bit in the same cycle that a write clears it, the event wins and the bit stays 1. `INT_EN` doesn't affect `STATUS`: the bits record events whether or not their interrupts are enabled.

INT_EN

One enable per `STATUS` bit, in the same positions. The interrupt is

$$\texttt{irq} = \bigvee_{i=0}^{2} \left(\texttt{STATUS}[i] \wedge \texttt{INT_EN}[i] \right) \tag{2.2}$$

and it's a plain combination of the two registers, with no extra delay: `irq` changes in the same cycle as the register bit that changes it. In example E4, a NOROUTE drop in cycle 2 makes `STATUS.NOROUTE` and `irq` 1 from cycle 3, and a write that clears the bit, completing in cycle 7, makes both 0 from cycle 8. `irq` stays 1 for as long as an enabled bit is set; only clearing the bit, clearing its enable, or a reset lowers it.

DROP_CNT

Two counters: `NOROUTE_CNT` counts packets dropped for a disabled route, and `FULL_CNT` counts packets dropped for a full output. Each counts a dropped packet once, at its header. A counter that reaches 255 stays there rather than wrapping.

A write of any value to `DROP_CNT` clears both counters; the value written is ignored. If a drop happens in the same cycle as the clearing write, it's counted after the clear, so that counter reads 1 afterwards (example E9).

LEVEL

The number of beats each output's buffer holds, 0 to 16, as described under Routing. Bits [7:5] and [31:13] read 0.

ROUTE0 to ROUTE3

Entry n of the routing table, used by packets whose `DST` is n. `EN` (bit 0) enables the route and `PORT` (bit 1) picks the output: 0 for output 0, 1 for output 1. `PORT` keeps its value while `EN` is 0, but it has no effect then: a packet for a disabled route is a NOROUTE drop.

Reset

`rst_n` is active low and asynchronous. When it falls, the whole router resets at once, without waiting for a clock edge: in example E8, `rst_n` goes low just after the edge that starts cycle 3, and `in_ready`, `out0_valid` and `LEVEL0` are already 0 in cycle 3. The router does nothing at an edge where `rst_n` is 0, and runs normally from the first rising edge after `rst_n` is 1.

Change `rst_n` just after a rising edge, both ways, as our testbenches do. Then every cycle is either wholly in reset or wholly out of it, and a monitor that samples at the falling edge never records a handshake that the reset cancels. (If `rst_n` falls in the middle of a cycle, nothing moves at the edge that ends it, whatever `valid` and `ready` were earlier in the cycle.)

Hold `rst_n` at 0 through at least one rising edge at the start of every simulation. Before its first reset the router's state is undefined. On Verilator it happens to start at all zeros, which is not something to rely on.

While `rst_n` is 0, and right after it:

Signal or register	Value
`in_ready`	0, and it stays 0 after reset until software sets `CTRL.EN`
`out0_valid`, `out1_valid`	0
`irq`	0
`pready`	1, with the bus idle as rule A4 requires (it's 0 only in the first access cycle of a read)
`pslverr`	0
`prdata`	0
every register	its reset value from the register map; the routing table is all disabled
both buffers	empty: `LEVEL` reads 0

`out0_data`, `out0_last`, `out1_data` and `out1_last` aren't reset; they mean nothing while their `valid` is 0.

A reset with packets in flight loses them. Whatever was in the buffers is gone, including a beat that was waiting on a stalled output, and nothing from before the reset ever comes out afterwards.

- If the sink had already taken some beats of a packet, the rest of it never comes. A scoreboard should throw away everything it expected from before the reset, including the rest of a partly delivered packet, and a monitor that assembles beats into packets should drop a half-assembled one.
- If a packet was half accepted, the beats already taken are discarded with everything else, and the next beat the router accepts is treated as a header. A source that was part way through a packet must start again with a header (rule I3).
- The configuration is gone too, so software must set up the routing table and `CTRL.EN` again before any packet gets through.

