DD105-B — Testbenches with UVM

Build a UVM testbench in Python, with pyuvm and cocotb on Verilator, for the same packet router as DD105-A.

UofT ASIC Team

Contents

1 DD105-B — Testbenches with UVM 5

I 1. Two frameworks

2 Getting started 11

3 What a framework buys you 17

4 From cocotb tests to pyuvm components 21

5 Lab 01: One ALU, two frameworks 27

II 2. A layered testbench

6 Python classes and cocotb concurrency 41

7 Driving and sampling on the clock 51

8 Lab 02: A layered testbench without pyuvm 57

III 3. Constraints and coverage

9 Constrained random with pyvsc 69

10 Functional coverage you can read 77

11 Lab 03: Constraints and coverage 83

IV 4. pyuvm

12 The pyuvm skeleton 99

13 Sequences, sequencers and drivers 105

14 Monitors, analysis ports and scoreboards 111

15 The factory and ConfigDB 117

16 Lab 04: The same testbench in pyuvm 123

V 5. Protocol rules

17 Assertions for interface rules 139

18 Reactive agents and backpressure 145

19 Virtual sequences 151

20 Lab 05: Protocol rules and backpressure 157

VI 6. Registers

21 Register models in pyuvm 171

22 Regressions and seeds 177

23 Lab 06: The register block 183

VII 7. Capstone

24 Verification plans and sign-off 197

25 Writing the verification report 201

26 Lab 07: Capstone 205

Appendices 217

A The router in one page 219

B The packet router 227

C The pyuvm register API 241

D Getting the tools outside the course image 251

1. DD105-B — Testbenches with UVM

In this course you build testbenches in Python for a small packet router and run them on Verilator. Each one has a driver that sends packets into the router, a monitor that turns the valid/ready handshakes on one of the router's streams into packets, and a scoreboard that predicts which packets should come out and checks the ones that do. In Lab 02 we give you the driver, and you write the monitor and the scoreboard as plain Python classes on cocotb. In Lab 03 you add functional coverage and constraints on the packet lengths to that testbench, and in Lab 04 you finish our port of it to pyuvm, a Python library of UVM's classes that runs on cocotb.

DD105-B is part of the UofT ASIC Team education materials, published at https://uoftasic.com/dd105-b/. The labs run in the workbench, which is the IIC-OSIC-TOOLS container that you start from the `workspace` clone you set up in IC101. In the workbench, the command `mod add dd105-b` clones this course's repository, and everything the labs run is in that repository's `labs/` folder. Lab 01, Part 1 walks you through `mod add dd105-b`.

Who this course is for

You have finished DD104, where you wrote cocotb tests in Python that caught all thirteen bugs planted in SimProc. Lab 01 starts on SimProc's ALU, which you tested in DD104 Lab 03, with the three bugs that DD104 planted in it.

From Lab 02 on you test the router. The router is about 730 lines of SystemVerilog. It was written for DD105-A, and this course has its own copy of it. 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 when you run a simulation you can switch one of them on. By the end of Lab 07 your pyuvm testbench catches all fifteen, and you have written a verification plan and a verification report. Everything you write for the testbenches is Python, apart from the assertions in Labs 05 and 07, which you write in SystemVerilog and the testbench binds into the router.

Track	Digital
Prerequisites	DD104, and through it IC101 and DD101 to DD103
Tools	Verilator 5.050, cocotb 2.0.1, pyuvm 4.0.1, Python 3.12.3 and Surfer 0.7.0, all in the workbench image `hpretl/iic-osic-tools:2026.08`. From Lab 03 on, the labs also use pyvsc 1.1.0 for constrained random and cocotb-coverage 2.0 for functional coverage. Neither is in the image, so in Lab 03 you install both into a Python venv with `make venv`.
Machine	The workbench you already have. Every time on these pages was measured with 2 CPUs and 3 GB. The router and its assertions build in about 8 s with an empty compiler cache, and an edit to a Python file rebuilds nothing. One run of a lab's graded test takes about 8 s in Lab 06 and about 12 s in Lab 07. The longest steps are the two regressions in Lab 07, which take about 9 minutes each.
Total time	21 h 10 min, which is the sum of the row times in the Course path below. The 14 h for the seven labs is the sum of the times that the lab writeups give. The 7 h 10 min for the eighteen guide pages is our estimate, at a minute for every 130 words, plus 2 minutes for each figure and 1 minute for

each 10 lines of code or output, with each Read row rounded to the nearest 5 minutes.

Course path

Work down this table in order. A Read row is a set of guide pages, and a Do row is one lab writeup. The guide pages are for reading only. Some of them show a command we ran and what it printed, but the lab writeups are the only pages that ask you to run anything.

#		What	Time
1	Read	Getting started · What a framework buys you · From cocotb tests to pyuvm components	1 h
2	Do	Lab 01: One ALU, two frameworks. Test SimProc's ALU with DD104′s cocotb test and with a pyuvm testbench, compare their lines, build times and run times with DD105-A's numbers for a SystemVerilog UVM testbench, and extend the pyuvm scoreboard until it catches A03, a bug in the zero flag.	1 h 15 min
3	Read	Python classes and cocotb concurrency · Driving and sampling on the clock	1 h 5 min
4	Do	Lab 02: A layered testbench without pyuvm. Write the monitor and the scoreboard of a plain-cocotb testbench for the router, catch B05 and B09 with it, and find out which part of the stimulus each catch depends on.	1 h 30 min
5	Read	Constrained random with pyvsc · Functional coverage you can read	50 min
6	Do	Lab 03: Constraints and coverage. Add functional coverage to the Lab 02 testbench, weight its packet lengths with pyvsc constraints, and write the stimulus that reaches the two cover-	1 h 30 min

#		What	Time
		age holes where B07 and B08 are.	
7	Read	The pyuvm skeleton · Sequences, sequencers and drivers · Monitors, analysis ports and scoreboards · The factory and ConfigDB	1 h 35 min
8	Do	Lab 04: The same testbench in pyuvm. Finish our pyuvm port of the Lab 03 testbench by adding an agent for output 1, a test that stalls one output at a time, and a factory override that replaces each random packet with one that goes to a disabled route one time in four, so that the port catches B06 and B13.	2 h
9	Read	Assertions for interface rules · Reactive agents and backpressure · Virtual sequences	1 h 15 min
10	Do	Lab 05: Protocol rules and backpressure. Write three SystemVerilog assertions for the router's output rules, finish a sink driver that answers each output beat, and write a virtual sequence whose oversize packets reach B11.	2 h
11	Read	Register models in pyuvm · Regressions and seeds	50 min
12	Do	Lab 06: The register block. Write the fields of seven registers, the adapter, the predictor, and the reset and bit-bash sequences for the router's pyuvm register model, then test the 246 addresses outside the register map and check `irq` in every cycle, so that the test-	2 h 15 min

#		What	Time
		bench catches B01 to B04 and B14.	
13	Read	Verification plans and sign-off · Writing the verification report	35 min
14	Do	Lab 07: Capstone. Hunt three mystery bugs, catch B15 by writing the assertions for the rules about reset and a sequence that resets the router with packets in flight, and keep your verification plan, your report, your testbench and its results in a GitHub repository of your own.	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 from start to finish:

- The router in one page
- The packet router, the router's full specification
- The pyuvm register API, for Lab 06
- Getting the tools outside the course image

DD105-A

DD105-A has the same router, the same fifteen bugs and the same seven labs in the same order as this course, but its testbenches are written in SystemVerilog, with the UVM library, on Verilator 5.050. You take one of the two courses, not both. In DD105-A's Lab 01, a SystemVerilog UVM testbench for SimProc's ALU took 102 s to build, because Verilator compiled the whole UVM library along with the ALU and the testbench, and an edit to its scoreboard took 99 s to rebuild. In this course's Lab 01, the simulator for the pyuvm testbench of the same ALU builds in 7.3 s with an empty compiler cache, because Verilator compiles only the ALU. The testbench is Python, which cocotb imports each time the simulator starts, so an edit to its scoreboard rebuilds nothing.

Getting help

Ask in the team Discord. 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's folder in `build/runs/` if the question is about a simulation.

I

1. Two frameworks

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 pyuvm 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 tests to pyuvm components 21**
4 DD104's t01 as one pyuvm test 21
4 Logged errors and BaseTest 22
4 Phases 22
4 Objections 24
4 From one test to components 24
4 cocotb and pyuvm side by side 25
4 Check yourself 25
4 Further reading 26
4 Answers 26

5 **Lab 01: One ALU, two frameworks . . 27**
5 Before you start 27
5 Part 1: Load the course 27
5 Part 2: The cocotb testbench 28
5 Part 3: The pyuvm 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 in the parts after it you'll see the builds, runs and verdicts that this page describes.

Requirements

You set up the workbench in IC101, and you used the same container all through DD104. DD105-B runs in that container too. Everything in the course is simulation, so it needs no PDK and no synthesis flow.

Setting up takes three steps:

- Start the noVNC desktop from your `workspace` clone with `./scripts/start_vnc.sh` (`scripts\start_vnc.bat` on Windows). IC101: Launch the noVNC desktop gives the ports and the password, and it tells you where to look if the desktop page comes up blank.
- In a terminal on the desktop, load the environment with `. /foss/designs/common/.designinit`. It defines the workbench's `mod` command. Every new terminal starts without it, so you load it in each one.
- Run `mod add dd105-b`, the first time only.

`mod add dd105-b` clones the course into `/foss/designs/modules/dd105-b` and prints:

```
Cloning https://github.com/uoftasic/dd105-b.git
     into /foss/designs/modules/dd105-b
Cloning into '/foss/designs/modules/dd105-b'...
OK  module ready: /foss/designs/modules/dd105-b
    run: mod dd105-b
cwd: /foss/designs/modules/dd105-b
```

From then on, `mod dd105-b` takes you to that folder and prints the same `cwd:` line. 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.

Labs 01 and 02 run on the image alone. Verilator, cocotb, pyuvm and Surfer are all in it. From Lab 03 on, the labs also use pyvsc, for constrained-random stimulus, and cocotb-coverage, for functional coverage. Neither of those is in the image. In Lab 03, Part 1 you install them once, with `make venv`, into a Python virtual environment (a *venv*) that also sees the image's own packages. pip downloads them, so that step needs the network. The venv goes in your workspace, at `/foss/designs/.venvs/dd105-b`, so it survives a restart of the container. Ours came to 37 MB. Activating a venv in your shell doesn't change the Python that the simulator starts, so whenever the venv exists, the labs' shared makefile tells cocotb to start the venv's Python, in every run.

Each lab's makefile works out its own paths, so you can also run any lab with `make` alone, from 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`
cocotb	`2.0.1`
pyuvm	`4.0.1`
Python	`Python 3.12.3`

Tool	What its version check prints
Surfer	`surfer 0.7.0 (git: v0.7.0)`
pyvsc and cocotb-coverage, in the venv	`pyvsc 1.1.0` and `cocotb-coverage 2.0`, as `make venv` lists them

You'll check the first five rows yourself in Lab 01, Part 1, and `make venv` prints the last one in Lab 03. Keep pyuvm's version in mind when you read about pyuvm elsewhere, because pyuvm's online documentation describes a newer release than 4.0.1. Its example test calls `self.raise_objection("Description")`, but in 4.0.1 `raise_objection()` takes no argument. It also describes logging through `self.uvm_report`, which 4.0.1 doesn't have. This course goes by the pyuvm source that is installed in the image, in `/usr/local/lib/python3.12/dist-packages/pyuvm`.

How long things take

Verilator compiles only the design, not the testbench. Every testbench in this course is Python, which cocotb runs inside the simulator each time the simulator starts, so editing a testbench never rebuilds anything. These are the times we measured for Lab 01 in the course image, with 2 CPUs and 3 GB:

Job	Wall time
The first build, with an empty compiler cache	7.3 to 7.6 s
The second build of the same ALU, for the pyuvm testbench	1.6 to 1.8 s
A build after you change the testbench	none: `Vtop is up to date.`, in 0.07 s
One run of the pyuvm test, 2000 vectors	0.54 s
`make check-all`: the test against the correct ALU and three bugs	2.0 s, 0.5 s a run
The cocotb test, all 393,216 vectors	11.0 to 11.7 s

The first build compiles seven C++ files, and only one of them, `Vtop__ALL.o`, is the ALU. The other six are Verilator's and cocotb's support code, and the compiler cache keeps all seven. The second build of the same ALU found six of them in the cache and compiled one, so it took under 2 s. Lab 02′s first build of the router took 2.2 s and compiled two files: the router, and cocotb's `main()` for the simulator. It compiled `main()` again because `main()` includes a header that Verilator writes for each design. Build times vary with the machine and with whatever else it's doing, so use the table to judge how long to wait, and don't expect your times to match it.

In each lab, one build serves all of a testbench's tests and all of the lab's planted bugs. Every lab from Lab 02 on has one testbench, and so one build. Lab 01 has two testbenches, the cocotb test and the pyuvm testbench, and it builds the ALU once for each. 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 either.

Runs take longer from Lab 03 on. Each run spends about 1.5 s importing pyvsc, and a graded test sends hundreds of packets, so a run of Lab 03′s graded test took us about 3 s, where Lab 02′s took 0.25 s. The longest steps are Lab 07′s two regressions, which took us 8 min 27 s and 9 min 21 s.

Where the builds go

Each lab builds into its own `build/` folder. The simulator is `build/sim_build/Vtop`, and for Lab 01′s ALU, `build/sim_build/` came to 1.5 MB. Lab 01′s cocotb test builds its own simulator in `build/cocotb/`. Each run gets a folder of its own under `build/runs/`, named after the test, the seed and the bug (for example `AluRandTest-s1-A03`). It holds `run.log`, cocotb's `results.xml`, the exact command (`cmd`), and the waves if you asked for them. After Lab 01′s two builds and a few runs, `build/` held 3.4 MB.

The compiler cache is ccache. It lives at the top of your clone, in `.ccache`, and 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 1 GB, and after Lab 01′s two builds it held 672 KB.

`make clean` in a lab folder removes that lab's `build/` and keeps the cache and the venv:

```
rm -rf /w/labs/lab-01-one-alu-two-frameworks/build
 removed /w/labs/lab-01-one-alu-two-frameworks/build. The ccache (/w/.ccache) and the
venv are kept.
```

We ran it with the course folder mounted at `/w`, so on the workbench your paths start with `/foss/designs/modules/dd105-b` instead. `make clean-cache` deletes the cache, and the next build after it is a cold one again. `make clean-venv` deletes the venv, and `make venv` makes it 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 and the current values of its variables.

Reading a lab's output

Every run starts with cocotb's own lines, which give the Verilator version, the seed (`Seeding Python random module with supplied seed 1`) and cocotb's version. Your test's lines come next, then cocotb's table of the tests it ran, and last the course's one-line verdict on the run, between two rules. The last of your test's lines, and the wording of the verdict, depend on what kind of test it is.

Every pyuvm test in the course derives from `BaseTest`, in `labs/common/py/dd105b_base.py`. Lab 01′s `AluRandTest` is a pyuvm test, and so is every test from Lab 04 on. pyuvm reports through Python's `logging`, and an error that a component logs with `self.logger.error()` doesn't fail anything by itself, so without `BaseTest` the test would still pass. `BaseTest` counts every error that any component logs, and at the end it prints one line with the count and fails the test if the count isn't 0. In a pyuvm run that passes, the last line that your test prints is this one:

```
DD105-B BaseTest: 0 errors logged, no bound assertions
```

Then come cocotb's line saying that the test passed and cocotb's table, and the run ends with the verdict:

```
-----------------------------------------------------------------------------
 ok: 0 errors   0.53 s   log: build/runs/AluRandTest-s1/run.log
-----------------------------------------------------------------------------
```

"No bound assertions" means that no SystemVerilog assertions are attached to the design. From Lab 05 on, assertions are bound into the router, and the line counts their failures too.

Lab 01′s cocotb test and the tests in Labs 02 and 03 are plain cocotb tests. They have no `BaseTest`, because a failed check in them raises `AssertionError`, and that fails the test by itself. When one of them passes, its verdict reads `ok: passed`.

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 is `BROKEN`, and a `BROKEN` run never counts as a catch. The other reasons include an exception that isn't a check (a `NameError` from a typo in your code, for example), a crash, a timeout and a failed build. A pyuvm test that passed without `BaseTest`'s line is `BROKEN` too, because its logged errors would have failed nothing. Lab 01′s shipped pyuvm testbench misses one of the lab's three bugs, so your first `make check-all` there ends like this:

```
  MISSED  tier 1  A03_zero_flag_low_nibble        0 errors                    0.5s
  caught  tier 1  A01_sub_operands_swapped        291 errors                  0.5s
  caught  tier 1  A02_nand_is_nor                 278 errors                  0.5s
-----------------------------------------------------------------------------
 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/pyuvm.mk:281: 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, and cocotb uses it to seed Python's `random`, so your counts should match the labs' exactly. If one doesn't, something is different and we'd like to know what. Wall-clock times change from one run to the next, so treat them only as a rough guide.

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 an error that a pyuvm component logs while its test passes, so a lot of what looks like your own mistake is really the tools being unhelpful.
- Quote the exact error, the command you ran, and the test, seed and bug. "My scoreboard fails" and "my scoreboard reports 48 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 -- your testbench reads which bug is on.` 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. Your pyuvm scoreboard logs an error, cocotb's table says `PASS`, and the verdict says `BROKEN`. What's the most likely cause?

   Answer: A3, in Answers at the end.

Further reading

- pyuvm's README at release 4.0.1, the version in the course image. Its table of the IEEE 1800.2 standard says, section by section, what pyuvm implements and what it leaves out. The same text on pyuvm.readthedocs.io describes a later release. Correction to those pages for this course: in 4.0.1, `raise_objection()` and `drop_objection()` take no argument, and there is no `self.uvm_report`.
- cocotb 2.0.1's documentation, the version in the image. The pages for other versions differ, so check that the address has `v2.0.1` in it.
- ChipVerify's Introduction to Verilator. Correction: it says that Verilator supports only synthesizable Verilog, and it sends you to Icarus for testbenches with delays or `fork`. In this course the testbench is Python and the delays are cocotb's, so Verilator compiles only the design, which is synthesizable. DD105-A, which writes its testbenches in SystemVerilog, runs delays and `fork` on Verilator 5.050 with Verilator's `--timing` option.

Next: What a framework buys you.

Answers

A1. Check yourself

No. `make clean` removes only that lab's `build/` folder. The compiler cache in `.ccache` at the top of your clone stays, and it holds the support code that every build compiles, so Lab 02's first build compiled two files, the router and cocotb's `main()`, and took 2.2 s. Only `make clean-cache` empties the cache.

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

The test doesn't end through `BaseTest`, so its log has no `DD105-B BaseTest:` line. Either it derives from pyuvm's `uvm_test` directly, or it overrides `final_phase` without calling `super().final_phase()`. Without `BaseTest`, the logged error failed nothing, and the verdict won't call that run a pass.

3. What a framework buys you

This page answers: *what is pyuvm for, and what does it cost compared with the cocotb tests you already write?*

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, and pyuvm is UVM written in Python. In Lab 01 you'll test SimProc's ALU twice, once with the cocotb test from DD104 Lab 03 and once with a pyuvm testbench, both on Verilator. 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.

DD105-A teaches the same course with UVM in SystemVerilog, and its Lab 01 measured a SystemVerilog UVM testbench of the same ALU. This course doesn't build that testbench, so the numbers this page gives for it come from DD105-A's runs.

What pyuvm is

UVM is standardised as IEEE 1800.2, which defines it as a library of SystemVerilog classes. pyuvm's README calls pyuvm "a clean implementation of IEEE 1800.2 in Python". pyuvm uses cocotb to talk to the simulator. The course image has pyuvm 4.0.1 installed beside cocotb 2.0.1, and because pyuvm is plain Python, none of it is compiled. Verilator builds a simulator from the ALU alone. When that simulator starts, cocotb imports your testbench, and your testbench imports pyuvm like any other Python package.

The library gives you a base class for each job a testbench does, and the code that builds those pieces, connects them and runs them in order. 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_sequencer`, `uvm_driver`, `uvm_monitor`, `uvm_subscriber` (for the 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 methods, in that order, and **objections**, which the test raises and drops to tell pyuvm when the run phase is over;
- `ConfigDB`, a table of named values. The code that sets a value gives a path, and only the components whose paths match it can read the value. Lab 01's test sets the object that holds the ALU's pins for the path `*`, which every component matches, so the driver and the monitor can both read it;
- **analysis ports**, which pass each transaction that the monitor writes to every component connected to the port. In Lab 01 the scoreboard is the only component connected to it;
- the **factory**, through which every component and transaction is made, with `AluDriver.create("drv", self)` rather than `AluDriver("drv", self)`. Lab 04 adds a factory override, which lets a test swap one class for another without editing the code that makes it;
- `@pyuvm.test()`, which turns a test class into a cocotb test named after the class, so that `make run TEST=AluDirectedTest` picks it when the simulator starts.

pyuvm leaves out some of SystemVerilog UVM. It has no macros, because Python doesn't need them to register a class with the factory or to print and copy an object. It reports through Python's `logging` instead of a report server, so when a component calls `self.logger.error()`, pyuvm prints the message and the test still passes. Every pyuvm test in the course derives from the course's `BaseTest`, which counts those errors and fails the test at the end if there were any. The course's verdict calls a pyuvm run `BROKEN` if it passed without printing `BaseTest`'s closing line. From cocotb tests to pyuvm components shows a test with and without it.

The two testbenches

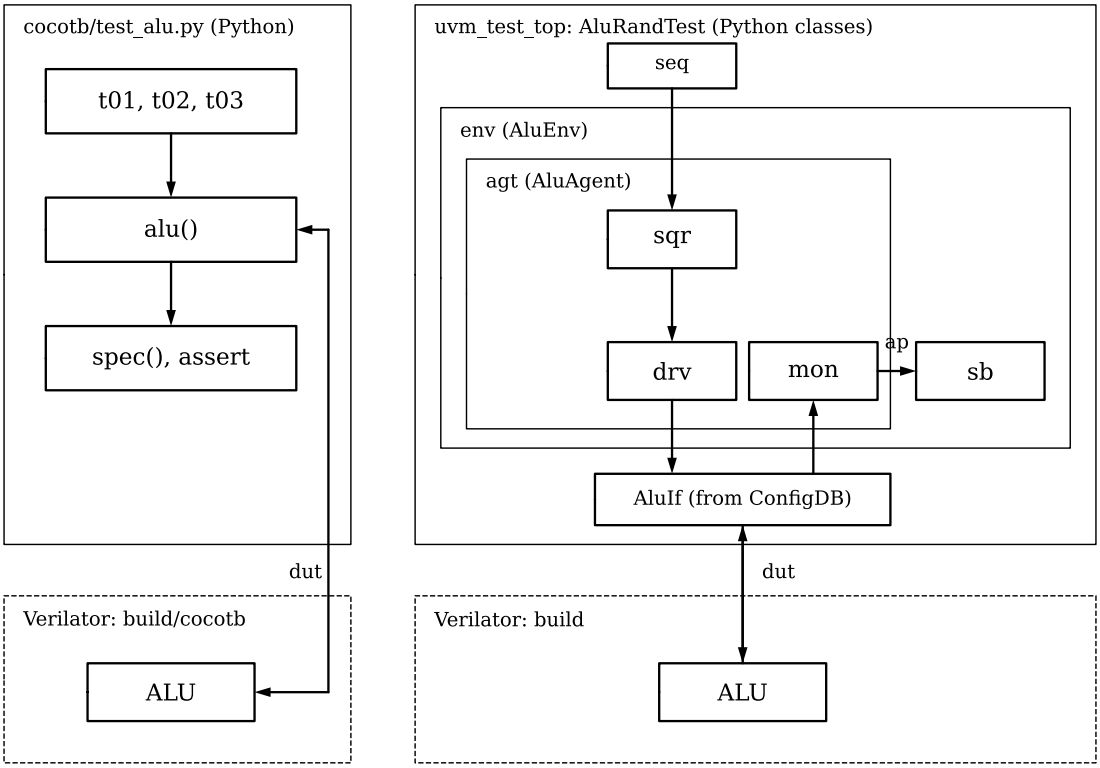


Figure 3.1: Lab 01′s cocotb test beside its pyuvm testbench

Lab 01′s two testbenches. Each is Python that cocotb runs inside a simulator of its own, and each simulator is Verilator's build of `rtl/alu.sv` alone. The cocotb test reaches the pins through `dut`; the pyuvm testbench reaches them through `AluIf`, which holds `dut`.

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 pyuvm testbench gives each job its own class. A sequence makes `AluItem` objects and hands them to the sequencer, which passes them to the driver. The ALU has no clock, so the driver and the monitor share a 10 ns period and a `valid` flag through `AluIf`. The driver holds each vector on the pins for one period and sets `valid` while it does. The monitor reads the pins 5 ns into every period, and in each period that carries a vector it copies the vector and the ALU's answer into a new `AluItem`. The monitor writes each item to its analysis port without knowing what the answer should be, because deciding that is the scoreboard's job. The scoreboard receives each item in its `write()` method, 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. In its check phase the environment also makes sure that the scoreboard checked every vector that 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 we measured in Lab 01, in the course image on 2 CPUs. The third column comes from DD105-A's Lab 01, on the same Verilator:

	cocotb	pyuvm	SystemVerilog UVM (DD105-A)
Lines of code (not counting blank lines, comments and doc-strings)	94	161	282
First build	7.3 to 7.6 s	7.3 s; 1.6 to 1.8 s after the cocotb build	102 s
Rebuild after you change the testbench	none	none	26 s to 99 s
Run	11.0 to 11.7 s, 10.6 to 11.2 s of it in `t02`'s 393,216 vectors	0.54 s for 2000 vectors	1.17 s for 2000 vectors
Time per vector	about 0.03 ms	about 0.1 ms	at most 0.6 ms

Lines. The pyuvm testbench has about 1.7 times the cocotb test's code lines. The extra lines go into pieces that the cocotb test doesn't need. The agent and the environment only create and connect the other pieces, `AluIf` carries the period and the `valid` flag, and the driver and the monitor between them do what `alu()` does in five lines. DD105-A's SystemVerilog testbench had three times the cocotb test's code lines. Lab 01, Part 3 asks you which of those lines Python doesn't need.

Build. Whichever testbench you run, Verilator compiles only the ALU, because both testbenches are Python. Most of the first build's 7.3 s goes on compiling the support code that Verilator and cocotb add to the simulator, and the compiler cache keeps that code for the next build (Getting started has the numbers). DD105-A's build took 102 s because Verilator compiled the whole UVM library along with the ALU and the testbench. Its build report counted 24.157 MB of source in 386 modules, and g++ spent 98.5 s of the 102.

Rebuilds. When you change either Python testbench, nothing has to be rebuilt before the next run, because cocotb imports the testbench each time the simulator starts. After the flag-check edit in Lab 01, Part 5, `make build` prints `Vtop is up to date.`, and it took 0.07 s when we timed it. In DD105-A the same edit to the SystemVerilog scoreboard cost a rebuild of 99 s. DD105-A also timed two one-line edits: a change inside a task's body rebuilt in 26 s, and one new class member in 93 s.

Speed. The table that cocotb prints at the end of a run puts `AluRandTest`'s 2000 vectors at 0.19 s of real time, about 0.1 ms each, and `t02`'s 393,216 at 10.64 s, about 0.03 ms each. In the pyuvm testbench each vector goes through a sequence, the sequencer, the driver, the monitor and its analysis port, and the scoreboard. In `t02` each vector is one call to `alu()` and one to `spec()`. The pyuvm run is still quick, because `AluRandTest` sends 2000 random vectors rather than sweeping all 393,216 as DD104's `t02` does. DD105-A's 2000 vectors in 1.17 s come to at most 0.6 ms each, because the 1.17 s also covers starting the simulator.

What it buys

For that cost, Lab 01's pyuvm testbench gives you two things, both 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 `AluRandTest` and `AluDirectedTest` use it without being edited, because neither test does any checking of its own. In `test_alu.py`, `t01` and `t02` each do their own comparison, so a new check would have to be written into each of them.
- **Tests that reuse the pieces.** All that `AluDirectedTest` adds is a short sequence that sends DD104's six `t01` vectors and a short test class that starts it. It reuses the driver, the monitor and the scoreboard as they are. You choose it when you run, without a rebuild, but cocotb chooses a plain cocotb test by name in the same way when the simulator starts, so pyuvm doesn't add that part.

In DD105-A, UVM's report server counted the errors by their ID, and printed `[VALUE]  282` at the end of a run with A01 switched on. pyuvm 4.0.1 has no report server, so in this course the scoreboard keeps its own counts by kind and prints them on the `[SB]` line in its `report_phase`. On A01 that line is `[SB] 2000 vectors checked: 291 value mismatches (SUB 291)`.

You still decide what the scoreboard checks, and Lab 01′s shipped scoreboard checks only the value. Against Lab 01′s three bugs it caught A01 with 291 errors and A02 with 278, but on A03 it logged no errors at all, so it missed that bug. The cocotb test caught all three, because its `t02` compares the flags as well as the value. Once you add the flag check, the pyuvm test catches A03 with 125 errors in 2000 vectors.

You get more back on a bigger design, and UVM was made for designs like the router you'll test from Lab 02 on. 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. In Labs 02 and 03 you build that testbench from plain Python classes on cocotb first, so that each pyuvm piece has an obvious job when it appears. In Lab 04 you finish our pyuvm port of it, count its lines against your Lab 03 testbench, and work out which pieces moved across, which changed shape and which are new.

Check yourself

1. In Lab 01′s scoreboard you change the body of `predict()`. How long will the rebuild take? How long did the same kind of change take in DD105-A?

   Answer: A1, in Answers at the end.

2. Lab 01′s shipped scoreboard logs 291 `[VALUE]` errors on A01. If `AluBaseTest` derived from pyuvm's `uvm_test` instead of the course's `BaseTest`, what would cocotb report for the test?

   Answer: A2, in Answers at the end.

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

   Answer: A3, in Answers at the end.

Further reading

- pyuvm's README at release 4.0.1: "IEEE 1800.2 and pyuvm" goes through the standard section by section and says what pyuvm keeps, changes or leaves out, and "The pyuvm testbench" walks through a testbench for an ALU.
- ChipVerify's UVM Tutorial: a map of the library's main pieces. Its code is SystemVerilog, written for commercial simulators.
- ChipVerify's 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. Lab 01′s `AluMonitor` makes its port in `build_phase`, with `uvm_analysis_port("ap", self)`.

Next: From cocotb tests to pyuvm components.

Answers

A1. Check yourself

Nothing is rebuilt. The simulator is built from `rtl/alu.sv` alone, and the scoreboard is Python that cocotb imports when the simulator starts, so `make build` prints `Vtop is up to date.` In DD105-A the whole testbench is compiled into the simulator, so any edit to it means a rebuild. DD105-A measured 26 s for a change inside a task's body.

A2. Check yourself

That it passed. pyuvm logs each error through Python's `logging`, and a logged error doesn't fail a test. In this course `BaseTest` counts the errors and fails the test at the end. Without `BaseTest`'s line in the log, the course's verdict calls the run `BROKEN`.

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 pyuvm testbench catches A03 too.

4. From cocotb tests to pyuvm components

This page answers: *I can write a cocotb test. What changes when the same checks are written as pyuvm components?*

Python is still the language and cocotb still runs the test, so `dut.A.value = a`, `await Timer(1, unit="ns")` and `int(dut.ALUout.value)` mean in a pyuvm testbench what they meant in your DD104 tests. DD104 ran its tests on Icarus, and this course runs everything on Verilator, the cocotb test and the pyuvm testbench alike. pyuvm changes how the test is organised. The sections below start from DD104′s `t01` written as a single pyuvm test, and end with the pieces that Lab 01 splits it into. In Lab 01, Part 3 you'll read the whole pyuvm testbench, and in Part 5 you'll extend its scoreboard.

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

DD104′s t01 as one pyuvm test

`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, with two helpers: `spec()` works out what the ALU should produce, and `alu()` drives one vector, waits 1 ns and reads the three outputs back. The test itself is a loop over the six vectors:

```
@cocotb.test()
async def t01_one_vector_per_operation(dut):
    ...
    for op, a, b in 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} "
            ...)
```

`T01Test`, in the example's `components_demo.py`, is the same loop, with the same `spec()`, `alu()` and six vectors, written as a pyuvm test:

```
@pyuvm.test()
class T01Test(BaseTest):
    async def run_phase(self):
        self.raise_objection()
        dut = cocotb.top
        for op, a, b in CASES:
            got = await alu(dut, op, a, b)
            want = spec(op, a, b)
            if got != want:
                self.logger.error(
                    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]}")
        self.logger.info(f"[T01] {len(CASES)} directed vectors checked")
        self.drop_objection()
```

Four things changed:

- **The test is a class.** `@pyuvm.test()` wraps the class in a cocotb test with the class's name. cocotb lists that test as `components_demo.T01Test`, and `make run TEST=T01Test` picks it. The class derives from the course's `BaseTest`, which derives from pyuvm's `uvm_test`.

- **There's no `dut` argument.** `cocotb.top` is the same handle to the design's top level, which here is the ALU. A test can use it directly, as this one does, but a component deeper in the tree gets its handle from `ConfigDB`, as From one test to components shows.
- **The body is `run_phase()`, between `raise_objection()` and `drop_objection()`.** pyuvm starts `run_phase()` as a cocotb task and ends the run phase when the last objection is dropped. The section on objections shows what happens without them.
- **The check logs an error instead of asserting.** The loop goes on after a mismatch, so one run checks every vector, and `BaseTest` fails the test at the end if anything logged an error.

On the correct ALU the test checks its six vectors and passes:

```
    6.00ns INFO     ..omponents/components_demo.py(74) [uvm_test_top]: [T01] 6 directed
vectors checked
DD105-B BaseTest: 0 errors logged, no bound assertions
```

Each of pyuvm's log lines names the file and line it came from (cocotb shortens a long path at the front, hence the `..`) and gives the full name of the component that logged it. pyuvm names every test `uvm_test_top`, so here that component is the test.

Bug A01 makes `SUB` compute `B - A`. With it switched on, the second vector fails, the loop carries on through the sixth, and `BaseTest` ends the test with an `AssertionError` that quotes the first error:

```
    2.00ns ERROR    ..omponents/components_demo.py(70) [uvm_test_top]: SUB(0xC8, 0x3C)
gave out=0x74 N=0 Z=0, expected out=0x8C N=1 Z=0
    6.00ns INFO     ..omponents/components_demo.py(74) [uvm_test_top]: [T01] 6 directed
vectors checked
DD105-B BaseTest: 1 error logged, no bound assertions
```

```
 caught: 1 error   0.28 s   log: build/runs/T01Test-s1-A01/run.log
   T01Test: AssertionError: 1 error logged, the first: SUB(0xC8, 0x3C) gave out=0x74
N=0 Z=0, expected out=0x8C N
```

The error message is word for word the one that `t01`'s `assert` builds. In Lab 01, Part 2 the cocotb test's `t01` stopped at the same vector with the same message, because an `assert` that fails ends the test there.

Logged errors and BaseTest

Nothing in pyuvm turns a logged error into a failing test. pyuvm reports through Python's `logging`, and every component's logger has `propagate = False`, so it doesn't pass its records on to any logger that would count them. `self.logger.error()` prints a line and does nothing else. The example's `T01UvmTest` has `T01Test`'s `run_phase` line for line, in a class derived from pyuvm's `uvm_test` instead. With A01 switched on it logs the same error, and cocotb says it passed:

```
    2.00ns ERROR    ..omponents/components_demo.py(89) [uvm_test_top]: SUB(0xC8, 0x3C)
gave out=0x74 N=0 Z=0, expected out=0x8C N=1 Z=0
    6.00ns INFO     ..omponents/components_demo.py(93) [uvm_test_top]: [T01] 6 directed
vectors checked
    6.00ns INFO     cocotb.regression                  components_demo.T01UvmTest passed
```

The course's verdict grades the run `BROKEN` instead, because the log has no `BaseTest` line:

```
 BROKEN: no BaseTest line   0.27 s   log: build/runs/T01UvmTest-s1-A01/run.log
   the test passed and the log has no 'DD105-B BaseTest:' line
```

So every pyuvm test you write in this course derives from `BaseTest`, or from `RouterBaseTest` for the router. If you override `final_phase`, which is where `BaseTest` checks its error count and prints its line, call `super().final_phase()` from yours.

Phases

A cocotb test is one coroutine, and cocotb runs it from top to bottom. A pyuvm test is the root of a tree of components. pyuvm makes the test and then calls a fixed sequence of methods, the **phases**, on every

component in the tree, finishing each phase everywhere before it starts the next. A component overrides the phases it needs.

The example's `PhaseOrderTest` builds a small tree. The test has one child, an environment called `env`, and `env` has two, `drv` and `sb`. Every component prints a line in each phase. The test raises an objection, waits 35 ns and drops it, and `drv`'s `run_phase` counts a tick every 10 ns and never returns. This is everything the run prints between cocotb's `running` and `passed` lines:

```
    0 ns  build_phase               uvm_test_top
    0 ns  build_phase               uvm_test_top.env
    0 ns  build_phase               uvm_test_top.env.drv
    0 ns  build_phase               uvm_test_top.env.sb
    0 ns  connect_phase             uvm_test_top.env.drv
    0 ns  connect_phase             uvm_test_top.env.sb
    0 ns  connect_phase             uvm_test_top.env
    0 ns  connect_phase             uvm_test_top
    0 ns  end_of_elaboration_phase  uvm_test_top
    0 ns  end_of_elaboration_phase  uvm_test_top.env
    0 ns  end_of_elaboration_phase  uvm_test_top.env.drv
    0 ns  end_of_elaboration_phase  uvm_test_top.env.sb
    0 ns  start_of_simulation_phase uvm_test_top
    0 ns  start_of_simulation_phase uvm_test_top.env
    0 ns  start_of_simulation_phase uvm_test_top.env.drv
    0 ns  start_of_simulation_phase uvm_test_top.env.sb
    0 ns  run_phase starts          uvm_test_top.env.drv
    0 ns  run_phase starts          uvm_test_top.env.sb
    0 ns  run_phase starts          uvm_test_top.env
    0 ns  run_phase starts          uvm_test_top
   35 ns  drop_objection()          uvm_test_top
   35 ns  extract_phase             uvm_test_top
   35 ns  extract_phase             uvm_test_top.env
   35 ns  extract_phase             uvm_test_top.env.drv
   35 ns  extract_phase             uvm_test_top.env.sb
   35 ns  check_phase               uvm_test_top
   35 ns  check_phase               uvm_test_top.env
   35 ns  check_phase               uvm_test_top.env.drv
   35 ns  check_phase               uvm_test_top.env.sb
   35 ns  report_phase              uvm_test_top
   35 ns  report_phase              uvm_test_top.env
   35 ns  report_phase              uvm_test_top.env.drv
            uvm_test_top.env.drv counted 3 ticks
   35 ns  report_phase              uvm_test_top.env.sb
   35 ns  final_phase               uvm_test_top
DD105-B BaseTest: 0 errors logged, no bound assertions
   35 ns  final_phase               uvm_test_top.env
   35 ns  final_phase               uvm_test_top.env.drv
   35 ns  final_phase               uvm_test_top.env.sb
```

- **`build_phase`** goes from the top down, because a component's children don't exist until its own `build_phase` makes them. The test's `build_phase` makes `env`, and only then can pyuvm call `env`'s, which makes `drv` and `sb`. This is the phase where a component makes its children and reads what it needs from `ConfigDB`.
- **`connect_phase`** goes from the bottom up. By then every component exists, so it's where a component connects the ports of the components below it to each other.
- **`end_of_elaboration_phase` and `start_of_simulation_phase`** come before simulated time moves. `BaseTest` uses them to attach its error counter to every component's logger.
- **`run_phase`** is the only phase that takes simulated time. pyuvm starts every component's `run_phase` as a cocotb task, all at 0 ns, and the run phase lasts until the last objection is dropped, here at 35 ns. `drv`'s `run_phase` was still running then and was never waited for; it had counted 3 ticks, at 10, 20 and 30 ns.
- **`extract_phase`, `check_phase`, `report_phase` and `final_phase`** take no time, and run once the run phase is over. They're where a component gathers and checks its results and prints its summary, and `BaseTest`'s line comes from the test's `final_phase`.

In this run `connect_phase` went from the bottom up, the `run_phase` tasks started in the same order, and every other phase went from the top down. Those are the orders that pyuvm 4.0.1′s source sets for each phase, in `s09_phasing.py`. SystemVerilog UVM runs every phase from the bottom up except `build_phase` and `final_phase`, so a SystemVerilog UVM reference will give you the opposite order for `end_of_elaboration_phase`, `start_of_simulation_phase`, `extract_phase`, `check_phase` and `report_phase`.

pyuvm also has only these nine phases. It leaves out SystemVerilog UVM's run-time phases (reset, configure, main and shutdown), which run alongside `run_phase` there. SystemVerilog UVM can keep the run phase open for a set time after the last objection is dropped, which it calls a drain time. pyuvm has no drain time, so its run phase ends at the drop, as it did at 35 ns here.

Objections

A cocotb test ends when its function returns. In pyuvm, `run_phase` returning doesn't end anything, because every component's `run_phase` is a separate task and some of them, like `drv`'s, never return. The run phase ends instead when no component holds an objection. A test raises one before its stimulus and drops it after.

`NoObjectionTest` has the same tree, and its `run_phase` waits 35 ns and then prints a line, without raising an objection first. pyuvm finds no objection raised, warns, and ends the run phase at 0 ns:

```
   0 ns  run_phase starts           uvm_test_top.env.drv
   0 ns  run_phase starts           uvm_test_top.env.sb
   0 ns  run_phase starts           uvm_test_top.env
   0 ns  run_phase starts           uvm_test_top
      0.00ns WARNING   root                                        You did not call
self.raise_objection() in any run_phase
   0 ns  extract_phase              uvm_test_top
```

The rest of the phases follow at 0 ns, `drv` counts 0 ticks, the line that the test meant to print after 35 ns never appears, and the test passes, with `ok: 0 errors`, although it checked nothing. Lab 01′s environment guards against the same mistake in its `check_phase`, which logs `[COUNT] the scoreboard checked no vectors` if the scoreboard ends the run with nothing checked.

From one test to components

Lab 01′s pyuvm testbench splits `t01`′s loop into the pieces that the What a framework buys you diagram shows. Once the pieces are separate classes, a vector has to get from one piece to the next, and the driver and the monitor have to reach the pins. pyuvm has a part for each of these jobs, and its factory makes the pieces themselves.

A transaction instead of tuples. `t01` passes `(op, a, b)` into `alu()` and gets a tuple back. In Lab 01 a vector is an `AluItem`, a class derived from `uvm_sequence_item` that has fields for the stimulus and for the ALU's answer. The sequence sets the stimulus in each item it sends. The monitor makes an `AluItem` of its own for each vector it reads back, and fills in every field from the pins.

A sequence hands items to the driver. The sequence makes each item, waits for the driver to be ready for it, and hands it over. `AluRandSeq.body()` makes its 2000 random vectors in a loop:

```
for _ in range(self.n):
    req = AluItem.create("req")
    await self.start_item(req)
    req.randomize()
    await self.finish_item(req)
```

and `AluDriver.run_phase()` takes the items at the other end, one at a time, with `await self.seq_item_port.get_next_item()`, drives the pins as `alu()` did, and calls `self.seq_item_port.item_done()` when the vector has been on the pins for its 10 ns. The sequencer between them is pyuvm's own `uvm_sequencer`, used as it comes, and the agent connects the driver to it in the agent's `connect_phase`.

An analysis port carries what the monitor sees. `AluMonitor` makes the port in its `build_phase`, with `self.ap = uvm_analysis_port("ap", self)`, and calls `self.ap.write(t)` for each vector it reads back. The scoreboard derives from `uvm_subscriber`, whose `write()` method receives every

item that's written to the port it's connected to. `AluEnv.connect_phase()` makes that connection, with `self.agt.mon.ap.connect(self.sb.analysis_export)`. The monitor doesn't know the scoreboard exists, so a coverage collector could subscribe to the same port without the monitor changing.

ConfigDB carries the pins. No component is handed `dut` the way `alu()` is. `AluBaseTest.build_phase()` puts an `AluIf` holding `cocotb.top` into `ConfigDB` under the name `"vif"`, for every path (`"*"`), and the driver and the monitor fetch it in their own `build_phase` with `ConfigDB().get(self, "", "vif")`. A `get()` of a name that nobody set raises `UVMConfigItemNotFound`, which the course's verdict counts as `BROKEN`.

The factory makes the components and the items. `AluAgent.build_phase()` makes its driver with `AluDriver.create("drv", self)` rather than `AluDriver("drv", self)`, and its sequencer and monitor the same way. The two calls make the same object until a test asks the factory to make a different class in its place. Lab 04 does that with a packet class, `NoroutePkt`, in place of `RouterPkt`. A component's `create()` takes only a name and a parent, so nothing else, `dut` included, can be passed to the component when it's made.

cocotb and pyuvm side by side

In a cocotb test	In a pyuvm testbench
`@cocotb.test()` on an `async def` function	`@pyuvm.test()` on a class derived from `BaseTest`
`dut`, the test's argument	`cocotb.top`, or an object from `ConfigDB`
the test function's body	the test's `run_phase()`, between `raise_objection()` and `drop_objection()`
the test ends when the function returns	the run phase ends when the last objection is dropped; the other phases follow
`assert got == want, "message"`	`self.logger.error("message")`, and `BaseTest` fails the test at the end
a tuple of values	a `uvm_sequence_item`
a loop that makes vectors	a `uvm_sequence`'s `body()`
a coroutine that drives the pins	a `uvm_driver`'s `run_phase()`
reading the outputs back	a `uvm_monitor` that writes each item to its analysis port
the model and the comparison	a `uvm_subscriber`'s `write()`
`cocotb.start_soon(coro())`	the same, and pyuvm starts every `run_phase` this way
a test function, picked by its name with `COCOTB_TEST_FILTER`	a test class, picked by its class name: `make run TEST=AluDirectedTest` sets `COCOTB_TEST_FILTER` for you
`random.randrange(256)`	the same. Lab 03 adds pyvsc's constraints to a plain cocotb testbench, and the pyuvm testbenches from Lab 04 on keep them

Check yourself

1. You write a pyuvm test that logs an error on a broken ALU, and cocotb's table says `PASS`. What's the most likely mistake?

   Answer: A1, in Answers at the end.

2. A test's `run_phase` starts a sequence and awaits it, but never raises an objection. What does the run do?

   Answer: A2, in Answers at the end.

3. Why must `build_phase` go from the top down, while ports are connected in `connect_phase` rather than in `build_phase`?

   Answer: A3, in Answers at the end.

4. In `PhaseOrderTest`, `drv`'s `run_phase` never returns, yet the test ended at 35 ns. Why?

   Answer: A4, in Answers at the end.

Further reading

- pyuvm's README at release 4.0.1, "The pyuvm testbench": an ALU testbench built from the same kinds of pieces as Lab 01′s, with notes on where pyuvm differs from SystemVerilog UVM and why.
- ChipVerify's UVM Phases. Its code is SystemVerilog. Correction for pyuvm 4.0.1: it says that only `build_phase` and `final_phase` run from the top down, but in pyuvm every phase except `connect_phase` and the start of `run_phase` does, as `PhaseOrderTest` shows. It also lists the run-time phases (reset, configure, main and shutdown), which pyuvm doesn't have.
- ChipVerify's UVM Objection. Its code is SystemVerilog. Its point that a run phase with no objection ends at time 0 holds in pyuvm too, and pyuvm also prints the warning shown above. Correction for pyuvm 4.0.1: `raise_objection()` is a method of the component and takes no argument, and there is no drain time.

Next: Lab 01: One ALU, two frameworks.

Answers

A1. Check yourself

The test doesn't derive from `BaseTest`, or it overrides `final_phase` without calling `super().final_phase()`. A logged error fails nothing on its own, as `T01UvmTest` showed. The course's verdict calls such a run `BROKEN: no BaseTest line`.

A2. Check yourself

pyuvm warns `You did not call self.raise_objection() in any run_phase` and ends the run phase at 0 ns, before the sequence has sent anything. The later phases run at 0 ns and the test passes, having checked nothing, unless a check such as Lab 01′s `[COUNT]` in `check_phase` notices.

A3. Check yourself

A component's children are made in its own `build_phase`, so pyuvm can't call a child's `build_phase` before its parent's has run. The two components that a parent connects might not both exist until every `build_phase` has run. `AluEnv`, for example, connects `agt.mon`'s analysis port to `sb`, and `agt.mon` is made in `agt`'s own `build_phase`, after `AluEnv`'s has finished. `connect_phase` comes after every `build_phase`, so by then both ends exist.

A4. Check yourself

The run phase ends when the last objection is dropped, which the test did at 35 ns. pyuvm doesn't wait for the components' `run_phase` tasks to return; it goes on to the extract, check, report and final phases, and cocotb ends the tasks that are still running when the test ends.

5. Lab 01: One ALU, two frameworks

You'll test SimProc's ALU from DD104 with two testbenches, both on Verilator. The first is the cocotb test from DD104 Lab 03. The second is written with pyuvm, a Python library that implements UVM's classes and runs on cocotb. For each one you'll time the build and the run, see what it reports on three planted bugs and count its lines. DD105-A made the same measurements on a SystemVerilog UVM testbench for this ALU, and you'll compare your numbers with DD105-A's. Last, you'll extend the pyuvm scoreboard so that it catches A03. A03 is a bug in the zero flag, and the scoreboard as shipped misses it.

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

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 tests to pyuvm components.
- Copy this table into your notes. You'll fill in the cocotb and pyuvm columns as you go. This course doesn't build the SystemVerilog testbench, so the third column already holds DD105-A's numbers, which were measured in its Lab 01 with the same 2 CPUs that we used for the times on this page. The questions at the end ask you to compare the columns.

	cocotb	pyuvm	SystemVerilog UVM (DD105-A)
Lines (all / code)			424 / 282
First build			102 s
Rebuild after you change the testbench			26 s to 99 s
Run			1.17 s for 2000 vectors
A01			caught, 282 `UVM_ERROR`s
A02			caught, 255 `UVM_ERROR`s
A03			missed

Time: about 1 h 15 min.

Part 1: Load the course

Can you reach DD105-B'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-b                     # first time only: clones github.com/uoftasic/
dd105-b
mod dd105-b                          # every time, including the first
```

`mod dd105-b` prints:

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

3. Check the tool versions:

```
verilator --version
cocotb-config --version
python3 -c "import pyuvm; print(pyuvm.__version__)"
python3 --version
surfer --version
```

You should see:

```
Verilator 5.050 2026-07-01 rev v5.050
2.0.1
4.0.1
Python 3.12.3
surfer 0.7.0 (git: v0.7.0)
```

The second line is cocotb's version and the third is pyuvm's.