Timing examples

These tables come from runs of the RTL (`labs/common/smoke/tb_examples.sv` prints them). Each column is one cycle, numbered from 1 at the start of the example, and each value is what the signal holds in that cycle. A dash means the value doesn't matter in that cycle, and the table shows a dash in place of whatever is left on the wires. The dashes are on

- `in_data` and `in_last` while `in_valid` is 0, and `out0_data` and `out0_last` while `out0_valid` is 0;
- `paddr` and `pwrite` while `psel` is 0, and `pwdata` outside a write.

Rows marked (reg) are values inside the router: what a read whose first access cycle is that cycle would return. Hex values have no `0x`. Each example starts from a reset, and its setup is given above its table.

E1: a write, a read and a bad read

Setup: nothing after the reset. The write of `0x1` to `CTRL` completes in cycle 2, so `CTRL.EN` and `in_ready` are 1 from cycle 3. The read of `ID` completes in cycle 5 and the read of the bad address `0x18` in cycle 8, with `pslverr` = 1 and `prdata` = 0. `prdata` changes only at the end of each read's first access cycle (rule A8), so it's `d1050001` from cycle 5 to cycle 7.

cycle	1	2	3	4	5	6	7	8	9
`psel`	1	1	1	1	1	1	1	1	0
`penable`	0	1	0	1	1	0	1	1	0
`pwrite`	1	1	0	0	0	0	0	0	-
`paddr`	04	04	00	00	00	18	18	18	-
`pwdata`	00000001	00000001	-	-	-	-	-	-	-
`pready`	1	1	1	0	1	1	0	1	1

cycle	1	2	3	4	5	6	7	8	9
`prdata`	00000000	00000000	00000000	00000000	d1050001	d1050001	d1050001	00000000	00000000
`pslverr`	0	0	0	0	0	0	0	1	0
`CTRL.EN` (reg)	0	0	1	1	1	1	1	1	1
`in_ready`	0	0	1	1	1	1	1	1	1

E2: a three-beat packet through an empty output

Setup: `ROUTE0` is `0x1` (enabled, output 0) and `CTRL.EN` is 1. The header `05` is accepted in cycle 2 and appears on output 0 in cycle 4.

cycle	1	2	3	4	5	6	7
`in_valid`	0	1	1	1	0	0	0
`in_ready`	1	1	1	1	1	1	1
`in_data`	-	05	11	22	-	-	-
`in_last`	-	0	0	1	-	-	-
`LEVEL0` (reg)	0	0	1	2	2	1	0
`out0_valid`	0	0	0	1	1	1	0
`out0_ready`	1	1	1	1	1	1	1
`out0_data`	-	-	-	05	11	22	-
`out0_last`	-	-	-	0	0	1	-

E3: a stalling sink

Setup as in E2, and the same packet, with `out0_ready` low in cycles 5 to 7 and 9. Beat `11` is on output 0 from cycle 5 and moves in cycle 8; beat `22` is on it from cycle 9 and moves in cycle 10. Neither changes while it waits (rules O1 and O2).

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
`in_valid`	0	1	1	1	0	0	0	0	0	0	0	0	0
`in_ready`	1	1	1	1	1	1	1	1	1	1	1	1	1
`in_data`	05	11	22	-	-	-	-	-	-	-	-	-	-
`in_last`	0	0	1	-	-	-	-	-	-	-	-	-	-
`LEVEL0` (reg)	0	1	2	2	2	2	2	1	1	0	0	0	0
`out0_valid`	0	0	1	1	1	1	1	1	1	0	0	0	0
`out0_ready`	1	1	1	0	0	0	1	0	1	1	1	1	1
`out0_data`	-	-	05	11	11	11	11	22	22	-	-	-	-
`out0_last`	-	-	0	0	0	0	0	1	1	-	-	-	-

E4: a NOROUTE drop and a write that clears it

Setup: `INT_EN` is `0x1`, `CTRL.EN` is 1, and every route is disabled. The header `47` (`DST` = 1) is accepted in cycle 2, so `STATUS.NOROUTE`, `NOROUTE_CNT` and `irq` change from cycle 3. The packet comes out of neither output. The write of `0x1` to `STATUS` completes in cycle 7.

cycle	1	2	3	4	5	6	7	8
`in_valid`	0	1	1	0	0	0	0	0
`in_ready`	1	1	1	1	1	1	1	1
`in_data`	-	47	99	-	-	-	-	-
`in_last`	-	0	1	-	-	-	-	-
`out0_valid`	0	0	0	0	0	0	0	0
`out1_valid`	0	0	0	0	0	0	0	0
`STATUS` (reg)	0	0	1	1	1	1	1	0
`NOROUTE_CNT` (reg)	0	0	1	1	1	1	1	1
`irq`	0	0	1	1	1	1	1	0
`psel`	0	0	0	0	0	1	1	0
`penable`	0	0	0	0	0	0	1	0
`pwrite`	-	-	-	-	-	1	1	-
`paddr`	-	-	-	-	-	08	08	-
`pwdata`	-	-	-	-	-	00000001	00000001	-
`pready`	1	1	1	1	1	1	1	1

E5: a FULL drop

Setup: `ROUTE0` is `0x1` and `CTRL.EN` is 1. Output 0 is stalled and holds nine 1-beat packets, `01` to `09`, so its level is 9. The packet `3a a0` for `DST` 0 is dropped in cycle 2. The sink starts taking beats in cycle 5.

cycle	1	2	3	4	5	6	7
`in_valid`	0	1	1	0	0	0	0
`in_ready`	1	1	1	1	1	1	1
`in_data`	-	3a	a0	-	-	-	-
`in_last`	-	0	1	-	-	-	-
`LEVEL0` (reg)	9	9	9	9	9	8	7
`out0_valid`	1	1	1	1	1	1	1
`out0_ready`	0	0	0	0	1	1	1
`out0_data`	01	01	01	01	01	02	03
`STATUS` (reg)	0	0	2	2	2	2	2
`FULL_CNT` (reg)	0	0	1	1	1	1	1

E6: a ten-beat packet

Setup: `ROUTE0` is `0x1` and `CTRL.EN` is 1. The 8th beat, `07`, is accepted in cycle 9 without `in_last`, so it goes out with `out0_last` = 1 and `STATUS.OVERSIZE` is set from cycle 10. Beats `08` and `09` are taken and discarded.

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
in_valid	0	1	1	1	1	1	1	1	1	1	1	0	0
in_ready	1	1	1	1	1	1	1	1	1	1	1	1	1
in_data	-	00	01	02	03	04	05	06	07	08	09	-	-
in_last	-	0	0	0	0	0	0	0	0	0	1	-	-
out0_valid	0	0	0	1	1	1	1	1	1	1	1	0	0
out0_ready	1	1	1	1	1	1	1	1	1	1	1	1	1
out0_data	-	-	-	00	01	02	03	04	05	06	07	-	-
out0_last	-	-	-	0	0	0	0	0	0	0	1	-	-
STATUS (reg)	0	0	0	0	0	0	0	0	0	4	4	4	4

E7: CTRL.EN cleared in the middle of a packet

Setup: `ROUTE0` is `0x1` and `CTRL.EN` is 1. A write of 0 to `CTRL` completes in cycle 4, while the packet `01 11 12 13` is coming in. The packet finishes in cycle 5 with `in_ready` still 1. The next header, `02`, waits from cycle 6 with `in_ready` = 0.

cycle	1	2	3	4	5	6	7	8	9
psel	0	0	1	1	0	0	0	0	0
penable	0	0	0	1	0	0	0	0	0
pwrite	-	-	1	1	-	-	-	-	-
paddr	-	-	04	04	-	-	-	-	-
pwdata	-	-	00000000	00000000	-	-	-	-	-
pready	1	1	1	1	1	1	1	1	1
CTRL.EN (reg)	1	1	1	1	0	0	0	0	0
in_valid	0	1	1	1	1	1	1	1	1
in_ready	1	1	1	1	1	0	0	0	0
in_data	-	01	11	12	13	02	02	02	02
in_last	-	0	0	0	1	0	0	0	0
out0_valid	0	0	0	1	1	1	1	0	0
out0_ready	1	1	1	1	1	1	1	1	1
out0_data	-	-	-	01	11	12	13	-	-