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-b` 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-b`	You ran `mod dd105-b` before `mod add dd105-b`.

For anything else, see IC101's Troubleshooting page.

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

Part 2: The cocotb testbench

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

1. Go to the lab folder:

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

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`
A02	`NAND` computes NOR instead of NAND	`B02_nand_is_nor`
A03	`Z` looks only at `ALUout[3:0]`	`B03_zero_flag_low_nibble`

One simulator holds all four ALUs. With no `+BUG` you get the golden ALU, and with `+BUG=A03` you get A03.

3. `cocotb/test_alu.py` is DD104 Lab 03's reference solution, with only its docstring changed. DD104 ran it on Icarus, and here it runs on Verilator, like everything else in this course. If you'd rather run your own test, copy it in under a new name and add `TB=test_alu_mine` to every `make cocotb`, `make cocotb-all` 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. `rtl/alu.sv` is 95 lines, and 53 of them are code. Write down how long you expect Verilator to take to build it, then build it and run the test on the golden ALU:

```
make cocotb
```

It took us about 19 seconds. The build comes first:

```
-----------------------------------------------------------------------
 BUILD  ALU   (Verilator, 2 jobs)
 Full log: /foss/designs/modules/dd105-b/labs/lab-01-one-alu-two-frameworks/build/
cocotb/build.log
-----------------------------------------------------------------------
-----------------------------------------------------------------------
   built   /foss/designs/modules/dd105-b/labs/lab-01-one-alu-two-frameworks/build/
cocotb/sim_build/Vtop in 7.3 s (1.5M in /foss/designs/modules/dd105-b/labs/lab-01-
one-alu-two-frameworks/build/cocotb/sim_build)
 ccache:  Hits: 0 / 7 ( 0.00%)  Misses: 7 / 7 (100.0%)
-----------------------------------------------------------------------
```

and you should see, at the end:

```
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      12360.42  **
                                                                                    **
test_alu.t02_every_vector_the_alu_can_be_given   PASS      393216.00          10.64
36965.46  **
                                              ** test_alu.t03_the_undefined_opcodes
PASS           2.00           0.00      15768.06  **
                                    ****************************************************************
                                                 ** TESTS=3 PASS=3 FAIL=0
SKIP=0                              393224.00          10.64      36957.32  **
                                    ****************************************************************
- :0: Verilog $finish
-----------------------------------------------------------------------
 ok: passed   10.94 s   log: build/cocotb/runs/test_alu-s1/run.log
-----------------------------------------------------------------------
-----------------------------------------------------------------------
 cocotb, test_alu on Verilator, the golden ALU: build 7.6 s, run 11.0 s
-----------------------------------------------------------------------
```

The `ok: passed` line is the course's verdict on the run, from DD104's verdict script adapted for this course. The line between the last two rules is the lab's makefile reporting how long the build and the

run took. The run is mostly `t02` driving all 393,216 vectors. Write both times in your table. Times are ballpark, but the counts in this lab are exact.

The build took 7.3 s, which is probably longer than you guessed for a 95-line file. Why does it take that long?

Answer: A1, in Answers at the end.

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 45 seconds. You should see:

```
*************************************************************************
ALU: no bug switched on, this is the golden ALU
** test_alu.t01_one_vector_per_operation           PASS          6.00          0.00
11120.56  **
** test_alu.t02_every_vector_the_alu_can_be_given   PASS     393216.00         10.52
37360.56  **
** test_alu.t03_the_undefined_opcodes              PASS          2.00          0.00
14768.68  **
** TESTS=3 PASS=3 FAIL=0 SKIP=0                              393224.00         10.53
37351.19  **
 cocotb, test_alu on Verilator, the golden ALU: build 0.1 s, run 10.8 s
*************************************************************************
ALU: bug A01 switched on
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
2797.14  **
** test_alu.t02_every_vector_the_alu_can_be_given   FAIL     393216.00         10.80
36394.25  **
** test_alu.t03_the_undefined_opcodes              PASS          2.00          0.00
15169.27  **
** TESTS=3 PASS=1 FAIL=2 SKIP=0                              393220.00         10.81
36364.68  **
 cocotb, test_alu on Verilator, A01: build 0.1 s, run 11.1 s
*************************************************************************
ALU: bug A02 switched on
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
4953.41  **
** test_alu.t02_every_vector_the_alu_can_be_given   FAIL     393216.00         10.56
37234.87  **
** test_alu.t03_the_undefined_opcodes              PASS          2.00          0.00
15224.33  **
** TESTS=3 PASS=1 FAIL=2 SKIP=0                              393222.00         10.57
37204.58  **
 cocotb, test_alu on Verilator, A02: build 0.1 s, run 10.9 s
*************************************************************************
ALU: bug A03 switched on
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
11549.25  **
** test_alu.t02_every_vector_the_alu_can_be_given   FAIL     393216.00         10.84
36288.13  **
** test_alu.t03_the_undefined_opcodes              PASS          2.00          0.00
14768.68  **
```

```
** TESTS=3 PASS=2 FAIL=1 SKIP=0                              393224.00          10.84
36259.91  **
 cocotb, test_alu on Verilator, A03: build 0.1 s, run 11.2 s
*******************************************************************************
```

You derived the counts 65,024 and 65,280 by hand in DD104 Lab 03, Part 4, and accounted for 25,726 in Part 6, so the copied ALU and its switch behave on Verilator the way DD104's three broken ALUs did on Icarus. 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: A2, in Answers at the end.

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

Part 3: The pyuvm testbench

What does the same job cost when it's built from UVM's pieces in pyuvm?

1. Build the pyuvm testbench's simulator:

   ```
   make build
   ```

   After a `BUILD` banner like Part 2's, it ends with:

   ```
   *******************************************************************************
      built   /foss/designs/modules/dd105-b/labs/lab-01-one-alu-two-frameworks/build/
   sim_build/Vtop in 1.7 s (1.5M in /foss/designs/modules/dd105-b/labs/lab-01-one-alu-
   two-frameworks/build/sim_build)
    ccache:  Hits: 6 / 14 (42.86%)  Misses: 8 / 14 (57.14%)
   *******************************************************************************
    /foss/designs/modules/dd105-b/labs/lab-01-one-alu-two-frameworks/build/sim_build/
   Vtop is up to date.
   ```

   This simulator is the ALU again, compiled into its own folder. The testbench and pyuvm itself are Python, so neither is compiled. The ccache's counts add up across builds, so subtract the cocotb build's 0 hits and 7 misses to get this build's own counts. Why did it take 1.7 s when the cocotb build of the same ALU took 7.3 s?

   Answer: A3, in Answers at the end.

   DD105-A's SystemVerilog UVM testbench for this ALU took 102 s to build with the same 2 CPUs. In that build, Verilator turned the whole UVM library into C++ along with the ALU and the testbench. It counted 24.157 MB of source in 386 modules, nearly all of it from the UVM library, and g++ spent 98.5 s of the 102 s compiling the C++ that Verilator made from it. Write your pyuvm build's time in your table.

2. Read `tb/alu_uvm.py`. Each piece is a class named after its counterpart in DD105-A's testbench, written in CamelCase as Python class names are (`alu_driver` becomes `AluDriver`). Most pieces do a job that the cocotb test also does:

pyuvm piece	Its job	In `test_alu.py`
`AluIf`	the ALU's pins, a 10 ns period and a `valid` flag	`dut` and the 1 ns `Timer` in `alu()`
`AluItem`	one vector and the ALU's answer	the `(op, a, b)` and `got` values
`AluRandSeq`, `AluDirectedSeq`	make the vectors	the loops in `t02`, the list in `t01`

pyuvm piece	Its job	In `test_alu.py`
`AluDriver`	puts each vector on the pins	the three `.value =` lines in `alu()`
`AluMonitor`	reads the pins back into an `AluItem`	the `return` line of `alu()`
`AluScoreboard`	`predict()` and the comparison	`spec()`, and the comparisons in `t01` and `t02`
`AluAgent`, `AluEnv`	create the pieces and connect them; `AluEnv` also checks that the scoreboard checked every vector the driver sent	nothing
`AluRandTest`, `AluDirectedTest`	choose the stimulus	the `@cocotb.test()` functions

The ALU is combinational, so the cocotb test sets the inputs and waits 1 ns. A driver and a monitor are separate pieces, so they need to agree on when a vector is on the pins. DD105-A added a clock and a `valid` signal to its interface for that. Here `AluIf` does the same job with a period and a `valid` flag in Python. The driver holds each vector on the pins for 10 ns and sets `valid` while it does, and the monitor reads the pins 5 ns into every period.

The tests derive from `BaseTest`, in `labs/common/py/dd105b_base.py`. pyuvm reports through Python's `logging`, and an error that a component logs with `self.logger.error()` is printed but doesn't fail the test. `BaseTest` counts every error that any component logs, prints one line with the count at the end, and fails the test if the count isn't 0. The cocotb test doesn't need it, because each of its checks raises `AssertionError` when it finds a mismatch, and that fails the test. So `cocotb/Makefile` sets `BASE_TEST := 0`, which tells the verdict not to look for `BaseTest`'s line.

3. Run the random test, which sends 2000 random vectors:

```
make run
```

You should see, among cocotb's lines:

```
 20000.00ns INFO     ..wo-frameworks/tb/alu_uvm.py(251) [uvm_test_top.env.sb]: [SB]
2000 vectors checked: 0 value mismatches
DD105-B BaseTest: 0 errors logged, no bound assertions
 20000.00ns INFO     cocotb.regression                  alu_uvm.AluRandTest passed
```

and, last:

```
**************************************************************************
 ok: 0 errors   0.54 s   log: build/runs/AluRandTest-s1/run.log
**************************************************************************
```

cocotb cuts a long file path to fit its column, so `..wo-frameworks/tb/alu_uvm.py(251)` is line 251 of `tb/alu_uvm.py`. The `[SB]` line is stamped `20000.00ns`, which is 20 µs, the time that 2000 vectors take at one every 10 ns. "No bound assertions" means that no SystemVerilog assertions are bound into this design. From Lab 05 on, the router has assertions bound into it. Write the run time in your table.

4. Run the other test, without rebuilding:

```
make run TEST=AluDirectedTest
```

It sends DD104's six `t01` vectors. You should see `[SB] 6 vectors checked: 0 value mismatches` and `ok: 0 errors   0.27 s`, and the whole command took 0.37 s when we timed it. The test is picked when the simulator starts, so trying another test, seed or bug never rebuilds anything, and every lab in the course works this way.

5. DD105-A's SystemVerilog testbench had three times as many code lines as the cocotb test. Before you count each testbench's lines, guess how the pyuvm testbench compares. Then count them:

```
make loc
```

You should see:

```
cocotb testbench:
  file                 lines    code
  cocotb/test_alu.py     189      94
pyuvm testbench:
  file            lines    code
  tb/alu_uvm.py     316     161
```

`code` leaves out blank lines and comments, and counts Python docstrings as comments. The pyuvm testbench has about 1.7 times as many code lines as the cocotb test (161 against 94), and DD105-A's had 3 times as many (282). Using the table in step 2, where do pyuvm's extra lines go, and which of the SystemVerilog testbench's lines does Python not need?

Answer: A4, in Answers at the end.

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

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

Add the six ports from `ALU` and look at the first 200 ns:



Figure 5.1: The first twenty vectors of AluRandTest on the golden ALU, in Surfer

The golden ALU. The first vector, `ALUop` 2 (`OR`) with `A=e3` and `B=6e`, gives `ef`. `Z` is high from 120 to 130 ns, where `SUB` with `A=B=b9` gives `00`.

Each vector holds the pins for 10 ns, starting at a multiple of 10 ns. The waves hold only the ALU's six ports, because `AluIf`'s period and `valid` flag exist only in Python.

Check: `2000 vectors checked: 0 value mismatches` and `ok`, and your table has a line count, a build time and a run time in the cocotb and pyuvm columns.

Part 4: Both testbenches on the bugs

Which bugs does the pyuvm testbench catch, and why?

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

```
make check-all
```

All four runs use the simulator you built in Part 3, and each takes about 0.5 s. You should see:

```
     ok  the golden design                    0 errors                    0.5s
-------------------------------------------------------------------------------
```

```
  MISSED  tier 1  A03_zero_flag_low_nibble        0 errors                    0.5s
  caught  tier 1  A01_sub_operands_swapped       291 errors                    0.5s
  caught  tier 1  A02_nand_is_nor                278 errors                    0.5s
----------------------------------------------------------------------
 2 of 3 caught.
 Still hiding from this test:
   A03_zero_flag_low_nibble
      hides behind: every check that compares results and not flags
```

`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 logs 291 errors. The first three are:

```
     95.00ns  ERROR      ..wo-frameworks/tb/alu_uvm.py(236)  [uvm_test_top.env.sb]:
[VALUE] SUB A=0c B=c1 -> ALUout=b5 N=1 Z=0, expected ALUout=4b
    135.00ns  ERROR      ..wo-frameworks/tb/alu_uvm.py(236)  [uvm_test_top.env.sb]:
[VALUE] SUB A=a4 B=e5 -> ALUout=41 N=0 Z=0, expected ALUout=bf
    195.00ns  ERROR      ..wo-frameworks/tb/alu_uvm.py(236)  [uvm_test_top.env.sb]:
[VALUE] SUB A=c7 B=52 -> ALUout=8b N=1 Z=0, expected ALUout=75
```

and near the end of the log:

```
 20000.00ns INFO     ..wo-frameworks/tb/alu_uvm.py(251) [uvm_test_top.env.sb]: [SB]
2000 vectors checked: 291 value mismatches (SUB 291)
DD105-B BaseTest: 291 errors logged, no bound assertions
```

followed by the `AssertionError` that `BaseTest` raises, which quotes the first error, and then the verdict, `caught: 291 errors`. That `AssertionError` is how the verdict knows a check fired. pyuvm has no report server, so the log has no table that counts errors by kind, like the "Report counts by id" table that DD105-A's UVM printed. The per-operation count on the `[SB]` line comes from the scoreboard's own counters.

4. The cocotb test's `t02` drove all 65,536 `SUB` vectors and found 65,024 of them wrong. The pyuvm test caught A01 on the tenth of its 2000 random vectors (`95.00ns` is 5 ns into the period that starts at 90 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: A5, in Answers at the end.

5. Write what the pyuvm 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 pyuvm 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 errors                    0.5s
  MISSED  A03_zero_flag_low_nibble             0 errors                    0.5s
----------------------------------------------------------------------
 VERDICT: FAIL -- your test passes A03_zero_flag_low_nibble too.
```

2. Extend `AluScoreboard` in `tb/alu_uvm.py`, 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 list like `n_bad_value` and `bad_value`, and set both up in `build_phase`. Log each flag mismatch with `self.logger.error()` and a message that starts with `[FLAGS]`, and add the flag count to the line that `report_phase` prints. pyuvm's messages have no IDs, so `[FLAGS]` is only text for you to read. `BaseTest` counts every error together, so the two counts on the `[SB]` line are the only place where the run keeps value and flag mismatches apart.
3. You changed one file. Write down how long you expect the rebuild to take, then run it:

   ```
   make build
   ```

   You should see:

   ```
    /foss/designs/modules/dd105-b/labs/lab-01-one-alu-two-frameworks/build/sim_build/
   Vtop is up to date.
   ```

   Why is there nothing to build?

   Answer: A6, in Answers at the end.

   Write the rebuild time in your table.
4. Ask for the verdict again:

   ```
   make check BUG=A03
   ```

   With the reference solution you should see:

   ```
        ok  the golden design                       0 errors                       0.5s
     caught  A03_zero_flag_low_nibble               125 errors                     0.5s
           AluRandTest: AssertionError: 125 errors logged, the first: [FLAGS] SHR A=80
   B=80 -> ALUout=80 N=1 Z=
   ----------------------------------------------------------------------------
    VERDICT: PASS
   ```

   If the golden row says `FAIL` instead of `ok`, your check fires on the golden ALU, so compare it with `predict()` again. If a row says `BROKEN`, your new code raised an exception, and `make run BUG=A03` shows the traceback.
5. Run all three bugs again:

   ```
   make check-all
   ```

   You should see `3 of 3 caught`. The reference solution logs one `[FLAGS]` error for each vector whose flags are wrong, and with it the counts are 582, 421 and 125 errors for A01, A02 and A03. Before your change, A01 and A02 had 291 and 278. Why is A01's count exactly double, when A02's isn't?

   Answer: A7, in Answers at the end.
6. Record the failure and open it:

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



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

A03. The vectors are the same as in Part 3's picture, because the seed is the same. `Z` *also goes high from 150 to 160 ns, where* `SHR` *of* `80` *by 0 gives* `80`*, and from 170 to 180 ns, where* `SHL` *of* `74` *by 3 gives* `a0`*.*

The first `[FLAGS]` error is stamped `155.00ns`, because the monitor read the `SHR A=80 B=80` vector 5 ns into the period that starts at 150 ns. In Part 3's picture, `Z` is high only from 120 to 130 ns, where the result really is `00`.

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

```
make run TEST=AluDirectedTest BUG=A03
```

You should see `[SB] 6 vectors checked: 0 value mismatches, 0 flag mismatches` and `MISSED: 0 errors  0.27 s`. 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_uvm.py`.

Questions:

1. Using your table, what did the UVM structure cost in this lab, in pyuvm and in DD105-A's SystemVerilog, and what did it give you that the cocotb test didn't?

   Answer: A8, in Answers at the end.

2. On A03 your scoreboard counted 125 flag mismatches in 2000 vectors, and the cocotb test counted 25,726 in 393,216. What can each number tell you?

   Answer: A9, in Answers at the end.

Submit: Nothing to submit.

Going further (optional)

Run your flag check on other seeds, each of which picks a different 2000 vectors:

```
make check-all SEED=2
```

With the reference solution, A03's count was 147 on seed 2, 143 on seed 3, 135 on seed 4 and 134 on seed 5, against 125 on seed 1. The starter's scoreboard missed A03 on all five seeds.

Next on the Course path: the reading on Python classes and cocotb concurrency and on driving and sampling on the clock, then Lab 02: A layered testbench without pyuvm.

Answers

A1. Part 2: The cocotb testbench

Verilator turns the ALU into C++, and g++ compiles that C++ into a simulator, together with Verilator's and cocotb's own support code. The ccache line counts seven compiled files. One of them, `Vtop__ALL.o`, is the ALU. The other six are the support code: `verilated.o`, `verilated_vpi.o` and `verilator.o` are three of them, and `build.log` lists them all. Part 3 shows how much of the 7.3 s those six take.

A2. 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.

A3. Part 3: The pyuvm testbench

This build looked up seven files and found six of them in the ccache, where the cocotb build had put them, so it compiled one file. With an empty ccache this build took 7.3 s too, in both of the runs we timed, so the six files from the ccache saved about 5.6 s.

A4. Part 3: The pyuvm testbench

The extra lines go 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 checked every vector the driver sent. `AluIf` carries the period and the `valid` flag, and the driver and the monitor take two classes to do what `alu()` does in five lines. The SystemVerilog testbench also spent lines on things that Python doesn't need: a `` `uvm_component_utils `` or `` `uvm_object_utils `` line and a constructor in every class, declared types and widths, an interface with two clocking blocks, and a top module. In `alu_uvm.py`, only `AluIf`, `AluItem` and `AluRandSeq` define `__init__`, and each does it only to set its own fields.

A5. 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.

A6. Part 5: Check the flags

The simulator is built from `rtl/alu.sv` alone, and `tb/alu_uvm.py` is a Python file that cocotb imports each time the simulator starts. The command took 0.07 s when we timed it. In DD105-A, the same edit to the SystemVerilog scoreboard cost a rebuild of 99 s (87 s in an earlier run). DD105-A also timed two one-line edits. Adding one member to a class rebuilt in 93 s, and a change inside a task's body rebuilt in 26 s.

A7. Part 5: Check the flags

A wrong value can bring wrong flags with it, and then the vector is logged 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 291 `[VALUE]` errors come with 291 `[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 278 wrong `NAND` values also have wrong flags. The `[SB]` line of `make run BUG=A02` counts 143 of them.

A8. Finishing

In pyuvm it cost about 1.7 times the code (161 code lines against 94). Its build cost no more than the cocotb test's, because the only thing compiled in either is the ALU (7.3 s from an empty ccache), and editing the testbench rebuilt nothing. In DD105-A's SystemVerilog it cost three times the code, a first build of about 100 s, and 26 to 99 s every time the testbench changed. Neither is faster per vector than the cocotb test. The table that cocotb prints at the end of each run puts `AluRandTest`'s 2000 vectors at 0.19 s of real time, about 0.1 ms each, and `t02`′s 393,216 at 10.64 s, about 0.03 ms each. DD105-A's 2000 vectors took 1.17 s, about 0.6 ms each. That time also covers starting the simulator, but DD105-A's directed test ran its six vectors in 0.04 s, so starting takes at most 0.04 s of it.

You saw what the structure gave you in Part 5. The flag check went into one class, and both tests used it without being edited, because the stimulus and the checking are separate pieces. In `test_alu.py`, `t01` and `t02` each do their own comparison. `AluDirectedTest` 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. On one combinational block that's a modest return. The structure is meant for designs like the router, which you'll test from Lab 02 on. It has a register bus, an input stream and two output streams.

A9. Finishing

125 of 2000 is 6.25%, 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 found no wrong values and some wrong flags, which tells you that the arithmetic is right and the bug is in the flag logic.

II 2. A layered testbench

6 **Python classes and cocotb concurrency 41**
6 Names and objects 41
6 The same object kept three times 42
6 None 43
6 Inheritance and super() 43
6 Class attributes 44
6 Tasks 44
6 Queues 46
6 Events 47
6 Stopping tasks 47
6 An exception in a task 47
6 Check yourself 48
6 Further reading 48
6 Answers 49

7 **Driving and sampling on the clock . 51**
7 Cycles and handshakes 51
7 The rule in a driver 51
7 What the pins read at each moment 52
7 A write at the edge 53
7 Reading in ReadOnly 53
7 Three monitors 53
7 The job a clocking block did 54
7 Check yourself 54
7 Further reading 54
7 Answers 54

8 **Lab 02: A layered testbench without pyuvm 57**
8 Before you start 57
8 Part 1: The starter 57
8 Part 2: The monitor 60
8 Part 3: The scoreboard 61
8 Part 4: What each catch depends on 63
8 Finishing 64
8 Going further (optional) 64
8 Answers 64

6. Python classes and cocotb concurrency

This page answers: *a testbench has to drive, watch and check all at once, and hand packets from one piece to the next. What do I need to know about Python objects and cocotb tasks before I write a monitor and a scoreboard?*

In Lab 02 you'll write the router's monitor and scoreboard. They're plain Python classes, and the testbench they go into has no pyuvm in it. The rest of that testbench comes with the lab: a generator that makes packets, a driver that offers them to the router's input, a sink for each output, and an environment that makes all of them and runs the test. When the test runs, seven pieces run at the same time and pass packets to each other through three queues. They're the generator, the driver, the two sinks, the scoreboard, and two monitors made from your class, one on the router's input and one on output 0. The first half of this page is about the objects those pieces pass around, and the second half is about running the pieces side by side as cocotb tasks.

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

Names and objects

The examples use a cut-down version of Lab 02's `Packet`, with a serial number added:

```
class Packet:
    n_made = 0                    # a class attribute: one for the class

    def __init__(self, data=None):
        self.id = Packet.n_made   # instance attributes: one set per object
        Packet.n_made += 1
        self.data = list(data or [])

    def __str__(self):
        return f"pkt {self.id}: " + " ".join(f"{b:02x}" for b in self.data)
```

Each call of `Packet(...)` makes a new object and runs `__init__` on it. Lab 02's generator makes a new `Packet` for every packet it sends, and your monitor will make one for every packet it sees on its stream. A Python variable is a name that refers to an object, and assigning one name to another gives the same object a second name. `handles_test` does that, and then makes two copies of the packet, one with `copy.copy()` and one with `copy.deepcopy()`:

```
a = None
print(f"a is {a}")
a = Packet([0x80, 0x11])
b = a
b.data.append(0x22)
c = copy.copy(a)
c.data.append(0x33)
d = copy.deepcopy(a)
d.data.append(0x44)
```

It then prints the four packets, four `is` comparisons and the count in `Packet.n_made`:

```
a is None
a: pkt 0: 80 11 22 33
b: pkt 0: 80 11 22 33
c: pkt 0: 80 11 22 33
d: pkt 0: 80 11 22 33 44
```

```
a is b: True, a is c: False, a.data is c.data: True, a.data is d.data: False,
Packet.n_made: 1
```

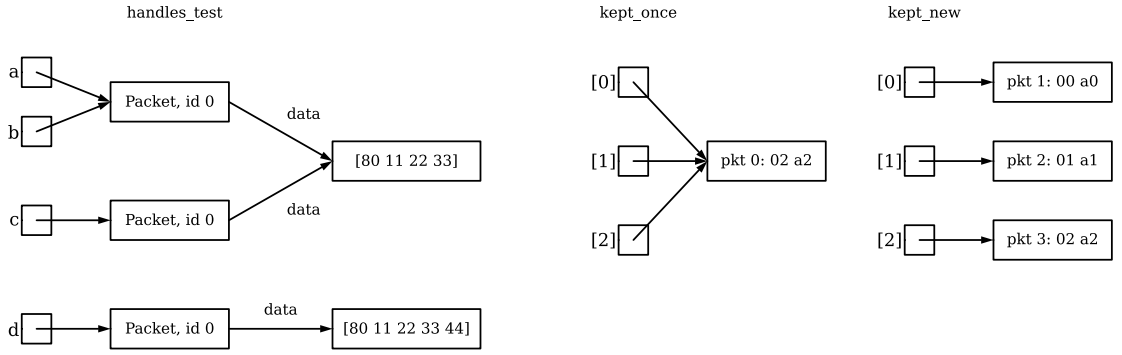


Figure 6.1: The objects that handles_test and reuse_test leave behind

Left, `handles_test`: `a` and `b` name one `Packet`; `c` is a second `Packet` that shares the first one's list; `d` is a third `Packet`, with a list of its own. Right, `reuse_test`, from the next section: `kept_once` holds three references to one reused `Packet`, and `kept_new` holds three `Packet`s. The values are the ones the tests printed.

`b = a` didn't make a packet, so the `22` appended through `b` shows up when you print `a`. `a is b` is `True`, because `is` asks whether two names refer to the same object. Two packets made separately are two objects even when their beats match, so `is` can't tell you whether two packets carry the same data. That's why Lab 02's `Packet` has a `same()` method, which compares the beats.

`copy.copy(a)` made a *shallow* copy. It made a new `Packet` and copied each of `a`'s attributes into it, but where an attribute refers to a list, it copied the reference and not the list. So `c.data` is `a.data`, and the `33` appended through `c` changed `a` too. `copy.deepcopy(a)` made a *deep* copy, which copies the list as well, so the `44` went into `d` alone. Neither copy ran `__init__`, so all three packets have `id` 0 and `Packet.n_made` is still 1.

You won't often copy packets in the labs, but you'll pass references to them all the time. In Lab 02, the `Packet` that your input monitor builds is the same object that the scoreboard takes out of the queue and, if it's predicted for output 0, keeps in its list of expected packets. A change made through any of those references shows through all of them.

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 packets, two bytes each, and keeps them in a list, as Lab 02's scoreboard keeps the packets it expects. The first loop makes one `Packet` before it starts and reuses it for every packet, and the second makes a new `Packet` for each one:

```
p = Packet()
for k in range(3):
    p.data.clear()
    p.data.append(k)
    p.data.append(0xA0 + k)
    kept_once.append(p)

for k in range(3):
    p = Packet()
    p.data.append(k)
    p.data.append(0xA0 + k)
    kept_new.append(p)
```

```
one Packet, reused: [0] pkt 0: 02 a2
one Packet, reused: [1] pkt 0: 02 a2
```

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

`kept_once` holds three references to one object. That object holds the beats of the last packet, `02 a2`, so the list shows them three times, and the first two packets' beats are gone, cleared by `p.data.clear()` when the loop reused the object. A queue behaves the same way, because what you put into it is a reference. That's why the `TODO` in Lab 02's `monitor.py` has you start a new packet when `p` is `None`, and set `p = None` once you've put a finished packet into the queue. The next packet then gets an object of its own, and the object that the scoreboard was given keeps the beats it had.

None

If you use an attribute of a name that still refers to `None`, Python raises an `AttributeError`, and the test stops there. `none_test` sets `p = None`, prints a line, and then appends a beat to `p.data`:

```
appending a beat to p
      0.00ns WARNING  cocotb.Test none_test.none_test     'NoneType' object has no
attribute 'data'
                                                             Traceback (most recent
call last):
                                                            File "/w/guide-examples/
python-classes-and-cocotb-concurrency/classes_demo.py", line 78, in none_test
                                                          p.data.append(0x80)
                                                          ^^^^^^
                                                        AttributeError: 'NoneType'
object has no attribute 'data'
```

and the course's verdict on the run is

```
 BROKEN: AttributeError   0.23 s   log: build/runs/none_test-s1/run.log
```

The traceback names the file and the line where the `None` was used. The graders mark a run like this `BROKEN`, because the exception is a mistake in the testbench, and a run counts as catching a bug only when a check fires, such as a failed `assert`. In Lab 02 you'll see this error if your monitor appends a beat to `p.data` before it has made `p`.

Inheritance and super()

A class can derive from another class, which is then its base class. It inherits the base class's methods, `__init__` among them, and it can override any of them with a version of its own. In Python every method call is looked up on the object's own class when the call happens, so when a method written in the base class calls another method, the call goes to the derived class's version if there is one. In `dispatch_test`, `Component` has a `report()` method that calls `self.kind()`, and `Scoreboard` derives from `Component` and overrides `kind()`:

```
class Component:
    def __init__(self, name):
        self.name = name
        self.n_errors = 0

    def kind(self):
        return "a Component"

    def report(self):
        print(f"report() of Component, on {self.name}: {self.kind()}, {self.n_errors}
errors")
```

```
class Scoreboard(Component):
    def __init__(self, name):
        super().__init__(name)
        self.expected = []

    def kind(self):
        return "a Scoreboard"
```

`Component("c").report()` and `Scoreboard("sb").report()` both ran `Component`'s `report()`, and the call to `self.kind()` inside it went to each object's own class:

```
report() of Component, on c: a Component, 0 errors
report() of Component, on sb: a Scoreboard, 0 errors
```

pyuvm relies on the same lookup. It calls `build_phase()` and `run_phase()` on every component, and when the component is Lab 01's `AluDriver`, those calls run the versions written in `AluDriver`.

`Scoreboard.__init__` starts with `super().__init__(name)`, which runs `Component`'s `__init__` and sets `name` and `n_errors`. If you leave that line out, Python raises nothing when the object is made. The error comes later, when something first uses an attribute that `Component.__init__` would have set. `super_test` makes a `ForgetfulScoreboard`. It derives from `Component`, as `Scoreboard` does, and its `__init__` is `Scoreboard`'s without the `super()` line. The test prints the object's attributes and then calls `report()`:

```
made a ForgetfulScoreboard; its attributes: ['expected']
     0.00ns WARNING  cocotb.Test super_test.super_test  'ForgetfulScoreboard' object
has no attribute 'name'
```

The traceback in the log points at the `print` in `report()`, not at the constructor that left the attribute out. The same rule applies in pyuvm to any method you override when the base class's version does work of its own. `RouterBaseTest.build_phase()` puts the router in `ConfigDB`, and `BaseTest.final_phase()` prints the closing line that the graders look for and fails the test if any component logged an error, so a test that overrides either one has to call the base version through `super()`.

Class attributes

An attribute assigned in the class body belongs to the class, so every object shares it. `Packet.n_made` is a class attribute on purpose, because it counts the packets made from the class. A list made in the class body is shared the same way, which is almost never what you meant. `class_attr_test` uses two classes. `Sink` has `beats = []` in its class body, and `OwnSink` sets `self.beats = []` in `__init__`. The test makes two sinks, `out0` and `out1`, from each class, appends `0x11` to `out0`'s list and `0x22` to `out1`'s, and prints both lists:

```
Sink: out0 took ['0x11', '0x22'], out1 took ['0x11', '0x22']
OwnSink: out0 took ['0x11'], out1 took ['0x22']
```

Lab 02's classes make every list and counter in `__init__`, and yours should too.

Tasks

A coroutine is an `async def` function, and `await coro()` runs it inside your own task until it returns. To run a coroutine alongside your own code instead, you pass `coro()` to `cocotb.start_soon()`, which wraps it in a **task** and returns the `Task` object. `start_soon()` returns straight away, and the task first runs when the code that started it reaches its next `await`. You can wait for a task later, or never.

`start_test` starts three tasks, A, B and C, which finish 10, 20 and 30 ns after they start, and it does this three times. The first time it waits for all three with `await Combine(a, b, c)`, the second time it waits for the first one to finish with `await First(a, b, c)`, and the third time it waits for none of them:

```
  0 ns: start A, B and C, and await Combine(a, b, c)
 10 ns:   A finished
 20 ns:   B finished
 30 ns:   C finished
 30 ns: the test goes on after Combine
 30 ns: start A, B and C, and await First(a, b, c)
```

```
40 ns:   A finished
40 ns: the test goes on after First
50 ns:   B finished
60 ns:   C finished
60 ns: Combine returns
60 ns: start A, B and C, and await none of them
60 ns: the test goes on at once
70 ns:   A finished
80 ns:   B finished
90 ns:   C finished
```

- `Combine` waits for every task, so the test went on at 30 ns, when the slowest one finished.
- `First` waits for the first task to finish, so the test went on at 40 ns, when A finished. `First` leaves the other two running, and B and C finished at 50 and 60 ns. The test then waited for them with `Combine` before it went on.
- When the test waited for none of them, it went on at once, at 60 ns, and the three tasks ran beside it.

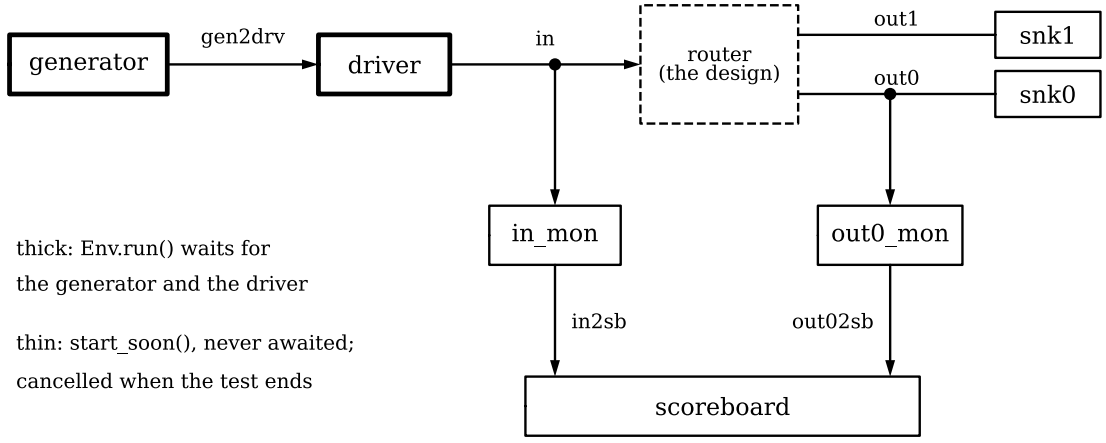


Figure 6.2: Lab 02′s environment as tasks and queues

From Lab 02′s `env.py`. Each solid box is an object whose `run()` method runs as a task, except the driver, whose `run()` runs inside the test's own task. The dashed box is the router. The arrows labelled `gen2drv`, `in2sb` and `out02sb` are the three queues, and `in`, `out0` and `out1` are the router's streams.

Lab 02′s `Env.run()` starts the two sinks, the two monitors and the scoreboard with `cocotb.start_soon()` and never waits for them, because none of their `run()` methods ever returns. Each sink, and each monitor once you've written it, runs a `while True` loop, and the scoreboard's `run()` waits for two such loops, which it starts as two more tasks. `Env.run()` starts the generator with `cocotb.start_soon()` too, and then runs the driver itself, with `await self.drv.run(self.cfg.n)`, so the driver and the generator work at the same time. When the driver has sent every packet, `await gen` waits for the generator's task, which by then has finished. Then the environment waits 50 cycles for the outputs to drain and calls the scoreboard's `report()`, while the sinks, the monitors and the scoreboard are still running.

Tasks started in a loop

`capture_test` starts three tasks in a loop, and then three more in a second loop. In the first loop each task is a coroutine defined inside the loop, which reads the loop variable `i`. In the second, each task is a coroutine that is given `i` as an argument:

```
for i in range(3):
    async def show():
        say(f"  i = {i}")
    cocotb.start_soon(show())
```

```
...
for i in range(3):
    cocotb.start_soon(show_arg(i))
```

```
  0 ns: three tasks, each a closure that reads i
  0 ns: the loop has finished
  0 ns:   i = 2
  0 ns:   i = 2
  0 ns:   i = 2
  5 ns: three tasks, each given i as an argument
  5 ns: the loop has finished
  5 ns:   j = 0
  5 ns:   j = 1
  5 ns:   j = 2
```

None of the six tasks ran before its loop had finished. The first three read `i` when they ran, by which time the loop had left it at 2. A function defined inside another one reads the outer function's variables when it runs, not when it's defined. The second three were each given the value of `i` at the moment `show_arg(i)` was called, so they printed 0, 1 and 2. Give a task what it needs as arguments. Lab 02′s environment starts its tasks in a loop too, with `cocotb.start_soon(piece.run())`, and that's safe, because `piece.run()` is called inside the loop, while `piece` still names that pass's object, and the coroutine it returns keeps that object as its `self`.

Queues

A `cocotb.queue.Queue` carries values from tasks that `put()` them to tasks that `get()` them, in the order they were put. `await q.get()` waits while the queue is empty, and if you gave the queue a `maxsize`, `await q.put()` waits while it's full. `queue_test` has a generator that makes four items as fast as it can, and a driver that spends 10 ns on each item. It runs them twice, first with an unbounded queue between them and then with a queue that has room for one item:

```
  0 ns: Queue(maxsize=0)
  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
 40 ns: Queue(maxsize=1)
 40 ns: gen put 0
 40 ns:   drv got 0
 40 ns: gen put 1
 50 ns:   drv got 1
 50 ns: gen put 2
 60 ns:   drv got 2
 60 ns: gen put 3
 60 ns: gen has made all four
 70 ns:   drv got 3
```

`maxsize=0`, the default, means no limit. With it, the generator made all four items at once, 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 20 ns after it started. The driver got the same items, 10 ns apart, both ways.

Lab 02′s queues are all unbounded (`Queue()`), so its generator makes every packet at the start and the driver works through them. The monitors' queues are the other way round, with the reader waiting for the writer. A monitor puts a packet into its queue each time one finishes on its stream, and the scoreboard's `get()` waits until the next one arrives. The scoreboard reads two of these queues, `from_in` and `from_out0`,

and each fills at its own pace. A task waiting in `from_out0.get()` can't take a packet from `from_in` at the same time, so the scoreboard's `run()`, which you write in Part 3, starts two tasks of its own. One predicts each input packet as it arrives, and the other checks each packet from output 0 against the predictions made so far.

Events

A `cocotb.triggers.Event` lets one task tell others that something has happened. `e.set()` sets it, and every task waiting in `await e.wait()` goes on. The event stays set until something calls `e.clear()`, so a task that calls `e.wait()` after the `set()` goes straight on. `event_test` starts a waiter, sets the event at 10 ns and starts a second waiter at the same time, then clears the event at 20 ns and starts a third:

```
 0 ns:   first: waiting
10 ns: done.set()
10 ns:   first: done.wait() returned
10 ns:   second: waiting
10 ns:   second: done.wait() returned
20 ns: done.clear()
20 ns:   third: waiting
30 ns: end of the test
```

The third waiter never returned, because nothing set the event again. Lab 02 has no events. You'll meet one in Lab 04, whose scoreboard sets an `Event` called `stop` at its 50th error. Lab 04's tests wait on that event, so the run ends there. pyuvm also uses an event to end the run phase, and sets it when the last objection is dropped.

Stopping tasks

`t.cancel()` stops a task. cocotb raises `CancelledError` inside the task, at the `await` where it's waiting, and unless the task catches the exception, the task ends there. `cancel_test` starts a monitor task that counts a tick every 10 ns, and a parent task that counts its own ticks and has started a child task that counts too. At 25 ns the test prints the three counts and cancels the monitor and the parent, and 50 ns later it prints the counts again:

```
25 ns: cancel the monitor and the parent: {'mon': 2, 'parent': 2, 'child': 2}
75 ns: 50 ns later: {'mon': 2, 'parent': 2, 'child': 7}
75 ns: mon.cancelled() is True, par.cancelled() is True
```

The monitor and the parent stopped at 2, and the child went on counting. cocotb doesn't keep track of which task started which, so cancelling a task leaves the tasks it started running. To stop those too, keep their `Task` objects and cancel each one.

You don't have to stop the tasks that run for the whole test, because when a test ends, cocotb cancels every task that is still running. `end_test` starts a monitor that catches `CancelledError`, prints a line and raises the exception again, and the test returns 35 ns later:

```
35 ns: the test returns
35 ns:   the monitor was cancelled
```

Lab 02's sinks, monitors and scoreboard are stopped this way. They're still running when `Env.run()` returns and the test makes its `assert`, and cocotb cancels them when the test function returns.

An exception in a task

A mistake in a task, such as a misspelt name, raises an exception in that task, and what happens next depends on whether anything is waiting for the task. The example's `broken_monitor()` waits 20 ns and then appends to a list called `pkts`, which doesn't exist. Four tests start it as a task:

Test	What it does after starting the task	Verdict
`unwatched_test`	waits 100 ns and never looks at the task	`BROKEN: NameError`, at 20 ns
`joined_test`	`await t`	`BROKEN: NameError`, at 20 ns
`first_test`	`await First(t.complete, Timer(100, unit="ns"))`	`ok: passed`, at 20 ns
`first_result_test`	the same, then `t.result()`	`BROKEN: NameError`, at 20 ns

When nothing waits for a task, cocotb fails the test with the task's exception as soon as it happens:

```
     20.00ns WARNING  cocotb.Task 2.broken_monitor       name 'pkts' is not defined
```

`await t` raises the task's exception in the task that waits for it, so `joined_test` failed the same way. `first_test` passed. While `First` waits on the task's `complete` trigger, cocotb counts the task as watched and leaves its exception to the code that is waiting. `First` returns once the task has finished, and nothing raises the exception again:

```
  20 ns: First returned; t.done() is True
  20 ns: the test reached its end
```

The test passed even though its monitor had crashed at 20 ns. A monitor that has crashed sends nothing more to the scoreboard, so a test like this can pass on a broken design, unless the scoreboard checks at the end that every packet it expected arrived, as Lab 02's `report()` does.

If you ever wait on `t.complete`, call `t.result()` once the task is done. `t.result()` raises the task's exception if it had one, as `first_result_test` shows. Lab 04's tests all derive from `RouterLabTest`, which does this. Its `run_phase()` waits with `First` for its stimulus task, a watchdog task or the scoreboard's `stop` event. Then, for each of the two tasks, it calls `result()` if the task has finished and `cancel()` if it hasn't. Lab 02's environment waits for only one of its tasks, with `await gen`, and starts the rest without waiting for them, so an exception in your monitor or your scoreboard fails the test as soon as it happens. The verdict is `BROKEN`, and the traceback is in the log.

Check yourself

1. Your scoreboard reports that every packet it expected has the same beats as the last packet that the input monitor saw. The monitor does `p = Packet()` once, in its `__init__`. What's wrong, and how do you fix it?

   Answer: A1, in Answers at the end.

2. A test starts eight tasks in a loop, `for k in range(8)`. Each task is a coroutine, defined inside the loop, that logs `k`. Every task logs 7. Why, and what's the fix?

   Answer: A2, in Answers at the end.

3. What would happen if `Env.run()` started the scoreboard with `await self.sb.run()` instead of `cocotb.start_soon(self.sb.run())`?

   Answer: A3, in Answers at the end.

4. Your scoreboard's output loop has a typo that raises `NameError` at the first packet from output 0. What does the verdict say in Lab 02, and what would it say if `Env.run()` had waited for the scoreboard's task with `First(task.complete, ...)`?

   Answer: A4, in Answers at the end.

Further reading

- The Python tutorial's chapter on classes. Its section on class and instance variables shows the shared-list mistake that `class_attr_test` makes.
- The `copy` module: shallow and deep copies, and how a class can define its own.

- The Python FAQ's answer to why lambdas defined in a loop all return the same result, which is `capture_test`'s first loop with a `lambda` in place of a coroutine.
- cocotb 2.0.1's Coroutines and Tasks: `start_soon`, `First` and `Combine`, and waiting for several things at once. It says that `First` doesn't cancel the tasks that hadn't finished, as `start_test` shows. It doesn't say what happens to tasks still running when a test ends, or to an exception in a task that nothing waits for; `end_test` and the four exception tests on this page show what cocotb 2.0.1 does.

Next: Driving and sampling on the clock.

Answers

A1. Check yourself

The monitor puts the same object into the queue for every packet and refills it each time, so every reference that the scoreboard kept refers to 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 queue, which is what setting `p = None` after the `put()` does.

A2. Check yourself

Each coroutine reads `k` when it runs, and none of the tasks runs until the test reaches its next `await`. By then the loop has finished and left `k` at 7. Pass `k` to the coroutine as an argument, as `capture_test`'s second loop passes `i` to `show_arg()`, so that each task gets the value that `k` had when the task was started.

A3. Check yourself

The scoreboard's `run()`, once you've written it in Part 3, waits for its two loops, and they never return, so `Env.run()` would wait there for ever. It would never start the generator or the driver, so no packet would be sent, and the run would go on until `make` stopped it after `RUN_TIMEOUT`, 300 s by default, and graded it `BROKEN`.

A4. Check yourself

In Lab 02 the exception fails the test as soon as it happens, and the verdict is `BROKEN: NameError`, with the traceback in the log. With `First(task.complete, ...)` on the scoreboard's task, the verdict is the same if your `run()` awaits the input loop's task before the output loop's. The typo raises in the output loop, which is a separate task, and nothing is waiting on that task when it raises.

If the exception does reach the scoreboard's own task, for example because `run()` waits for both loops with `Combine`, `First` counts that task as watched and the exception is dropped. The test then runs to the end, where `report()` finds every packet predicted for output 0 still expected and no packet checked. Those errors fail the test with an `AssertionError`, which the verdict counts as a check that fired, so the golden router's run says `FAIL` instead of `BROKEN`. Calling `task.result()` after `First` returns would raise the `NameError` and make the verdict `BROKEN` again.

7. Driving and sampling on the clock

This page answers: *the router's flip-flops change at every rising edge. When should a testbench change the router's inputs, and when should it read the pins, so that every value it reads is the value that the specification gives for that cycle?*

Lab 01′s ALU has no clock, so its driver and monitor kept time with a 10 ns period that they shared through `AluIf`. The router has a clock, and its specification states every rule cycle by cycle. In Lab 02, the driver, the sinks and the APB master that the lab ships all change the router's inputs right after a rising edge, and every class that reads the pins, including the monitor you'll write, reads them at the falling edge in the middle of the cycle. Lab 02′s code calls this the timing rule. In DD105-A the same timing came from SystemVerilog clocking blocks. This course has no clocking blocks, so each class keeps the rule in its own code. This page first shows the rule at work on the router. Then it compares what the same pins read at two other moments that a testbench could choose: as soon as the rising edge comes, and in cocotb's `ReadOnly` phase after it.

The examples on this page drive the golden router from `labs/common/rtl`, and we ran them in the course image. Their source is in `guide-examples/driving-and-sampling-on-the-clock/`, and the section for this page in `guide-examples/README.md` gives the command behind each output.

Cycles and handshakes

The specification's Cycles and handshakes section numbers the clock periods: cycle n is the 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, and most of its outputs come straight from flip-flops, so they change just after a rising edge. A **handshake** on a stream is a cycle in which `valid` and `ready` are both 1, and the beat moves at the rising edge that ends that cycle.

So if a testbench changes an input right after rising edge n, the input has its new value in cycle $n + 1$, and the router acts on it at the edge that ends that cycle. At the falling edge in the middle of a cycle no signal changes, because the router's outputs and the testbench's inputs both changed just after the rising edge before it. Every signal then holds the value it has for the whole cycle, which is the value the specification calls its value in that cycle.

The rule in a driver

The example's `send_beat()` offers one beat on the router's input the way Lab 02′s `Driver` does. It's called right after a rising edge:

```
async def send_beat(dut, data, last):
    dut.in_valid.value = 1
    dut.in_data.value = data
    dut.in_last.value = int(last)
    while True:
        await FallingEdge(dut.clk)
        if dut.in_ready.value:
            break
        await RisingEdge(dut.clk)
    await RisingEdge(dut.clk)                    # the beat moves at this edge
```

The three writes give the beat's signals their values in the cycle that has just started. At the falling edge in the middle of that cycle the loop looks at `in_ready`. If it's 1, `in_valid` is 1 in the same cycle, so this cycle is the handshake, and the beat moves at the rising edge that ends it. `send_beat()` returns right after that edge, so the caller can offer the next beat in the cycle that edge starts. If `in_ready` is 0, the loop waits for the rising edge with the beat still on offer, and looks again at the next falling edge. The beat is driven once, before the loop, and stays on the pins while the loop runs. Each pass of the loop takes one clock

cycle, from just after one rising edge to just after the next. The example's `apb_write()` is a copy of Lab 02's `ApbMaster.write()` and works the same way. It reads `pready` at the falling edge of each access cycle.

What the pins read at each moment

`edge_test` holds the router in reset through the rising edges at 0, 10 and 20 ns, as Lab 02's test does. Then it writes `ROUTE0` (enabled, `PORT` 0) and `CTRL.EN` with `apb_write()`, and offers a two-beat packet, `00` and then `5a`, to `DST` 0 with `send_beat()`. While it does, a watcher task reads `in_valid`, `in_ready` and `out0_valid` three times around every rising edge: at the falling edge before it, as soon as the rising edge comes (`await RisingEdge(dut.clk)` and then a read), and in cocotb's `ReadOnly` phase after it. `ReadOnly` is cocotb's name for the end of a time step, after the simulator has applied everything for it, so every value has settled. The watcher's first line names the three columns, and then it prints the three readings for each edge from 70 ns to 120 ns, in that order. The lines in between are the test's own, printed when the `CTRL.EN` write finished and as each beat moved:

```
  20 ns: (in_valid in_ready out0_valid)
  70 ns: falling edge before: 0 0 0 | at the edge: 0 0 0 | ReadOnly: 0 0 0
  80 ns: CTRL.EN written
  80 ns: falling edge before: 0 0 0 | at the edge: 0 0 0 | ReadOnly: 1 1 0
  90 ns: beat 00 moved
  90 ns: falling edge before: 1 1 0 | at the edge: 1 1 0 | ReadOnly: 1 1 0
 100 ns: beat 5a moved
 100 ns: falling edge before: 1 1 0 | at the edge: 1 1 0 | ReadOnly: 0 1 1
 110 ns: falling edge before: 0 1 1 | at the edge: 0 1 1 | ReadOnly: 0 1 1
 120 ns: falling edge before: 0 1 1 | at the edge: 0 1 1 | ReadOnly: 0 1 0
```

Here is the same run in Surfer, from 60 ns to 130 ns:



Figure 7.1: The router's APB write of CTRL.EN and a two-beat packet, driven right after rising edges, in Surfer

The `CTRL` write's setup cycle runs from 60 to 70 ns and its access cycle from 70 to 80 ns, and `in_ready` rises at the 80 ns edge, when the write takes effect. `in_valid` also rises at 80 ns, because the test drove it right after that edge. The router takes `00` at the 90 ns edge and `5a` at the 100 ns edge, and `out0_valid` is high from 100 to 120 ns.

Each beat came out on output 0 two cycles after its handshake on the input, as rule O8 of the specification gives for a sink that takes every beat as soon as it appears. `00`'s handshake was the cycle from 80 to 90 ns, and `out0_valid` is 1 with `00` on `out0_data` in the cycle from 100 to 110 ns.

Here is the cycle that each of the three readings belongs to:

- **The falling edge before edge *n*** reads the values in cycle *n*, which is the cycle that edge *n* ends. At 85 ns, the falling edge before the 90 ns edge, the watcher read `in_valid` and `in_ready` both 1. So the cycle from 80 to 90 ns is `00`'s handshake, and the beat moved at 90 ns, as the test printed.
- **At the edge**, on Verilator 5.050 in the course image, Python got the values from before the edge, the same ones that the falling edge before it read. At 80 ns it read `in_ready` as 0.
- **In `ReadOnly` after edge *n*** the watcher read the values in cycle $n + 1$, the cycle that edge *n* starts. At 80 ns it read `in_ready` as 1, and `in_valid` as 1 too, because the test's write right after that edge had already been applied.

So at 80 ns `in_ready` read 0 or 1 depending on when, within that one time step, the read was made. The waveform shows `in_ready` rising at 80 ns and can't tell you which of the two a read would get, because Surfer draws a change made at the edge and a change made just after it at the same instant.

A write at the edge

A value you write at an edge isn't there yet for another task that reads as soon as the same edge comes. `same_edge_test` starts three tasks that wait for the same rising edge: a reader, a writer that sets `in_data` to `aa`, and a second reader, in that order. The second reader runs after the write and still reads the old value, and `aa` first shows up in the `ReadOnly` reading at the end of that time step:

```
   55 ns: at the falling edge: in_data 00, in_ready 1
   60 ns: reader 1: in_data reads 00, in_ready reads 1
   60 ns: writer: in_data = aa
   60 ns: reader 2: in_data reads 00, in_ready reads 1
   60 ns: ReadOnly: in_data aa, in_ready 1
```

cocotb's documentation says that writes "are not applied immediately, but delayed until the next write cycle". The router doesn't see the write at that edge either, so a value written right after rising edge n is the input's value in cycle $n + 1$. In `edge_test`, `send_beat()` wrote `5a` to `in_data` right after the 90 ns edge, when `in_ready` was already 1, and the router took `00` at that edge and `5a` at the next one, at 100 ns.

Reading in ReadOnly

DD104's *Clocks, resets and triggers* page listed `ReadOnly` as one way to read settled values after an edge, and in every row of `edge_test` it read the next cycle's values. In a testbench that also drives the pins, it has two drawbacks. The first is that you can't write in it. `readonly_test` waits for a rising edge, waits for `ReadOnly`, and writes `in_valid`:

```
   30 ns: in the ReadOnly phase after the rising edge; writing in_valid
      30.00ns WARNING  ..Test readonly_test.readonly_test Attempting settings a value
during the ReadOnly phase.
```

The test stops there with a `RuntimeError`, and `make run` reports `BROKEN: RuntimeError`. A loop that reads in `ReadOnly` has to wait for another trigger before it can drive anything.

The second drawback is that a `ReadOnly` reading is stamped with the time of the edge that starts the cycle it describes. A log line at 80 ns describes the cycle from 80 to 90 ns, while a falling-edge reading of the same cycle is stamped 85 ns.

Three monitors

`monitor_test` sends 73 beats in 20 random packets through the router, with an idle gap of 1 to 3 cycles before about one beat in four, as Lab 02's generator does. It runs three copies of an input monitor, which are the same code except for when they read the pins: at the falling edge, as soon as the rising edge comes, or in `ReadOnly` after the rising edge. Each one records the beats of every handshake, and the time at which it saw each packet's last beat:

```
1260 ns: the driver sent 73 beats in 20 packets; the first: 00 75 96 ab
1260 ns:  falling monitor: 73 beats in 20 packets, 20 the same as sent; the first: 00
75 96 ab at 215 ns
1260 ns:   rising monitor: 73 beats in 20 packets, 20 the same as sent; the first: 00
75 96 ab at 220 ns
1260 ns: readonly monitor: 73 beats in 20 packets, 20 the same as sent; the first: 00
75 96 ab at 210 ns
```

In this run, on Verilator, all three got every beat right. Each one reads all four of the stream's signals at the same moment, so the four values it gets always come from the same cycle. They stamp the first packet's last beat at three different times, all for the cycle from 210 to 220 ns: the falling-edge monitor at 215 ns, in the middle of that cycle; the rising-edge monitor at 220 ns, the edge that ends it; and the `ReadOnly` monitor at 210 ns, the edge that starts it.

One piece of code that reads at two of these moments can get its readings from two cycles. In `edge_test`'s row for 80 ns, the falling-edge reading `0 0 0` and the `ReadOnly` reading `1 1 0` come from one pass of the watcher's loop, 5 ns apart, and they belong to the cycles before and after the 80 ns edge. A tracker that reads the input's handshakes at one moment and an output's at another puts one of them in the wrong cycle. Lab 03's tracker works out, cycle by cycle, how many beats are waiting in output 0 from

the handshakes on the input and on output 0. So it reads both streams at the same falling edge, and within each cycle it handles the input's handshake before the output's. Every class in the course's testbenches reads the router's outputs at the falling edge, and every monitor reads all of its signals there. That is also why the times that Lab 02's monitors log all end in 5, such as `265.00ns` for the first packet into the router.

The job a clocking block did

In DD105-A, each class reached the pins through an interface with a clocking block, which sampled its inputs just before each rising edge and drove its outputs 1 ns after it. The rule on this page does the same job. Reading at the falling edge gets the cycle's value half a period before the edge, where the clocking block got it just before the edge. Writing right after the rising edge gives the router a value that it first acts on at the next edge, as the 1 ns output skew did, though in the waves the change is drawn at the edge itself.

Plain Python does the interface's other job, which was to give a class a handle to one stream's signals. Lab 02's `Monitor` is given the prefix of a stream's signal names, such as `in` or `out0`, and looks up each of its four signals with `getattr()`, as in `getattr(dut, f"{prefix}_valid")`, so one class watches either stream. It never writes to the signals it holds, though Python would let it. In DD105-A a monitor's interface made every signal an input, so a monitor couldn't drive the stream it watched.

Check yourself

1. A driver writes `in_valid = 1` right after the rising edge at 80 ns, and `in_ready` is 1 from then on. At which falling edge does a monitor see the handshake, and at which edge does the beat move?

   Answer: A1, in Answers at the end.

2. In the waveform, `in_valid` and `in_ready` both go to 1 at 80 ns, the same instant as a rising edge. Did the router take a beat at the 80 ns edge?

   Answer: A2, in Answers at the end.

3. Your monitor reads `valid` and `ready` at the falling edge, but reads `data` after `await RisingEdge(dut.clk)` and `await ReadOnly()`. What can go wrong?

   Answer: A3, in Answers at the end.

Further reading

- The router specification's Cycles and handshakes and Valid/ready rules, which state every rule in the cycle numbering this page uses.
- DD104's Clocks, resets and triggers, "Reading at the clock edge". It reads a counter's output at the rising edge on Icarus 14.0 and gets the old value, as this page did on Verilator, and it lists the falling edge as the best of three ways to read a settled value.
- cocotb 2.0.1's Writing Testbenches, "Assigning values to signals", which says that writes are delayed, as `same_edge_test` shows.
- ChipVerify's Clocking Blocks and Clocking Blocks Part 2: what DD105-A's clocking blocks do, with input and output skews and `#1step`. They are SystemVerilog, and nothing in this course uses one.

Next: Lab 02: A layered testbench without pyuvm.

Answers

A1. Check yourself

The monitor sees `in_valid` and `in_ready` both 1 at the falling edge at 85 ns, so the cycle from 80 to 90 ns is the handshake, and the beat moves at the 90 ns edge. `edge_test` printed `beat 00 moved` at 90 ns.

A2. Check yourself

No. Both changed at that edge, so in the cycle that the 80 ns edge ends they were both 0, as the falling edge at 75 ns read them. The first cycle with both at 1 is the one from 80 to 90 ns, and the router took `00` at 90 ns. A waveform draws the change at the same instant as the edge, so read a cycle's values in the middle of the cycle, where nothing changes.

A3. Check yourself

The data it reads belongs to the next cycle, so whenever the driver puts a new beat on the pins right after the edge, the monitor pairs this cycle's handshake with the next beat's data. Read every signal of a stream at the same falling edge.

8. Lab 02: A layered testbench without pyuvm

You'll finish a testbench for the router. Its pieces are plain Python classes on cocotb, with no pyuvm. You'll 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, B05 and B09, and rerun it with different plusargs to find out what each catch depends on.

Package: `labs/lab-02-a-layered-testbench-without-pyuvm/`

Before you start

- Finish Lab 01. This lab uses the same `make` targets, the same terminal and the same course folder.
- Read Python classes and cocotb concurrency and Driving and sampling on the clock.
- Read The router in one page, and keep the specification open. You'll use its Cycles and handshakes, Packets, Routing and The outputs sections, and the stream rules under Valid/ready rules.

Time: about 1 h 30 min.

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-b
cd labs/lab-02-a-layered-testbench-without-pyuvm
```

2. Build the simulator:

```
make build
```

You should see:

```
python3 ../common/inject.py --all --out build/rtl
faulty build with 15 bugs -> build/rtl/router.f
==========================================================================
 BUILD  router   (Verilator, 2 jobs)
  Full  log:  /foss/designs/modules/dd105-b/labs/lab-02-a-layered-testbench-without-
pyuvm/build/build.log
==========================================================================
==========================================================================
 built /foss/designs/modules/dd105-b/labs/lab-02-a-layered-testbench-without-pyuvm/
build/sim_build/Vtop in 2.2 s (1.9M in /foss/designs/modules/dd105-b/labs/lab-02-a-
layered-testbench-without-pyuvm/build/sim_build)
 ccache:  Hits: 11 / 21 (52.38%)  Misses: 10 / 21 (47.62%)
==========================================================================
       /foss/designs/modules/dd105-b/labs/lab-02-a-layered-testbench-without-pyuvm/
build/sim_build/Vtop is up to date.
```

In the first two lines, `inject.py` writes a copy of the router's RTL with all fifteen planted bugs in it, each behind the `+BUG` switch, as Lab 01's ALU had its three bugs. So the simulator you just built is the one that every run in this lab uses, whichever bug you switch on.

The ccache's counts add up over every build in your clone. Before this build, our counts covered only Lab 01's two builds, which came to 6 hits and 8 misses, so this build's own counts were 5 hits and 2 misses. The two files that this build compiled are `Vtop__ALL.o`, the router, and `verilator.o`, cocotb's `main()` for the simulator, which includes the header that Verilator writes for whichever design it built. The other five are Verilator's support code, which Lab 01's builds had already put in the ccache. With an empty ccache, the build took us 7.6 to 7.7 s.

3. Read the files in `tb/`. The generator hands its packets to the driver through a queue, and each monitor hands the packets it sees to the scoreboard through a queue of its own. Each queue is a `cocotb.queue.Queue`. One end calls `await q.put(p)`, and the other calls `await q.get()`, which waits until there's a packet to take.

File	Class	Its job
`packet.py`	`Packet`	one packet: its beats, header first
	`RouterCfg`	the test's knobs, read from plusargs
`generator.py`	`Generator`	makes the packets and puts them in a queue
`driver.py`	`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
	`ApbMaster`	writes the routing table and `CTRL.EN`
`monitor.py`	`Monitor`	yours: turns a stream's handshakes into packets
`scoreboard.py`	`Scoreboard`	yours: predicts output 0 and checks it
`env.py`	`Env`	makes the pieces from `Generator` to `Scoreboard`, connects them with queues, and runs the test
`test_router.py`	`router_class_test`	the clock, the reset, the `Env`, and the assert at the end, which fails the test if the testbench reported errors

Read `Env.run()` closely. It waits for the reset to end and writes the routing table and `CTRL.EN`. Then it starts the sinks, the monitors, the scoreboard and the generator with `cocotb.start_soon()`, which runs each one's `run()` as a task of its own beside the test. It runs the driver itself. When the driver has sent every packet, `Env.run()` waits 50 cycles for the outputs to drain and asks the scoreboard for its report. Nothing in `Env.run()` waits for the monitors or the scoreboard, so the test ends after those 50 cycles however much of the monitor and the scoreboard you've written.

`router_class_test` starts a 10 ns clock, holds `rst_n` low through the rising edges at 0, 10 and 20 ns, and then makes the `Env` and runs it. The driver and the scoreboard log each error they find and count it. A logged error fails nothing by itself, so the test's last line is an `assert` that the count is 0. When the count isn't 0, the assert raises an `AssertionError`, which the verdict counts as a check that fired. This

lab has no pyuvm and no `BaseTest`, so its Makefile sets `BASE_TEST := 0`, as Lab 01's `cocotb/Makefile` did. That setting tells the verdict not to look for the line that `BaseTest` prints at the end of a test.

4. Run the test:

```
make run
```

After cocotb's start-up lines you should see:

```
        0.00ns  INFO        cocotb.regression                              running
test_router.router_class_test (1/1)
    20.00ns INFO     tb.env                              [ENV] 100 packets, DST from
0123, routes 0110, 1 to 8 beats
  7960.00ns INFO     tb.env                              [ENV] sent 100 packets (476
beats); output 0 took 243 beats and output 1 took 233
  7960.00ns INFO     tb.sb                             [SB] 0 packets in, 0 predicted
for output 0, 0 checked, 0 errors
  7960.00ns INFO     cocotb.regression                test_router.router_class_test
passed
```

and, last:

```
----------------------------------------------------------------------------
 ok: passed   0.25 s   log: build/runs/router_class_test-s1/run.log
----------------------------------------------------------------------------
```

The third column is the logger that wrote the line. The environment, the driver, the two monitors and the scoreboard each have a logger of their own, named `tb.` followed by a short name for the piece: `tb.env`, `tb.drv`, `tb.in`, `tb.out0` and `tb.sb`. 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`, and each character is the output that the entry sends packets to. 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                       passed                       0.2s
----------------------------------------------------------------------------
  MISSED  tier 2  B05_dst_field_off_by_one        passed                       0.2s
  MISSED  tier 3  B09_write_pointer_power_of_two   passed                       0.2s
----------------------------------------------------------------------------
 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 took 243 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.py`. A `Monitor` is given the prefix of one stream's signal names, so the same class watches `in_valid`, `in_ready`, `in_data` and `in_last` on the input and `out0_valid`, `out0_ready`, `out0_data` and `out0_last` on output 0. `Driver`, `Sink` and `ApbMaster` change the router's inputs right after a rising edge, and every class that reads the pins, `Monitor` included, reads them at the falling edge in the middle of the cycle. The router's outputs come from flip-flops and change just after a rising edge too, so at the falling edge every signal holds the value it has for the whole cycle. That's what the specification means by a signal's value "in cycle n". A cycle in which `valid` and `ready` are both 1 is a handshake, and the beat moves at the rising edge that ends it. The monitor sees the handshake at the falling edge in the middle of that cycle, so the times it records are 5 ns before the edges at which the beats moved.
2. This is the golden router taking the first packet on its input and delivering it on output 0:

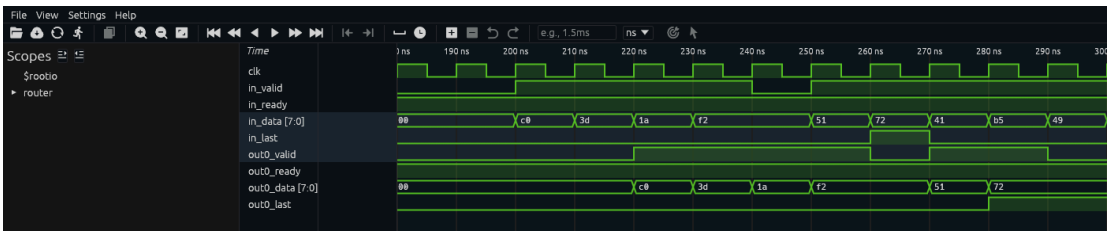


Figure 8.1: The packet c0 3d 1a f2 51 72 going in and coming out of output 0, in Surfer

Seed 1, 180 to 300 ns. `in_valid` goes high at 200 ns with `c0`, the header, and the router takes it at the rising edge at 210 ns. `c0` is on `out0_data` from 220 ns. `out0_ready` is 1 throughout.

Find `f2` on `in_data`. At how many falling edges is it on `in_data`, and how many beats is it?

Answer: A2, in Answers at the end.

3. Write `Monitor.run()` as the `TODO` describes. Each time round a `while True` loop, wait for the next falling edge with `await FallingEdge(self.dut.clk)`; drop any half-assembled packet while `rst_n` is 0; and in a handshake cycle add the beat to the packet `p`, starting a new one when `p` is `None`. When the beat has `last` = 1, the packet is finished, so put it in `to_sb` with `await self.to_sb.put(p)` and set `p` back to `None` for the next one. Log each finished packet with the `self.log.info(f"{p}")` line and the condition in the `TODO`, so that you can compare your output with the one below.
4. Run `make build`. It prints `Vtop is up to date.`, because the simulator is built from the router's RTL alone and `tb/` is Python, as in Lab 01. So you'll find a mistake in your code only when you run the test. If `run()` raises an exception, such as a `NameError`, the test ends there, the verdict is `BROKEN`, and the traceback is in the log.
5. `+SHOW=5` makes each monitor log the first 5 packets it sees, each at the time it saw the packet's last handshake. 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 each `tb.out0` line should come after the `tb.in` line of the same packet. Then run it:

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

You should see, with the reference monitor:

```
    20.00ns INFO     tb.env                                  [ENV] 100 packets, DST from
0123, routes 0110, 1 to 8 beats
   265.00ns INFO     tb.in                                   dst 3, 6 beats: c0 3d 1a f2
51 72
   285.00ns INFO     tb.out0                                 dst 3, 6 beats: c0 3d 1a f2
51 72
   365.00ns INFO     tb.in                                   dst 1, 8 beats: 41 b5 49 a0
```

```
a6 fc 48 0e
   405.00ns INFO     tb.in                              dst 3, 1 beat: c2
   425.00ns INFO     tb.out0                            dst 3, 1 beat: c2
   485.00ns INFO     tb.in                             dst 0, 4 beats: 03 03 16 14
   505.00ns INFO     tb.out0                           dst 0, 4 beats: 03 03 16 14
   525.00ns INFO     tb.in                             dst 0, 4 beats: 04 57 6c 1b
   545.00ns INFO     tb.out0                           dst 0, 4 beats: 04 57 6c 1b
   555.00ns INFO     tb.out0                            dst 0, 1 beat: 05
  7960.00ns INFO     tb.env                            [ENV] sent 100 packets (476
beats); output 0 took 243 beats and output 1 took 233
  7960.00ns INFO     tb.sb                          [SB] 0 packets in, 0 predicted
for output 0, 0 checked, 0 errors
```

The header's low six bits are the packet's sequence number, so `c0` is packet 0, `41` packet 1 and `c2` packet 2. Why is packet 1 missing from output 0, why does each `tb.out0` line come 20 ns after the `tb.in` line of the same packet, and why does `05` come out of output 0 with no `tb.in` line?

Answer: A3, in Answers at the end.

The `[SB]` line says 0 packets in, because you haven't written the scoreboard yet.

Check: your `tb.in` and `tb.out0` lines match the ones above. Your seed is the same as ours, so your packets are the same too.

Part 3: The scoreboard

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

1. Read the docstring at the top of `tb/scoreboard.py`. 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. (The full rule also allows a changed `out_last` on a packet that the router cut for being oversize, but every packet in this lab is 1 to 8 beats, so none is cut.) 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()`: two loops that each run forever, each in an `async def` method of its own. One takes packets from `from_in` and calls `predict()` with each, and the other takes packets from `from_out0` and calls `check()` with each. Start both with `cocotb.start_soon()` and await both.
   - `predict()`: append the packet to `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 `self.error()`, which logs the message on the `tb.sb` logger and counts it in `n_errors`:

   ```
   self.error(f"out0 delivered {got} at {got.t_last} ns; expected {want}, which went
   in at {want.t_last} ns")
   ```
3. 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` and `ok: passed`. The golden router is correct, so if an error fires here, the mistake is in your scoreboard or your monitor.
4. Run it on five seeds:

```
make seeds SEEDS="1 2 3 4 5"
```

You should see `5 ok  (5 runs)`. The command took us 1.5 s.

5. Write down what you now expect from `make check-all`, then run it. With the reference scoreboard you should see:

```
     ok  the golden design                      passed                      0.3s
------------------------------------------------------------------------------
  caught  tier 2  B05_dst_field_off_by_one       AssertionError              0.3s
  caught  tier 3  B09_write_pointer_power_of_two  AssertionError              0.3s
------------------------------------------------------------------------------
 2 of 2 caught.
 Every bug in the list, each one caught by a check that fired.
```

Each caught row names the exception that ended the run, which is the `AssertionError` from the test's last line. The number of errors is in that assert's message, which you'll see in the next step.

6. Look at the first error on B05:

```
make run BUG=B05
```

With the reference scoreboard, the first line after `[ENV]` is:

```
   505.00ns ERROR    tb.sb                                out0 delivered dst 0, 4
beats: 03 03 16 14 at 505 ns; expected dst 3, 6 beats: c0 3d 1a f2 51 72, which went
in at 265 ns
```

and the end of the run is:

```
  7960.00ns INFO     tb.sb                                [SB] 100 packets in, 51
predicted for output 0, 49 checked, 47 errors
  7960.00ns WARNING  ..ter_class_test.router_class_test the testbench reported 47
errors; the first: out0 delivered dst 0, 4 beats: 03 03 16 14 at 505 ns; expected
dst 3, 6 beats: c0 3d 1a f2 51 72, which went in at 265 ns
                                                    assert 47 == 0
```

followed by the traceback of the failed assert and the verdict, `caught: AssertionError`. In the `WARNING` line, cocotb prints the assert's message, and under it the comparison that failed.

In Part 2, on the golden router, `c0 3d 1a f2 51 72` came out of output 0, 20 ns after it went in. Its header, `c0`, is `1100_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.

7. Now the first error on B09:

```
make run BUG=B09
```

```
   775.00ns ERROR    tb.sb                            out0 delivered dst 3, 6 beats:
c0 3d 1a f2 51 72 at 775 ns; expected dst 0, 1 beat: 05, which went in at 535 ns
```

You've seen this packet before, in your Part 2 printout. Where, and which beat into output 0 is the first bad one?

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 none of these runs rebuilds anything.

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 -- your test passes B05_dst_field_off_by_one too.` Why does the test miss B05 now?

   Answer: A6, in Answers at the end.

2. `+DSTS=0` makes the generator give every packet `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              AssertionError               0.3s
             router_class_test: AssertionError: the testbench reported 68 errors; the
   first: out0 delivered dst 0
   ```

   and `make run BUG=B05 PLUSARGS="+DSTS=0"` shows its first error in full:

   ```
     4845.00ns ERROR    tb.sb                                 out0 delivered dst 0, 6
   beats: 00 77 ce 9b 6b 1e at 4845 ns; expected dst 0, 8 beats: 20 27 e4 05 51 e4 6b
   4f, which went in at 2345 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          passed                       0.2s
   ```

   ```
     caught  B09_write_pointer_power_of_two         AssertionError               0.2s
          router_class_test: AssertionError: the testbench reported 1 error; the first:
   out0 delivered dst 0,
   ```

   `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 &
   ```

   Surfer lists two scopes at the top. `$rootio` holds the router's ports a second time, and `router` is the router itself, which is cocotb's top level. Add `clk`, `in_valid`, `in_ready`, `in_data`, `out0_valid` and `out0_data` from `router`, and `wr_ptr_q` and `rd_ptr_q` from `router.u_buf0`, output 0's buffer:

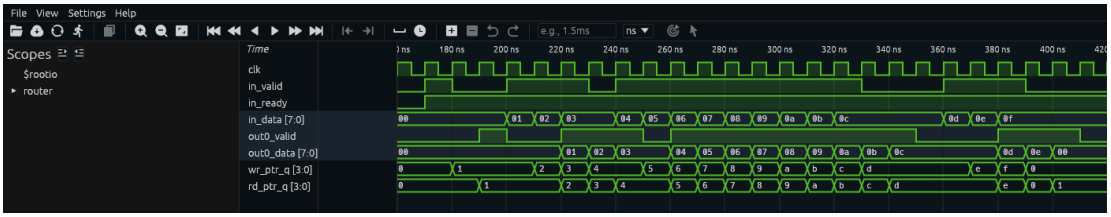


Figure 8.2: Sixteen one-beat packets through output 0 with B09 switched on, in Surfer

B09, 160 to 420 ns. The first beat, `00`, is offered from 170 ns; `in_data` already held `00`, so only `in_valid` shows it. 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.py` and `tb/scoreboard.py`.

Questions:

1. In Lab 01 the pyuvm testbench was split into the same kinds of pieces. Which jobs did pyuvm 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 3, 1 beat: c2 at 785 ns; expected dst 3, 6 beats: c8 5c f3 ea 11 09, which went in at 765 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 to submit.

Going further (optional)

Watch output 1 as well. The `Env` needs a third `Monitor`, on the prefix `out1`, and a fourth queue to carry that monitor's packets to the scoreboard. The scoreboard needs a second list of expected packets, which `predict()` fills for routes with `PORT` 1, and a third loop that takes output 1's packets and checks them against it. Nothing needs rebuilding. Run `make seeds SEEDS="1 2 3 4 5"` to check that the golden router still passes.

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

Answers

A1. Part 1: The starter

The starter's monitor and scoreboard are empty, so nothing compared those 243 beats with anything. That's why the `[SB]` line says the scoreboard took in 0 packets and checked 0. A pass means only that no check fired, and this testbench has no checks that could fire, so it passes on all three routers.

A2. Part 2: The monitor

It's on `in_data` at two falling edges, 235 ns and 245 ns, and it's one beat. The driver put it there at 230 ns, the router took it at the edge at 240 ns, and the driver then left one idle cycle with `in_valid` at 0. So at 245 ns `in_data` still shows `f2`, but with `in_valid` at 0 there's no handshake. A monitor that looked only

at `in_data` would record it twice. Output 0 shows the same idle cycle two cycles later, with `out0_valid` at 0 from 260 to 270 ns.

A3. Part 2: The monitor

Packet 1′s header, `41`, has `DST` 1 in bits 7:6, and `ROUTE1` 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 is taken from output 0 two cycles after it went in. `05` is packet 5. The five packets that the input monitor logged were packets 0 to 4, including packet 1, which went to output 1, so output 0′s fifth packet is packet 5, which went in after the input monitor had stopped logging.

A4. Part 3: The scoreboard

`DST` is bits 7:6 of the header, and in `c0` they're `11`, so the router should have used `ROUTE3`, which sends packets to output 0. Bits 6:5 of `c0` are `10`, so this router read the field one bit too low and used `ROUTE2`, which sends packets to output 1. That's B05, `dst_field_off_by_one`. Packet 0 never came out of output 0, so the scoreboard compared it with the first packet that did, `03 03 16 14`, and reported the mismatch. Packet 3′s header, `03`, has `00` in both bits 7:6 and bits 6:5, so it went where it should.

A5. Part 3: The scoreboard

It's packet 0, `c0 3d 1a f2 51 72`. It was the first packet that output 0 delivered, at 285 ns, in your Part 2 printout and in this run alike. At 775 ns its beats came out of the buffer a second time. Before that, output 0 had delivered 15 beats: packet 0′s 6 beats, then `c2`, then `03 03 16 14` and `04 57 6c 1b`. So the 16th beat into output 0, `05`, 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 `+ROUTES=0000`, every entry sends packets to output 0, so whichever entry the router reads, each 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 bits has an effect of its own on where a packet goes. With the default `0110`, changing either bit of `DST` changes the output a packet goes to, and with `0000` neither bit does.

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. A run of 32 packets sends only those. Packet 32′s header is `20`, which is `0010_0000`, so its bits 6:5 are `01`. The router uses `ROUTE1` for packets 32 to 63, so it sends all of them to output 1. In the first error, 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`.

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. For the first 15 beats, 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

In Lab 01, `AluEnv.connect_phase()` connected the monitor's analysis port to the scoreboard. Here `Env.__init__()` makes three queues and passes each one to its two ends. In Lab 01, pyuvm's phases started every component's `run_phase()`, and the test's objection decided when the test was over. Here `Env.run()` starts the sinks, the monitors, the scoreboard and the generator with `cocotb.start_soon()`, runs the driver itself, and ends the test after a fixed 50-cycle drain. In Lab 01 `BaseTest` counted every error any component logged, and failed the test if the count wasn't 0. Here the driver and the scoreboard each keep their own `n_errors`, `Env.n_errors()` adds them up, and the test's `assert` fails the test if the sum isn't 0. In Lab 01 each test was its own class, picked with `TEST=`, and both tests reused the same driver, monitor and scoreboard. Here there is one test, its variations are the plusargs that `RouterCfg` reads, and anything those can't express needs a change to `Env.run()`. In Lab 04 you'll finish a pyuvm port of this testbench as it stands after Lab 03, and compare the two piece by piece.

A10. Finishing

It's the same bug. `c2` is packet 2, which was the second packet that output 0 delivered, and here its beat has come out of the buffer a second time. After the first wrong packet, the buffer keeps handing back old beats, and the packets that output 0 delivers no longer line up with the ones in `expected`. Read the first error first. Most of the later ones are comparisons that failed because the delivered packets and `expected` had fallen out of step, so their number says little about how many packets the router got wrong.

III 3. Constraints and coverage

9 Constrained random with pyvsc . . . 69
9 randobj, random fields, constraints and randomize() . . . 69
9 How pyvsc picks the values . . . 70
9 Attributes that a constraint reads . . . 71
9 dist . . . 72
9 soft . . . 72
9 solve_order . . . 73
9 unique and sum . . . 73
9 Seeds . . . 73
9 Speed . . . 74
9 Counting what came out . . . 74
9 Check yourself . . . 74
9 Further reading . . . 74
9 Answers . . . 75

10 Functional coverage you can read . . 77
10 Coverpoints, bins and crosses . . . 77
10 Ranges and rel= . . . 78
10 coverage.xml and make cov . . . 78
10 The percentage . . . 79
10 Ignore bins and illegal values . . . 79
10 Three kinds of empty bin . . . 80
10 Check yourself . . . 80
10 Further reading . . . 80
10 Answers . . . 81

11 Lab 03: Constraints and coverage . . . 83
11 Before you start . . . 83
11 Part 1: The venv and the testbench you start from . . . 83
11 Part 2: Coverage you can read . . . 86
11 Part 3: Constraints the solver honours . . . 88
11 Part 4: Close the two holes . . . 90
11 Finishing . . . 93
11 Going further (optional) . . . 93
11 Answers . . . 93

9. Constrained random with pyvsc

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

Lab 02's generator chose each packet's DST and length itself, with calls to Python's `random` written into its loop. In Lab 03 the generator makes each packet as a `RandPacket` instead, and pyvsc randomizes it. The class declares which of its fields are random and the rules that they must follow, and the generator calls `randomize()`, which hands those rules to a solver and lets it 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 solver chooses among the values your rules allow, and pyvsc doesn't always choose evenly.

pyvsc 1.1.0 isn't in the course image. Lab 03, Part 1 has you make the course venv with `make venv`, and that installs pyvsc along with its solver, Boolector. The examples on this page are in `guide-examples/constrained-random-with-pyvsc/`. Each is a small class built on SimProc's ALU operations, randomized inside a pyuvm test that builds and runs with the course's `pyuvm.mk`, the same makefile that every lab uses. In Lab 03, Part 3 you'll run the same kind of experiment on classes shaped like the router's packet, and then count what comes out of the packet itself.

randobj, random fields, constraints and randomize()

A pyvsc class is an ordinary Python class with the `@vsc.randobj` decorator. Its random fields are pyvsc types, such as `vsc.rand_bit_t(8)` for an 8-bit value and `vsc.rand_enum_t(...)` for an enum, and each method marked `@vsc.constraint` holds rules that the fields must follow. Here is a class for one ALU vector:

```
class AluOp(enum.IntEnum):
    ADD = 0
    SUB = 1
    OR = 2
    NAND = 3
    SHL = 4
    SHR = 5


@vsc.randobj
class AluVec:
    def __init__(self):
        self.op = vsc.rand_enum_t(AluOp)
        self.A = vsc.rand_bit_t(8)
        self.B = vsc.rand_bit_t(8)

    @vsc.constraint
    def c_shift(self):
        with vsc.implies(self.op.inside(vsc.rangelist(AluOp.SHL, AluOp.SHR))):
            self.B < 4
```

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. The lines in a constraint method look like Python comparisons, but pyvsc never evaluates them as `True` or `False`. It runs the method once, when the object is made (at the end of its `__init__`), and records each line as a rule for the solver. A call to `v.randomize()` asks for values of `op`, `A` and `B` that meet every constraint that is switched on, all at the same time. With `randomize_with()`, a call can add rules of its own, which apply on top of the class's and only for that call:

```
with v.randomize_with() as it:
    it.op == AluOp.SUB
    it.A == it.B
```

A constraint can be switched off with `v.c_shift.constraint_mode(False)`, and on again with `True`.