E8: reset with a beat waiting on output 0

Setup: `ROUTE0` is `0x1` and `CTRL.EN` is 1. Output 0 is stalled with the packet `01 aa` in its buffer, and the packet `02 b0 b1 ...` has had two beats accepted when `rst_n` goes low for cycles 3 and 4. Everything is gone at once, including the waiting beat `01`, and nothing comes out when the sink becomes ready in cycle 6. After reset `CTRL.EN` is 0, so `in_ready` stays 0.

cycle	1	2	3	4	5	6	7
rst_n	1	1	0	0	1	1	1

cycle	1	2	3	4	5	6	7
`in_valid`	1	1	1	1	0	0	0
`in_ready`	1	1	0	0	0	0	0
`in_data`	02	b0	b1	b1	-	-	-
`in_last`	0	0	0	0	-	-	-
`LEVEL0` (reg)	2	3	0	0	0	0	0
`out0_valid`	1	1	0	0	0	0	0
`out0_ready`	0	0	0	0	0	1	1
`out0_data`	01	01	-	-	-	-	-
`CTRL.EN` (reg)	1	1	0	0	0	0	0

E9: a write to DROP_CNT in the same cycle as a drop

Setup: `CTRL.EN` is 1, every route is disabled, and two drops have made `NOROUTE_CNT` 2. A write to `DROP_CNT` completes in cycle 2, the same cycle a 1-beat packet for a disabled route is accepted. The counter is cleared and then counts the drop, so it reads 1.

cycle	1	2	3	4
`psel`	1	1	0	0
`penable`	0	1	0	0
`pwrite`	1	1	-	-
`paddr`	10	10	-	-
`pwdata`	00000000	00000000	-	-
`pready`	1	1	1	1
`in_valid`	0	1	0	0
`in_ready`	1	1	1	1
`in_data`	-	00	-	-
`in_last`	-	1	-	-
`NOROUTE_CNT` (reg)	2	2	1	1
`STATUS` (reg)	1	1	1	1

Building the router

The RTL is five files in `labs/common/rtl/`. `router.f` lists them in compile order, with the package first, and its paths are relative to its own folder, so pass it to Verilator with `-F` (not `-f`):

```
verilator --lint-only -Wall -F labs/common/rtl/router.f --top-module router
```

That lint passes with no warnings. The files have no `` `timescale `` and the design has no delays, so it runs at any clock period; our testbenches pass `--timescale 1ns/1ps` and use a 10 ns clock.

File	What it holds
`router_pkg.sv`	Sizes, register offsets and the ingress state type
`router_fifo.sv`	One output buffer and the stream it drives

File	What it holds
`router_ingress.sv`	The input stream: route lookup, admission, drops and the length limit
`router_regs.sv`	The APB port and every register
`router.sv`	The top module, `router`, which connects the four

The labs also use a **faulty build** of the router, made by `labs/common/inject.py --all --out DIR`. It compiles from `DIR/router.f` in exactly the same way and has the same ports, and with no bug switched on it behaves exactly like the RTL above. With `+BUG=Bnn` on the simulator's command line, it switches on one of the course's planted bugs at time 0, so you can change `+BUG` without rebuilding. An unknown value stops the simulation with a fatal error. The faulty build prints nothing about which bug is on, so a log can't give it away. Add `+BUG_ECHO` and it prints one line at time 0 that names the bug, or says that none is on.

C. The UVM register layer

This page describes the parts of UVM's register layer that Lab 06 uses, so that you can look up an argument or a message while you work through the lab. Everything here is as the course's UVM library (UVM 2020-3.2, in `third_party/uvm`) implements it. The router's own register map, with the UVM access policy of every field and whether the field is volatile, is in the specification.