`CrItemTest` randomizes one `AluVec` a few times:

```
[ITEM] randomize():                          ADD  A=98 B=dd
[ITEM] randomize():                          SHR  A=5c B=03
[ITEM] randomize():                          NAND A=c3 B=b1
[ITEM] randomize():                          SUB  A=98 B=dc
[ITEM] randomize_with(op == SUB, A == B):    SUB  A=8a B=8a
[ITEM] randomize_with(op == SUB, A == B):    SUB  A=61 B=61
[ITEM] before:                               SUB  A=61 B=61
Solve failure: set 'solve_fail_debug=1' for more details
[ITEM] randomize_with(op == SHL, B == 5) raised SolveFailure: solve failure
[ITEM] after:                                SUB  A=b6 B=61
[ITEM] the same with c_shift off:            SHL  A=87 B=05
```

The call with `op == SHL` and `B == 5` can't be met, because `c_shift` says a shift has `B` below 4. SystemVerilog's `randomize()` returns 0 when that happens, but pyvsc raises an exception, `SolveFailure`, and prints the `Solve failure` line first. The failed call also changed the object. `A` went from `61` to `b6` while `op` and `B` kept their old values, because pyvsc gives a field that no constraint mentions, such as `A`, its new value before it hands the other fields to the solver. So after the failure the object held a combination that no call had asked for. With `c_shift` switched off, the same call succeeds.

So a test shouldn't carry on past a failed `randomize()`. A `SolveFailure` that the test doesn't catch ends the run, and the course's graders call such a run BROKEN, because a test that stopped in the middle of its stimulus says nothing about the design. Lab 04's `RouterRandSeq` catches the failure only to raise a clearer error, which names the packet's class and the `route_en` bits that the sequence was given. Lab 04, Part 4 describes a run that the graders called `BROKEN: SolveFailure`, and why it failed.

How pyvsc picks the values

pyvsc's constraints are meant to work as SystemVerilog's do, and Lab 03, Part 3 has you write down what SystemVerilog's rules say the first six of its experiments should show. The standard makes every combination of values that meets the constraints equally likely. pyvsc chooses differently, as its source, `vsc/model/solvegroup_swizzler_partsel.py`, shows. It takes the random fields that the constraints mention (up to four of them) in a random order, and gives each a random target inside the range of values that it has worked out the field can take. An enum's target goes to the solver as one piece, and a wider field's is split into pieces of a bit or a few bits each. Then pyvsc offers all the pieces to the solver one at a time, in a random order, keeps each one that the constraints still allow, and drops the rest. Wherever a piece was dropped, the field gets whatever value the solver finds there, and that value isn't random.

`CrSkewTest` counts what comes out of 1,000 vectors with no constraint at all, and of 1,000 with `c_shift`:

```
[SKEW] no constraint:     ADD 177  SUB 166  OR 172  NAND 159  SHL 165  SHR 161   B <
4 in 18
[SKEW] with c_shift:      ADD 392  SUB 150  OR 168  NAND 179  SHL  50  SHR  61   B <
4 in 119
```

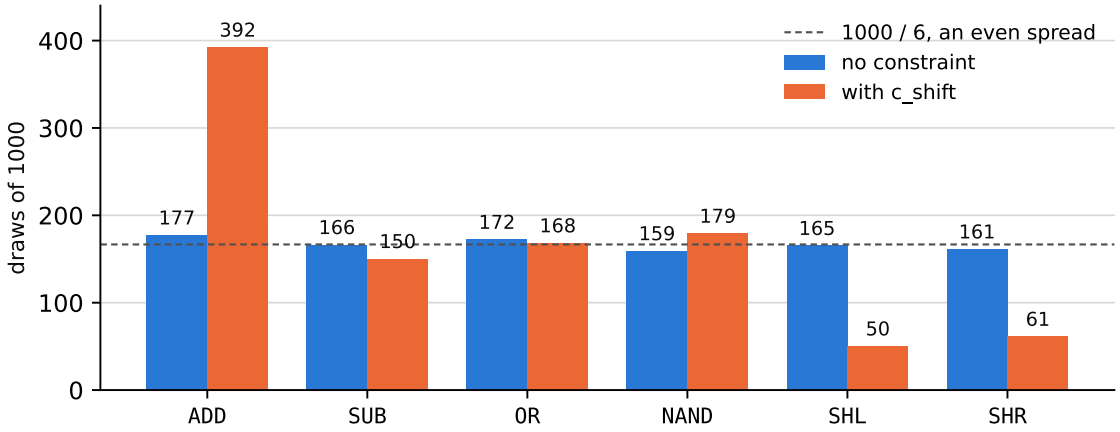


Figure 9.1: How often each operation came out in 1,000 randomizations, with no constraint and with c_shift

The two `[SKEW]` lines as bars. The dashed line is 1,000 / 6, an even spread.

Without the constraint, each operation came out close to 1,000 / 6 times, and `B` was below 4 in 18 draws, about the 16 that 4 values in 256 would give. With `c_shift`, a target of `SHL` or `SHR` survives only if the solver is offered it before any piece of `B`'s target that would put `B` at 4 or more, and 252 of the 256 targets that `B` can get are 4 or more. When such a piece goes first, the shift is dropped, and the solver puts `ADD` in its place. So the two shifts came out 50 and 61 times, and `ADD` 392. Every draw met `c_shift`, so pyvsc had nothing to warn you about. A constraint says which values can come out, not how often each one does, and to know that you count them, as Lab 03's coverage does for the router's packet.

The operations in the first line came out even because pyvsc draws each field that no constraint mentions without the solver, evenly over its values. (In DD105-A, on Verilator 5.050, an enum whose four names fill its two bits came out as its first name 813 times in 1,000.)

Attributes that a constraint reads

A constraint can also read attributes that aren't random. In Lab 03's packet, `c_dst` reads `route_en`, which holds the routing table's enable bits, to keep each packet on an enabled route. pyvsc runs the constraint methods only once, when the object is made, so the way you declare such an attribute decides whether the solver sees a change made to it after that. `CrAttrTest` uses this class:

```
@vsc.randobj
class Limited:
    def __init__(self):
        self.A = vsc.rand_bit_t(8)
        self.B = vsc.rand_bit_t(8)
        self.a_max = 4                # a plain Python attribute
        self.b_max = vsc.uint8_t(4)   # a non-random pyvsc attribute

    @vsc.constraint
    def c_lim(self):
        self.A < self.a_max
        self.B < self.b_max
```

It randomizes the object 200 times, sets both limits to 200, and randomizes it 200 times more:

```
[ATTR] a_max = 4, b_max = 4, 200 draws:     the largest A 3, the largest B 3
[ATTR] a_max = 200, b_max = 200, 200 draws: the largest A 3, the largest B 199
```

When pyvsc read `c_lim`, at the end of `__init__`, `self.a_max` was the number 4, and it went into the rule as the number 4, so changing the attribute later changed nothing. `self.b_max` is a pyvsc field, so the rule refers to the field and reads its value at every call. Lab 03's packet holds `route_en` and its three length weights in `vsc.uint8_t` attributes for that reason. The generator sets `route_en` on each new packet after

making it, and the weights are fields so that a test can change them before it randomizes. pyvsc gives no error or warning when a plain attribute has gone into a rule as a fixed number.

dist

`vsc.dist()` gives each value or range a weight, and the solver picks among them in proportion to the weights. `AluVecWeighted` makes `ADD` five times as likely as each other operation:

```
@vsc.constraint
def c_op(self):
    vsc.dist(self.op, [vsc.weight(AluOp.ADD, 5), vsc.weight(AluOp.SUB, 1),
                       vsc.weight(AluOp.OR, 1), vsc.weight(AluOp.NAND, 1),
                       vsc.weight(AluOp.SHL, 1), vsc.weight(AluOp.SHR, 1)])
```

A `dist` is one more constraint, so a call can add others on the same variable. `CrDistTest` draws 1,000 vectors, then asks for `SUB` 1,000 times with the `dist` still on:

```
[DIST] op dist ADD 5, others 1:                    ADD 516  SUB  89  OR 103  NAND 100
SHL 104  SHR  88
[DIST] the same, then randomize_with(op == SUB):  failed 0 times in 1000; ADD   0  SUB
1000  OR   0  NAND   0  SHL   0  SHR   0
```

The weights ask for `ADD` in 5 draws of 10, and it came out in 516 of 1,000, so they came out right. Asking for `SUB` with the `dist` still on never failed. pyvsc draws a target from the `dist` by its weights, up to four times, and keeps the first one that the other constraints allow. If none of the four fits, it falls back to a target from the field's own range of values. The weights in a `dist` can be enum names, as here, or numbers.

The labs put the `dist` on a selector and choose the other fields from it with `vsc.if_then`. Lab 03's packet has a `kind`, weighted 25 / 50 / 25, and a `length` that `c_len` chooses from the kind. `VecKind` does the same for `A`, with four kinds, weighted 10 / 20 / 30 / 40, that make `A 00`, make it `ff`, make it equal to `B`, or leave it free. Then the test asks for `A < 0x10`, which rules out `K_ONES`:

```
[DIST] kind dist 10/20/30/40:                     K_ZERO  89  K_ONES 194  K_SAME 295
K_ANY 422
[DIST] the same, then randomize_with(A < 0x10):   failed 0 times in 1000; K_ZERO 127
K_ONES   0  K_SAME 374  K_ANY 499
```

No call failed, and the other three kinds shared the draws in about the ratio 10 : 30 : 40 of their weights. In DD105-A, Verilator failed 921 and 199 times in 1,000 on the same two shapes, because it behaves as if it picks an entry of the `dist` first and solves the rest afterwards. On pyvsc you can constrain a packet's length inline whatever its weights, and line 3 of Lab 03's experiments shows it on a class with the same `kind` and `length` constraints as the router's packet.

soft

A soft constraint holds unless something else contradicts it, and then it gives way instead of failing the call. `SoftVec` makes `B` 1 by default, and has two soft constraints on `A` that disagree:

```
@vsc.constraint
def c_b(self):
    vsc.soft(self.B == 1)

@vsc.constraint
def c_a(self):
    vsc.soft(self.A == 0x11)
    vsc.soft(self.A == 0x22)
```

`CrSoftTest` randomizes it 100 times, then 100 times with `B == 7` inline:

```
[SOFT] randomize(), 100 times:              A=22 B=01 x100
[SOFT] randomize_with(B == 7), 100 times:   failed 0; A=22 B=07 x100
```

The inline `B == 7` is a hard constraint, and it beat the soft `B == 1` without a failure, as the SystemVerilog standard says a hard constraint always does. Of the two soft constraints that disagreed, the later one won

every time, which is the order that the standard fixes: a later soft constraint beats an earlier one. The standard also puts an inline soft constraint ahead of the class's soft ones, which `CrSoftTest` doesn't try.

solve_order

`vsc.solve_order(a, b)` asks for `a` to be chosen before `b`. That changes how often each value of `a` comes out, without changing which values are legal. `ShiftSel` has a bit, `is_shift`, that picks between the two shifts and the four other operations, and `ShiftSelOrdered` adds `vsc.solve_order(self.is_shift, self.op)`:

```
[SOLVE] solve_order(is_shift, op): is_shift in 509 of 1000;   ADD 251  SUB  60  OR  89
NAND  91  SHL  82  SHR 427
[SOLVE] without solve_order:       is_shift in 412 of 1000;   ADD 224  SUB 105  OR 133
NAND 126  SHL 144  SHR 268
```

By the standard, `is_shift` should come out about 500 times in 1,000 with `solve_order`, because it's chosen first and has two values, and about 333 without, because two of the six operations are shifts. pyvsc gave 509, and 412. So `solve_order` gave `is_shift` the spread that the standard asks for, but the operations within each half are far from even, with `SHR` at 427 and `SHL` at 82. Once `is_shift` is fixed, `op`'s target is still one of all six operations, and a target that disagrees with `is_shift` is dropped. When `is_shift` is 1, the target is an operation that isn't a shift two times in three, and each time it's dropped the solver puts `SHR`. When `is_shift` is 0, a target of `SHL` or `SHR` is dropped, and the solver puts `ADD`.

unique and sum

Lab 03's experiments also try `vsc.unique()` on a list and a constraint on a list's `sum`, the two constraints that DD105-A found Verilator gets wrong on an array. Lab 03, Part 4 uses the second, in a `FillPlan` whose eight lengths add up to 8.

Seeds

Every run starts Python's `random` from a seed, which `make` passes as `COCOTB_RANDOM_SEED` from its `SEED` variable (1 unless you set it). cocotb seeds `random` for each test from that seed and the test's name, so two tests with the same body send different values. A pyvsc object takes its own random state from Python's `random` once, at its first `randomize()`, and `CrSeedTest` shows it. Each of its three rows starts from the random state that the test started with. The first row makes an object and randomizes it six times, the second calls `random.seed(99)` between making the object and its first `randomize()`, and the third calls it after the first `randomize()`. On seed 1:

```
[SEED] make, then randomize() 6 times:          NAND SUB SHL SHL SHL NAND
[SEED] make, random.seed(99), then randomize(): SUB SHL ADD SUB OR OR
[SEED] make, randomize(), random.seed(99), ...: NAND SUB SHL SHL SHL NAND
```

and on seed 2:

```
[SEED] make, then randomize() 6 times:          SHR SUB SHR SHL OR SHL
[SEED] make, random.seed(99), then randomize(): SUB SHL ADD SUB OR OR
[SEED] make, randomize(), random.seed(99), ...: SHR SUB SHR SHL OR SHL
```

The run's seed decided the first and third rows, and reseeding after the first `randomize()` changed nothing. Reseeding before it decided the second row, whatever the run's seed. So an object that the test makes and randomizes repeats with the seed, and you can rerun a failing seed and watch the same failure happen again. Lab 03's generator makes and randomizes every packet inside the test, so the run's seed decides them all.

A seed reproduces a run only when the testbench draws the same numbers in the same order. Every random value in the labs comes from Python's `random`, one draw after another, including pyvsc's state for each new packet, the payload bytes and the idle cycles. A change that draws one number more changes every value after it, and for this reason Lab 03 says that your own constraints may give different counts from ours with the same seed.

Speed

pyvsc is slow next to Python's own `random`. `CrSpeedTest` timed 1,000 calls of `randomize()` on one `AluVec`, 1,000 on a new `AluVec` each time, and 100,000 of three `random.randrange()` calls:

```
[SPEED] per second: randomize() on one AluVec 531; a new AluVec each time 494; three
random.randrange() calls 1037162
```

That's about 1.9 ms a call, against about 1 µs for three `random.randrange()` calls, so the labs give the solver only the fields that a constraint mentions. Lab 03's packet randomizes its DST, its kind and its length with pyvsc. pyvsc then calls the packet's `post_randomize()`, as it does after every `randomize()`, and that method fills in the payload bytes and the idle cycles with `random`.

Importing pyvsc takes time too, and every run that uses pyvsc pays it once. `make import-time` timed `import vsc` in the venv's Python three times:

```
import vsc: 1.41 s
import vsc: 1.00 s
import vsc: 0.99 s
```

Lab 03's graded test takes about 5 s once Part 3's constraints are in, where Lab 02's took 0.25 s. Part of the difference is the import and the solver, and part is that Lab 03's test sends 800 packets to Lab 02's 100.

Counting what came out

In a testbench with a design in it, a skew like `CrSkewTest`'s wouldn't fail anything either, because the scoreboard checks the design's answers and not how often each input was sent. In Lab 03 you count with coverpoints. Their first run shows a third of the router's packets 1 beat long and a third 8 beats long (263 of 800 each), because the packet's length comes from `kind`, an enum with three names, and nothing weights it yet. Functional coverage you can read explains how to read those counts.

Check yourself

1. A packet class has a constraint `self.length <= self.max_len`, with `self.max_len = 8` set in `__init__`. A test sets `p.max_len = 4` before it randomizes, and still gets 8-beat packets. Why, and what do you change?

   Answer: A1, in Answers at the end.

2. A sequence calls `randomize_with()` with an inline constraint that contradicts one of the class's constraints, catches the `SolveFailure`, and sends the item anyway. What does the design get?

   Answer: A2, in Answers at the end.

3. You add a monitor that calls `random.random()` once at the start of its `run_phase`, and a test that sent the same packets on seed 3 every time now sends different ones. Which values change, and why?

   Answer: A3, in Answers at the end.

Further reading

pyvsc's own documentation is the authority on its API. ChipVerify's SystemVerilog pages explain the constructs that pyvsc copies, with longer examples. Their code is SystemVerilog, and their output comes from commercial simulators, so where pyvsc behaves differently, trust the counts on this page and in Lab 03.

- pyvsc's Constraints, Methods and Data Types pages. The Methods page says that an object's random state is made at its first `randomize()` if nothing has set it, as `CrSeedTest` shows. It doesn't say what a failed call does; `CrItemTest` shows that.
- ChipVerify's SystemVerilog Randomization and Inline Constraints. Both say that a failed `randomize()` returns 0 and leaves the variables as they were. pyvsc raises `SolveFailure` instead, and the failed call in `CrItemTest` changed `A`.

- Soft Constraints. Correction: it says that when two soft constraints on one variable conflict, which one wins depends on the solver. The standard fixes the order (the later one wins), and pyvsc gave the later one in all 100 draws of `CrSoftTest`.
- solve before explains correctly what the standard means. pyvsc's `solve_order()` gave 509 in 1,000 for a bit that the standard puts at 50%.
- Disable Constraints, for `constraint_mode()`.

Next: Functional coverage you can read.

Answers

A1. Check yourself

`max_len` is a plain Python attribute. pyvsc read the constraint at the end of `__init__`, when `max_len` was 8, so the rule holds the number 8, and setting the attribute afterwards changes nothing. Declare it as a pyvsc field, `self.max_len = vsc.uint8_t(8)`, and the solver reads its value at every call, as it reads `b_max` in `CrAttrTest` and the weights in Lab 03's packet.

A2. Check yourself

An item that no call asked for. The failed call can change some fields and leave others as they were, as `A` changed from `61` to `b6` in `CrItemTest` while `op` and `B` didn't. Let the exception end the run, or switch the class's constraint off with `constraint_mode(False)` before that call, and on again after it.

A3. Check yourself

Every packet whose first `randomize()` comes after that call, and every payload byte and idle cycle drawn after it. Each packet takes its random state from Python's `random` at its first `randomize()`, so one extra draw before it moves every value after it.

10. Functional coverage you can read

This page answers: *how do I find out what my random stimulus reached, and how do I read the numbers cocotb-coverage gives me?*

In DD104's Coverage as a question, 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. cocotb-coverage 2.0 keeps the same kind of list in Python. You name the values you care about, and it counts how often the stimulus produces each one. From Lab 03 on, each run in the labs writes those counts to a file, and `make cov` merges the files of all your runs and reports them. In Lab 03, Part 1 you'll see `router_rand_test` pass and catch neither of the router bugs that belong to that lab, B07 and B08. In Lab 03, Part 2 you write coverpoints for the router's packets, and their counts show you the holes in the test's stimulus where those two bugs are.

cocotb-coverage, like pyvsc, is in the course venv that Lab 03, Part 1 makes. The example for this page, `guide-examples/functional-coverage-you-can-read/`, sends random vectors to SimProc's ALU, the design you tested in Lab 01, and samples each vector.

Coverpoints, bins and crosses

Here is the coverage that the example collects for each vector it sends to the ALU. It counts the operations, three kinds of value for `A`, and whether the two operands were equal:

```
OPS = ["add", "sub", "or_op", "nand_op", "shl", "shr"]


def in_range(value, rng):
    """rel= for a bin written as (low, high)."""
    return rng[0] <= value <= rng[1]


@CoverPoint("alu.op", xf=lambda op, a, b: op, bins=list(range(6)), bins_labels=OPS)
@CoverPoint("alu.a", xf=lambda op, a, b: a, bins=[(0, 0), (1, 254), (255, 255)],
rel=in_range,
            bins_labels=["zero", "other", "ones"])
@CoverPoint("alu.eq",   xf=lambda   op,   a,   b:   int(a   ==   b),   bins=[0,   1],
bins_labels=["differ", "equal"])
@CoverCross("alu.x_op_eq", items=["alu.op", "alu.eq"])
def sample(op, a, b):
    pass
```

Each **bin** is one question, and its count is the number of samples that answered yes. A **coverpoint** groups the bins for one value. In cocotb-coverage a coverpoint is a decorator on a function, here `sample()`, and every call to that function is one sample. The test calls `sample(op, a, b)` once for every vector it sends. At each call, the coverpoint's `xf=` function takes the value out of the call's arguments, and the coverpoint adds one to the count of the bin that the value matches. A **cross** makes one bin for every combination of the bins of the coverpoints it crosses, so `alu.x_op_eq` has 6 × 2 = 12 bins, named by the pairs of labels they stand for, from `('add', 'differ')` to `('shr', 'equal')`. DD104 asked whether `SUB` and `NAND` had been run with two different operands, and this cross asks that of every operation. The cross has to go below the coverpoints it crosses, because at each call the decorators run from the top down, each one counting its value before it calls the one below it, and the cross reads which bin each of its coverpoints has just counted.

Every name is a dotted path, and the part before the last dot names a **group**. `alu` is the group of everything whose name starts with `alu.`, and its counts of bins and of bins hit are the sums of theirs.

SimProc programs use equal operands on purpose, because `SUB rX, rX` is how you zero a register and `NAND rX, rX` is how you write NOT. Equal operands also hide two of Lab 01's ALU bugs. 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`.

Ranges and rel=

cocotb-coverage puts a value in a bin when `value == bin`. That's fine for single values, but no number is equal to a tuple, so a bin written as a range such as `(1, 254)` needs a relation function, given as `rel=`, that says what "in the bin" means. `alu.a` has one, `in_range()`. The example also has a coverpoint with two single values and a range written as a tuple between them, and no `rel=`:

```
@CoverPoint("quirk.a_no_rel", xf=lambda op, a, b: a, bins=[0, (1, 254), 255],
            bins_labels=["zero", "other", "ones"])
```

After 300 random vectors, `CovPlainTest` prints its counts:

```
[COV] quirk.a_no_rel: {'zero': 1, 'other': 0, 'ones': 1}
```

Two values of `A` landed in the single-value bins, and the other 298 matched no bin at all, so `other` stayed at 0, with no warning. You find a mistake like this only by reading a bin's count against what you know the test sent. A `rel=` applies to every bin of its coverpoint, so write every bin as a range, `(0, 0)` included, as `alu.a` does. In Lab 03, Part 2 you run into the same thing with the router's level coverpoint.

coverage.xml and make cov

Each test in the example ends by writing every bin's count to `coverage.xml` in its run folder:

```
coverage_db.export_to_xml("coverage.xml")
```

`make cov` finds every `coverage.xml` under `build/runs`, merges them with cocotb-coverage's own `merge_coverage()`, and prints each group, coverpoint and cross with the bins that were never hit. Here is what it printed after one run of `CovPlainTest`, which sends 300 vectors with `A` and `B` uniformly random:

```
=========================================================================
 COVERAGE   1 run, merged by cocotb-coverage into build/coverage/merged.xml
=========================================================================
  top                                          81.48 %   22 of 27 bins
    top.alu                                      82.61 %   19 of 23 bins
      top.alu.op                                  100.00 %   6 of 6 bins
      top.alu.a                                   100.00 %   3 of 3 bins
      top.alu.eq                                  100.00 %   2 of 2 bins
      top.alu.x_op_eq                              66.67 %   8 of 12 bins
              not hit: ('sub', 'equal'), ('or_op', 'equal'), ('nand_op', 'equal'),
('shr', 'equal')
    top.quirk                                    75.00 %   3 of 4 bins
      top.quirk.a_no_rel                          66.67 %   2 of 3 bins
          not hit: other
      top.quirk.op_legal                          100.00 %   1 of 1 bins
          not hit: FAIL
=========================================================================
```

`top` comes from cocotb-coverage's `export_to_xml()`, which puts every group under one group of that name. `quirk` holds the example's other two items, `a_no_rel` from the section above and `op_legal` from the one below, in a group of their own so that they don't change `alu`'s counts.

In `alu`, every operation and every kind of `A` came up, but only `ADD` and `SHL` had a vector with 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. `CovBiasedTest` makes `A == B` in one vector in eight, and one run of it filled the cross:

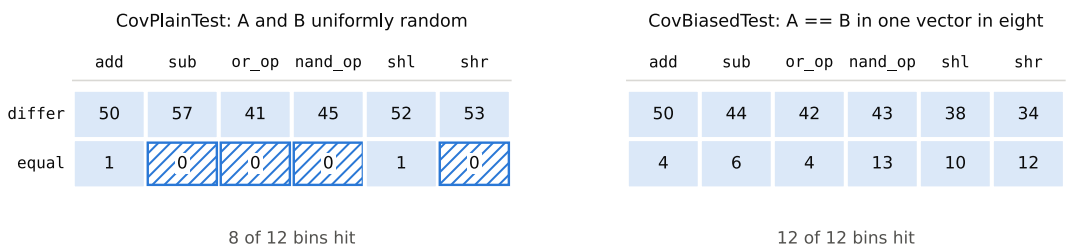


Figure 10.1: The cross alu.x_op_eq as a grid of counts after CovPlainTest, with four empty bins in the equal row, and after CovBiasedTest, with all twelve hit.

The `[COV] x_op_eq` lines of the two runs, as grids. A hatched bin is empty.

`make cov` names only the empty bins. The merged file, `build/coverage/merged.xml`, has every bin with its count, and Lab 03, Part 2 has you read the length bins' counts from it with `grep`.

The percentage

cocotb-coverage's `82.61 %` for `alu` is the number of bins hit divided by the number of bins, 19 of 23, with every bin of every coverpoint and cross under the group counted the same. The SystemVerilog standard works it out another way, and gives a group the average of its coverpoints' and crosses' coverage. For `alu` that's the average of 100, 100, 100 and 66.67%, which is 91.67%. The two differ because the cross has 12 of the group's 23 bins, so it makes up more than half of cocotb-coverage's number, where in the standard's average it is one item of four. `top` comes to 81.48%, 22 of 27, because it counts `quirk`'s four bins too.

Lab 03, Part 2 does the same sums for the router's 41 bins, which come to 46.34% by cocotb-coverage's formula and 61% by the standard's. Twelve of those bins can't be hit by any test in that lab, and they count in both percentages, so in Lab 03 neither percentage can reach 100%, however good your stimulus is. The list of bins that were never hit is the same whichever formula you use, so read that list to find the holes, as you did above for the four `equal` bins.

Ignore bins and illegal values

In cocotb-coverage a coverpoint has no ignore bins. A value that matches none of its bins isn't counted anywhere, so the way to ignore a value is to leave it out of the bins. SimProc's ALU has no operations 6 and 7, and `alu.op` has no bins for them. A cross can ignore combinations of bins. `CoverCross` takes `ign_bins`, a list of label pairs such as `("full", "empty")`, and leaves those bins out of the cross and out of its count. In the course's reference testbench for the router, `labs/ref`, one cross has the seven combinations that the router can't produce in its `ign_bins`, so that cross has 13 bins, and the reference testbench's regression hits all 13. Lab 03 leaves its impossible bins in, and has you work out from the specification which ones they are.

A value that should never happen at all goes in a `CoverCheck`. At each sample it counts a failure if its `f_fail` function returns true, and otherwise a pass if its `f_pass` function does. The example's check fails on an operation the ALU doesn't have, and its `f_pass` passes every other sample:

```
@CoverCheck("quirk.op_legal", f_fail=lambda op, a, b: op > 5, f_pass=lambda op, a,
b: True)
```

`CovIllegalTest` sends `CovPlainTest`'s kind of vectors and then one vector with `ALUop` 6. Among its `[COV]` lines are:

```
[COV] quirk: 2 of 4 bins, 50.00%
[COV] quirk.a_no_rel: 2 of 3 bins, 66.67%
[COV] quirk.op_legal: 0 of 1 bins, 0.00%
```

and the check's counts, `{'PASS': 300, 'FAIL': 1}`. The run still ended `ok`, because a `CoverCheck` raises no error. The check is one of `quirk`'s four bins, and a check that has failed counts as an empty bin, so

`quirk` is at 2 of 4 here, where `CovPlainTest`'s run had 3 of 4. If the check were in `alu`, its failure would take a bin off `alu`'s count in the same way. Then `make cov` merged all three runs:

```
top.quirk                                  100.00 %   4 of 4 bins
  top.quirk.a_no_rel                        66.67 %   2 of 3 bins
      not hit: other
  top.quirk.op_legal                       200.00 %   2 of 1 bins
```

The merge counted the check's `PASS` and `FAIL` as two bins hit, out of a size of 1, so a check that failed came out at 200%. That lifted `quirk` to 100%, with `other` still empty below it. So read a check by its `FAIL` count. If an illegal value should fail the test, give the scoreboard a check of its own for it, rather than leaving it to the coverage. (In the merged report of the passing runs, `make cov` lists the check's `FAIL` bin as "not hit", which for a check is the result you want.)

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. In Lab 03, seven of the router's cross bins are like that. Three of them, such as `('full', 'empty')`, pair a FULL drop with a level of 8 or less, and the router drops a packet as FULL only when the output its route names holds more than 8 beats. The other four pair a forwarded packet with a level of 9 or more, and the router admits a packet only at a level of 8 or less. 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', 'low')` stays empty until Lab 04 gives output 1 its real level, and no test in Lab 03 asks for a FULL drop at all. 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 for an output's level has `bins=[0, (1, 7), 8, 9, (10, 16)]` and no `rel=`. A run's tracker says the levels went up to 12. Which bins can be hit, and what do you change?

   Answer: A1, in Answers at the end.

2. After a regression, `make cov` says a group is at 100%, but one bin under it is listed as not hit. How can that be?

   Answer: A2, in Answers at the end.

3. After `CovPlainTest`, `make cov` reports `19 of 23 bins` for `alu`. Which of the four empty bins would more seeds of the same test fill?

   Answer: A3, in Answers at the end.

Further reading

For cocotb-coverage's API, go to its own documentation. ChipVerify's coverage pages explain the same ideas in SystemVerilog's covergroup syntax. Where they give a percentage, it's the standard's average, not cocotb-coverage's share of bins hit.

- cocotb-coverage's Tutorials, for `rel=` and `ign_bins`, under "Translating SystemVerilog Constructs to cocotb-coverage". Correction: in "Using CoverCheck as Assertions", the callback that should fail the test is a `lambda` with a `raise` statement in it, which isn't valid Python. Write it as a function with `def`.
- cocotb-coverage's Reference, for `CoverPoint`, `CoverCross`, `CoverCheck` and `merge_coverage()`. It says that a failed `CoverCheck` is set to 0 "permanently". That holds within one run; after `merge_coverage()`, the example's failed check came out at 200%.
- ChipVerify's Covergroup and Coverpoint, for what bins are for.
- ChipVerify's Cross Coverage. On that page, a value that falls in `illegal_bins` raises an error at run time. cocotb-coverage's nearest thing, a `CoverCheck`, raises none, as `CovIllegalTest` shows.

Next: Lab 03: Constraints and coverage.

Answers

A1. Check yourself

Only the three written as numbers, 0, 8 and 9. No level is equal to a tuple, so the two range bins stay empty, with no warning. Write every bin as a range, `(0, 0)`, `(1, 7)`, `(8, 8)`, `(9, 9)` and `(10, 16)`, and give the coverpoint a `rel=` that tests `low <= value <= high`.

A2. Check yourself

Something under the group counts more than its size. In the example, a `CoverCheck` that failed in one run of three came out as `2 of 1` after the merge, and that extra bin made up for the empty `other`. Read each coverpoint, cross and check on its own, and read a check by its `FAIL` count rather than by the group's percentage.

A3. Check yourself

All four, but slowly. They are `equal` bins, and only about one vector in 256 has equal operands, so each operation gets one about once in every 1,500 vectors. A test that makes `A == B` on purpose, as `CovBiasedTest` does, fills them in one run.

11. Lab 03: Constraints and coverage

You'll use cocotb-coverage to add functional coverage to the Lab 02 testbench, and read the coverage to find which kinds of drop and which output levels the testbench's random packets never produce. Then you'll add constraints to the packet so that pyvsc's solver makes 1-beat, 2-to-7-beat and 8-beat packets in the proportions you ask for. Last, you'll write stimulus that reaches the two coverage holes, because that's where this lab's planted bugs, B07 and B08, 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.py` and `scoreboard.py` in `tb/`. Around them it adds a packet that pyvsc randomizes, a sink that can stall, an APB read, a tracker that predicts what the router does with each packet, and a check on `DROP_CNT`, and Part 1 walks you through each of them. If your own monitor and scoreboard passed Lab 02's `make check-all`, you can copy them over ours. The scoreboard's error messages on this page come from ours, so yours may read differently. Run the copy from the lab folder, which Part 1's first step takes you to:

```
cp  ../lab-02-a-layered-testbench-without-pyuvm/tb/monitor.py  ../lab-02-a-layered-testbench-without-pyuvm/tb/scoreboard.py tb/
```

- Read Constrained random with pyvsc 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.

Time: about 1 h 30 min. Every file you change is Python, so once Part 1 has built the simulator, none of your changes makes it rebuild. A run of the graded test took us about 3 s as shipped and about 5 s once Part 3's constraints were in, against 0.25 s for Lab 02's test.

Part 1: The venv and the testbench you start from

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

1. Open a terminal on the workbench desktop, as in Lab 02, and go to the lab folder:

```
. /foss/designs/common/.designinit   # in every new terminal
mod dd105-b
cd labs/lab-03-constraints-and-coverage
```

2. The packets in this lab come from pyvsc, a Python library in which you write constraints and a solver picks values that meet them, and the coverage is counted by cocotb-coverage. Neither is in the course image, so make the course venv, a Python virtual environment that holds both:

```
make venv
```

   You should see:

```
=============================================================================
 VENV  /foss/designs/.venvs/dd105-b
   from /foss/designs/modules/dd105-b/labs/common/requirements.txt
=============================================================================
 imports: vsc, cocotb_coverage, cocotb 2.0.1 pyuvm 4.0.1
 created in 12.2 s, 37M
```

```
cocotb                          2.0.1
cocotb-coverage                 2.0
pyuvm                           4.0.1
pyvsc                           1.1.0
```

You make it once, for this lab and the ones after it. pip downloads the two packages and the ones they depend on, so this step needs the network. The venv is in your workspace under `/foss/designs`, so it survives a restart of the container. It sees the image's own packages too, which is why it lists the image's cocotb 2.0.1 and pyuvm 4.0.1 beside pyvsc 1.1.0 and cocotb-coverage 2.0. Every run in this lab uses the venv's Python. Until the venv exists, `make check-all`, `make cov` and `make traps` stop at once with the message below, and `make run` builds the simulator first and then stops with the same message:

```
======================================================================
 This lab needs the course venv (pyvsc and cocotb-coverage), and it isn't there:
   /foss/designs/.venvs/dd105-b
 Create it once with:  make venv
======================================================================
```

3. Build the simulator:

```
make build
```

It ends with:

```
  built  /foss/designs/modules/dd105-b/labs/lab-03-constraints-and-coverage/build/
sim_build/Vtop  in  2.1  s  (1.9M  in  /foss/designs/modules/dd105-b/labs/lab-03-
constraints-and-coverage/build/sim_build)
 ccache:  Hits: 17 / 28 (60.71%)  Misses: 11 / 28 (39.29%)
```

This is the same router as Lab 02′s, with the same fifteen bugs behind `+BUG`. The ccache's counts add up over every build in your clone, as in Lab 02. Ours include Lab 01′s and Lab 02′s builds, which came to 11 hits and 10 misses, so this build's own were 6 hits and 1 miss. The one file it compiled is `Vtop__ALL.o`, from the C++ that Verilator wrote for the router. Verilator writes the RTL's file names, paths included, into that C++, and this lab's copy of the RTL is in its own `build/rtl/`, so the C++ differs from Lab 02′s and the ccache had no compiled copy of it.

4. Read the files in `tb/`. Most of them are Lab 02′s:

File	What it is
`monitor.py`, `scoreboard.py`	Lab 02′s monitor and scoreboard: our solution, or yours
`driver.py`	Lab 02′s `Driver`, `Sink` and `ApbMaster`, with two additions: `ApbMaster.read()`, and `Sink.stall`, which holds the sink's ready at 0 from the next rising edge on, for as long as it is `True`
`packet.py`	Lab 02′s `Packet` and `RouterCfg`, and a new `RandPacket`, whose DST (`hdr_dst`) and length kind (`kind`) come from pyvsc's `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`. `RouterCfg` still reads `+N`, `+ROUTES` and `+SHOW`, but not Lab 02′s `+DSTS` or `+LEN`, because `RandPacket` now picks the DST and the length.
`generator.py`	randomizes `RandPackets`, where Lab 02′s generator called Python's `random`

File	What it is
`tracker.py`	new: follows output 0's level and decides what happens to each packet
`coverage.py`	new: `sample_pkt()`, which the tracker calls once for every packet and which has no coverpoints yet, and `report()`, which writes the coverage file at the end of the test. Part 2 adds the coverpoints.
`env.py`	Lab 02's `Env`, with `run()` split into steps a test can call one by one, and `check_drops()`
`test_router.py`	`router_rand_test`, `router_fill_test` and Lab 02's `router_class_test`, which now checks `DROP_CNT` at the end too. `close_holes()` is empty: Part 4 fills it in.

The tracker reads the pins of the input and of output 0 at the falling edge of every cycle, as the monitors do. Within a cycle it handles the input handshake before the output handshake, so the level that a header sees counts only the handshakes of earlier cycles, which is how the Routing section defines the level. From the routing table and that level, the tracker decides each packet's fate: NOROUTE, FULL, or forwarded to output 0 or 1. Nothing in the testbench watches output 1, and its sink is always ready, so its level stays far below 8, and the tracker takes it as 0. The tracker counts the drops it predicts, and `Env.check_drops()` reads `DROP_CNT` over APB and logs an error if the router's count and the tracker's 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. It ends with an `assert` like the one in Lab 02's test, which now fails the test if the driver, the scoreboard or the `DROP_CNT` check logged an error.

5. Run the test:

```
make run
```

After cocotb's start-up lines you should see:

```
   0.00ns INFO    cocotb.regression            running test_router.router_rand_test
(1/1)
    50.00ns INFO     tb.test                           [PHASE] 1 of 4: routes 0-11,
200 packets
 14410.00ns INFO     tb.env                               [DROP] DROP_CNT 00000000
as expected
 14440.00ns INFO     tb.test                           [PHASE] 2 of 4: routes 0-1-,
200 packets
 28470.00ns INFO     tb.env                               [DROP] DROP_CNT 00000000
as expected
 28500.00ns INFO     tb.test                           [PHASE] 3 of 4: routes -011,
200 packets
 42490.00ns INFO     tb.env                               [DROP] DROP_CNT 00000000
as expected
 42520.00ns INFO     tb.test                           [PHASE] 4 of 4: routes 1-01,
200 packets
 57060.00ns INFO     tb.env                               [DROP] DROP_CNT 00000000
as expected
 57130.00ns INFO     tb.env                               [DROP] DROP_CNT 00000000
as expected
 57130.00ns INFO     tb.env                           [ENV] sent 800 packets (3621
```

```
beats); output 0 took 1408 beats and output 1 took 2213
 57130.00ns INFO     tb.trk                              [TRK] 800 packets: 306 to
output 0, 494 to output 1, 0 NOROUTE, 0 FULL; the highest level a header saw was 2
 57130.00ns INFO     tb.sb                               [SB] 800 packets in, 306
predicted for output 0, 306 checked, 0 errors
 57130.00ns INFO     tb.env                              [COV] no coverpoint named
router.pkt.* yet, so no coverage.xml
 57130.00ns INFO    cocotb.regression            test_router.router_rand_test passed
```

and last, `ok: passed  4.14 s  log: build/runs/router_rand_test-s1/run.log`. A second run took us 3.04 s. The `[PHASE]` lines give the routes in the form Lab 02's `+ROUTES` took, one character per route: `0` or `1` for the output, and `-` for disabled.

6. This lab's two bugs are B07 and B08. B07 counts a packet for a disabled route as a FULL drop as well as a NOROUTE drop when the output that 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                        passed                        3.0s
   ------------------------------------------------------------------------------
    MISSED  tier 2  B07_full_counted_on_disabled_route passed                     3.0s
    MISSED  tier 3  B08_admission_off_by_one        passed                        3.1s
   ------------------------------------------------------------------------------
   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 5, 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: Coverage you can read

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

1. Read `RandPacket` in `tb/packet.py`: how `kind` is declared, and `post_randomize()`, which pyvsc calls after every `randomize()`. 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.py`. The tracker calls `sample_pkt()` once for every packet, with a `PktRecord` (declared in `tracker.py`) that holds the packet's length, its DST, its fate (`NOROUTE`, `FULL`, `OUT0` or `OUT1`, which are 0 to 3), and the level of the output that its route names, in the cycle in which its header was accepted. In cocotb-coverage a coverpoint is a decorator on the function that receives the values. The coverpoint's `xf=` takes its value out of the function's arguments, and each call of the function adds one to the bin that the value falls in. The length's coverpoint is:

   ```
   @CoverPoint("router.pkt.len", xf=lambda rec: rec.length, bins=list(range(1, 9)),
               bins_labels=[f"len{n}" for n in range(1, 9)])
   ```

Put it above `def sample_pkt(rec):`, and add the others under it, with these names and labels, because the outputs below use them:

Coverpoint	Value	Bins	Labels
`router.pkt.len`	`rec.length`	1 to 8	`len1` to `len8`
`router.pkt.dst`	`rec.dst`	0 to 3	`dst0` to `dst3`
`router.pkt.fate`	`rec.fate`	0 to 3	`noroute`, `full`, `out0`, `out1`
`router.pkt.level`	`rec.level`	`[0, (1, 7), 8, 9, (10, 16)]`	`empty`, `low`, `at_8`, `at_9`, `high`

and last the cross of the fate and the level:

```
@CoverCross("router.pkt.x_fate_level",
items=["router.pkt.fate", "router.pkt.level"])
```

Write the level's bins exactly as the table gives them, with the two ranges as tuples.

3. Each run's `report()` writes the hit count of every bin to `coverage.xml` in its run folder, and `make cov` merges every `coverage.xml` under `build/runs` with cocotb-coverage's `merge_coverage()`, so a run left over from earlier would be counted too. Delete the old runs first, then run the test and read its coverage:

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

The run's `[COV]` line says `router.pkt: 17 of 41 bins hit`, and `make cov` prints:

```
**********************************************************************
 COVERAGE   1 run, merged by cocotb-coverage into build/coverage/merged.xml
**********************************************************************
  top                                           41.46 %   17 of 41 bins
    top.router                                    41.46 %   17 of 41 bins
      top.router.pkt                                41.46 %   17 of 41 bins
        top.router.pkt.len                           100.00 %   8 of 8 bins
        top.router.pkt.dst                           100.00 %   4 of 4 bins
        top.router.pkt.fate                           50.00 %   2 of 4 bins
            not hit: noroute, full
        top.router.pkt.level                          20.00 %   1 of 5 bins
            not hit: low, at_8, at_9, high
        top.router.pkt.x_fate_level                   10.00 %   2 of 20 bins
                  not hit: ('noroute', 'empty'), ('noroute', 'low'), ('noroute',
'at_8'), ('noroute', 'at_9'), ('noroute', 'high'), ('full', 'empty'), ('full',
'low'), ('full', 'at_8'), ('full', 'at_9'), ('full', 'high'), ('out0', 'low'),
('out0', 'at_8'), ... (6 more)
**********************************************************************
```

The cross names each of its 20 bins by the pair of labels it stands for. The `[TRK]` line says headers saw levels up to 2, so some packets were decided at a level of 1 or 2. Why is `low` empty?

Answer: A2, in Answers at the end.

4. Write every level bin as a range, `bins=[(0, 0), (1, 7), (8, 8), (9, 9), (10, 16)]`, and give the coverpoint `rel=lambda value, rng: rng[0] <= value <= rng[1]`. Then delete the runs, run and merge again. The `[COV]` line now says `router.pkt: 19 of 41 bins hit`, and `make cov` prints:

```
  top                                              46.34 %   19 of 41 bins
    top.router                                       46.34 %   19 of 41 bins
      top.router.pkt                                   46.34 %   19 of 41 bins
        top.router.pkt.len                              100.00 %   8 of 8 bins
        top.router.pkt.dst                              100.00 %   4 of 4 bins
        top.router.pkt.fate                              50.00 %   2 of 4 bins
            not hit: noroute, full
        top.router.pkt.level                             40.00 %   2 of 5 bins
            not hit: at_8, at_9, high
        top.router.pkt.x_fate_level                      15.00 %   3 of 20 bins
                  not hit: ('noroute', 'empty'), ('noroute', 'low'), ('noroute',
'at_8'), ('noroute', 'at_9'), ('noroute', 'high'), ('full', 'empty'), ('full',
'low'), ('full', 'at_8'), ('full', 'at_9'), ('full', 'high'), ('out0', 'at_8'),
('out0', 'at_9'), ... (5 more)
```

5. `make cov` names only the empty bins, but the merged file, `build/coverage/merged.xml`, has every bin with its count. Look at the length's bins:

```
grep 'pkt.len.bin' build/coverage/merged.xml
```

```
        <bin0 bin="len1" hits="263" abs_name="top.router.pkt.len.bin0" />
        <bin1 bin="len2" hits="46" abs_name="top.router.pkt.len.bin1" />
        <bin2 bin="len3" hits="42" abs_name="top.router.pkt.len.bin2" />
        <bin3 bin="len4" hits="41" abs_name="top.router.pkt.len.bin3" />
        <bin4 bin="len5" hits="51" abs_name="top.router.pkt.len.bin4" />
        <bin5 bin="len6" hits="41" abs_name="top.router.pkt.len.bin5" />
        <bin6 bin="len7" hits="53" abs_name="top.router.pkt.len.bin6" />
        <bin7 bin="len8" hits="263" abs_name="top.router.pkt.len.bin7" />
```

   Compare the two 263s 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: A3, in Answers at the end.

6. 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: A4, in Answers at the end.

Check: `router.pkt: 19 of 41 bins hit`, and `len1` at 263.

Part 3: Constraints the solver honours

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

1. Open `traps/traps.py`. It has eight experiments, numbered 1 to 8, and each one randomizes a small pyvsc class over and over, most of them 1000 times. Each experiment prints one or more lines that start with its number, so this page calls them lines 1 to 8. Lines 1 to 3 use the shapes of this lab's packet, and lines 4 to 6 use shapes that a test might use. Line 7 times the solver, and line 8 shows when a pyvsc object takes its random state. pyvsc's constraints are meant to work as SystemVerilog's do, so for each of lines 1 to 6, write down what SystemVerilog's rules say it should show. DD105-A ran the same experiments on Verilator's solver, which got several of them wrong.

2. Run it:

```
make traps
```

   It runs with the venv's Python and no simulator, and took us 13 s. You should see:

```
1  three names, no constraint names kind     LEN_ONE  336  LEN_MID  342  LEN_MAX  322
1  four names, no constraint names op         OP_ADD  220  OP_SUB  252  OP_AND  275
OP_OR  253
2  kind dist, weights on LEN_ONE ... LEN_MAX  LEN_ONE  257  LEN_MID  505  LEN_MAX  238
2  kind dist, weights on 0, 1, 2             LEN_ONE  248  LEN_MID  515  LEN_MAX  237
3  length dist {...} and length != 7           randomize() failed 0 times in 1000;
1:280 2:91 3:81 4:100 5:83 6:94 8:271
3  kind dist, length from kind, length == 3  randomize() failed 0 times in 1000; 3:1000
4  solve_order(big, length)                    big in 469 of 1000
4  the same without solve_order                big in 342 of 1000
5  lens.sum == 8, list of 8                    randomize() failed 0 times in 20; the
sum was 8 in 20
5  lens.sum == 8, list of random size          randomize() failed 0 times in 20; the
sum was 8 in 20
6  unique on a list of four 4-bit values       a value repeated in 0 of 1000
6  unique on w, x, y, z                        a value repeated in 0 of 1000
7  T3Sel randomize(), per second              one object 571; a new object each time
521; random.randint() 2804954
8  seed 7, make T3Sel, 6 x randomize()         [6, 7, 2, 1, 8, 8]
8  the same, seed 99 before the first          [1, 7, 5, 3, 1, 8]
8  the same, seed 99 after the first           [6, 7, 2, 1, 8, 8]
======================================================================
 traps: 13.0 s, seed 1
======================================================================
```

Compare each line with what you wrote.

Answer: A5, in Answers at the end.

3. Now give the packet its constraints. In `RandPacket` in `tb/packet.py`:
   - make `length` random: `self.length = vsc.rand_uint8_t()`;
   - add three weights in `__init__`, `self.w_one = vsc.uint8_t(25)`, `self.w_mid = vsc.uint8_t(50)` and `self.w_max = vsc.uint8_t(25)`. The solver reads them at every `randomize()`, so a test can change them first;
   - add a constraint `c_kind` with a `dist` on `kind`:

```
@vsc.constraint
def c_kind(self):
    vsc.dist(self.kind, [vsc.weight(LenKind.LEN_ONE, self.w_one),
                         vsc.weight(LenKind.LEN_MID, self.w_mid),
                         vsc.weight(LenKind.LEN_MAX, self.w_max)])
```

   - add a constraint `c_len` that picks `length` from `kind`, with `vsc.if_then`, `vsc.else_if` and `vsc.else_then`: `self.length == 1` for `LEN_ONE`, `self.length.inside(vsc.rangelist((2, 7)))` for `LEN_MID`, and `self.length == 8` for `LEN_MAX`;
   - in `post_randomize()`, delete the `if`, `elif` and `else` branches that set `length`, so that only the call to `build()` is left.
4. Before you run, write down how many of the 800 packets you expect in each length bin. Then run and read the coverage:

```
rm -rf build/runs
make run
make cov
grep 'pkt.len.bin' build/coverage/merged.xml
```

With the reference solution's `coverage.py` and `packet.py` you should see:

```
        <bin0 bin="len1" hits="218" abs_name="top.router.pkt.len.bin0" />
        <bin1 bin="len2" hits="62" abs_name="top.router.pkt.len.bin1" />
        <bin2 bin="len3" hits="78" abs_name="top.router.pkt.len.bin2" />
        <bin3 bin="len4" hits="63" abs_name="top.router.pkt.len.bin3" />
        <bin4 bin="len5" hits="82" abs_name="top.router.pkt.len.bin4" />
        <bin5 bin="len6" hits="53" abs_name="top.router.pkt.len.bin5" />
        <bin6 bin="len7" hits="42" abs_name="top.router.pkt.len.bin6" />
        <bin7 bin="len8" hits="202" abs_name="top.router.pkt.len.bin7" />
```

and the run's `[ENV]` line says `sent 800 packets (3466 beats)`. The three kinds came out 218, 380 and 202 times, close to the 200, 400 and 200 you asked for. The run took us 4.67 s. Your own constraints may give different counts with the same seed, but they should be about as close.

5. The six lengths from 2 to 7 share their kind's 380 packets, and came out between 42 and 82 times each. What would SystemVerilog's rules give for `self.length.inside(vsc.rangelist((2, 7)))`, and how would you find out whether 42 against 82 is more than chance?

   Answer: A6, 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: A7, in Answers at the end.

2. In `tb/test_router.py`, where the first `TODO (Part 4)` is, write a pyvsc class `FillPlan` with `self.lens = vsc.rand_list_t(vsc.uint8_t(), sz=8)`, and a constraint that keeps each length at most 8 (`with vsc.foreach(self.lens, idx=True) as i:` and `self.lens[i] <= 8`) and makes `self.lens.sum == 8`. A 0 means "no packet". Line 5 of the experiments showed that pyvsc solves a sum on a list of fixed size and on one of random size, so a list of random size would work too, and the fixed size of 8 is only our choice.

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

   - `await env.write_routes("0---")`: `ROUTE0` enabled to output 0, the other three disabled (with `PORT` 0);
   - `env.snk0.stall = True`, then `await env.cycles(2)`, so that `out0_ready` is already 0 before any beat can reach output 0;
   - randomize a `FillPlan`, and for each length `n` that isn't 0, put `RandPacket.make(0, n)` on a list of packets;
   - one random packet for DST 0: `p = env.gen.new_packet()`, then `with p.randomize_with() as it:` and `it.hdr_dst == 0`, and onto the list;
   - one random packet for DST 1, the same way but with `q.c_dst.constraint_mode(False)` before it is randomized, because `c_dst` would otherwise forbid a disabled route and the solver would find no solution;
   - `await env.send(pkts)`, which returns when the last packet has gone in; then `env.snk0.stall = False` and `await env.drain()`.

   Both tests that call `close_holes()` check `DROP_CNT` after it returns, so `close_holes()` doesn't need a check of its own. `RandPacket.make()` builds a fill packet without the solver, with its beats back to back. You could also make each fill packet with `randomize_with()` and `it.length == n` (line 3 of the experiments showed that this works), but `build()` would then put idle cycles before about one beat in four. A log line 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 `close_holes()` sent one more random packet for DST 0 at that point. What would the golden router do with it, and what would this testbench report? Write down your answer before you open ours.

   Answer: A8, in Answers at the end.

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

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

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:

```
   190.00ns INFO     tb.test                            [HOLES] output 0 stalled and
filled to 8 by packets of 8 beats, then 2 beats for DST 0 and 1 for DST 1
   840.00ns INFO     tb.env                               [DROP] DROP_CNT 00000001
as expected
   840.00ns INFO     tb.env                               [ENV] sent 3 packets (11
beats); output 0 took 10 beats and output 1 took 0
   840.00ns INFO     tb.trk                             [TRK] 3 packets: 2 to output
0, 0 to output 1, 1 NOROUTE, 0 FULL; the highest level a header saw was 10
   840.00ns INFO     tb.sb                            [SB] 3 packets in, 2 predicted
for output 0, 2 checked, 0 errors
   840.00ns INFO     tb.env                             [COV] router.pkt: 13 of 41
bins hit; coverage.xml written
```

and `ok: passed  1.79 s`. On seed 1 this plan is a single 8-beat packet. If your run fails here, read the first error in the log. The router is the golden one, so the mistake is in your stimulus.

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

```
make run
make cov
```

The `[HOLES]` line of `router_rand_test` says `filled to 8 by packets of 2 2 3 1 beats, then 7 beats for DST 0 and 7 for DST 1`. That's a different plan from `router_fill_test`'s on the same seed, because cocotb seeds Python's `random` for each test from the seed and the test's name, and because `router_rand_test` has drawn 800 packets from `random` before `close_holes()` starts. `make cov` reports:

```
  top                                          58.54 %   24 of 41 bins
    top.router                                   58.54 %   24 of 41 bins
      top.router.pkt                               58.54 %   24 of 41 bins
        top.router.pkt.len                          100.00 %   8 of 8 bins
        top.router.pkt.dst                          100.00 %   4 of 4 bins
        top.router.pkt.fate                          75.00 %   3 of 4 bins
            not hit: full
        top.router.pkt.level                         80.00 %   4 of 5 bins
            not hit: at_9
        top.router.pkt.x_fate_level                  25.00 %   5 of 20 bins
           not hit: ('noroute', 'empty'), ('noroute', 'low'), ('noroute', 'at_8'),
('noroute', 'at_9'), ('full', 'empty'), ('full', 'low'), ('full', 'at_8'), ('full',
'at_9'), ('full', 'high'), ('out0', 'at_9'), ('out0', 'high'), ('out1', 'low'), ...
(3 more)
```

That leaves 17 bins never hit, and five of the 29 bins that this testbench can reach are among them. Which are they, and what would hit each one?

Answer: A9, 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                          passed                          4.7s
--------------------------------------------------------------------------------
  caught  tier 2  B07_full_counted_on_disabled_route AssertionError                  4.8s
  caught  tier 3  B08_admission_off_by_one           AssertionError                  4.7s
--------------------------------------------------------------------------------
 2 of 2 caught.
 Every bug in the list, each one caught by a check that fired.
```

`make run BUG=B07` shows the check that fired on B07:

```
 56020.00ns ERROR    tb.env                                  DROP_CNT read 00000101,
expected 00000001 (FULL_CNT 0, NOROUTE_CNT 1)
```

`FULL_CNT` is bits 15:8 and `NOROUTE_CNT` is bits 7:0, so the router counted the packet for DST 1 as NOROUTE and as FULL. `make run BUG=B08` logs the same `DROP_CNT` line first, because B08 counts the packet for DST 0 that it drops at level 8 as a FULL drop. Then comes the scoreboard's report of that packet:

```
 56020.00ns ERROR    tb.sb                                dst 0, 7 beats: 24 28 19 4b
53 55 0b, which went in at 55355 ns, never came out of output 0
```

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

```
make waves TEST=router_fill_test
make waves TEST=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`, `in_valid`, `in_data`, `in_last`, `out0_valid`, `out0_ready`, `out0_data` and `level0` from `router`, and look at 160 to 420 ns. `level0` is the router's own count of the beats in output 0, which is the value that `LEVEL` reads in its `LEVEL0` field. Surfer shows it in hex.

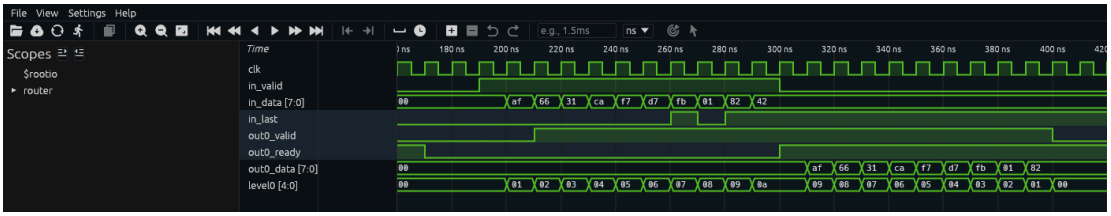


Figure 11.1: router_fill_test on the golden router, 160 to 420 ns, in Surfer

The golden router. `out0_ready` falls at 170 ns. The fill packet `00 af 66 31 ca f7 d7 fb` goes in from 190 ns (`in_data` already held `00`, so only `in_valid` shows the header), and `level0` counts up to 08. The header `01` is accepted at level 8, so its packet `01 82` is admitted and `level0` climbs to 0a. The packet `42`, for DST 1, is dropped. `out0_ready` rises at 300 ns, and output 0 delivers all 10 beats.

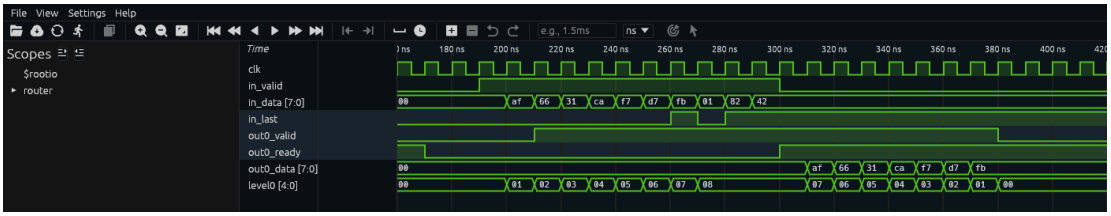


Figure 11.2: The same run with B08 switched on

B08. The same packets go in, and `level0` stops at 08, because the packet `01 82` is dropped. Output 0 delivers 8 beats.

Which rising edge ends the cycle in which the header `01` is accepted, what is `level0` in that cycle, and what does `level0` do at that edge on each router?

Answer: A10, 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 coverpoints in `tb/coverage.py`, your constraints in `tb/packet.py` and your `close_holes()` in `tb/test_router.py`.

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: A11, in Answers at the end.

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

   Answer: A12, in Answers at the end.

Submit: Nothing to submit.

Going further (optional)

- Run `make traps SEED=2` and see which lines move. Then change a class in `traps/traps.py`, predict its line, and run it again.
- Hit the five reachable bins that Part 4 step 6 lists, and check with `make cov`.
- Give the cross the seven impossible bins as `ign_bins`, a list of label pairs such as `("full", "empty")`, and see what happens to the number of bins `make cov` counts.

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

Answers

A1. Part 1: The venv and 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 that 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: Coverage you can read

cocotb-coverage puts a value in a bin when `value == bin`, unless the coverpoint is given a `rel=` function that says what "in the bin" means. The bins 0, 8 and 9 are numbers, so a level of 0 lands in `empty`. The bins `(1, 7)` and `(10, 16)` are tuples, and no number is equal to a tuple, so `low` and `high` can never be hit. cocotb-coverage doesn't warn you about it, so you have to find it by reading the list of empty bins against what the test did, as you just did with the `[TRK]` line.

A3. Part 2: Coverage you can read

SystemVerilog makes an unconstrained `rand` enum uniform over its named values, and pyvsc does the same for a `rand_enum_t`, so each of the three kinds should come out about 267 times. Here they came out 263, 274 and 263. The six lengths from 2 to 7 share `LEN_MID`'s 274, so each is about 45. A third of the packets are single beats and a third are 8 beats long only because the enum has three names. In Part 3 you give the kinds weights of your own.

A4. Part 2: Coverage you can read

B08 needs a packet admitted to output 0 at a level of exactly 8, which is `('out0', 'at_8')`. B07 needs a NOROUTE drop while the output that its route names holds more than 8 beats, which is `('noroute', 'at_9')` or `('noroute', 'high')`.

Seven bins are impossible by the specification, because a FULL drop happens only at a level of 9 or more (`('full', 'empty')`, `('full', 'low')`, `('full', 'at_8')`), and a packet is admitted only at 8 or less (`('out0', 'at_9')`, `('out0', 'high')`, `('out1', 'at_9')`, `('out1', 'high')`). This testbench can't hit two more, `('out1', 'low')` and `('out1', 'at_8')`, because the tracker takes output 1's level as 0. Three more, `full`, `('full', 'at_9')` and `('full', '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. cocotb-coverage's 46.34% is the number of bins hit over all 41 bins, the cross's included. SystemVerilog's formula instead averages the coverage of the five items (100, 100, 50, 40 and 15%), which gives 61%.

A5. Part 3: Constraints the solver honours

1. An unconstrained enum is uniform over its names, so about 333 each for three names and 250 each for four, and the counts came out close to that. (In DD105-A, Verilator's enum whose four names fill its two bits came out as its first name 809 times in 1000.)
2. Both should give about 250, 500 and 250, and both did. In DD105-A, Verilator gave the weight of 50 to `LEN_MAX` when the weights were written with the enum's names.
3. Neither should fail. `length != 7` leaves a value in every weighted range, and with `length == 3` the solver should choose `LEN_MID`. Neither failed once, where Verilator, in DD105-A, failed 72 and 478 times in 1000.
4. With `solve_order`, `big` is chosen first, at 50/50, so about 500, and it came out 469. Without it, SystemVerilog makes each of the 8 solutions equally likely, so `big` is 1 in 8, about 125. pyvsc gave 342, so it doesn't choose evenly among the solutions.
5. A sum should work on a list of fixed size and on one whose size is random, and it worked on both.
6. Neither should repeat a value, and neither did.

Line 7 measures pyvsc's speed. A two-field class was randomized 571 times a second, about 1.8 ms a call, while `random.randint()` ran 2.8 million times a second. That's why the packet's payload bytes and idle cycles come from Python's `random`, in `build()`, and pyvsc picks only the packet's random fields: the DST, the kind and, once step 3 makes it random, the length.

Line 8 shows that a pyvsc object takes its random state from Python's `random` at its first `randomize()`. Seeding `random` again between making the object and that first call gave a different list from the first row's, and seeding it again after the first call gave the same list as the first row. The generator makes and randomizes every packet inside the test, after cocotb has seeded Python's `random` from the run's seed, so the seed decides every packet.

A6. Part 3: Constraints the solver honours

SystemVerilog makes each of the six equally likely, about 63 each of 380. In a sample this size some spread is chance, so draw more packets and see whether the same lengths stay high. We drew 3000 packets from this class outside the simulator, twice: lengths 2 to 5 came out 269 to 298 times each, and 6 and 7 came out 180 to 197 times, against about 250 for an even spread. So pyvsc's solver doesn't pick evenly among the values that a constraint allows, either here or in line 4 of the experiments. The constraint decides which lengths can come out, and to find out how often each one does, you count them with the length coverpoint.

A7. 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', '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', 'at_9')` or `('noroute', 'high')`, B07's bin), and the specification says it's never counted as FULL as well.

A8. 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 in a copy of the solution, with a 7-beat packet, and `router_fill_test` printed:

```
    910.00ns INFO     tb.env                                  [DROP] DROP_CNT 00000101
as expected
   910.00ns ERROR    tb.sb                              dst 0, 7 beats: 03 2f 67 b4 02
99 32, which went in at 365 ns, never came out of output 0
```

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.

A9. Part 4: Close the two holes

`at_9` and `('noroute', 'at_9')` need the packet for DST 1 to arrive while output 0 holds exactly 9 beats, which happens when the random packet for DST 0 before it is 1 beat long. A packet for a disabled route would hit `('noroute', 'empty')`, `('noroute', 'low')` or `('noroute', 'at_8')` if it arrived while output 0 held 0, 1 to 7 or 8 beats. Both of our packets for DST 1 arrived at a level of 10 or more, so they hit `('noroute', 'high')`.

A10. Part 4: Close the two holes

The edge at 280 ns. In the cycle that edge ends, `level0` is 08, because the eight beats of the fill packet have gone in and none has left. The specification admits a packet at a level of 8 or less, so the golden router writes `01` 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.

A11. 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 in the starter's runs the router with either bug switched on did the same thing as the golden router. Once the stimulus reached those bins, the same checks caught both bugs.

A12. Finishing

No. On every seed the sinks are always ready, so output 0 never reaches 8. More seeds help only with holes that the random stimulus can reach, and Part 1′s answer explains why it can't reach these two.

IV 4. pyuvm

12 **The pyuvm skeleton 99**
12 Objects and components 99
12 @pyuvm.test() and TEST= 100
12 Phases 100
12 Objections 102
12 After the run phase 103
12 Check yourself 103
12 Further reading 104
12 Answers 104

13 **Sequences, sequencers and drivers . 105**
13 The pieces 105
13 One item through the handshake 105
13 Two sequences on one sequencer 107
13 Sequences that start sequences 107
13 get_next_item() and item_done() 108
13 Check yourself 108
13 Further reading 108
13 Answers 109

14 **Monitors, analysis ports and scoreboards 111**
14 The monitor 111
14 Analysis ports 112
14 The scoreboard 113
14 One object per write 114
14 Coverage as a subscriber 114
14 Check yourself 115
14 Further reading 115
14 Answers 115

15 **The factory and ConfigDB 117**
15 The factory 117
15 ConfigDB 119
15 Check yourself 120
15 Further reading 121
15 Answers 121

16 **Lab 04: The same testbench in pyuvm 123**
16 Before you start 123
16 Part 1: Run the port 123
16 Part 2: Watch the second output 127
16 Part 3: Stall one output at a time 128
16 Part 4: A factory override for disabled routes . 129
16 Part 5: Count what the port cost 132
16 Finishing 133
16 Answers 133

12. The pyuvm skeleton

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

In Lab 01 you read a whole pyuvm testbench, the ALU's `tb/alu_uvm.py`, from its tests down to its driver, monitor and scoreboard, and you added the flag check to its scoreboard. Labs 02 and 03 were plain cocotb. Each test was a function marked `@cocotb.test()`. An `Env` class made the pieces in its constructor and handed them queues, and one of its methods then started each piece's coroutine. In Lab 04 you finish our pyuvm port of the Lab 03 router testbench, and in Part 5 of that lab you count its lines against those of your own Lab 03 testbench.

The example for this page, `guide-examples/the-pyuvm-skeleton/`, has no design to test. Lab 01's ALU is its toplevel only because cocotb needs one. 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 pyuvm calls them.

Objects and components

Every pyuvm class in a testbench derives from 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 beat that a monitor saw. 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 `RouterPkt`, `StreamBeat` and `ApbItem` derive from `uvm_sequence_item`, which is a `uvm_object`. Its `OutCfg`, the knobs of one output agent, is a plain Python class, since nothing makes it through the factory or sends it through a port.

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 it when it's made, as in `SrcAgent.create("src", self)`, and the parents link the components into a tree. Lab 04's tree looks like this:

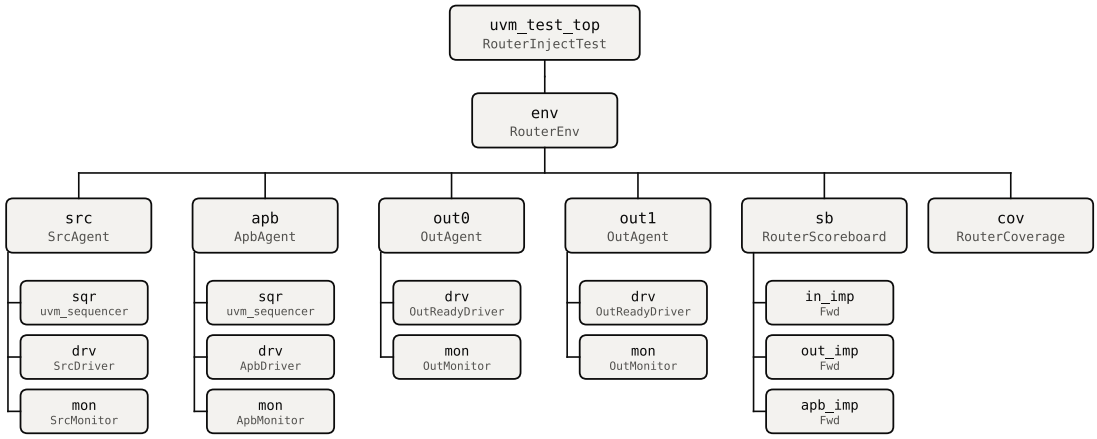


Figure 12.1: The components of Lab 04's testbench. Each box shows an instance name and, under it, the class.

The components that `RouterLabTest`, `RouterEnv` and the agents build in Lab 04, once you've added `out1` in Part 2, drawn from a printout of the solution's tree. The test drawn at the top is `RouterInjectTest`, one of the tests built on `RouterLabTest`. The printout also had the ports, which are left out here: each monitor's `ap`, the scoreboard's `fate_ap`, the `seq_item_port` of `SrcDriver` and of `ApbDriver`, each sequencer's

`seq_item_export`, and an `analysis_export` under each `Fwd` and under `cov`. When a component logs a message, the line names it by its path down this tree, such as `uvm_test_top.env.sb` for the scoreboard.

The test is at the top of everything the testbench builds, and pyuvm always names it `uvm_test_top`. Above it sits pyuvm's own root, `uvm_root`, which has no name in a path. The test makes the environment, `env`, and the environment makes the agents, the scoreboard and the coverage. 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.

In pyuvm, ports and exports are components too, so they're in the tree with everything else. The example's `mon` makes an analysis port called `ap`, and the tree printed below has it under `mon`. The course's `BaseTest` walks this whole tree, ports included, when it adds its error counter to every component's logger.

pyuvm has a class for each job: `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 it gets items through, `seq_item_port`, and `uvm_subscriber` adds the export it receives transactions on, `analysis_export`; the next two pages describe both. `uvm_agent`'s `build_phase` reads an `is_active` setting. `uvm_test`, `uvm_env`, `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, `OutReadyDriver` takes no items, so it derives from `uvm_component` rather than `uvm_driver`.

pyuvm has no registration lines like SystemVerilog UVM's `` `uvm_component_utils ``. It registers every class that derives from one of its own with the factory as soon as Python defines the class, under the class's name. A component made with `ClassName.create(name, parent)` goes through the factory, so a test can tell the factory to make something else in its place. In Lab 04 the environment, the agents, the drivers, the monitors, the scoreboard and the coverage are made that way. The sequencers, the scoreboard's `Fwd`s and the ports are made with the constructor, as in `uvm_sequencer("sqr", self)`, and a test can't swap those. The factory page explains how an override works.

@pyuvm.test() and TEST=

cocotb runs functions marked `@cocotb.test()`. `@pyuvm.test()` wraps a test class in such a function and gives the function the class's name. When cocotb calls that function, the function asks pyuvm's `uvm_root` to make an instance of the class, name it `uvm_test_top`, and run the phases. `make run TEST=SkPhaseTest` passes cocotb a filter that matches the test name `SkPhaseTest` exactly, and cocotb runs only the test whose name matches, so one build serves all of a testbench's tests.

`@pyuvm.test()` adds the cocotb test to the module that the decorator is written in, and cocotb looks for tests only in the modules that the lab's `TEST_MODULES` names. So write each test class, with its decorator, in one of those modules. A test module that imports a decorated class from another module doesn't get the class's test. A class that derives from a decorated test is a test only if it has a decorator of its own, as Lab 04′s `RouterStallTest`, which derives from `RouterRandTest`, does. The class that a test derives from can be in any module, as `BaseTest` is in `dd105b_base`.

If the name matches no test, nothing runs. `make run TEST=SkPhaseTset`, with two letters of the name swapped, printed:

```
     0.00ns WARNING  cocotb.regression                 No tests left after filtering
with: (^|\.)SkPhaseTset$
```

and the verdict `BROKEN: every test skipped`, with a hint to compare the name with `make tests`.

Phases

Where Lab 03′s `Env` started each piece's coroutine itself, pyuvm calls a fixed list of methods, the **phases**, on every component in the tree, and finishes each phase in the whole tree before it starts the next. A component overrides the phases it needs. pyuvm 4.0.1 has these nine, in this order:

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

Phase	Kind	Order in the tree	What goes in it
`end_of_elaboration_phase`	function	top down	print or check the finished tree
`start_of_simulation_phase`	function	top down	last settings before time moves
`run_phase`	`async`	all at once	everything that takes simulated time
`extract_phase`, `check_phase`, `report_phase`	function	top down	collect results, look for leftovers, print summaries
`final_phase`	function	top down	the very end

The function phases take no simulated time. `run_phase` is a coroutine, and pyuvm starts every component's `run_phase` at the same moment, each as a cocotb task of its own. SystemVerilog UVM runs every phase but `build_phase` and `final_phase` from the bottom up, and has twelve more run-time phases (`reset_phase`, `main_phase` and so on) beside `run_phase`. In pyuvm 4.0.1's `s09_phasing.py`, only `connect_phase` runs from the bottom up, the `run_phase` tasks are started in that order too, and there are no run-time phases. So if you read SystemVerilog UVM material, expect the order of the later phases to differ. Here is the start of `SkPhaseTest`'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.mon
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.a0
0    ns  build_phase              uvm_test_top.env.a0.child
0    ns  connect_phase            uvm_test_top.env.mon
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.a0.child
0    ns  connect_phase            uvm_test_top.env.a0
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 pyuvm can't call `env`'s `build_phase` until the test's has created `env`. Within each parent, pyuvm visits the children in the order the parent made them. `env` made `mon`, then `a1`, then `a0`, and that's the order in the output above. (DD105-A's SystemVerilog UVM library visited them in the order of their names.)

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 not in `build_phase`, because during its own `build_phase` the components below its children don't exist yet. pyuvm finishes `build_phase` in the whole tree before any `connect_phase` starts, so every component exists by the time it connects.

pyuvm has no method that prints the tree, so the example's test walks it in `end_of_elaboration_phase`, printing each component's name and class:

```
uvm_test_top           SkPhaseTest
  env                  SkEnv
    mon                SkWatcher
      ap               uvm_analysis_port
    a1                 SkLongAgent
      child            SkLongLeaf
    a0                 SkAgent
      child            SkLeaf
```

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` to return. Instead, pyuvm keeps the run phase going while any component holds an **objection**. A component that is in the middle of something calls `self.raise_objection()`, and calls `self.drop_objection()` when it's done, and the run phase ends as soon as every objection has been dropped. SystemVerilog UVM can also keep the phase open for a drain time after the last drop; pyuvm 4.0.1 has no drain time.

In `SkPhaseTest` the test holds an objection for 100 ns, `env.a1.child` holds one for 150 ns, and `env.mon` loops forever, printing every 40 ns:

```
0    ns  run_phase: still looping   uvm_test_top.env.mon
0    ns  run_phase starts           uvm_test_top.env.a1.child
0    ns  run_phase starts           uvm_test_top.env.a0.child
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
150  ns  extract_phase              uvm_test_top
```

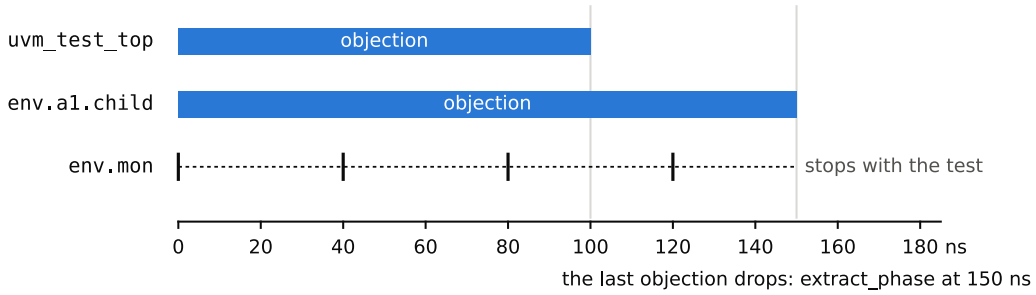


Figure 12.2: Objections in SkPhaseTest: the test's from 0 to 100 ns, a1.child's from 0 to 150 ns; the run phase ends at 150 ns.

The run phase of `SkPhaseTest`. When the test dropped its objection at 100 ns, `a1.child` still held one. The phase ended when that one dropped, at 150 ns, and the remaining phases ran in the same time step. `mon`'s loop, which had been running the whole time without holding the phase open, printed nothing more, because the test ended at 150 ns.

`SkNoObjTest` builds the same tree with no component holding an objection. Every `run_phase` still starts at 0 ns, pyuvm logs a warning, and the phase ends in the same time step:

```
0    ns  run_phase starts           uvm_test_top
      0.00ns WARNING   root                                    You did not call
self.raise_objection() in any run_phase
0    ns  extract_phase              uvm_test_top
```

A new 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, and the only sign of it that pyuvm gives is that warning. Lab 04's `RouterLabTest` raises its objection at the start of `run_phase`, resets the router, runs the test's `body()` and drops the objection when `body()` returns, or sooner if the watchdog below fires or the scoreboard logs its 50th error. The tests built on it override `body()` and leave `run_phase` alone, so none of them has to raise an objection itself.

A test that never drops its objection never ends on its own. `RouterLabTest` runs a watchdog beside `body()`, and if the test is still running after `+MAX_CYCLES` clock cycles (200,000 unless you set it), the watchdog logs an error that starts `[WATCHDOG]`, and the test ends. cocotb can also stop a test at a time limit that you set on the decorator. `SkHangTest`, marked `@pyuvm.test(timeout_time=1, timeout_unit="us")`,

raises an objection and waits 10 µs, and cocotb stopped it at 1,000 ns with the verdict `FAIL: SimTimeoutError`.

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 log an error for each output that still has beats it expected and never saw, and one more if it checked no beat at all. It uses `report_phase` to print the `[SB]` lines you'll read in every run.

pyuvm reports through Python's `logging`, and every component has a logger, `self.logger`. A logged error prints a line and fails nothing. `SkPlainErrorTest` derives from pyuvm's `uvm_test` and logs one error at 50 ns, and with `make run TEST=SkPlainErrorTest BASE_TEST=0` it printed:

```
    50.00ns ERROR    ..-pyuvm-skeleton/sk_tests.py(184) [uvm_test_top]: a check failed
at 50 ns
    50.00ns INFO     cocotb.regression                 sk_tests.SkPlainErrorTest passed
```

and the verdict `ok: passed`. The `ERROR` line has the shape of every message that the labs' components log: the time, the level, the file and line that logged it, the component's full name in brackets, and the text. cocotb cuts a long file path to fit its column, which is why the path starts with `..`.

Every pyuvm test in the labs derives from the course's `BaseTest`, which makes a logged error fail the test. In `end_of_elaboration_phase` it adds a counting handler to the logger of every component in the tree, and in `final_phase` it prints one line with the count and raises `AssertionError` if the count isn't 0. From Lab 05 on, the labs bind assertions into the router, and the same line also gives the number of assertion failures, which fails the test too if it isn't 0. The example binds no assertions, so its line says `no bound assertions`. `SkErrorTest` logs the same error and derives from `BaseTest`:

```
    50.00ns ERROR    ..-pyuvm-skeleton/sk_tests.py(175) [uvm_test_top]: a check failed
at 50 ns
DD105-B BaseTest: 1 error logged, no bound assertions
```

and the verdict was `FAIL: 1 error`, with `AssertionError: 1 error logged, the first: a check failed at 50 ns` under it. The graders count that `AssertionError` as a check that fired. They also call a test BROKEN if it passes without `BaseTest`'s line, as `make run TEST=SkPlainErrorTest` without `BASE_TEST=0` did (`BROKEN: no BaseTest line`), because a logged error in such a test would have failed nothing.

`final_phase` runs from the top down, so the test's `final_phase` runs before its children's. In `SkPhaseTest` the `DD105-B BaseTest:` line came straight after the `final_phase` line for `uvm_test_top` and before the one for `env`. An error that a component below the test logs in its own `final_phase` therefore comes after `BaseTest` has printed its count and decided whether to raise `AssertionError`. `SkFinalErrorTest` has a component that does that, and the run printed `DD105-B BaseTest: 0 errors logged` before the error and ended `ok: 0 errors`. So put your checks in `check_phase` or `report_phase`.

Check yourself

1. In Lab 04, `RouterEnv.build_phase` makes the agents and the scoreboard. Why can't it connect `self.src.mon.ap` to the scoreboard there, straight after making them?

   Answer: A1, in Answers at the end.

2. A test's `run_phase` starts a sequence of 200 packets and doesn't raise an objection. What happens, and which of Lab 04's checks would notice?

   Answer: A2, in Answers at the end.

3. A scoreboard compares its counts in `final_phase` and logs an error when they disagree. The run logs the error and passes, as `SkFinalErrorTest` did. Why?

   Answer: A3, in Answers at the end.

Further reading

- pyuvm's documentation walks through a whole testbench for a small ALU, from its tests to its monitor and driver, and UVM and Python explains what pyuvm changes from SystemVerilog UVM. The pages are for pyuvm's latest release, which is newer than the course's 4.0.1. The SV-UVM-style reporting they describe, which `set_sv_uvm_style_reporting_enabled()` turns on, isn't in 4.0.1.
- pyuvm's phases are in its source, `s09_phasing.py` at 4.0.1, and each phase class says whether it runs from the top down or the bottom up.
- ChipVerify's UVM Phases and UVM Objection describe SystemVerilog UVM. They give its order of phases, with every phase but build and final from the bottom up, and its twelve run-time phases, and neither matches pyuvm 4.0.1. Their objections take the component as an argument, `phase.raise_objection(this)`, where pyuvm's are the component's own methods, and pyuvm has no drain time.
- ChipVerify's UVM Component and UVM Object describe the two kinds of class, in SystemVerilog UVM.

Next: Sequences, sequencers and drivers.

Answers

A1. Check yourself

`src.mon` doesn't exist yet. `SrcAgent.build_phase` makes the monitor, and pyuvm calls it only after `RouterEnv.build_phase` has returned, because the build phase goes from the top down. The scoreboard's inputs are made in its own `build_phase` too. By `connect_phase`, every component in the tree has been built.

A2. Check yourself

The run phase ends at 0 ns, as `SkNoObjTest`'s did, before any packet reaches the pins, and pyuvm logs only a warning. Lab 04's scoreboard expected nothing and saw nothing, so it finds no leftover beats, but its `check_phase` also logs an error when it checked no beat at all, and `BaseTest` fails the test on it.

A3. Check yourself

`final_phase` runs from the top down, so `BaseTest`'s `final_phase` in `uvm_test_top` had already counted the errors, printed its line and returned before the scoreboard's `final_phase` ran. Move the comparison to `check_phase` or `report_phase`.

13. 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 cocotb `Queue`, and the driver took them out one at a time and drove them. In Lab 04 a **sequence** makes the packets, a **sequencer** hands them to the driver, and the **driver** puts each one on the pins. Where the generator called the queue's `put()`, the sequence calls `start_item()` and `finish_item()`, and where the driver called the queue's `get()`, it now calls `get_next_item()` and `item_done()`. The idle cycles that the driver leaves before beats, and the way it drives the pins, are the same as in Lab 03. The table in Lab 04's Part 5 goes through the two testbenches piece by piece.

This page follows one item from a sequence, through the sequencer, to the pins. Its example drives SimProc's ALU as Lab 01's pyuvm testbench did, holding each vector on the pins for one 10 ns period, and at every step it prints a line that starts with the simulation time. The example's files are in `guide-examples/sequences-sequencers-and-drivers/`.

The pieces

An item is an object of a class derived from `uvm_sequence_item`. Lab 04's item class is `RouterPkt`. The example's is `AluItem`, which holds an operation and two operands, and also holds the ALU's answer after the driver has put the vector on the pins. Both classes are decorated with `@vsc.randobj` as well, which makes them pyvsc classes, so a sequence can call `randomize()` on them.

A sequence is an object too, not a component. Its class derives from `uvm_sequence`, and its `body()` coroutine makes the items and sends them. A test makes a sequence when it needs one, sets its knobs, and runs it with `await seq.start(sequencer)`, which returns when `body()` has finished. A pyuvm sequence has no logger of its own. The example's sequences print their lines with `print()`, and the register sequences in the course's reference testbench, `labs/ref`, report through a component's logger.

The sequencer is a component inside the agent. Any number of sequences can send items to it, and it hands them to its one driver one at a time. You don't write a sequencer class in this course. In Lab 04 the input agent and the APB agent each make one of pyuvm's own, with `uvm_sequencer("sqr", self)`.

The driver is a component too. Lab 04's `SrcDriver` derives from `uvm_driver`, and that base class gives it a port called `seq_item_port`, which the agent connects to the sequencer in `connect_phase`:

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

The port is on the driver because the driver sets the pace. It asks for the next item only when it's ready to put one on the pins, and the sequence waits until it does.