Configuring a field

You create a field with the factory and then configure it:

```
f = uvm_reg_field::type_id::create("noroute");
f.configure(parent, size, lsb, access, volatile, reset, has_reset, is_rand,
individually_accessible);
```

Argument	Meaning
`parent`	the register the field belongs to, which is `this` in the register's `build()`
`size`	the field's width in bits
`lsb`	the position of its lowest bit in the register
`access`	the UVM access policy; the router's fields use `"RW"`, `"RO"`, `"W1C"` and `"WC"`
`volatile`	1 if the design can change the field on its own
`reset`	its value after reset
`has_reset`	1 if the field has a reset value (every field on the router has one)
`is_rand`	1 if `randomize()` may pick a value for it. Pass 1 only for a field that software writes with a value of its own choosing, which on the router means an `RW` field
`individually_accessible`	1 if UVM may read or write the field without the rest of its register. Pass 1 only for a field that is alone in its register, which on the router means the fields of `ID` and `CTRL`

The register layer never compares a volatile field when it reads one, because the design may have changed the field since the model last looked.

Mirror, desired value and prediction

The model keeps two values for every field. The **mirror** is what the model believes the design holds. The **desired** value is what the testbench last asked for. A `write()` sets the desired value first and then starts the transfer.

Way the mirror changes	When
auto-predict (`set_auto_predict(1)` on the map)	after each operation that the model itself sends, but not after a transfer that goes around the model

Way the mirror changes	When
a `uvm_reg_predictor` connected to a monitor's analysis port	after every transfer the monitor reports, whoever started it
`peek()` and `poke()`	at once, since neither uses the bus. `peek()` sets the mirror to the value it found, and `poke()` sets it to the value it wrote, without applying the field's access policy

A prediction from a read sets both the mirror and the desired value to the value read.

Check	What it compares
`r.mirror(status, UVM_CHECK)`	the mirror with the value it reads, for every field that isn't volatile
check-on-read (`set_check_on_read(1)` on the map)	the mirror with the value of every read that the predictor sees, for the same fields

Either check reports a mismatch as

```
[RegModel] Register "regs.CTRL" value read from DUT (0x0000000000000001) does not match
mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000001)
```

where the valid bit mask has a 1 for every bit that was compared.

The built-in sequences

Neither sequence needs a sequencer of its own. To run either one, set its `model` to the register block and start it with `start(null, this)`. It sends its reads and writes through the model, and so through your adapter.

Sequence	What it does	What it skips
`uvm_reg_hw_reset_seq`	resets the model to its reset values. Then, for each register, it prints `Verifying reset value of register ...` and reads the register with `mirror(UVM_CHECK)`	a register in which no field would be compared, which on the router means one whose fields are all volatile
`uvm_reg_bit_bash_seq`	prints `Verifying bits in register ...` for each register. Then, for each bit it can check, it writes the register twice, each time with that bit the opposite of the desired value, reads the register back after each write, and compares the read with the desired value	the bits of volatile fields. It still tests a bit that belongs to no field, as a read-only bit that should read 0, so both of its writes set that bit to 1

A bit-bash failure is reported as

```
[uvm_reg_bit_bash_seq] Writing a 1 in bit #0 of register "regs.ROUTE3" with initial
value 'h0000000000000000 yielded 'h0000000000000002 instead of 'h0000000000000001
```

The back door

A back-door access reads (`peek`) or writes (`poke`) a register through the name of the signal that holds it, with no bus transfer and no simulation time. The model finds the signal from HDL paths. `add_hdl_path()` on the block names the design instance that holds the registers, and `add_hdl_path_slice(signal, lsb,`

`width)` on a register names the signal that holds each of its bit ranges. In Lab 06 the block's path is `router_tb_top.dut.u_regs`, and `CTRL` is held in `en_q` inside it.

UVM reaches the signal through VPI, and Verilator lets VPI see a signal only if it is marked public. The lab's `tb/router_regs.vlt` marks as public the five signals that hold `CTRL`, `STATUS`, `INT_EN` and `DROP_CNT` (`DROP_CNT`'s two counters are separate signals), with one line for each signal. This is the line for `CTRL`:

```
public_flat_rw -module "router_regs" -var "en_q"
```

The lab names the signals one at a time because on Verilator 5.050 the UVM build breaks with `--public-flat-rw`, the command-line flag that marks every signal public. Lab 06, Part 6, measures what `peek` and `poke` do on the router.

D. 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