One item through the handshake

Here are the loops in the example's sequence and driver, without the lines that print each step:

```
# AluRandSeq.body()
for _ in range(self.n):
    req = AluItem.create("req")
    await self.start_item(req)          # wait until the driver takes it
    req.randomize()
    await self.finish_item(req)         # hand it over; wait until the driver is done

# AluDriver.run_phase()
while True:
    req = await self.seq_item_port.get_next_item()   # wait for an item
    d.ALUop.value = int(req.op)
    d.A.value = int(req.A)
    d.B.value = int(req.B)
```

```
    vif.valid = True
    await Timer(vif.PERIOD_NS, "ns")    # the vector holds for one 10 ns period
    req.ALUout = int(d.ALUout.value)
    vif.valid = False
    self.seq_item_port.item_done()      # let finish_item() return
```

The example's files also tag each item with the name of the sequence that sent it, and that name is the `a` in the lines below. `SqTimelineTest` runs this sequence for three items, and these are the lines it prints up to the driver's `item_done()` for the second item:

```
0    ns  [drv]     get_next_item: waiting for an item
0    ns  [seq a]   start_item(req): waiting for the driver
0    ns  [seq a]   start_item returned; randomize()
0    ns  [drv]     get_next_item returned a: SHR A=bd B=14
10   ns  [drv]     item_done, ALUout was bd
10   ns  [drv]     get_next_item: waiting for an item
10   ns  [seq a]   finish_item returned: a: SHR A=bd B=14 is done
10   ns  [seq a]   start_item(req): waiting for the driver
10   ns  [seq a]   start_item returned; randomize()
10   ns  [drv]     get_next_item returned a: SHL A=ce B=eb
20   ns  [drv]     item_done, ALUout was 70
```

`start_item()` puts the item in the sequencer's queue and returns once the driver's `get_next_item()` has taken it. At time 0 the driver was already waiting, so `start_item()` returned at once. The sequence then randomized the item and called `finish_item()`, which hands the item over and waits. `get_next_item()` returned after that, in the same time step, because pyuvm's `get_next_item()` waits for `finish_item()` before it gives the driver the item. The driver held the vector on the pins for one period and called `item_done()` at 10 ns, which let `finish_item()` return in the same time step. So the sequence's loop runs at the driver's pace, which here is one item every 10 ns:

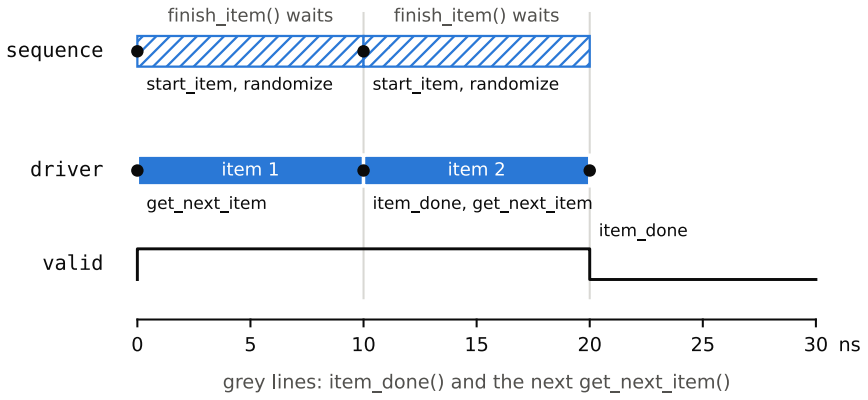


Figure 13.1: The first two items of SqTimelineTest: the sequence waits in finish_item from 0 to 10 ns and from 10 to 20 ns while the driver holds each vector for one period.

The handshake for the first two items of `SqTimelineTest`. The sequence is blocked in `finish_item()` for the whole of each period, while the driver holds the vector on the pins. `valid` is the Python flag that the driver sets while a vector is on the pins, as in Lab 01. It stays high from 0 to 20 ns, because the driver puts the second vector on in the same time step as it calls `item_done()` for the first.

The sequence randomizes the item after `start_item()` returns, so the item gets its values when the driver is ready for it rather than when the sequence made it. The constraints can use anything that the sequence has put in the item by then. Lab 04's `RouterRandSeq`, for example, 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.

The ALU has no clock, so the example's driver drives the vector as soon as it has the item. The router has a clock, and Lab 04's drivers follow the timing rule that the course set in Lab 02, Part 2. Every driver changes the router's inputs right after a rising edge, and every monitor reads the pins at the falling edge in the middle of the cycle. So when `get_next_item()` returns, `SrcDriver` looks at the clock before it drives the packet's first beat. If the clock is low, it waits for the next rising edge, and if the clock is high, it's already just after one. `ApbDriver` always waits for the next rising edge, and starts the transfer's setup cycle there. The sequence waits in `finish_item()` all the while.

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. `SqTwoTest` starts two sequences of two items each, `a` and `b`, as two cocotb tasks, and waits for both. These are the lines it prints up to 20 ns, without the driver's `get_next_item: waiting` and `item_done` lines:

```
0    ns  [seq a]   start_item(req): waiting for the driver
0    ns  [seq b]   start_item(req): waiting for the driver
0    ns  [seq a]   start_item returned; randomize()
0    ns  [drv]     get_next_item returned a: ADD A=6d B=23
10   ns  [seq a]   finish_item returned: a: ADD A=6d B=23 is done
10   ns  [seq a]   start_item(req): waiting for the driver
10   ns  [seq b]   start_item returned; randomize()
10   ns  [drv]     get_next_item returned b: ADD A=85 B=c8
20   ns  [seq b]   finish_item returned: b: ADD A=85 B=c8 is done
20   ns  [seq b]   start_item(req): waiting for the driver
20   ns  [seq a]   start_item returned; randomize()
20   ns  [drv]     get_next_item returned a: SUB A=22 B=c4
```

The items alternate between the two sequences. pyuvm's sequencer hands the driver items in the order that their `start_item()` calls arrived, whichever sequence made them, 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. That order is the only one that pyuvm 4.0.1's sequencer has. SystemVerilog UVM's sequencer has arbitration modes, priorities and locks, and pyuvm's source has none of them. Lab 05 starts a sequence that runs for the whole test on each output agent, so another sequence started on one of those sequencers would take turns with it.

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. `start()` stores the sequencer in the sequence's `sequencer` attribute, so a child sequence is started with `await child.start(self.sequencer)`. (SystemVerilog UVM calls that attribute `m_sequencer`, and has a `p_sequencer` as well; pyuvm has neither.) `SqNestedTest` runs a sequence that starts two others, first one that sends two random vectors and then one that sends two directed `SUB`s. The run prints these lines from the point where the random sequence finishes, at 20 ns, to the driver's `item_done()` for the second `SUB`:

```
20   ns  [seq r]   finish_item returned: r: SHL A=ff B=ec is done
20   ns  [drv]     get_next_item returned sub: SUB A=5a B=5a
30   ns  [drv]     item_done, ALUout was 00
30   ns  [drv]     get_next_item: waiting for an item
30   ns  [seq sub] sent sub: SUB A=5a B=5a
30   ns  [drv]     get_next_item returned sub: SUB A=5a B=21
40   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. Lab 04's `close_holes()` builds its fill packets by hand too, with `RouterPkt.make()`, and sends each one with `RouterOneSeq`, a sequence that only calls `start_item()` and `finish_item()`.

Lab 04′s base test writes a register by starting a one-item sequence on the APB agent's sequencer, and it reads a register the same way:

```
async def write_reg(self, addr, data):
    await ApbRwSeq("wr", True, addr, data).start(self.env.apb.sqr)

async def read_reg(self, addr):
    s = ApbRwSeq("rd", False, addr)
    await s.start(self.env.apb.sqr)
    return s.t.rdata
```

`read_reg()` takes the value it returns from 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 `finish_item()` has. `SqReadTest` does the same on the ALU, whose driver fills in `ALUout`:

```
[READ] SUB(c8, 3c) = 8c
[READ] NAND(c3, 1d) = fe
[READ] SHR(f8, 03) = 1f
```

get_next_item() and item_done()

A SystemVerilog UVM driver can call `try_next_item()`, which returns at once, with no item, when none is waiting. pyuvm 4.0.1′s `seq_item_port` has no `try_next_item()`. The only call it has that gets an item is `get_next_item()`, and that call waits until there is one. So a driver that must act in every cycle, whether or not an item has arrived, needs a sequence that always has the next item ready. The sink driver that you finish in Lab 05, Part 3 is one of these. Lab 05′s base test starts a `SinkSeq` on each sink, and that sequence never ends.

A driver must also call `item_done()` before it asks for the next item. `SqDoubleGetTest` has a driver that calls `get_next_item()` twice without calling `item_done()` in between, and pyuvm stopped the test at the second call:

```
 BROKEN: UVMSequenceError   1.88 s   log: build/runs/SqDoubleGetTest-s1/run.log
     SqDoubleGetTest: UVMSequenceError: You must call item_done() before calling
get_next_item again
```

The graders call this run BROKEN rather than FAIL, because the exception came from the testbench, and a run that ends that way tells you about the testbench and nothing about the design.

The time on the first line is how long the simulator ran. We built the simulator twice and ran `SqDoubleGetTest` five times. The first run after each build took us 3.54 s and 3.56 s, and each of the other three runs took 1.88 s.

Check yourself

1. In `SqTimelineTest`, `finish_item()` for the first item returns at 10 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()` stalls output 0 and fills it with packets for DST 0 whose lengths come from a `FillPlan` and add up to exactly 8. Why does it build them with `RouterPkt.make()` and send them with `RouterOneSeq`, instead of starting `RouterRandSeq`?

   Answer: A3, in Answers at the end.

Further reading

pyuvm's sequences are in its source, `s14_15_python_sequences.py` at 4.0.1. Its comments sketch both sides of the handshake: a driver's loop that calls `get_next_item()` and `item_done()`, and a sequence's `body()` that calls `start_item()` and `finish_item()`. ChipVerify's walkthrough builds the same pieces in

SystemVerilog UVM, for a register block that it calls `reg_block`: UVM Sequence, UVM Sequencer and UVM Driver.

- Using get_next_item(), for the handshake in more detail. It also covers `try_next_item()`, which pyuvm 4.0.1 doesn't have.
- How to execute sequences via start(), for the hooks around `body()`. pyuvm's `start()` takes only the sequencer and a flag for whether to call `pre_body()` and `post_body()`, with no parent sequence or priority.
- UVM Sequence Arbitration describes arbitration modes that pyuvm 4.0.1's sequencer doesn't have; it hands out items in the order they were asked for.

Next: Monitors, analysis ports and scoreboards.

Answers

A1. Check yourself

The driver's `item_done()`. The driver got the item at 0 ns, held it on the pins for one 10 ns period, and then called `item_done()`, which let `finish_item()` return.

A2. Check yourself

At 10 ns. Its `start_item()` can't return until the driver takes it, and the driver asks for another item only after it has held the first one for its period and called `item_done()`.

A3. Check yourself

`RouterRandSeq` makes each packet and calls `randomize()` on it, so the solver would choose the DST and the length, and the packet's `post_randomize()` would put idle cycles before about one beat in four. `RouterRandSeq` also makes each packet with `RouterPkt.create()`, through the factory, so a test's override, such as the one that swaps in `NoroutePkt` for `RouterInjectTest`, would reach the fill packets too. `make()` sets the DST and the length by hand, puts the beats back to back and uses the constructor, which no override reaches, and `RouterOneSeq` sends the packet it's given without changing it. (Lab 03, Part 4 says that `randomize_with()` with an inline `length == n` would work too, as line 3 of its experiments shows, and the packets would then have the idle cycles.)

14. 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 pyuvm 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()` method. Lab 02′s router testbench was built without pyuvm, and its classes passed packets through queues. Your monitor turned the handshakes on a stream into whole packets and put each one in a queue, and your scoreboard took them out in two loops of its own, one for each monitor's queue. Lab 03 added a tracker that read the input's and output 0′s pins itself, in a loop of its own.

In Lab 04 the router testbench is rewritten in pyuvm, 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 coroutine at all. It's a component with three write methods, `write_in()`, `write_out()` and `write_apb()`, and every write to a monitor's analysis port ends in a call to one of them.

The example for this page tests SimProc's ALU with a complete agent, made of a sequencer, a monitor and the driver from the example for Sequences, sequencers and drivers. Its source is in `guide-examples/monitors-analysis-ports-and-scoreboards/`. 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 the scoreboard, the coverage collector, the history and any subscriber you add later, without a change to the monitor. The example's monitor reads the pins 5 ns into every 10 ns period, as Lab 01′s did, while the driver holds them steady:

```
def next_item(self):
    return AluItem.create("seen")          # a new item every time

async def run_phase(self):
    vif, d = self.vif, self.vif.dut
    await Timer(vif.PERIOD_NS // 2, "ns")
    while True:
        if vif.valid:
            t = self.next_item()
            t.op = AluOp(int(d.ALUop.value))
            t.A = int(d.A.value)
            t.B = int(d.B.value)
            t.ALUout = int(d.ALUout.value)
            t.N = int(d.N.value)
            t.Z = int(d.Z.value)
            self.n_seen += 1
            self.ap.write(t)
        await Timer(vif.PERIOD_NS, "ns")
```

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

Lab 04′s monitors do the same job for the router. The monitor on each stream writes one `StreamBeat` for every handshake, stamped with the number of the cycle it happened in, and the APB monitor writes

one `ApbItem` for every transfer that completes. All of them read the pins at the falling edge in the middle of each cycle, as the course's monitors have done since Lab 02.

Analysis ports

The monitor makes its port in `build_phase`, with `self.ap = uvm_analysis_port("ap", self)`. The port's `write(t)` is an ordinary method, not a coroutine, so it can't wait. It calls the `write()` of every subscriber connected to the port, one after another in the order they were connected, before it returns. A port with no subscribers is allowed, and `write()` then does nothing. Nothing in the monitor depends on who, if anyone, is connected to its port. The environment makes the connections, in its `connect_phase`:

```
def connect_phase(self):
    self.agt.mon.ap.connect(self.sb.analysis_export)
    self.agt.mon.ap.connect(self.cov.analysis_export)
    self.agt.mon.ap.connect(self.hist.analysis_export)
```

All three subscribers derive from `uvm_subscriber`. Its constructor makes the subscriber's `analysis_export`, an export that passes every item written to it on to the subscriber's `write(t)`, so `write(t)` is the only method you have to write. The port has to be connected to that export, not to the subscriber itself. `MonConnectTest` swaps in an environment whose `connect_phase` calls `self.agt.mon.ap.connect(self.sb)` instead, and the run stopped there with this error:

```
 BROKEN: UVMTLMConnectionError   1.83 s   log: build/runs/MonConnectTest-s1/run.log
   MonConnectTest: UVMTLMConnectionError: uvm_test_top.env.sb must be a subclass of
uvm_export_base
```

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

```
   2000.00ns INFO       ..-and-scoreboards/mon_env.py(184) [uvm_test_top]: [MON] the
monitor wrote 200 items
  2000.00ns INFO      ..-and-scoreboards/mon_env.py(106) [uvm_test_top.env.sb]: [SB]
200 vectors checked: 0 wrong
  2000.00ns INFO     ..-and-scoreboards/mon_env.py(126) [uvm_test_top.env.cov]: [COV]
alu: 20 of 20 bins hit
  2000.00ns INFO    ..-and-scoreboards/mon_env.py(141) [uvm_test_top.env.hist]: [HIST]
kept 200 handles to 200 different objects
```

A scoreboard with several inputs

A subscriber has one export, and every item written to it arrives in the same method, `write()`. A scoreboard that listens to several monitors needs a separate method for each input, so that it knows which monitor an item came from. SystemVerilog UVM declares a new imp class for each input with the macro `` `uvm_analysis_imp_decl ``, and each of those classes calls a different method of the scoreboard. pyuvm 4.0.1 has no such macro. Instead, Lab 04 gives its scoreboard a small subscriber for each input, a `Fwd`, which passes every item written to it on to one method of the scoreboard:

```
class Fwd(uvm_subscriber):
    def __init__(self, name, parent, fn):
        super().__init__(name, parent)
        self.fn = fn

    def write(self, t):
        self.fn(t)
```

`RouterScoreboard.build_phase` makes three of them as its children:

```
self.in_imp = Fwd("in_imp", self, self.write_in)
self.out_imp = Fwd("out_imp", self, self.write_out)
self.apb_imp = Fwd("apb_imp", self, self.write_apb)
```

and the environment connects each monitor to the right one. The figure shows which monitors write to each.

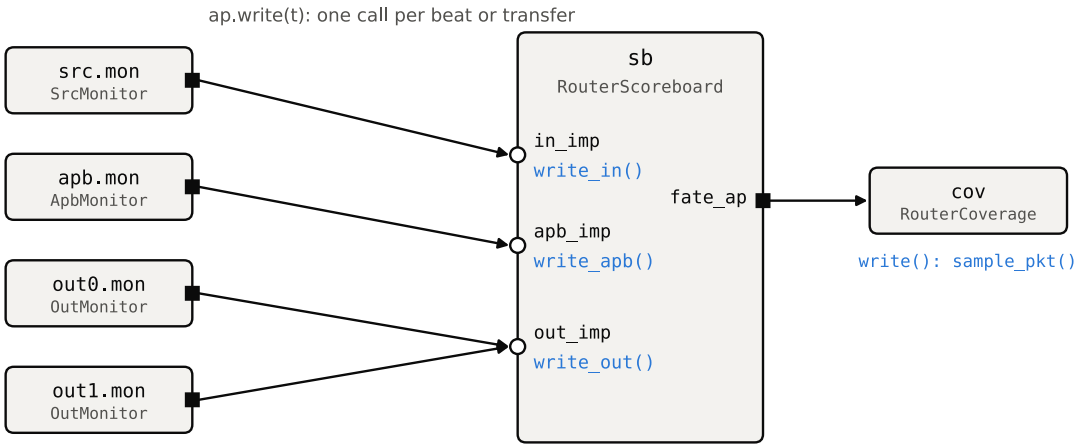


Figure 14.1: Lab 04′s analysis connections: four monitors write to three Fwd inputs on the scoreboard, and the scoreboard's own analysis port feeds the coverage.

The connections that Lab 04′s `RouterEnv.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 to `out_imp`. Both output monitors write to the same `out_imp`, and each `StreamBeat` 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 record of it to its own analysis port, `fate_ap`, which the coverage subscribes to.

The scoreboard

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

```
def write(self, t):
    out, n, z = predict(t)
    bad = t.ALUout != out
    if self.check_flags:
        bad = bad or t.N != n or t.Z != z
    self.n_checked += 1
    if bad:
        self.n_bad += 1
        self.logger.error(f"[SB] {t}, expected ALUout={out:02x} N={n} Z={z}")
```

With bug A01 on, `MonRandTest` logged 30 errors, and this is the first:

```
     75.00ns ERROR    ..s-and-scoreboards/mon_env.py(97) [uvm_test_top.env.sb]: [SB]
SUB A=9a B=db -> ALUout=41 N=0 Z=0, expected ALUout=bf N=1 Z=0
```

In pyuvm, logging an error doesn't fail the test by itself. The test derives from the course's `BaseTest`, which gives every component's logger a handler that counts the errors logged through it. Here the handler counted 30, so `BaseTest` failed the test in `final_phase`, and the verdict was `caught: 30 errors`. 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.

Checking inside `write()` works for the ALU because the ALU is combinational. The monitor reads its inputs and its result in the same period, and both arrive in the same item. In the router, a packet goes in on one stream and comes out on another stream two or more cycles later, and the scoreboard can't wait inside a write method for it to come out. So Lab 04′s scoreboard splits the work between its write

methods. 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 its copy of the routing table, which it keeps from the APB writes it sees, and from the level of the output that the packet's route names. `level()` works out that level from the beats written into the output and taken from it. 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 that the scoreboard has counted since `DROP_CNT` was last written. At the end of the test, `check_phase` reports any beats still left in the queues, because those are beats that the router never delivered. (As shipped, `write_in()` queues only output 0's beats, and `level()` takes output 1's level as 0. Part 2 of Lab 04 has you change the first, and Part 3 the second.)

All the monitors read the pins at the same falling edge, and cocotb runs them in no fixed order. So in the cycle that a header comes in, an output's monitor may already have reported a beat taken from that output in the same cycle, and `write_out()` will have counted it. The specification's level counts only the handshakes in earlier cycles, so `level()` adds that beat back.

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 only `ALUout`. Against Lab 01's three bugs, `make check-all` gave:

```
  caught  tier 1  A01_sub_operands_swapped        30 errors                  2.5s
  caught  tier 1  A02_nand_is_nor                 25 errors                  2.5s
  MISSED  tier 1  A03_zero_flag_low_nibble        0 errors                   2.5s
```

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 ConfigDB, 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 `AluItem` for every vector. `MonAliasTest` 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:

```
  2000.00ns INFO    ..-and-scoreboards/mon_env.py(141) [uvm_test_top.env.hist]: [HIST]
kept 200 handles to 1 different objects
  2000.00ns INFO    ..-and-scoreboards/mon_env.py(145) [uvm_test_top.env.hist]: [HIST]
vector 0: OR A=e7 B=01 -> ALUout=e7 N=1 Z=0
  2000.00ns INFO    ..-and-scoreboards/mon_env.py(145) [uvm_test_top.env.hist]: [HIST]
vector 1: OR A=e7 B=01 -> ALUout=e7 N=1 Z=0
  2000.00ns INFO    ..-and-scoreboards/mon_env.py(145) [uvm_test_top.env.hist]: [HIST]
vector 2: OR A=e7 B=01 -> ALUout=e7 N=1 Z=0
  2000.00ns INFO    ..-and-scoreboards/mon_env.py(145) [uvm_test_top.env.hist]: [HIST]
vector 199: OR A=e7 B=01 -> ALUout=e7 N=1 Z=0
```

All 200 handles point at one object, which holds the last vector. The scoreboard in the same run still reported `200 vectors checked: 0 wrong`, 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, finds a later vector's values in the object by the time it looks. So a monitor should make a new object for every write. A subscriber that keeps what it's given should copy the values it needs, as Lab 04's scoreboard does when it queues `(data, last)` for each expected beat. It can keep the handle instead only when the monitor makes a new object for every write, and then it must never change the object, because every other subscriber has the same one.

Coverage as a subscriber

A coverage collector is a simple subscriber, because its `write()` only samples the item. The example's collector passes each item to `sample(t)`, a function decorated with a coverpoint for the operation, a coverpoint for whether `A` equals `B`, and a cross of the two, all from cocotb-coverage. In `report_phase` the

collector writes `coverage.xml` into the run's folder, where `make cov` finds it. Functional coverage you can read explains the coverpoints.

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 `RouterCoverage` covers each packet's length, its DST, its fate, and the level of its route's output in the cycle that its header came in. Only the scoreboard works out the fate and the level, so `RouterCoverage` subscribes to the scoreboard's `fate_ap`.

Check yourself

1. An analysis port's `write()` is an ordinary method. 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 `MonAliasTest` the scoreboard passed and the history was wrong. Would a scoreboard that kept each input item on a list and compared it when an output arrived have passed?

   Answer: A2, in Answers at the end.

3. Two output monitors write `StreamBeat`s to the same `out_imp`. How does `write_out()` tell output 0's beats from output 1's?

   Answer: A3, in Answers at the end.

Further reading

pyuvm 4.0.1 defines its analysis ports in `s12_uvm_tlm_interfaces.py` and `uvm_subscriber` in `s13_predefined_component_classes.py`. ChipVerify's pages are about SystemVerilog UVM:

- 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`. Such a monitor can't do anything useful, and in pyuvm a `ConfigDB().get()` that finds nothing and has no default raises `UVMConfigItemNotFound`, which ends the test, as the next page shows.
- UVM TLM Analysis Port shows a broadcast to three subscribers. It warns that every receiver gets the same handle, which is what `MonAliasTest` shows. Its log comes from UVM 1.1d.
- Using _decl macro in TLM builds a two-input scoreboard with a macro that pyuvm doesn't have; Lab 04's `Fwd` does the same job. Correction: it says the macro can't go inside a class because SystemVerilog doesn't allow a class inside another class. SystemVerilog does allow nested classes.
- In UVM Scoreboard, the only check at the end of the test prints a count of mismatches, so a scoreboard that was never called passes it. The example's scoreboard and Lab 04's check for that in `check_phase`.

Next: The factory and ConfigDB.

Answers

A1. Check yourself

The scoreboard's `write()` can't wait for anything, so it can't `await` a clock edge or a queue. 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 coroutine of its own for any of that.

A2. Check yourself

No. Every handle on its list 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 every beat with its port number, which it takes from its `OutCfg`, and `write_out()` uses `b.port` to pick which queue of expected beats to compare with.

15. The factory and ConfigDB

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

In pyuvm a test uses the factory to change what the environment builds, and `ConfigDB` to change how it's set up, and Lab 04 has you use both. In Part 2 you make a second output agent from the same class, and `ConfigDB` tells each agent which output it's on. The stall test you write in Part 3 stalls one output at a time by changing a knob that the output drivers read. Part 4's test sends packets to disabled routes by telling the factory to make a subclass of the packet class instead, and the sequence that makes the packets doesn't change. Part 5 then asks what the extra lines of the pyuvm testbench bought you, and its answer is these three changes and how few lines each one took.

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

The factory

As soon as Python defines a class that derives from a pyuvm class, pyuvm registers it with the factory under the class's name. The labs make most of their components and many of their sequence items by asking the factory for them, as the example's sequence does for each vector:

```
req = AluItem.create("req")
```

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

`FacEqualTest` uses an override to give every vector equal operands. The override puts `AluEqualItem` in place of `AluItem`, and `AluEqualItem` is a subclass of `AluItem` that replaces `AluItem`'s constraint `c_same`:

```
@vsc.randobj
class AluEqualItem(AluItem):
    @vsc.constraint
    def c_same(self):
        self.same == 1
        with vsc.implies(self.same == 1):
            self.B == self.A
```

`AluItem`'s `c_same` makes `same` 1 in one vector in eight, and makes `B` equal to `A` when it is. `AluEqualItem`'s makes `same` 1 in every vector. A pyvsc constraint is a method, and a Python method with the same name as one in the parent class replaces it, so `AluEqualItem` has this `c_same` and not its parent's. Lab 04's `NoroutePkt` replaces `RouterPkt`'s `c_dst` the same way. The test sets the override in its `build_phase`, before `super().build_phase()` builds the environment:

```
def build_phase(self):
    uvm_factory().set_type_override_by_type(AluItem, AluEqualItem)
    super().build_phase()
```

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

```
--- overrides ---

Overrides:
AluItem                 : Type Override: AluEqualItem ||
```

The sequence still calls `AluItem.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`. The history's first three vectors were `OR A=29 B=29`, `NAND A=e8 B=e8` and `ADD A=5d B=5d`. 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
```

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. DD104 showed the same two bugs hiding behind equal operands. The scoreboard doesn't check the flags, so A03 stays hidden too. The coverage collector reported `[COV] alu: 13 of 20 bins hit`, because no vector had unequal operands, so the `differ` bin and the six cross bins that include it were never hit.

The constructor and the factory

An override changes only the objects that the factory makes. `FacCtorTest` sets the same override and then makes one item each way:

```
[CTOR] AluItem.create('x') made AluEqualItem
[CTOR] AluItem('x') made AluItem
```

Lab 04's `close_holes()` picks the DST of every packet it sends, so it makes its packets as plain `RouterPkt`s, with the constructor, in `new_packet()` and in `RouterPkt.make()`, where a test's override can't reach them. When we made its two random packets with `create()` instead, `RouterInjectTest`'s override turned them into `NoroutePkt`s too, and the test ended `BROKEN: SolveFailure` on 2 of seeds 1 to 20 (Lab 04, Part 4, step 3).

Components go through the factory too, when they're made with `create()`. On the monitors page, `MonAliasTest` swapped in a different monitor with an override of `AluMonitor`, and `MonConnectTest` swapped in a different environment with an override of `MonEnv`. Both overrides took effect because `MonAgent` makes its monitor with `AluMonitor.create()` and the base test makes the environment with `MonEnv.create()`. A component made with its constructor can't be swapped that way. In Lab 04, the input and APB agents make their sequencers like that, with `uvm_sequencer("sqr", self)`.

pyuvm clears every override at the start of each test, so one test's overrides never reach the next. An override can also apply to a single component, named by its path in the tree, with `uvm_factory().set_inst_override_by_type(original, override, path)`, but the course's labs use only type overrides.

An override from the command line

pyuvm 4.0.1 reads no plusargs of its own, so it can't take an override from the command line the way SystemVerilog UVM's `+uvm_set_type_override` does. Lab 04's base test reads a plusarg that the course defines, `+OVERRIDE=<class>,<class>`, and before it builds anything it calls `uvm_factory().set_type_override_by_name()` with the two class names. The example's base test, `MonBaseTest`, does the same, so the override also works on `MonRandTest`, which knows nothing about `AluEqualItem`:

```
make run TEST=MonRandTest BUG=A01 PLUSARGS=+OVERRIDE=AluItem,AluEqualItem
```

On the ALU with bug A01 the run logged

```
     0.00ns INFO     ..-and-scoreboards/mon_env.py(171) [uvm_test_top]: [OVERRIDE] the
factory makes AluEqualItem for every AluItem
```

and ended `MISSED: 0 errors`. The same test without the plusarg ends `caught: 30 errors`. A plusarg goes to the simulator when it starts, so changing one rebuilds nothing. Lab 04 has you use `+OVERRIDE` the same way on the router, in Part 4, step 6.

ConfigDB

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

```
ConfigDB().set(self, "env.sb", "check_flags", True)                 # in the test
self.check_flags = ConfigDB().get(self, "", "check_flags", False)   # 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`. Its last argument, `False`, is the value it returns if nobody set one.

The example's scoreboard checks only `ALUout` unless `check_flags` is `True`, and then it checks `N` and `Z` as well. `FacFlagsTest` sets it to `True` for `"env.sb"`, and it catches all three bugs:

```
      ok  the golden design                       0 errors                    2.5s
--------------------------------------------------------------------------------
  caught  tier 1  A01_sub_operands_swapped        32 errors                   2.5s
  caught  tier 1  A02_nand_is_nor                 18 errors                   2.5s
  caught  tier 1  A03_zero_flag_low_nibble        19 errors                   2.5s
--------------------------------------------------------------------------------
 3 of 3 caught.
```

A path in a `set()` can hold wildcards. They're globs, as in a shell, not regular expressions. A `*` matches any run of characters, dots included, so `"env.*"` reaches every component below `env`. A `get()` names one component, and pyuvm raises `UVMError` for a wildcard in its path. Lab 01's pyuvm test puts its `AluIf` in `ConfigDB` with `None` as the context and the path `"*"`, so that every component can find it. The course's `RouterBaseTest`, which Lab 04's base test derives from, does the same with the router's handle, under the name `DUT`.

When nothing reads the value

`FacTypoTest` makes the same `set()` for `"env.scoreboard"`, and the environment has no component by that name. The scoreboard's `get()` found nothing, so `check_flags` kept its default of `False` and the test missed A03 (`2 of 3 caught.`). The run printed no warning, and it passed on the golden design. To see what went wrong, turn on `ConfigDB`'s trace with `ConfigDB().is_tracing = True`, which logs every `set()` and every `get()` that finds a value. The test does that before its `set()`:

```
    0.00ns INFO    uvm.logger_holder126772875082400   CFGDB/SET Context: uvm_test_top
--  uvm_test_top.env.scoreboard check_flags=True
[TEST] uvm_test_top set check_flags = True for "env.scoreboard"
     0.00ns INFO     uvm.logger_holder126772875082400   CFGDB/SET Context:    --  *
vif=<sq_agent.AluIf object at 0x734c9c2fba40>
        0.00ns  INFO        uvm.logger_holder126772875082400     CFGDB/GET  Context:
uvm_test_top.env.agt.drv  --  uvm_test_top.env.agt.drv vif=<sq_agent.AluIf object at
0x734c9c2fba40>
        0.00ns  INFO        uvm.logger_holder126772875082400     CFGDB/GET  Context:
uvm_test_top.env.agt.mon  --  uvm_test_top.env.agt.mon vif=<sq_agent.AluIf object at
0x734c9c2fba40>
      0.00ns  INFO      uvm.logger_holder126772875082400    CFGDB/GET  Context:    --
vif=<sq_agent.AluIf object at 0x734c9c2fba40>
```

The `set()` was for `uvm_test_top.env.scoreboard`, and there's no `GET` line for `check_flags` at all, because the trace leaves out a `get()` that finds nothing. Compare the path in each `SET` line with the full names of the components that should read it. (The last `GET` is the sequence's. A sequence has no place in the component tree, so its `get()` passes `None` as the context.)

A knob with a sensible default can be read the way `check_flags` is, with a default in the `get()`. A component that can't work without a value should get it with no default, and then a `get()` that finds

nothing raises `UVMConfigItemNotFound` and ends the run. `FacMissingTest` swaps in a scoreboard that gets `check_flags` with no default, and nothing sets it:

```
 BROKEN: UVMConfigItemNotFound   1.86 s   log: build/runs/FacMissingTest-s1/run.log
   FacMissingTest: UVMConfigItemNotFound: "Component uvm_test_top.env.sb has no key:
check_flags
   ConfigDB().get() asked for a name nobody set. That is configuration,
   not a check.
```

Every driver and monitor in Lab 04 gets the router's handle with no default, and the output agents' drivers and monitors get their `OutCfg` the same way, so if either one is missing, the run stops in the build phase.

When two components set the same value

`FacPrecTest` sets `check_flags` to `True` for `"env.sb"`. It also overrides the environment with a subclass, `FalseEnv`, which sets `check_flags` to `False` for its own `"sb"` before it builds the scoreboard. The environment is built after the test, so its `set()` comes second:

```
[TEST] uvm_test_top set check_flags = True for "env.sb"
[ENV] uvm_test_top.env set check_flags = False 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 `True` beat the environment's `False`. pyuvm stores each `set()` made during `build_phase` with a precedence of 1000 minus the depth of the component that made it, and a `get()` takes the value with the highest precedence. With that rule, an environment can give its components defaults, and a test can override them.

`FacGlobTest` is the same test, except that the test's `set()` is for `"env.*"`:

```
[TEST] uvm_test_top set check_flags = True for "env.*"
[ENV] uvm_test_top.env set check_flags = False for "sb"
```

This time the scoreboard reported `200 vectors checked: 0 wrong`, without the flags, and the test missed A03. Of the paths that match, pyuvm's `get()` first finds the most specific one that has a value named `check_flags`, here `uvm_test_top.env.sb` rather than `uvm_test_top.env.*`, and compares precedences only among the `set()`s for that one path. So a test that wants to override an environment's default has to name the component at least as exactly as the environment does. After the build phase, every `set()` gets the same precedence, 1000, so a later one for the same path replaces an earlier one.

Objects by handle

`ConfigDB` can hold an object as well as a flag like `check_flags`. It stores the object itself, so the component that gets the value and the one that set it share one object. Lab 04 uses that for the output agents. `RouterLabTest` makes an `OutCfg` for each output, sets output 0's for `"env.out0*"` and output 1's for `"env.out1*"`, and keeps the two handles:

```
self.ocfg = [OutCfg(0), OutCfg(1)]
for p in (0, 1):
    ConfigDB().set(self, f"env.out{p}*", "cfg", self.ocfg[p])
```

Both output agents are the same class, `OutAgent`. The driver and the monitor in `out1`, for example, have full names that start with `uvm_test_top.env.out1`, which matches `"env.out1*"`, so they get output 1's `OutCfg`, and its `port` tells them which pins to use. Nothing else differs between the two agents. Because the test kept the handles, your Part 3 stall test can change `self.ocfg[p].ready_pct` in the middle of a run. Each `OutReadyDriver` reads `ready_pct` again after every rising edge, so it follows the change at the next one.

Check yourself

1. `FacEqualTest` and `MonRandTest` both derive from `MonBaseTest`, and `FacEqualTest` adds a type override. It catches none of the three bugs, and `MonRandTest` 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 the constructor. What happened in `RouterInjectTest` when they were made with `create()` instead?

   Answer: A3, in Answers at the end.

Further reading

In pyuvm 4.0.1's source, the factory is in `s08_factory_classes.py`, and `ConfigDB` is at the end of `s13_uvm_component.py`, where a comment says why it matches globs and not regular expressions. ChipVerify's pages are about SystemVerilog UVM:

- UVM Factory Override covers type and instance overrides, and the difference in argument order between the factory's methods and the component's. pyuvm has only the factory's, called on `uvm_factory()`. The page's last example calls methods on a variable named `factory` that it never declares there.
- UVM config database, for the API, the wildcards and the precedence rules. Its `get()` returns 0 when it finds nothing; pyuvm's raises `UVMConfigItemNotFound` unless you give it a default. Its precedence rules leave out the first step of pyuvm's `get()`, which picks the most specific matching path, as `FacGlobTest` shows.
- Configure Components, on configuration objects like Lab 04's `OutCfg`.

Next: Lab 04: The same testbench in pyuvm.

Answers

A1. Check yourself

No. The sequence's code is the same, and it still asks the factory for an `AluItem` and randomizes it. The factory hands back an `AluEqualItem` instead, and that class's `c_same` decides the operands. Every item the sequence made had `A == B`, which hides A01 and A02. The scoreboard didn't check the flags, so it missed A03, as it does in `MonRandTest`.

A2. Check yourself

Turn on the trace with `ConfigDB().is_tracing = True` and compare the path in the `CFGDB/SET` line with the scoreboard's full name. If there's no `CFGDB/GET` line for the knob, the scoreboard's `get()` found nothing, so the `set()` didn't name the component that reads the value, as in `FacTypoTest`. If the paths match, look for another `set()` for a more specific path, as in `FacGlobTest`.

A3. Check yourself

The test's override reached them too, so the packet for DST 0 was a `NoroutePkt`. Whenever its `pre_randomize()` set `to_disabled`, its `c_dst` asked for a disabled route, while the inline constraint asked for DST 0, the one enabled route. The solver found no solution, and the run ended `BROKEN: SolveFailure`, on seeds 11 and 15 of seeds 1 to 20 (Lab 04's solutions README).

16. Lab 04: The same testbench in pyuvm

In this lab you'll finish our pyuvm port of the Lab 03 testbench. As shipped, it checks only output 0, as Lab 03's testbench did, and it misses this lab's two bugs, B06 and B13. You'll add an agent for output 1 and make the scoreboard check that output's beats. You'll write a test that stalls one output at a time, and fix the scoreboard so that it can judge that test. Then you'll write `NoroutePkt`, a subclass of `RouterPkt` that sends one packet in four to a disabled route, and a factory override that puts it in place of every packet the random sequence makes. Last, you'll count the lines in the port and in your Lab 03 testbench.

Package: `labs/lab-04-the-same-testbench-in-pyuvm/`

Before you start

- Finish Lab 02 and Lab 03. Keep your Lab 03 files in that lab's `tb/`, because Part 5 counts them.
- Read The pyuvm skeleton, Sequences, sequencers and drivers, Monitors, analysis ports and scoreboards and The factory and ConfigDB.
- Look again at `tb/alu_uvm.py` from Lab 01. The port has the same kinds of pieces (sequences, drivers, monitors, agents, an environment, and tests that derive from `BaseTest`). It also leans on `ConfigDB` and `create()`, which that file used too. From cocotb tests to pyuvm components shows how Lab 01's testbench uses both, and The factory and ConfigDB has a section on each.
- Keep the router specification open. Parts 3 and 4 use its Routing section.
- Copy this table into your notes and fill it in as you go:

	`make check-all`
The port as shipped (Part 1)	
After Part 2	
After Part 3	
After Part 4	

Time: about 2 h. Every file you change is Python, so nothing in this lab rebuilds the simulator, and a run of the graded test takes about 5 s.

Part 1: Run the port

Does the port do what the Lab 03 testbench did, and which of this lab's bugs does it miss?

1. Open a terminal on the workbench desktop, as in Lab 03, go to the lab folder, and build the simulator:

```
. /foss/designs/common/.designinit   # in every new terminal
mod dd105-b
cd labs/lab-04-the-same-testbench-in-pyuvm
make build
```

It ends with:

```
 built /foss/designs/modules/dd105-b/labs/lab-04-the-same-testbench-in-pyuvm/build/
sim_build/Vtop in 2.2 s (1.9M in /foss/designs/modules/dd105-b/labs/lab-04-the-same-
testbench-in-pyuvm/build/sim_build)
 ccache:  Hits: 23 / 35 (65.71%)  Misses: 12 / 35 (34.29%)
==========================================================================
```

```
      /foss/designs/modules/dd105-b/labs/lab-04-the-same-testbench-in-pyuvm/build/
sim_build/Vtop is up to date.
```

This is the same router as in Labs 02 and 03. The ccache counts on that line include our builds of Labs 01 to 03, which came to 17 hits and 11 misses, so this build's own were 6 hits and 1 miss, as in Lab 03. The course venv you made in Lab 03 is still there, and this lab uses it.

2. Open the files in `tb/`. Every piece of the Lab 03 testbench is in one of these three files, and most of its classes are now pyuvm classes:

File	What's in it
`router_agents.py`	the packet, `RouterPkt`, which is Lab 03's `RandPacket` 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, `OutAgent`; the sequences
`router_env.py`	the scoreboard, which is Lab 03's scoreboard and tracker in one component; the coverage, which is our Lab 03 solution's; and `RouterEnv`, which builds and connects everything
`test_router.py`	`FillPlan`; the base test, `RouterLabTest`, 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 `RouterClassTest` (Lab 02's), `RouterRandTest` and `RouterFillTest` (Lab 03's), and the two that this lab adds, `RouterStallTest` and `RouterInjectTest`

Your Lab 02 and Lab 03 testbenches watched only output 0, and so does the port. `RouterEnv` makes only one `OutAgent`, 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. With no agent on output 1, `out1_ready` would stay at the 0 that the base test sets it to before the reset, and output 1 would never be ready. So `RouterEnv` holds it at 1, as Lab 03's sink did. Find the `TODO (Part n)` comments. There are four in `router_env.py` and three in `test_router.py`.

3. Run Lab 02's test, then the default test, then Lab 03's random test:

```
make run TEST=RouterClassTest
make run
make run TEST=RouterRandTest
```

`TEST=` picks the test when the simulator starts, so none of these rebuilds anything. You should see, from the first run:

```
    20.00ns INFO     ..-in-pyuvm/tb/test_router.py(299) [uvm_test_top]: [ENV] 100
packets, routes 0110
  6865.00ns INFO    ..h-in-pyuvm/tb/router_env.py(207) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00000000 as expected
  6870.00ns INFO     ..h-in-pyuvm/tb/router_env.py(224) [uvm_test_top.env.sb]: [SB]
100 packets in: 55 to output 0, 45 to output 1, 0 NOROUTE, 0 FULL; the highest level
a header saw was 2
  6870.00ns INFO     ..h-in-pyuvm/tb/router_env.py(227) [uvm_test_top.env.sb]: [SB]
beats checked: 233 on output 0, 0 on output 1; 1 DROP_CNT reads checked
  6870.00ns INFO     ..h-in-pyuvm/tb/router_env.py(230) [uvm_test_top.env.sb]: [SB]
193 beats went to output 1, which nothing watches
  6870.00ns INFO    ..h-in-pyuvm/tb/router_env.py(268) [uvm_test_top.env.cov]: [COV]
```

```
router.pkt: 19 of 41 bins hit; coverage.xml written
DD105-B BaseTest: 0 errors logged, no bound assertions
```

and `ok: 0 errors 3.61 s`. After the time and `INFO`, each line names the file and line of the logging call and then, in brackets, the component that logged it: `uvm_test_top` is the test, and `uvm_test_top.env.sb` is the scoreboard. The `[DROP]` line comes from Lab 03's `DROP_CNT` check, which the scoreboard now makes. It compares every read of `DROP_CNT` that the APB monitor reports with the drops it predicted.

The default test is `RouterInjectTest`, which for now does what `RouterRandTest` does: Lab 03's four phases and its `close_holes()`. The second run prints:

```
    0.00ns INFO    cocotb.regression            running test_router.RouterInjectTest
(1/1)
     50.00ns INFO      ..-in-pyuvm/tb/test_router.py(250) [uvm_test_top]: [PHASE] 1
of 4: routes 0-1-, 200 packets; output 0 ready 100%, output 1 ready 100%
 14165.00ns INFO    ..h-in-pyuvm/tb/router_env.py(207) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00000000 as expected
 14200.00ns INFO      ..-in-pyuvm/tb/test_router.py(250) [uvm_test_top]: [PHASE] 2
of 4: routes --10, 200 packets; output 0 ready 100%, output 1 ready 100%
 27255.00ns INFO    ..h-in-pyuvm/tb/router_env.py(207) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00000000 as expected
 27290.00ns INFO      ..-in-pyuvm/tb/test_router.py(250) [uvm_test_top]: [PHASE] 3
of 4: routes -1-0, 200 packets; output 0 ready 100%, output 1 ready 100%
 41625.00ns INFO    ..h-in-pyuvm/tb/router_env.py(207) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00000000 as expected
 41660.00ns INFO      ..-in-pyuvm/tb/test_router.py(250) [uvm_test_top]: [PHASE] 4
of 4: routes 1-0-, 200 packets; output 0 ready 100%, output 1 ready 100%
 54425.00ns INFO    ..h-in-pyuvm/tb/router_env.py(207) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00000000 as expected
 54600.00ns INFO       ..-in-pyuvm/tb/test_router.py(229) [uvm_test_top]: [HOLES]
output 0 stalled and filled to 8 by packets of 4 3 1 beats, then 6 beats for DST 0
and 1 for DST 1
 55015.00ns INFO    ..h-in-pyuvm/tb/router_env.py(207) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00000001 as expected
 55020.00ns INFO     ..h-in-pyuvm/tb/router_env.py(224) [uvm_test_top.env.sb]: [SB]
805 packets in: 417 to output 0, 387 to output 1, 1 NOROUTE, 0 FULL; the highest
level a header saw was 14
 55020.00ns INFO     ..h-in-pyuvm/tb/router_env.py(227) [uvm_test_top.env.sb]: [SB]
beats checked: 1884 on output 0, 0 on output 1; 5 DROP_CNT reads checked
 55020.00ns INFO     ..h-in-pyuvm/tb/router_env.py(230) [uvm_test_top.env.sb]: [SB]
1720 beats went to output 1, which nothing watches
 55020.00ns INFO    ..h-in-pyuvm/tb/router_env.py(268) [uvm_test_top.env.cov]: [COV]
router.pkt: 24 of 41 bins hit; coverage.xml written
DD105-B BaseTest: 0 errors logged, no bound assertions
```

ending in `ok: 0 errors 5.13 s`. The routes are written as in Lab 03, with one character per route: `0` or `1` for the output, `-` for disabled.

The third run, `RouterRandTest`, runs the same code on different packets. Its first phase has the routes `-110`, and it ends with:

```
 51770.00ns INFO       ..-in-pyuvm/tb/test_router.py(229) [uvm_test_top]: [HOLES]
output 0 stalled and filled to 8 by packets of 1 1 6 beats, then 8 beats for DST 0
and 1 for DST 1
 52240.00ns INFO     ..h-in-pyuvm/tb/router_env.py(224) [uvm_test_top.env.sb]: [SB]
805 packets in: 314 to output 0, 490 to output 1, 1 NOROUTE, 0 FULL; the highest
level a header saw was 16
 52240.00ns INFO     ..h-in-pyuvm/tb/router_env.py(227) [uvm_test_top.env.sb]: [SB]
beats checked: 1328 on output 0, 0 on output 1; 5 DROP_CNT reads checked
```

```
 52240.00ns INFO     ..h-in-pyuvm/tb/router_env.py(230) [uvm_test_top.env.sb]: [SB]
2006 beats went to output 1, which nothing watches
```

Why are its packets not the second run's, when both tests run the same steps on seed 1?

Answer: A1, in Answers at the end.

4. Look at the coverage of the three runs:

```
make cov
```

You should see:

```
**************************************************************************
 COVERAGE   3 runs, merged by cocotb-coverage into build/coverage/merged.xml
**************************************************************************
  top                                          58.54 %   24 of 41 bins
    top.router                                   58.54 %   24 of 41 bins
      top.router.pkt                               58.54 %   24 of 41 bins
        top.router.pkt.len                          100.00 %   8 of 8 bins
        top.router.pkt.dst                          100.00 %   4 of 4 bins
        top.router.pkt.fate                          75.00 %   3 of 4 bins
            not hit: full
        top.router.pkt.level                         80.00 %   4 of 5 bins
            not hit: at_9
        top.router.pkt.x_fate_level                  25.00 %   5 of 20 bins
           not hit: ('noroute', 'empty'), ('noroute', 'low'), ('noroute', 'at_8'),
('noroute', 'at_9'), ('full', 'empty'), ('full', 'low'), ('full', 'at_8'), ('full',
'at_9'), ('full', 'high'), ('out0', 'at_9'), ('out0', 'high'), ('out1', 'low'), ...
(3 more)
**************************************************************************
```

The three bins behind `... (3 more)` are `('out1', 'at_8')`, `('out1', 'at_9')` and `('out1', 'high')`. The 17 empty bins fall into four groups:

- seven that no test can hit, as you found in Lab 03: `('full', 'empty')`, `('full', 'low')`, `('full', 'at_8')`, `('out0', 'at_9')`, `('out0', 'high')`, `('out1', 'at_9')` and `('out1', 'high')`;
- `full`, `('full', 'at_9')` and `('full', 'high')`: no packet was dropped as FULL;
- `('out1', 'low')` and `('out1', 'at_8')`: the scoreboard takes output 1's level as 0;
- `at_9`, `('noroute', 'empty')`, `('noroute', 'low')`, `('noroute', 'at_8')` and `('noroute', 'at_9')`: the only NOROUTE drops are `close_holes()`'s packets for DST 1, one in each of the last two runs. They met levels of 14 and 16, and they were the only headers that met a level above 8.

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 errors. The scoreboard stops a test at its 50th error, so 50 means it found at least that many.

6. Now check this lab's two bugs, B06 and B13:

```
make check-all
```

You should see:

```
      ok  the golden design                      0 errors                    5.1s
--------------------------------------------------------------------------------
```

```
 MISSED  tier 2  B06_disabled_route_forwards     0 errors                    5.1s
 MISSED  tier 5  B13_admission_checks_other_output 0 errors                  5.1s
-------------------------------------------------------------------------------
 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. Write the result in your table. 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: A2, 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 `OutAgent`, `OutMonitor` and `OutReadyDriver` in `router_agents.py`. Nothing in them says which output they're on. The driver and the monitor each get an `OutCfg` from `ConfigDB`, and its `port` says which pins to use (`out0_ready` or `out1_ready`, and so on). The base test's `build_phase` puts an `OutCfg` in `ConfigDB` for the paths `env.out0*` and `env.out1*`, so output 1′s is already there for an agent named `out1`.
2. In `RouterEnv`, do the two Part 2 TODOs: make the agent for output 1 with `OutAgent.create("out1", self)` and add it to `self.out`, and connect its monitor's `ap` to the scoreboard's `out_imp.analysis_export`, as output 0′s is. Leave the scoreboard alone for now. Before you run, write down what you expect `RouterRandTest` to report on the golden router.
3. Run it:

   ```
   make run TEST=RouterRandTest
   ```

   It should fail on the golden router, with 50 errors that start:

   ```
     195.00ns ERROR   ..h-in-pyuvm/tb/router_env.py(105) [uvm_test_top.env.sb]: output
   1: got 40 last in cycle 20, and no beat was expected
     325.00ns ERROR   ..h-in-pyuvm/tb/router_env.py(105) [uvm_test_top.env.sb]: output
   1: got 42 last in cycle 33, and no beat was expected
     335.00ns ERROR   ..h-in-pyuvm/tb/router_env.py(105) [uvm_test_top.env.sb]: output
   1: got 83 in cycle 34, and no beat was expected
   ```

   and then `50 errors: stopping the test`, `DD105-B BaseTest: 50 errors logged` and `FAIL: 50 errors`. The cycle in each message is the monitor's count of falling edges since the start of the test, so cycle n is the clock period that holds the nth falling edge, which is where the monitor saw the beat. What is wrong, the router or the testbench?

   Answer: A3, in Answers at the end.
4. Do the Part 2 TODO in `RouterScoreboard.write_in`: queue every forwarded beat on `self.expected[self.in_port]`. Run `RouterRandTest` again. You should see:

   ```
    52280.00ns INFO     ..h-in-pyuvm/tb/router_env.py(221) [uvm_test_top.env.sb]: [SB]
   805 packets in: 314 to output 0, 490 to output 1, 1 NOROUTE, 0 FULL; the highest
   level a header saw was 16
    52280.00ns INFO     ..h-in-pyuvm/tb/router_env.py(224) [uvm_test_top.env.sb]: [SB]
   beats checked: 1328 on output 0, 2006 on output 1; 5 DROP_CNT reads checked
   ```

and `ok`. The 2006 beats that went unwatched in Part 1 are now checked. The number in parentheses after each file name is the line of the logging call, so it moves when you add or delete lines above that call. We left each TODO comment in place when we filled it in. If you deleted yours, your line numbers here and in Part 3 are a few lower, such as 218 and 221 for these two lines.

These are the same 805 packets as Part 1's run of this test, although the new agent's `OutReadyDriver` runs in every cycle. Read its `run_phase`: why did it change none of them?

Answer: A4, in Answers at the end.

5. Predict whether `make check-all` catches B06 or B13 now, then run it, and write the result in your table.

   Answer: A5, in Answers at the end.

Check: `RouterRandTest` 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: A6, in Answers at the end.

2. Do the Part 3 TODO in `test_router.py`: in `RouterStallTest`, override `set_sinks(k)` so that in phase `k` one output is ready in 5% of cycles and the other in 100%. Output 0 stalls in phases 1 and 3, and output 1 in phases 2 and 4. `RouterRandTest` calls `set_sinks(k)` at the start of each phase, with `k` from 1 to 4. Set `self.ocfg[p].ready_pct`. The base test made those `OutCfg` objects and kept them, and each `OutReadyDriver` reads its `ready_pct` again after every rising edge, so a change takes effect at the next 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. 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 TEST=RouterStallTest
   ```

   You should see phase 1 end with its `DROP_CNT` check, and phase 2 fail:

   ```
    13575.00ns INFO    ..h-in-pyuvm/tb/router_env.py(204) [uvm_test_top.env.sb]: [DROP]
   DROP_CNT 00007900 as expected
    13610.00ns INFO      ..-in-pyuvm/tb/test_router.py(250) [uvm_test_top]: [PHASE] 2
   of 4: routes 1-10, 200 packets; output 0 ready 100%, output 1 ready 5%
    16505.00ns ERROR   ..h-in-pyuvm/tb/router_env.py(105) [uvm_test_top.env.sb]: output
   1: got 9d in cycle 1651, expected 0b
    16995.00ns ERROR   ..h-in-pyuvm/tb/router_env.py(105) [uvm_test_top.env.sb]: output
   1: got 3d in cycle 1700, expected fe
   ```

   `FULL_CNT` is bits 15:8 of `DROP_CNT`, so phase 1 dropped 0x79, or 121, 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 TEST=RouterStallTest` stops after the golden run:

   ```
       FAIL  the golden design                       50 errors                    3.5s
           RouterStallTest: AssertionError: 50 errors logged, the first: output 1: got
   9d in cycle 1651, expect
   ----------------------------------------------------------------------
    VERDICT: FAIL -- your test does not pass the GOLDEN design.
   ```

   Why does phase 2 fail on the golden router?

Answer: A7, in Answers at the end.

5. Do the Part 3 TODO in `level()`, so that it gives output 1′s level the way it gives output 0′s. Run the stall test again. You should see:

```
 13575.00ns INFO    ..h-in-pyuvm/tb/router_env.py(202) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00007900 as expected
 28245.00ns INFO    ..h-in-pyuvm/tb/router_env.py(202) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00007400 as expected
 41775.00ns INFO    ..h-in-pyuvm/tb/router_env.py(202) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00002700 as expected
 54985.00ns INFO    ..h-in-pyuvm/tb/router_env.py(202) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00004c00 as expected
 55485.00ns INFO    ..h-in-pyuvm/tb/router_env.py(202) [uvm_test_top.env.sb]: [DROP]
DROP_CNT 00000001 as expected
 55490.00ns INFO     ..h-in-pyuvm/tb/router_env.py(219) [uvm_test_top.env.sb]: [SB]
804 packets in: 206 to output 0, 245 to output 1, 1 NOROUTE, 352 FULL; the highest
level a header saw was 16
 55490.00ns INFO     ..h-in-pyuvm/tb/router_env.py(222) [uvm_test_top.env.sb]: [SB]
beats checked: 935 on output 0, 1014 on output 1; 5 DROP_CNT reads checked
```

and `ok`. The four phases dropped 121, 116, 39 and 76 packets as FULL, 352 in all, and the scoreboard predicted each one. The random test dropped none.

6. Ask for the verdict on B13:

```
make check BUG=B13 TEST=RouterStallTest
```

You should see:

```
      ok  the golden design                        0 errors                    5.2s
  caught  B13_admission_checks_other_output       50 errors                    2.4s
       RouterStallTest: AssertionError: 50 errors logged, the first: output 0: got
cf in cycle 125, expecte
--------------------------------------------------------------------------------
 VERDICT: PASS
```

Phase 1 stalls output 0 and routes DST 2 and DST 3 to it (`-100`), and DST 1 to output 1. The first error is on output 0 in cycle 125, when `cf` came out. Which of B13′s two wrong decisions from step 1 put a wrong beat on output 0?

Answer: A8, in Answers at the end.

7. Run `make check-all` and write the result in your table. It prints `1 of 2 caught.`, because `RouterInjectTest` derives from `RouterStallTest` and so stalls the outputs too.

Check: `RouterStallTest` passes on the golden router with 352 FULL drops, and `make check BUG=B13 TEST=RouterStallTest` 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 `RouterRandSeq` in `router_agents.py`. It makes each packet with `RouterPkt.create("pkt")`, which asks the factory for a `RouterPkt`. If a test has told the factory to substitute a subclass of `RouterPkt`, the sequence gets the subclass without knowing it. Then read `RouterPkt`'s `c_dst` constraint, which is Lab 03′s.

2. Do the first Part 4 TODO: write `NoroutePkt`, which extends `RouterPkt` and sends one packet in four to a disabled route. Decorate it with `@vsc.randobj`, as `RouterPkt` is, and give it a constraint of its own named `c_dst`. A pyvsc constraint is a method, and a Python method with the same name as one in the parent class replaces the parent's, so `RouterPkt`'s `c_dst` no longer applies. Decide which packets

go to a disabled route with a non-random attribute, `self.to_disabled = vsc.uint8_t(0)`, which `pre_randomize()` sets from Python's `random`. pyvsc calls `pre_randomize()` before it solves, so `c_dst` reads `to_disabled` as a fixed value, the way it reads `route_en`. Draw it only when some route is disabled (`int(self.route_en) != 0xF`). Every routing table that `random_routes()` writes has a disabled route, but `RouterClassTest`'s `0110` doesn't, and if `to_disabled` were 1 with no route disabled, `c_dst` couldn't be met.

pyvsc also solves a `dist` beside another constraint (line 3 of Lab 03's experiments), so a random `to_disabled` with a `dist` works as well. We tried it, and it caught both bugs on seeds 1 to 5. The counts below come from the `pre_randomize()` version.

3. Do the second Part 4 TODO: in `RouterInjectTest`, add a `build_phase` that calls `uvm_factory().set_type_override_by_type(RouterPkt, NoroutePkt)` and then `super().build_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, so they have to stay plain `RouterPkt`s. It makes both in `new_packet()` with the constructor, `RouterPkt("pkt")`, which the override can't reach. When we made them with `create()` instead, `RouterInjectTest` ended `BROKEN: SolveFailure` on 2 of seeds 1 to 20. The packet for DST 0 was then a `NoroutePkt`, and whenever `pre_randomize()` set its `to_disabled` to 1, no DST could meet both its `c_dst` and `hdr_dst == 0`, since `ROUTE0` is the one enabled route.

4. Write down what you expect the golden run's first `[SB]` line to say about NOROUTE drops. Then run the default test:

   ```
   make run
   ```

   With the reference solution you should see:

   ```
    53890.00ns INFO     ..h-in-pyuvm/tb/router_env.py(212) [uvm_test_top.env.sb]: [SB]
   804 packets in: 193 to output 0, 196 to output 1, 218 NOROUTE, 197 FULL; the highest
   level a header saw was 16
    53890.00ns INFO     ..h-in-pyuvm/tb/router_env.py(215) [uvm_test_top.env.sb]: [SB]
   beats checked: 860 on output 0, 818 on output 1; 5 DROP_CNT reads checked
   ```

   and `ok: 0 errors  5.29 s`. 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 such a packet while its output had room. Your counts match ours only if your `pre_randomize()` makes the same draws from `random` as ours, so if they differ, compare your draw with the reference solution's once step 5 says `2 of 2 caught.` The line numbers in this output come from the files in `solutions/`, which have the TODO comments removed. If you kept yours, as we did for the runs in Parts 2 and 3, your line numbers are higher, such as 219 and 222.

5. Ask for the lab's verdict:

   ```
   make check-all
   ```

   You should see:

   ```
        ok  the golden design                        0 errors                    5.3s
   ------------------------------------------------------------------------------
     caught  tier 2  B06_disabled_route_forwards     50 errors                   2.6s
     caught  tier 5  B13_admission_checks_other_output 50 errors                 2.2s
   ------------------------------------------------------------------------------
    2 of 2 caught.
   ```

   Write the result in your table.

6. You can also apply a factory override from the command line, without a test class. pyuvm 4.0.1 has no plusarg for this, so the base test reads one of its own, `+OVERRIDE`, and calls `uvm_factory().set_type_override_by_name()` with the two class names before it builds anything.

Apply your override to Lab 03's test that way, on the golden router and then on B06, and time the first run:

```
time make run TEST=RouterRandTest PLUSARGS=+OVERRIDE=RouterPkt,NoroutePkt
make run TEST=RouterRandTest PLUSARGS=+OVERRIDE=RouterPkt,NoroutePkt BUG=B06
```

The first prints `[OVERRIDE] the factory makes NoroutePkt for every RouterPkt (+OVERRIDE)` at 0 ns, then `806 packets in: 333 to output 0, 264 to output 1, 209 NOROUTE, 0 FULL` and `ok`, in 5.2 s. The second is caught with 50 errors, and its first error is:

```
  545.00ns ERROR   ..h-in-pyuvm/tb/router_env.py(105) [uvm_test_top.env.sb]: output
0: got 04 in cycle 55, 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 TEST=RouterRandTest PLUSARGS=+OVERRIDE=RouterPkt,NoroutePkt
make waves TEST=RouterRandTest PLUSARGS=+OVERRIDE=RouterPkt,NoroutePkt BUG=B06
surfer build/runs/RouterRandTest-s1/waves.fst &
surfer build/runs/RouterRandTest-s1-B06/waves.fst &
```

Add `clk`, `in_valid`, `in_ready`, `in_data`, `in_last`, `out0_valid`, `out0_ready`, `out0_data` and `out0_last` from `router`, and `route_en_q`, `status_q` and `cnt_noroute_q` from `router.u_regs`, and look at 480 to 600 ns:

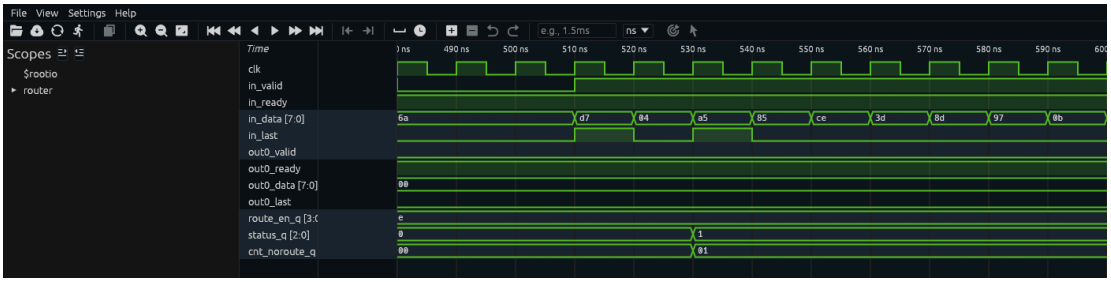


Figure 16.1: A packet for a disabled route on the golden router, in Surfer

The golden router. The header `04` (DST 0) is accepted at the edge at 530 ns, while `route_en_q` is `e` (routes 1, 2 and 3 enabled, route 0 disabled). `status_q` and `cnt_noroute_q` change at that edge (bit 0 of `status_q` is `STATUS.NOROUTE`), and `out0_valid` stays low.

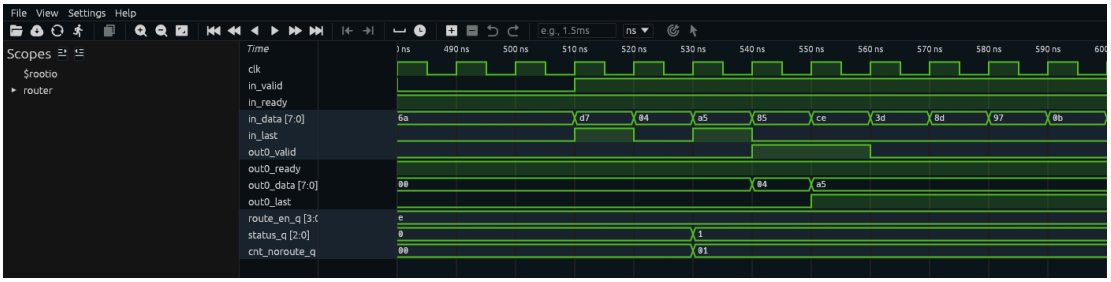


Figure 16.2: The same packet with B06 on, in Surfer

B06. The router counts the same NOROUTE drop, and then puts `04` and `a5` on output 0 in the cycles that end at 550 and 560 ns. The header comes out two cycles after the cycle in which it went in, which is when rule O8 says a forwarded beat first appears on an output.

Both pictures have the same input and the same register values. 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"
```

It took us about 97 s. You should see `25 ok  (25 runs)`, and then the coverage of those runs:

```
  top                                              82.93 %   34 of 41 bins
    top.router                                       82.93 %   34 of 41 bins
      top.router.pkt                                   82.93 %   34 of 41 bins
        top.router.pkt.len                              100.00 %   8 of 8 bins
        top.router.pkt.dst                              100.00 %   4 of 4 bins
        top.router.pkt.fate                             100.00 %   4 of 4 bins
        top.router.pkt.level                            100.00 %   5 of 5 bins
        top.router.pkt.x_fate_level                      65.00 %   13 of 20 bins
            not hit: ('full', 'empty'), ('full', 'low'), ('full', 'at_8'), ('out0',
'at_9'), ('out0', 'high'), ('out1', 'at_9'), ('out1', 'high')
```

The seven empty bins are the ones no test can hit, so this is as far as the percentage goes. The stall test's FULL drops hit the three bins of Part 1's second group, and the drops at a level of 9 hit the fourth group's `at_9` as well. The two bins of the third group are hit because `level()` now gives output 1's real level. The fourth group's four NOROUTE bins are hit now that `RouterInjectTest` sends packets to disabled routes while one output is stalled.

Check: `2 of 2 caught.` from `make check-all`, and the bin `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 coverage, the constraints and the `close_holes()` you wrote:

```
make loc FILES="../lab-03-constraints-and-coverage/tb/*.py"
```

With the reference solution's edits, the pyuvm part prints:

```
This lab's pyuvm testbench:
  file                  lines    code
  tb/router_agents.py     439     261
  tb/router_env.py        284     173
  tb/test_router.py       355     196
  total                  1078     630
```

Code lines leave out blank lines, comments and docstrings. As shipped, the pyuvm testbench was 1062 lines and 616 code lines, so the reference solution's edits added 14 code lines. With our Lab 03 solution in Lab 03's `tb/`, the `FILES:` part totals 1033 lines and 595 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 pyuvm piece that does its job, and write down whether its code moved across nearly unchanged or changed shape. Then list the pyuvm pieces that have no Lab 03 counterpart at all.

   Answer: A9, in Answers at the end.

3. Using Parts 2 to 4, what did the extra lines buy you in this lab?

   Answer: A10, in Answers at the end.

Check: your notes have a line count for both testbenches, and your table has a `make check-all` result for each part.

Finishing

You're done when `make check-all` prints `2 of 2 caught.` with the `RouterStallTest`, `NoroutePkt` and `RouterInjectTest` 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: A11, in Answers at the end.

2. In Part 4, step 7, why does the B06 run's `cnt_noroute_q` agree with the golden run's?

   Answer: A12, in Answers at the end.

Submit: Nothing to submit.

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: Run the port

cocotb seeds Python's `random` for each test from the seed and the test's name, as you saw in Lab 03, Part 4, step 6. pyuvm gives each test its class's name, so `RouterInjectTest` and `RouterRandTest` start from different seeds, and every routing table, packet and fill plan differs.

A2. Part 1: Run the port

`RouterPkt`'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 (14 on this seed: the fill of 4, 3 and 1 beats, then a 6-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 (`RouterClassTest` reported a highest level of 2 in step 3). In `close_holes()` output 0 fills, but the packet for DST 0 arrives when output 0 holds exactly 8 beats. B13 admits that packet too, because it judges from output 1, which is empty. The last packet goes to a disabled route, which is a NOROUTE drop on either router. No packet for an enabled route met an output holding more than 8 beats, which is also why `full` was never hit.

A3. 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.

A4. Part 2: Watch the second output

Every packet comes from Python's `random`, one draw after another, so a component that draws a number in between changes every packet after it, as you saw in Lab 03. `OutReadyDriver` draws only when `ready_pct` is between 0 and 100. Output 1's is 100, so its driver sets `out1_ready` to 1 in every cycle without drawing anything.

A5. 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.

A6. 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.

A7. Part 3: Stall one output at a time

`level()` returns 0 for output 1. That shortcut was safe while output 1 was always ready, which it was in every test before this one. With output 1 ready in only 5% of cycles, output 1 fills past 8 beats, and the router drops the next packets for it as FULL, as the specification says, while the scoreboard still expects them to come out. The packet starting `0b` has DST 0, which `1-10` routes to output 1, and the router dropped it in cycle 1396, when output 1 held 13 beats. So the next header to come out of output 1, `9d` (DST 2, also routed to output 1), was compared with `0b`. Output 0's level was right all along, which is why phase 1 passed.

A8. 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 (`make waves TEST=RouterStallTest BUG=B13`), the first such packet was the one with header `83` (DST 2), which arrived in cycle 40, from 390 to 400 ns. Output 0 held 11 beats then and output 1 none; the router's own counts, `level0` and `level1`, read `0b` and `00` in Surfer's hex. The golden router dropped the packet as FULL. B13 admitted it, and more packets for output 0 after it, and in cycle 59, from 580 to 590 ns, `level0` read `14`, which is 20 beats. The buffer holds 16, and it writes each beat at the next place in a ring, so the beats beyond 16 wrote over beats that were still waiting. Output 0 was ready in only 5% of cycles, so the first wrong beat took until cycle 125 to come out.

B13 also made the other wrong decision, from cycle 75 on. It dropped packets for output 1 as FULL because output 0 held more than 8 beats, and the first of them was `47` (DST 1). Those packets never came out of output 1, and the scoreboard's first error on output 1 came in cycle 185, when `5a` came out where it expected `47`. That was 60 cycles after the first error on output 0.

A9. Part 5: Count what the port cost

Lab 03	Lab 04	What happened
`Packet` and `RandPacket`	`RouterPkt`	moved: the fields and the constraints `c_dst`, `c_kind` and `c_len` are the same; it derives from `uvm_sequence_item` and is made through the factory
`Generator` and the queue to the driver	`RouterRandSeq` on the input agent's sequencer	changed shape: `start_item()` and `finish_item()` instead of a queue's `put()`
`Driver`	`SrcDriver`	moved: `get_next_item()` and `item_done()` replace the queue's `get()`, and the idle cycles and the drives are the same
`Sink` and its `stall` knob	`OutReadyDriver` and `OutCfg.ready_pct`	moved: the stall grew into a percentage, which the driver reads after every rising edge
`Monitor` and the queues to the scoreboard	`SrcMonitor`, `OutMonitor` and their analysis ports	changed shape: each beat goes to `ap.write()` as it moves, instead of a whole packet into a queue
`Scoreboard` and `Tracker`, each with its own `run()` coroutine	`write_in`, `write_out`, `write_apb`, `level()`	changed shape: one component, which the monitors call through its three `Fwd` inputs, with no coroutine; it predicts FULL drops as well, from the tracker's levels
`coverage.py`	the same coverpoints, and `RouterCoverage` as a subscriber	moved: the same bins, sampled from the scoreboard's `fate_ap`
`ApbMaster` with `write()` and `read()`	`ApbAgent`, `ApbRwSeq`, `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`, ...)	`RouterEnv`, and the steps as methods of `RouterLabTest`	changed shape: `build_phase` and `connect_phase` instead of a constructor that makes the queues

Lab 03	Lab 04	What happened
`FillPlan` and `close_holes()`	the same, in `test_router.py`	moved: the same steps in the same order, now sent through the agents' sequences
`RouterCfg` and its plusargs	plusargs read in the tests	moved, except `+SHOW`, which is gone
`n_errors()` and the `assert` at the end of each test	`self.logger.error()` and `BaseTest`	changed: `BaseTest` counts every component's errors, and the scoreboard stops the test at 50
three test functions	`RouterLabTest` and five test classes	moved: the three tests run the same steps, and two tests are new; `TEST=` picks one at run time, as it did in Lab 03
nothing	the agent classes, `OutCfg` and `ConfigDB`, the `Fwd` inputs	new

The code that touches the router (the drives, the sampling, the prediction) does the same work in both testbenches. In the port, the pieces find each other through `ConfigDB` and the factory instead of through the environment's constructor, and they pass data through sequencers and analysis ports instead of queues.

A10. Part 5: Count what the port cost

Watching output 1 took two lines in `RouterEnv` and one in the scoreboard, where the code from Lab 03 had queued only output 0's beats. Two lines were enough in `RouterEnv` because the output agent's driver and monitor get from `ConfigDB` everything that differs between the two outputs. Stalling one output took a short method in a new test, which you pick at run time, and it works because the test holds the `OutCfg` objects 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, and you could apply it from a test class or from the command line. Lab 03's testbench could have done each of these, but each would have meant editing classes that already worked. In DD105-A, whose testbench is SystemVerilog, each of these edits also cost a rebuild of 30 to 105 s, and applying the override from the command line saved one. Here none of them rebuilt anything, and the port has 35 more code lines than our Lab 03 solution (630 against 595).

A11. 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.

A12. 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

17 Assertions for interface rules 139
17 Three rules 139
17 A rule as a property 139
17 Binding the checker into the design 140
17 Failure messages on Verilator 140
17 Reading the count in the test 141
17 When the antecedent never holds 142
17 Which rules suit an assertion 142
17 Check yourself 143
17 Further reading 143
17 Answers 143

18 Reactive agents and backpressure . 145
18 Who starts a transfer 145
18 A sequence that never ends 146
18 A driver that can't wait 146
18 Measured waits for each sink 147
18 Check the distribution you got 149
18 The sink's configuration object 149
18 Check yourself 149
18 Further reading 149
18 Answers 150

19 Virtual sequences 151
19 A sequence given the environment 151
19 Doing two things at once 152
19 A loop that ends with the traffic 153
19 Two sequences on one sequencer 154
19 Configuration changes and the scoreboard . . . 154
19 Check yourself 154
19 Further reading 154
19 Answers 155

20 Lab 05: Protocol rules and backpressure 157
20 Before you start 157
20 Part 1: The Lab 04 testbench against this lab's bugs 158
20 Part 2: Assertions for the output rules 160
20 Part 3: A sink that answers each beat 162
20 Part 4: A virtual sequence 164
20 Finishing 166
20 Answers 167

17. 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. That lab starts from our Lab 04 testbench, which already catches two of Lab 05's three planted bugs, B10 and B12. On B10, the first error it logs is `output 0: got e5 last in cycle 34, expected 80`. 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. If a beat is changed or lost while it waits on a stalled output, the scoreboard can't tell until the sink next takes a beat. Part 1 of the lab has you work out the cycle in which the router broke its rule, and how many cycles before that line it was.

A SystemVerilog **concurrent assertion** is a check that watches the design's pins. It fires in the cycle in which a rule is broken, and its message can say which rule it was. Assertions are the only SystemVerilog you write in this course. The testbench stays in Python, and the assertions go in a module of their own, which a `bind` statement puts inside the design. 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.

The examples on this page use a smaller design of our own, a one-beat valid/ready stage, so that they don't write those three assertions for you. 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 pyuvm 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 `make run BUG=` switches one of them on when the simulator starts, as it does for the router's bugs. 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.

The test, `StageTest`, runs for 200 cycles. In each cycle where it has no beat waiting on the input, it puts a new one there with probability 70%, and in every cycle it sets `out_ready` to 1 with probability 50%. The test checks nothing itself. The checker does all the checking, and at the end of the test `BaseTest` reads how many assertion failures the checker counted.

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. Your drivers change the inputs right after a rising edge and your monitors read the pins at the falling edge in the middle of the cycle, and nothing changes between that falling edge and the next rising edge. So in each cycle the property sees the same values that your monitor reads, which is also what the specification means by the value of a signal "in a cycle".

The expression to the left of `|=>` is the **antecedent**, here `out_valid && !out_ready`. It picks the cycles that the rule talks about, which for S1 are 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 adds one to the assertion's own counter, `n_S1`, 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 `|->`, and 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`. Its inputs are the stage's pins. It keeps one counter for each assertion, and `n_fail` is the sum of the three. 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 example's Makefile lists the file in `SVA_SRCS`, and `labs/common/pyuvm.mk` compiles it with the design. cocotb's build passes Verilator `--public-flat-rw`, which makes every signal reachable from Python, including the bound module's counters, so `BaseTest` reads the failure count through the design as `cocotb.top.u_sva.n_fail`. Lab 05 binds its checker into the router the same way, with the `bind` statement in a file of its own, `tb/router_sva_bind.sv`. Part 2 of that lab asks you why the testbench attaches its checker with `bind`.

Failure messages on Verilator

The checker is plain SystemVerilog, so the simulator reports its failures itself, and the messages don't look like the lines your pyuvm components log. Here is the `drop` bug with Verilator's own error limit, which is 1. The course's makefile raises the limit for every run of a folder with `SVA_SRCS`, and `ERROR_LIMIT=1` sets it back to 1, so the command is `make run BUG=drop ERROR_LIMIT=1`:

```
[720000] %Error: stage_sva.sv:41: Assertion failed in stage.u_sva.a_S1: S1: out_valid
fell without a handshake
```

That line is followed by a second `%Error` line, which says `Verilog $stop`, and then `Aborting...`. The number in brackets is the simulation time in picoseconds, so the failure was at 720 ns. Our clock has a 10 ns period, and, as in the labs, cycle n is the period that ends at the rising edge at $10n$ ns, so the failure was found at the edge that ends cycle 72. A `|=>` failure is reported in the cycle of the consequent, so the beat was still waiting in cycle 71, and `out_valid` was 0 in cycle 72. The waveform of the same run shows both cycles:



Figure 17.1: The stage with the drop bug, 640 to 750 ns, in Surfer

The sink takes `21` at the edge at 670 ns, and `22` goes onto the output at the same edge. `22` waits in cycles 68 to 71, from 670 to 710 ns, with `out_ready` at 0. At 710 ns, the start of cycle 72, `out_valid` falls although the sink never took `22`, and at the edge at 720 ns, which ends cycle 72, the assertion fails and `n_S1` counts it. (`23` went in during cycle 72, so the output is busy again from 720 ns until the sink takes `23` at the edge at 730 ns.)

The message names the assertion by its full path, `stage.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, at 720 ns of the 2,020 ns that a full run takes, and the simulator's exit status was 1. cocotb marked `StageTest` failed, and `BaseTest`'s line at the end of the test never printed, because the test never got there. The course's grader reads the log as well as cocotb's results, and it called the run caught, with a note:

```
 caught: 1 assertion   0.26 s   log: build/runs/StageTest-s1-drop/run.log
   stage_sva.sv:41: Assertion failed in stage.u_sva.a_S1: S1: out_valid fell without
a handshake
   note: Verilator stopped the run at its error limit, so the rest of the test didn't
run. Runs through make raise it (+verilator+error+limit+N).
```

The makefile sets the limit with the run-time option `+verilator+error+limit+<value>`. Without `ERROR_LIMIT=1`, `pyuvm.mk` passes `+verilator+error+limit+100000`. No run comes near that many failures, so `make run BUG=drop` prints every failure of the bug:

```
[720000] %Error: stage_sva.sv:41: Assertion failed in stage.u_sva.a_S1: S1: out_valid
fell without a handshake
[1010000] %Error: stage_sva.sv:41: Assertion failed in stage.u_sva.a_S1: S1: out_valid
fell without a handshake
[1850000] %Error: stage_sva.sv:41: Assertion failed in stage.u_sva.a_S1: S1: out_valid
fell without a handshake
[1910000] %Error: stage_sva.sv:41: Assertion failed in stage.u_sva.a_S1: S1: out_valid
fell without a handshake
[1990000] %Error: stage_sva.sv:41: Assertion failed in stage.u_sva.a_S1: S1: out_valid
fell without a handshake
DD105-B BaseTest: 0 errors logged, 5 assertion failures (u_sva.n_fail)
```

After the first failure Verilator also prints, once, an `-Info` line saying that the `$stop` was ignored because of `+verilator+error+limit`. This time the test ran to its end. `BaseTest` read `n_fail` in its `final_phase`, printed the line above and raised `AssertionError`, and the grader said `caught: 0 errors, 5 assertions`. In that line, `0 errors` means that no pyuvm component logged an error, and `5 assertions` means the checker's five failures.

Reading the count in the test

pyuvm knows nothing about the checker. A failed assertion prints an `%Error` line and adds to a counter, and neither of those raises anything in Python, so a test that never reads the counter passes as if no assertion had failed. The example's `+UNCOUNTED` makes `StageTest` skip `BaseTest`'s read of `n_fail`, to show what happens in a test that never reads it. On the same seed the stimulus is the same, so `make run BUG=drop PLUSARGS=+UNCOUNTED` printed the same five `%Error` lines, and then:

```
DD105-B BaseTest: 0 errors logged, no bound assertions
```

and cocotb's summary said `PASS` for the test, as `results.xml` did. The course's grader also searches the log for failed assertions, so it still said `caught: 0 errors`, with the note `5 assertion failure(s) are`

`in the log and the test passed: it doesn't read u_sva.n_fail, as BaseTest does.` A grader that read only cocotb's results would have taken the run for a pass, and so would you if you read only cocotb's summary. A testbench that relies on assertions therefore has to turn their count into a failing test itself. Every pyuvm test in the labs derives from `BaseTest`, which reads `cocotb.top.u_sva.n_fail` at the end of every test of a design that has a checker bound into it, and fails 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 (`make run BUG=drop PLUSARGS=+READY_PCT=100`):

```
[STAGE] 147 beats in, 146 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. Your O6 assertion in Lab 05 misses B11 at first, 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 has another counter, `n_tried`, for the cycles in which the antecedent of S1 and S2 holds. An `always` block beside the assertions adds one to it in each of those cycles, and on the golden stage, with the sink ready half the time, it counted 83. The checker also counts the same cycles a second way, with a cover property whose pass statement adds one to `n_covered`:

```
c_S1: cover property (@(posedge clk) disable iff (!rst_n) out_valid && !out_ready)
  n_covered++;
```

With the course's build flags that statement never ran, and `n_covered` was 0 in every run on this page, the golden one included. It does run in a build with Verilator's user coverage switched on. We built the stage once with `make build VFLAGS=--coverage-user` and ran it with the same `VFLAGS`. In that run `n_covered` counted the same 83 cycles as `n_tried`, and the run wrote Verilator's own coverage file, `coverage.dat`. The labs build without that switch, and `make cov` reads cocotb-coverage's files, not `coverage.dat`, so in this course the evidence that a rule was exercised comes from a counter that you write yourself, like `n_tried`, or from a cocotb-coverage bin. Lab 05′s coverage group `router.wait` has bins of that kind. For each packet that an output delivers, it records the longest that any of the packet's 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. An assertion can't check them directly, and it catches a design that breaks one of them only if the design breaks another rule as well.
- 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 which beat came out wrong when a sink took it. An assertion names the rule that was broken and the cycle in which it was broken, which tells you where and when to look. A testbench needs both kinds of check, and in Lab 05 you'll see both fire on the same bugs, several cycles apart.

Check yourself

1. The `drop` run's first failure printed `[720000]`. 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, then `DD105-B BaseTest: 0 errors logged, no bound assertions`, and cocotb reports `PASS`. What is 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, which is the same in this course because the assertions are SystemVerilog. Their example output comes from commercial simulators, so the failure messages look different from Verilator's, and their testbenches are SystemVerilog modules, where yours is a pyuvm test reading a counter.

- 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.

Next: Reactive agents and backpressure.

Answers

A1. Check yourself
720 ns is the edge that ends cycle 72 ($10 \times 72 = 720$), so `out_valid` was 0 in cycle 72. The antecedent held in cycle 71, 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 72.

A2. Check yourself
The test isn't reading the checker's count. The `%Error` lines show that a checker is bound and its assertions fail, but "no bound assertions" means that `BaseTest` didn't find `cocotb.top.u_sva.n_fail`. Either the checker is bound under an instance name other than `u_sva`, or its sum of the counters isn't called `n_fail`, or the test has changed `SVA_FAILS`, as `+UNCOUNTED` does in the example. A failed assertion raises nothing in Python, so without that read the test passes.

A3. Check yourself
`a beat waited in 0 cycles` is `n_tried`. Give every assertion you rely on a count like it, or a coverage bin that shows its antecedent held. The cover property's count beside it is 0 too, but it was 0 on the golden stage as well, so it tells you nothing in this build.

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.

18. 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 gave the router backpressure, but the only thing you could set was `ready_pct`, and with it you can't make a particular beat wait a number of cycles that you choose. In Lab 05, Part 1, the coverage group `router.wait` takes each packet that an output delivers and records the longest that any of its beats waited for `out_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 the group crosses them with the two outputs. With Lab 04's sinks, `make cov` reports:

```
      top.router.wait                                92.86 %   13 of 14 bins
        top.router.wait.port                          100.00 %   2 of 2 bins
        top.router.wait.wait                          100.00 %   4 of 4 bins
        top.router.wait.x_port_wait                    87.50 %   7 of 8 bins
            not hit: ('out1', 'short_wait')
```

No packet on output 1 had a longest wait of 1 to 3 cycles. In Part 3 you'll finish a sink driver that answers each beat separately, with an item from a sequence. On this page we build an agent like that in pyuvm for a small design of our own, and measure how long the beats wait with each kind of sink. The files are in `guide-examples/reactive-agents-and-backpressure/`.

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, which `+MODE=` picks. In `coin` mode its driver does what Lab 04's did. In `beat` mode the driver answers each beat with one `SnkItem`, and the item says how many cycles the beat waits after it first appears:

```
class SnkItem(uvm_sequence_item):
    def __init__(self, name="snk_item"):
        super().__init__(name)
        self.delay = 0
        self.max_delay = 12

    def randomize(self):
        self.delay = random.randint(0, self.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:

```
async def body(self):
    make = vsc_item_class() if self.cfg.item == "vsc" else SnkItem
    while True:
        it = make("snk_item")
        await self.start_item(it)
        it.max_delay = self.cfg.max_delay
        self.timed(it.randomize)
        await self.finish_item(it)
```

(`vsc_item_class()` makes the pyvsc version of the item that one of the measurements below uses, and `timed()` calls `randomize()` and adds up how long it took.) The test starts this sequence with `cocotb.start_soon(seq.start(self.snk.sqr))`, so the sequence runs beside the test's `run_phase`, and the test never waits for it to return. A pyuvm sequence raises no objection of its own. The test raises one, and it drops it once the monitor has counted 2,000 beats, so the test's objection alone decides when the run ends. Lab 05's base test starts a `SinkSeq` on each output the same way, before it resets the router.

pyuvm hands the item over in two steps. `start_item()` puts the item in the sequencer's queue and waits there until the driver's `get_next_item()` takes it. When `start_item()` returns, the sequence sets the item's fields and calls `finish_item()`, which tells `get_next_item()` that the item is ready and then waits until the driver calls `item_done()`. So the sequence is never more than one item ahead of the driver, and it decides each item's values only when the driver asks for that item. If the sequence is already waiting in `start_item()`, no simulation time passes between the driver's call to `get_next_item()` and its return. In the run with per-beat delays from `random.randint`, the driver's `[DRV]` line at the end said:

```
[DRV] get_next_item() returned 2000 items, 1 of them at a later time than it was called
(called at 5 ns, returned at 10 ns)
```

The one late item was the first, which the driver asked for at 5 ns, before the test had started the sequence at 10 ns. Every other `get_next_item()` returned in the time step in which the driver called it.

A driver that can't wait

The driver's loop has two halves, one for each half of the clock cycle. Here it is without the countdown of the item's delay or the expression for the next cycle's `ready`, because in Part 3 you'll write both of them for the router's sink driver:

```
while True:
    await RisingEdge(d.clk)
    if cfg.mode == "coin":
        rdy = int(random.randrange(100) < cfg.ready_pct)
    d.ready.value = rdy                          # ready for the cycle that starts here
    if cfg.mode == "coin":
        continue
    await FallingEdge(d.clk)                      # the middle of the cycle
    v = bool(d.valid.value)
    if v and rdy:                                 # a handshake at the next edge
        self.seq_item_port.item_done()
        it = None
    ...                                           # one more cycle of this beat's wait
    if it is None:
        it = await self.seq_item_port.get_next_item()
        ...
    rdy = ...                                     # ready in the next cycle
```

It drives `ready` just after the rising edge and looks at `valid` at the falling edge in the middle of the cycle. That's the timing rule that the course has followed since Lab 02. At the falling edge the driver can see `valid`, and it knows the `ready` it drove for this cycle, so it can tell whether a beat moves at the next rising edge. If one does, that beat has had its answer, so the driver calls `item_done()` and asks for the item for the next beat.

A pyuvm driver gets its items with `get_next_item()`, which waits until its sequence hands one over. pyuvm has no call that returns at once when the sequence has nothing ready. The sink's driver can't afford to wait, because it has to drive `ready` at the next edge whatever happens. It gets away with `get_next_item()` because the sequence never ends and is always waiting in `start_item()` with the next item, so the call comes back in the same time step, as the `[DRV]` line showed. When no sequence is running on the sink, the call doesn't come back at all. With `+NOSEQ` the test starts no sequence on the sink. We ran it that way in beat mode for 3,000 cycles (`+MAX_CYCLES=3000`), and at the end it printed:

```
[WAIT] per beat, delays 0 to 12 from random.randint: 0 beats taken in 3001 cycles
[DRV] get_next_item() returned 0 items, 0 of them at a later time than it was called;
it has been waiting in get_next_item() since 5 ns
```

The driver drove `ready` to 0 at the first rising edge, called `get_next_item()` at the first falling edge, and never got past it, so `ready` stayed 0 and the first beat waited for the whole run. Lab 05's `SinkDriver` calls `get_next_item()` in the same place, at the falling edge, and Part 3 asks you why it needs a sequence that never ends.

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 `router.wait`'s bins, but it counts every beat, where `router.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%	82	289	911	718	18.98
coin flip, 50%	973	890	137	0	1.06
coin flip, 95%	1890	110	0	0	0.06
per beat, delays 0 to 12 from `random.randint`	164	465	1371	0	6.03
per beat, delays 0 to 12 from pyvsc's `randomize()`	143	380	1477	0	6.48

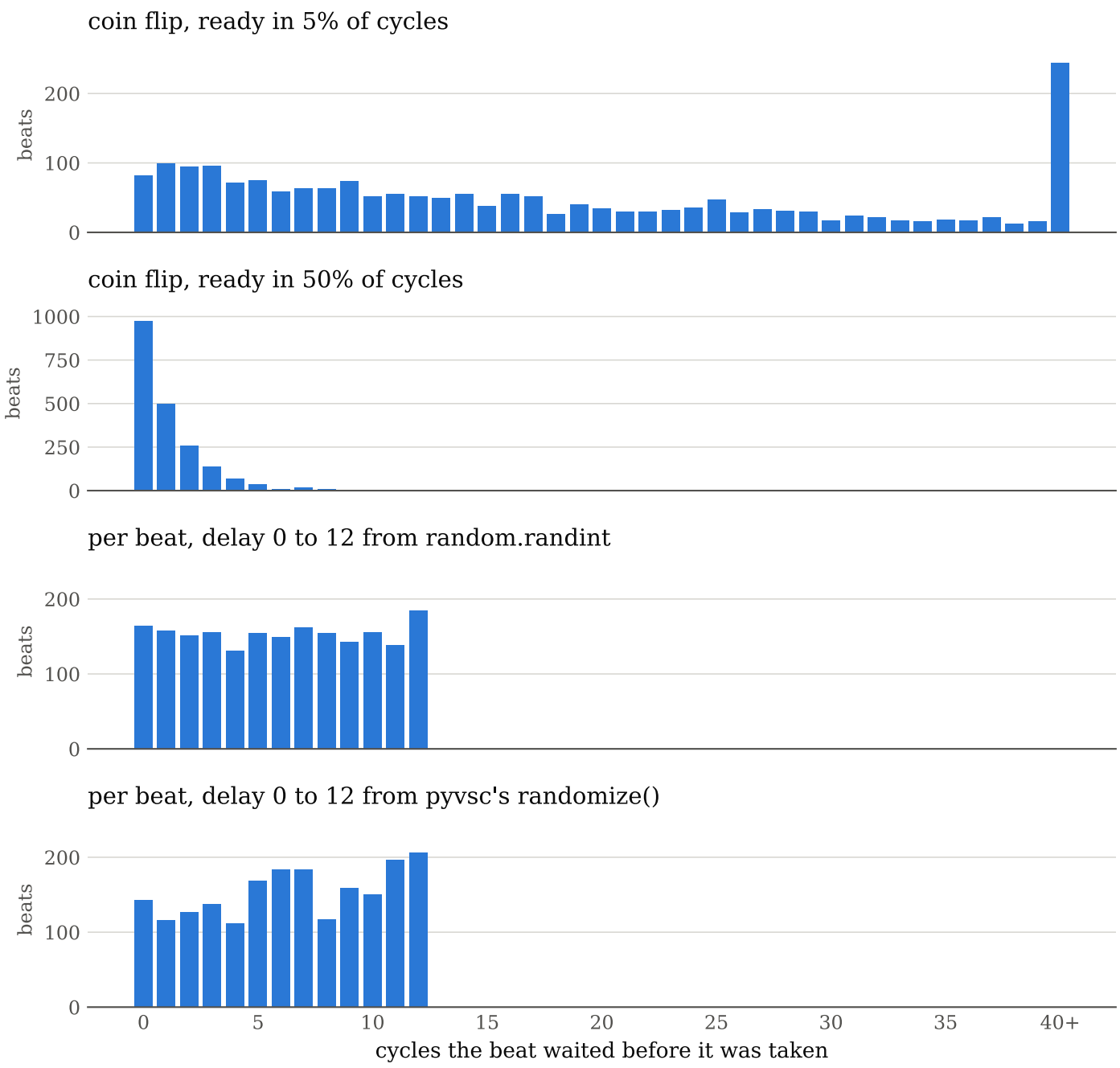


Figure 18.1: How long each of 2,000 beats waited, for four of the sinks in the table

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 $p = 5\%$, 1 cycle at 50% and 0.05 at 95%, against our measured 18.98, 1.06 and 0.06. 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%, 1,629 of the 2,000 beats waited 4 cycles or more, and only 289 waited 1 to 3. At 50%, no beat waited more than 10 cycles. At 95%, 1,890 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 in the run at the top of this page output 1 delivered none.

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, 1,371 of the beats with `random.randint` and 1,477 with pyvsc waited 4 to 20 cycles, and none waited more than 20. Lab 05′s `OutCfg.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

If the sequence picked each of the thirteen delays from 0 to 12 equally often, each would come up about 154 times in 2,000. With `random.randint` every value came up between 131 and 184 times on seed 1. The pyvsc item leaves the delay to pyvsc's solver, with the constraint `self.delay <= self.max_delay` on an 8-bit random field, and its values weren't spread evenly. On seed 1 the delays 11 and 12 came up 196 and 206 times, and 1 and 4 only 116 and 112 times. On seeds 2 and 3, 12 came up 225 and 235 times, and on all three seeds each of the values 0 to 4 and 8 came up fewer than 154 times. As in Lab 03, a constraint tells you which values are allowed, not how often each one comes out, so count what the stimulus actually produced.

The solver is also slow for a job that runs once per beat. The 2,000 calls to pyvsc's `randomize()` took 2.02 s, which is 988 a second, and the run as a whole took 6.36 s against 1.04 s with `random.randint`. The router's sinks make one item for every beat that leaves the router, so Lab 05′s `ReadyItem` draws from Python's `random` instead. Its `randomize()` makes half the items `pre_ready`, which means `ready` is already 1 when the beat appears. It picks each item's delay in two steps, first the range (0 for half the items, 1 to 3 for three in ten, 4 to 12 for the rest) and then a value inside it. A `ReadyItem`'s delay counts from the beat's first cycle on offer, so a beat whose item isn't `pre_ready` waits one cycle more than the delay, 1 to 13 cycles in all. In Lab 05, Part 3 you see the waits that these items give in `router.wait`.

The sink's configuration object

The sink's driver reads its mode and `ready_pct` from a configuration object, `SnkCfg`, 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, by writing their `OutCfg`, 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. `SinkTest` starts the sink's sequence with `cocotb.start_soon(seq.start(self.snk.sqr))`. What would happen if it wrote `await seq.start(self.snk.sqr)` instead?

   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

ChipVerify's pages are about SystemVerilog UVM. The ideas carry over, but pyuvm's driver has `get_next_item()` and no `try_next_item()`, and a pyuvm sequence raises no objection.

- 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()`, which pyuvm doesn't have.
- 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.

- 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

The sequence's `body()` loops forever, so `seq.start()` never returns, and the test's `run_phase` would wait at that line for the rest of the run. It would never reach the wait for 2,000 beats or drop its objection, so the run would go on until something stopped it from outside. In the course that's `make`'s run timeout, and a run that it stops is graded BROKEN.

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. With a coin-flip sink, a particular beat is sure to wait 0 cycles only when `ready_pct` is 100.

19. 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?*

In Lab 04 the tests did the work themselves, one step at a time. In each phase a test set the sinks, wrote the routing table, sent the packets, drained, read `DROP_CNT` and cleared it, and each step waited for the one before it to finish. In Lab 05, Part 4 you'll want to move routes to the other output while packets are still going in, and to change how both sinks behave at the same time. The APB agent, the source agent and both sinks' agents then all have work to do at once, and you decide what each of them does and when. 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 sequence, `SrcSeq`, sends 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, and 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 sequence given the environment

An ordinary sequence runs on its agent's sequencer, which passes its items to the driver. A virtual sequence has no driver to feed, so in pyuvm it runs on no sequencer at all, and it needs some other way to reach every agent that it steers. (In SystemVerilog UVM a virtual sequence usually runs on a virtual sequencer, which holds a handle to each agent's sequencer, and the sequence reaches the virtual sequencer through a handle called `p_sequencer`.) pyuvm has no `p_sequencer`, so the sequence is given the environment when it's made, and keeps it:

```
class BaseVseq(uvm_sequence):
    def __init__(self, name="vseq", env=None):
        super().__init__(name)
        self.env = env
        self.traffic_done = False

    def info(self, msg):
        self.env.logger.info(f"[VSEQ] {msg}")
    ...
    async def send(self, n, base):
        await SrcSeq(f"src_{base:02x}", n, base).start(self.env.src.sqr)

    def sink(self, pct):
        self.env.snk_cfg.ready_pct = pct
        self.info(f"sink ready {pct}% of cycles")
```

`send()` runs the source agent's own sequence on the source agent's sequencer, as a test would. `sink()` only writes the knob, without starting anything, and the sink's driver reads the new value at its next rising edge. The test starts the virtual sequence with `start()` and no sequencer:

```
v = cocotb.start_soon(self.VSEQ("v", self.env).start())
```

Lab 05's `RouterTest` starts its virtual sequence the same way, with `await RouterLab05Vseq("vseq", self.env).start()`. Lab 05's `RouterBaseVseq` gives you helpers of the same kind, among them `route()`, `ctrl()`, `burst()`, `send_rand()`, `sink()`, `stall()` and `drain()`, and most of them are the base test's steps from Lab 04, rewritten as methods of a sequence.

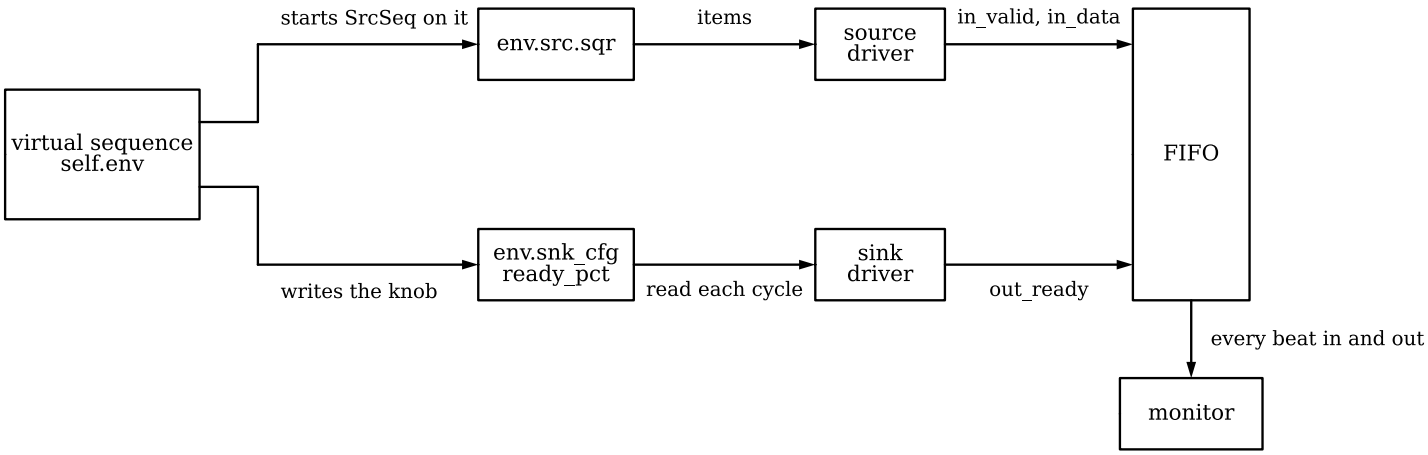


Figure 19.1: The example's testbench: one virtual sequence steering two drivers

Our example's testbench. Using the environment it was given, the virtual sequence reaches the source through the source's sequencer and the sink through the sink's configuration object. The monitor checks every beat, whatever caused it.

Because it runs on no sequencer, a virtual sequence can't send an item itself. `VseqItemTest` runs a virtual sequence whose `body()` calls `start_item()`. pyuvm raised a `UVMSequenceError` at that first call, and the graders called the run BROKEN:

```
BROKEN: UVMSequenceError   0.27 s   log: build/runs/VseqItemTest-s1/run.log
  VseqItemTest: UVMSequenceError: Tried start_item in a virtual sequence v
```

A pyuvm sequence has no logger of its own, because it isn't a component, so `info()` reports through the environment's logger. Every message from the virtual sequence therefore carries the environment's path, `[uvm_test_top.env]`, and you tell its lines from the environment's own by the `[VSEQ]` tag at the start of each message. In Lab 06 you'll pass the environment's logger to each register sequence you make, and the register sequences report through it.

Doing two things at once

The test starts our virtual sequence, `DemoVseq`. Its first step stalls the sink, and then sends six beats in one branch while a second branch waits 15 cycles and makes the sink ready in every cycle again. `par()` starts each coroutine it's given as a cocotb task, and then waits for each task in turn, so it returns when all of them are done:

```
@staticmethod
async def par(*coros):
    tasks = [cocotb.start_soon(c) for c in coros]
    for t in tasks:
        await t
```

and the first step is

```
self.sink(0)

async def beats():
    await self.send(6, 0x10)
    self.info("step 1: all 6 beats have gone in")

async def release():
    await self.cycles(15)
    self.info(f"step 1: the FIFO holds {int(d.level.value)} beats")
    self.sink(100)
```

```
await self.par(beats(), release())
```

It printed:

```
    10.00ns INFO    ..tual-sequences/test_vseq.py(172) [uvm_test_top.env]: [VSEQ] sink
ready 0% of cycles
   160.00ns INFO    ..tual-sequences/test_vseq.py(172) [uvm_test_top.env]: [VSEQ] step
1: the FIFO holds 4 beats
   160.00ns INFO    ..tual-sequences/test_vseq.py(172) [uvm_test_top.env]: [VSEQ] sink
ready 100% of cycles
   200.00ns INFO    ..tual-sequences/test_vseq.py(172) [uvm_test_top.env]: [VSEQ] step
1: all 6 beats have gone in
```

By 160 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 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 `+SERIAL` the example runs them in that order, and the test's own limit of 2,000 cycles ended the run:

```
  20010.00ns ERROR     ..tual-sequences/test_vseq.py(281) [uvm_test_top]: [TEST] the
virtual sequence is still running after 2000 cycles; the FIFO holds 4 beats
 20010.00ns INFO     ..tual-sequences/test_vseq.py(147) [uvm_test_top.env.mon]: [MON]
4 beats in, 0 out, 0 out of order
```

With `par()` the two branches run at the same time, and the step ended at 200 ns, when the last beat had gone in.

`par()` awaits each task, and awaiting a cocotb task re-raises any exception that the task raised, so a branch that fails also fails the sequence and the test. One of `labs/ref`'s probes of cocotb 2.0.1 waits on a task's `complete` trigger instead, and never calls the task's `result()`. That test passes, and the task's exception is lost. So if you wait on `complete`, as our test does to give up after 2,000 cycles, call the task's `result()` afterwards, which raises the exception if there was one.

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`:

```
self.traffic_done = False

async def traffic():
    await self.send(30, 0x40)
    self.traffic_done = True
    self.info("step 2: all 30 beats have gone in")

async def backpressure():
    while not self.traffic_done:
        self.sink(random.choice([0, 30, 100]))
        await self.cycles(random.randint(4, 12))

await self.par(traffic(), backpressure())
```

The loop set the sink eleven times between 200 ns and 1,070 ns, and the last beat went in at 1,130 ns. The loop saw the flag the next time it checked, at 1,150 ns, and then `par()` returned. 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 `par()` 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. At the end of the run, after all three steps, it printed `[MON] 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`:

```
await self.par(self.send(4, 0x80), self.send(4, 0xA0))
```

and the beats came out of the FIFO as

```
step 3: the sink took 80 a0 81 a1 82 a2 83 a3
```

pyuvm's sequencer keeps one queue. Each `start_item()` puts its item at the back of it, and the sequencer hands items to the driver from the front, in the order they were asked for. Each sequence asks for its next item only after the driver has finished its last one, 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 if you start a second sequence on a sequencer while another one is running there, the two share the driver, and pyuvm 4.0.1′s sequencer has no way for one sequence to keep the driver to itself. Each of Lab 05′s sinks runs a sequence that never ends, and Part 4 asks you why `sink()` and `stall()` change the sink's `OutCfg` rather than starting a sequence on the sink's sequencer.

Configuration changes and the scoreboard

With a virtual sequence you can 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 cycle in which the write completes still uses the old route. So in Lab 05 the APB monitor stamps each transfer with the cycle it completed in, and the scoreboard applies a route write only at the first header of a later cycle. Part 4 asks you why it goes by the cycle rather than by the order in which the two monitors report.

Adding a virtual sequence to a test can also change the random values that the test draws after it. Every random value in the labs comes from Python's `random`, one after another, so a sequence that draws values early in a test moves every value drawn after it. Making a pyuvm sequence and starting it draws nothing, though, so only the sequence's own draws move the values after them. Regressions and seeds has the measurements. In Lab 06, Part 5 you'll see the packets of Labs 04 and 05 change because of eight values drawn early in `RouterTest`.

Check yourself

1. In step 1, why can't `self.sink(100)` simply follow `await self.send(6, 0x10)` in the same branch?

   Answer: A1, in Answers at the end.

2. What does `par()` 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

ChipVerify's pages describe SystemVerilog UVM, where a virtual sequence usually runs on a virtual sequencer and reaches the agents' sequencers through `p_sequencer`. pyuvm has neither, and the labs use the pattern on this page, a sequence given the environment.

- UVM Virtual Sequence: both of SystemVerilog UVM's styles, `p_sequencer` and `start(null)`; the second is the one closest to pyuvm's. 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 Virtual Sequencer: the sequencer that holds handles, which pyuvm doesn't need. Its log was made from a different version of the code than the page shows, so the names in it don't match.

- UVM Sequence Arbitration: how a SystemVerilog UVM sequencer chooses between sequences that run on it at the same time. pyuvm 4.0.1's sequencer has one order, first come first served.
- 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. The `+SERIAL` run puts it there, and after 2,000 cycles its monitor had counted 4 beats in and 0 out.

A2. Check yourself

It waits for both tasks 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. In our example that's the test's limit of 2,000 cycles; in the labs it's the test's watchdog (`+MAX_CYCLES`) or `make`'s run timeout.

A3. Check yourself

Await them one after the other instead of in `par()`: `await self.send(4, 0x80)` and then `await self.send(4, 0xA0)`. The second sequence then starts only when the first has returned.

20. Lab 05: Protocol rules and backpressure

In this lab you'll write three SystemVerilog assertions for the router's output rules. The testbench binds them into the router, and the base test fails any test in which one of them fails. Then you'll finish a sink driver that answers each output beat with an item from a sequence, and write a virtual sequence that changes the routing table, the traffic and the backpressure together. You start from our Lab 04 testbench, which already catches two of this lab's three planted bugs, B10 and B12. Your assertions fail on both of them as well, and each failure names a rule that the router broke. On B10 your first assertion failure comes 8 cycles before the scoreboard's first error. The third bug, B11, shows only on packets of more than 8 beats. No Lab 04 test sends packets that long, so you'll send them from your virtual sequence.

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 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 agent for output 1, the scoreboard's queue of output 1's beats, a `level()` that gives output 1's real level, `RouterStallTest`, `NoroutePkt` and the factory override. If your own Lab 04 printed `2 of 2 caught.`, your code does the same job, but use ours anyway, because this lab's additions are already in our copies of the same three files, `router_agents.py`, `router_env.py` and `test_router.py`. We added these pieces 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_agents.py`	`SinkDriver`, which replaces `OutReadyDriver` and can answer each beat with a `ReadyItem` from the sink's sequence, `SinkSeq` (you finish the driver in Part 3); an oversize length kind in `RouterPkt`, switched off until a sequence gives it a weight; how long each output beat waited, which the output monitor stamps on the beat; the cycle in which each APB transfer completed, which the APB monitor stamps on the transfer; and `gap_one_in` and `w_over` on `RouterRandSeq`, which `send_rand()` sets (Part 4)
`tb/router_env.py`	the coverage group `router.wait`, an `oversize` bin in `router.pkt.len`, and a scoreboard that follows route writes made while packets are coming in
`tb/router_vseq.py`	the virtual sequences; you write `RouterLab05Vseq` (Part 4)
`tb/test_router.py`	`RouterTest` (Lab 04's `RouterInjectTest`, then your virtual sequence), `RouterLab05Test` (the sequence alone) and `RouterSinkTest` (Part 3); the base test now starts a `SinkSeq` on each output

Time: about 2 h. Part 2's edit is SystemVerilog, so it rebuilds the simulator, which took us 2.4 s. Every other file you change is Python, and editing one rebuilds nothing. A run of `RouterTest` takes about 6 to 7 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 in which the router went wrong?

1. Open a terminal on the workbench desktop, as in Lab 04, go to the lab folder, and build the simulator:

```
. /foss/designs/common/.designinit   # in every new terminal
mod dd105-b
cd labs/lab-05-protocol-rules-and-backpressure
make build
```

   Near the top it prints `BUILD  router with 2 bound file(s)`, which are `tb/router_sva.sv` and `tb/router_sva_bind.sv`, and it ends with:

```
  built  /foss/designs/modules/dd105-b/labs/lab-05-protocol-rules-and-backpressure/
build/sim_build/Vtop in 2.4 s (2.0M in /foss/designs/modules/dd105-b/labs/lab-05-
protocol-rules-and-backpressure/build/sim_build)
 ccache:  Hits: 29 / 42 (69.05%)  Misses: 13 / 42 (30.95%)
================================================================================
  /foss/designs/modules/dd105-b/labs/lab-05-protocol-rules-and-backpressure/build/
sim_build/Vtop is up to date.
```

   Our counts include the builds of Labs 01 to 04, which came to 23 hits and 12 misses, so this build's own were 6 hits and 1 miss.

2. Run the graded test, `RouterTest`. It runs everything `RouterInjectTest` did in Lab 04 and then `RouterLab05Vseq`, which is empty for now:

```
make run
```

   You should see `ok: 0 errors, 0 assertions   7.14 s`, and above it:

```
 53830.00ns  INFO       ..kpressure/tb/test_router.py(407)  [uvm_test_top]:  [VSEQ]
RouterInjectTest's steps are done; starting RouterLab05Vseq
 53830.00ns INFO     ..ckpressure/tb/router_env.py(229) [uvm_test_top.env.sb]: [SB]
806 packets in: 184 to output 0, 189 to output 1, 179 NOROUTE, 254 FULL; the highest
level a header saw was 16
 53830.00ns INFO     ..ckpressure/tb/router_env.py(232) [uvm_test_top.env.sb]: [SB]
beats checked: 788 on output 0, 828 on output 1; 5 DROP_CNT reads checked
 53830.00ns INFO    ..ckpressure/tb/router_env.py(276) [uvm_test_top.env.cov]: [COV]
router.pkt: 34 of 42 bins hit; router.wait: 13 of 14; coverage.xml written
DD105-B BaseTest: 0 errors logged, 0 assertion failures (u_sva.n_fail)
```

   In Lab 04 the `BaseTest` line ended `no bound assertions`. Now `BaseTest` reads the bound assertions' failure count, `cocotb.top.u_sva.n_fail`, at the end of every test, and fails the test if it isn't 0. The packets differ from the ones `RouterInjectTest` sent in Lab 04, because cocotb seeds each test from its name (you saw this in Lab 04, Part 1, step 3), and this test is `RouterTest`. The run's first `[PHASE]` line shows the routes `--01`, with output 0 ready in 5% of cycles, and step 7 of this part uses that phase.

3. Look at the coverage of that run:

```
make cov
```

   You should see, among the other lines:

```
        top.router.pkt.len                              88.89 %   8 of 9 bins
            not hit: oversize
```

   and at the end:

```
      top.router.wait                                92.86 %   13 of 14 bins
        top.router.wait.port                          100.00 %   2 of 2 bins
        top.router.wait.wait                          100.00 %   4 of 4 bins
        top.router.wait.x_port_wait                    87.50 %   7 of 8 bins
            not hit: ('out1', 'short_wait')
```

For each packet that an output delivers, `router.wait` records the longest time that any of its beats waited for `out_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. `oversize` counts packets of 9 beats or more on the input, before the router cuts them. Lab 04′s sink decides `out_ready` anew in every cycle, without looking at what is on the output. At 100% it is always 1, so a beat never waits, and at 5% it is 1 in one cycle in twenty, so a beat waits about 20 cycles on average. So a packet whose longest wait is 1 to 3 cycles is rare, and output 1 delivered none in this run.

4. Now run Lab 04′s graded test on its own:

```
make run TEST=RouterInjectTest
```

You should see `ok`, and these two lines:

```
 53890.00ns INFO     ..ckpressure/tb/router_env.py(229) [uvm_test_top.env.sb]: [SB]
804 packets in: 193 to output 0, 196 to output 1, 218 NOROUTE, 197 FULL; the highest
level a header saw was 16
 53890.00ns INFO     ..ckpressure/tb/router_env.py(232) [uvm_test_top.env.sb]: [SB]
beats checked: 860 on output 0, 818 on output 1; 5 DROP_CNT reads checked
```

They are the lines `RouterInjectTest` printed in Lab 04, Part 4, step 4, number for number. (The line numbers after `router_env.py` differ, because the file has grown.) This lab's base test starts a new sequence on each sink, and each sink has a new driver. Why didn't they change any of Lab 04′s packets?

Answer: A1, in Answers at the end.

5. Run the grader against this lab's three bugs:

```
make check-all
```

You should see:

```
     ok  the golden design                      0 errors, 0 assertions         5.8s
-------------------------------------------------------------------------------
  MISSED  tier 4  B11_cut_packet_has_no_last      0 errors, 0 assertions        5.8s
  caught  tier 4  B10_output_reloads_while_stalled 50 errors, 0 assertions       3.0s
  caught  tier 5  B12_out1_uses_out0_ready       50 errors, 0 assertions        2.0s
-------------------------------------------------------------------------------
 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.` B12 shows only when the two `out_ready` signals differ, and B10 only when a beat has to wait for `out_ready`. Lab 04′s stall test already makes both happen. B11 needs a packet of more than 8 beats, and none of Lab 04′s tests sends one.

6. 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 errors, 0 assertions   2.98 s   log: build/runs/RouterTest-s1-B10/run.log
    RouterTest: AssertionError: 50 errors logged, the first: output 0: got e5 last
in cycle 34, expected 80
```

and from the second:

```
 caught: 50 errors, 0 assertions   2.05 s   log: build/runs/RouterTest-s1-B12/run.log
    RouterTest: AssertionError: 50 errors logged, the first: output 1: got c2 in
cycle 34, expected 3b
```

7. Predict a cycle before you go on. In phase 1, routes `--01` send DST 2 to output 0, which is ready in 5% of cycles. The router accepts the 2-beat packet `80 e5` into an empty output 0, the header `80` in cycle 21 and `e5` in cycle 24, and output 0's sink is next ready in cycle 34. By rule O8, in which cycle is `80` 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 `80` waits? Write that cycle down; Part 2 checks it.

   Answer: A2, 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`?

   Answer: A3, in Answers at the end.
4. Rebuild and run on the golden design:

```
make build
make run
```

   The rebuild took us 2.4 s, with 6 of its 7 files from the ccache. You should see `ok`, the same two `[SB]` lines as in Part 1, and `DD105-B BaseTest: 0 errors logged, 0 assertion failures (u_sva.n_fail)`. 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, which is 1. The course's makefile gives every run of a lab with assertions `+verilator+error+limit+100000`, and `ERROR_LIMIT=1` makes that `+verilator+error+limit+1`:

```
make run BUG=B10 ERROR_LIMIT=1
cat build/runs/RouterTest-s1-B10/exit_status
```

You should see the run stop at the first failure, and `cat` print `1` after it. The failure looks like this:

```
[260000] %Error: router_sva.sv:53: Assertion failed in router.u_sva.u_out0.a_O2: O2:
out_data or out_last changed while the beat waited
%Error:  /foss/designs/modules/dd105-b/labs/lab-05-protocol-rules-and-backpressure/
tb/router_sva.sv:53: Verilog $stop
Aborting...
```

and, at the end:

```
 caught: 1 assertion   1.98 s   log: build/runs/RouterTest-s1-B10/run.log
   router_sva.sv:53: Assertion failed in router.u_sva.u_out0.a_O2: O2: out_data or
out_last changed while the bea
  note: Verilator stopped the run at its error limit, so the rest of the test didn't
run. Runs through make raise it (+verilator+error+limit+N).
```

The line number is the line of the `$error` call in your `a_O2`, so yours may differ, here and later. The time, 260000, is in picoseconds, so the failure was at 260 ns. The clock's period is 10 ns, so cycle n is the period that ends at the rising edge at $10n$ ns, and its falling edge, where the monitors read the pins and count the cycle, is at $10n - 5$ ns. This failure was sampled at the edge that ends cycle 26. Compare it with your prediction from Part 1.

With a limit of 1, the first failing assertion ends the run with exit status 1, and nothing after it is checked. The makefile raises the limit so that every failure prints, each assertion's counter keeps counting, and `BaseTest` 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/RouterTest-s1-B10/run.log
grep -m1 ERROR build/runs/RouterTest-s1-B10/run.log
```

You should see `caught: 50 errors, 284 assertions`, then:

```
[260000] %Error: router_sva.sv:53: Assertion failed in router.u_sva.u_out0.a_O2: O2:
out_data or out_last changed while the beat waited
  335.00ns ERROR   ..ckpressure/tb/router_env.py(116) [uvm_test_top.env.sb]: output
0: got e5 last in cycle 34, expected 80
```

The assertion fired at the edge that ends cycle 26, and the scoreboard reported its first error 8 cycles later, in cycle 34.

7. To see the rule break, record both runs and open them in Surfer:

```
make waves
make waves BUG=B10
surfer build/runs/RouterTest-s1/waves.fst &
surfer build/runs/RouterTest-s1-B10/waves.fst &
```

Add `clk`, `in_valid`, `in_ready`, `in_data`, `in_last`, `out0_valid`, `out0_ready`, `out0_data`, `out0_last` and `level0` from `router`, and `n_O2` from `router.u_sva.u_out0`, and look at 190 to 360 ns:

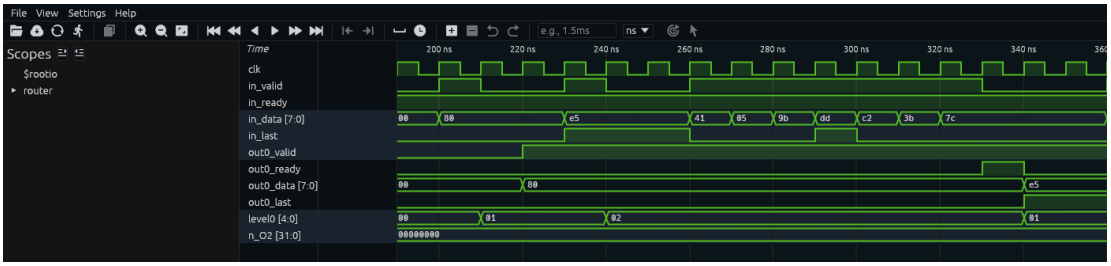


Figure 20.1: The golden router in phase 1 of RouterTest, in Surfer

The golden router. The header `80` is taken at the edge at 210 ns and `e5` at the edge at 240 ns. `80` reaches `out0_data` at the edge at 220 ns, the start of cycle 23, and stays there while `out0_ready` is 0; `level0` goes to 2 when `e5` arrives. `out0_ready` is 1 in cycle 34, from 330 to 340 ns, the sink takes `80` at the edge at 340 ns, and `e5` follows. `n_O2` stays 0.

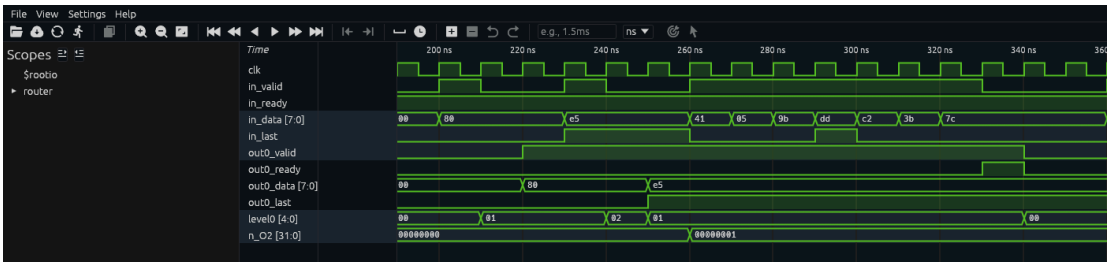


Figure 20.2: The same window with B10 on, in Surfer

B10. `out0_data` changes from `80` to `e5` at the edge at 250 ns, while `out0_ready` is 0, and `level0` drops from 2 to 1. `n_O2` counts its first failure at the edge at 260 ns, which ends cycle 26. When `out0_ready` rises in cycle 34, the beat on the output is `e5`, and after the sink takes it `out0_valid` falls, so `80` is lost.

8. Run the grader again:

```
make check-all
```

You should see B10 caught by your assertions as well, B12 with three assertion failures, and B11 still missed:

```
  MISSED  tier 4  B11_cut_packet_has_no_last      0 errors, 0 assertions         5.6s
  caught  tier 4  B10_output_reloads_while_stalled 50 errors, 284 assertions     2.9s
  caught  tier 5  B12_out1_uses_out0_ready        50 errors, 3 assertions        2.0s
------------------------------------------------------------------------
 2 of 3 caught.
```

The three failures on B12 are all `a_O6`, on output 1, and the first was at the edge at 400 ns, which ends cycle 40. Run `grep -m1 "%Error" build/grade/B12/run.log` to see it. The scoreboard's first error was in cycle 34. B11 is the bug that breaks O6, but your `a_O6` passes on B11 and fails on B12. Why?

Answer: A4, in Answers at the end.

Check: `make check-all` prints `50 errors, 284 assertions` for B10, and the cycle you predicted in Part 1 matches cycle 26.

Part 3: A sink that answers each beat

What changes when each sink answers each beat at its own pace, instead of setting `out_ready` in every cycle whatever is on the output?

1. Read the output agent in `tb/router_agents.py`: `OutCfg`, `ReadyItem`, `SinkSeq`, `SinkDriver` and `OutAgent`. `OutCfg.mode` picks how the sink answers. In `SNK_PCT`, the default, the driver works as Lab 04's did, and sets `out_ready` to 1 right after every rising edge 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 a `ReadyItem` from `SinkSeq`. 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. In those modes the driver looks at the output at the falling edge in the middle of each cycle, decides the next cycle's `ready` there, and drives it just after the rising edge; that loop is written for you.

   pyuvm's driver has `get_next_item()`, which waits until its sequence hands over an item. It doesn't have `try_next_item()`. The base test starts a `SinkSeq` on each sink before it resets the router, and the sequence never ends. Why does the driver need a sequence like that, when the source driver can work with sequences that send a few packets and stop?

   Answer: A5, in Answers at the end.

2. `RouterSinkTest` is Lab 03's random 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 TEST=RouterSinkTest
make cov
```

   You should see `ok` in about 6 s, a first `[PHASE]` line that ends `output 0 SNK_RANDOM, output 1 SNK_RANDOM`, and:

```
 54960.00ns INFO     ..ckpressure/tb/router_env.py(229) [uvm_test_top.env.sb]: [SB]
805 packets in: 456 to output 0, 348 to output 1, 1 NOROUTE, 0 FULL; the highest
level a header saw was 16
```

```
      top.router.wait                                50.00 %   7 of 14 bins
        top.router.wait.port                          100.00 %   2 of 2 bins
        top.router.wait.wait                           50.00 %   2 of 4 bins
            not hit: short_wait, stalled
        top.router.wait.x_port_wait                    37.50 %   3 of 8 bins
                  not hit: ('out0', 'short_wait'), ('out0', 'stalled'), ('out1',
'short_wait'), ('out1', 'long_wait'), ('out1', 'stalled')
```

   As shipped, the driver is always ready in the per-beat modes, whatever the item says, so the packets on output 1 never waited, and no FULL drop happened. The `long_wait` packets on output 0 come from `close_holes()`, which stalls output 0 in Lab 04's mode, `SNK_PCT`.

3. Fill in the two TODOs in `SinkDriver.run_phase`: the countdown while a beat waits, and the expression for `rdy`, the next cycle's `ready`. 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, in `SNK_PCT` too, since the driver still holds the item. One more count would take `wait_left` below 0, and once the sink was back in a per-beat mode, a test for `wait_left == 0` would never let it raise `ready` for that beat.

4. Before you run it again, predict what happens to the `[SB]` line's `FULL` count and to the `router.wait` bins.

   Answer: A6, in Answers at the end.

5. Run it again and look at the coverage. Your edit is Python, so nothing rebuilds:

```
rm -rf build/runs
make run TEST=RouterSinkTest
make cov
```

   If your driver does what the TODOs say, you should see `ok` and:

```
 53660.00ns INFO     ..ckpressure/tb/router_env.py(229) [uvm_test_top.env.sb]: [SB]
804 packets in: 369 to output 0, 319 to output 1, 1 NOROUTE, 115 FULL; the highest
level a header saw was 16
```

```
      top.router.wait                                92.86 %   13 of 14 bins
        top.router.wait.port                          100.00 %   2 of 2 bins
        top.router.wait.wait                          100.00 %   4 of 4 bins
        top.router.wait.x_port_wait                    87.50 %   7 of 8 bins
            not hit: ('out1', 'stalled')
```

The run passed, so every phase's `DROP_CNT` read matched the scoreboard's count, which means the scoreboard predicted all 115 FULL drops. The packets differ from step 2's (804 of them instead of 805), because each item `SinkSeq` makes in `SNK_RANDOM` draws from `random`, and your driver asks for its items in different cycles from the shipped driver. Output 1 has no `stalled` packet, because a `ReadyItem`'s longest wait is 13 cycles and nothing stalls output 1 in this test.

6. Run the grader:

```
make check-all
```

You should see `2 of 3 caught.` again, with the same counts as in Part 2. `RouterTest` keeps its sinks in `SNK_PCT` until its virtual sequence starts, and that sequence is still empty, so your driver changes none of its stimulus, and B11 still needs a packet longer than 8 beats.

Check: `RouterSinkTest` passes with `115 FULL` in its `[SB]` line, 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 `RouterBaseVseq` and `RouterLab05Vseq` in `tb/router_vseq.py`. pyuvm has no `p_sequencer`, so a virtual sequence is given the environment when it's made, and it runs on no sequencer of its own: `RouterTest` starts yours with `await RouterLab05Vseq("vseq", self.env).start()`. The helpers you'll use are `route(n, en, port)`, `ctrl(en)`, `sink(p, mode)`, `stall(p)`, `burst(dst, n, length)`, `send_rand(n, gap_one_in, w_over)`, `backpressure()`, `cycles(n)`, `drain()` and `rd(DROP_CNT)`. Most of them are the base test's steps from Lab 04, rewritten as methods of a sequence, and they start the same Lab 04 sequences that the steps started. Why do `sink()` and `stall()` change the sink's `OutCfg`, and not start a sequence on the sink's sequencer?

   Answer: A7, in Answers at the end.

2. Run the test that runs your sequence alone, as it is now:

```
make run TEST=RouterLab05Test
```

You should see `FAIL: 1 error, 0 assertions   1.90 s`, and above it:

```
    20.00ns ERROR    ..ckpressure/tb/router_env.py(116) [uvm_test_top.env.sb]: no
beat was checked: neither output delivered a beat that matched a prediction
```

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 the APB monitor stamps each transfer with the cycle it completed in, `write_apb` puts a route write in `pending` with that cycle, and `write_in` applies it at the first header of a later cycle. Why does the scoreboard go by the cycle, and not by the order in which the two calls arrive?

   Answer: A8, in Answers at the end.

4. Write `RouterLab05Vseq.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. All at once: random packets with oversize ones among them (`send_rand()`'s `w_over`), `backpressure()`, and a coroutine 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. `par()` starts each coroutine and waits until all of them are done, and `backpressure()` loops until `traffic_done` is set:

```
self.traffic_done = False

async def traffic():
    await self.send_rand(...)          # the packets
    self.traffic_done = True

async def moves():
    ...                                # route changes, with cycles() between them

await self.par(traffic(), self.backpressure(), moves())
```

`send_rand()` takes two settings. `gap_one_in` makes one beat in that many wait 1 to 3 idle cycles first, and it's 4, as in Lab 04, unless you give it. `w_over` is the weight of oversize packets, against 100 for the other lengths.

5. Predict how each oversize packet should come out, and which bug your sequence now reaches.

   Answer: A9, in Answers at the end.

6. Run both tests and look at the coverage. Your sequence is Python, so nothing rebuilds:

```
rm -rf build/runs
make run TEST=RouterLab05Test
make run
make cov
```

You should see `ok` from both runs, and `router.pkt.len` with all 9 bins hit. The counts depend on your sequence. Our `RouterLab05Test` sent 317 packets, and `RouterTest` said:

```
 77350.00ns INFO     ..ckpressure/tb/router_env.py(229) [uvm_test_top.env.sb]: [SB]
1123 packets in: 282 to output 0, 286 to output 1, 179 NOROUTE, 376 FULL; the highest
level a header saw was 16
 77350.00ns INFO     ..ckpressure/tb/router_env.py(232) [uvm_test_top.env.sb]: [SB]
beats checked: 1283 on output 0, 1254 on output 1; 6 DROP_CNT reads checked
```

The first 806 packets are the ones from Part 1, and our sequence sent the other 317. Our `make cov` of the two runs showed `top` at `87.50 %  49 of 56 bins` and `router.wait` at `14 of 14 bins`. The seven empty bins are the seven of `x_fate_level` that the specification rules out, so no test can hit them.

7. Run the grader:

```
make check-all
```

You should see `3 of 3 caught.` Our run printed:

```
     ok  the golden design                      0 errors, 0 assertions         7.1s
-------------------------------------------------------------------------------
  caught  tier 4  B11_cut_packet_has_no_last      20 errors, 18 assertions       7.2s
  caught  tier 4  B10_output_reloads_while_stalled 50 errors, 284 assertions     2.9s
  caught  tier 5  B12_out1_uses_out0_ready        50 errors, 3 assertions        2.0s
-------------------------------------------------------------------------------
```

```
 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 5496, and the scoreboard reported the same beat in the same cycle:

```
[54960000] %Error: router_sva.sv:58: Assertion failed in router.u_sva.u_out1.a_O6:
O6: an output packet is longer than 8 beats
 54955.00ns ERROR   ..ckpressure/tb/router_env.py(116) [uvm_test_top.env.sb]: output
1: got 55 in cycle 5496, expected 55 last
```

`grep -m1 "%Error" build/grade/B11/run.log` and `grep -m1 ERROR build/grade/B11/run.log` show them, and `BaseTest` ended the test with `DD105-B BaseTest: 20 errors logged, 18 assertion failures (u_sva.n_fail)`. What did O6 add that the scoreboard's error didn't already give you?

Answer: A10, 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 TEST=RouterLab05Test BUG=B12
grep -m1 "%Error" build/runs/RouterLab05Test-s1-B12/run.log
grep -m1 ERROR build/runs/RouterLab05Test-s1-B12/run.log
```

Ours said `caught: 50 errors, 39 assertions`, and:

```
[330000] %Error: router_sva.sv:53: Assertion failed in router.u_sva.u_out1.a_O2: O2:
out_data or out_last changed while the beat waited
  3515.00ns ERROR   ..ckpressure/tb/router_env.py(116) [uvm_test_top.env.sb]: output
1: got 46 in cycle 352, expected 44
```

Our O2 failed in cycle 33 and the scoreboard complained in cycle 352. In `RouterTest` the scoreboard was first. Why is it so much later here?

Answer: A11, 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)`, at about 7 s a run.

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: A12, 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: A13, in Answers at the end.

Submit: Nothing to submit.

Next on the Course path: the reading on register models and regressions, then Lab 06: The register block.

Answers

A1. Part 1: The Lab 04 testbench against this lab's bugs

Every packet comes from Python's `random`, one draw after another, so anything that drew one more number would change every packet after it. In Lab 04′s tests, none of this lab's new code draws a number. Every sink in them stays in `SNK_PCT`, Lab 04′s mode, where `SinkDriver` runs Lab 04′s loop, which draws only when `ready_pct` is between 0 and 100 and asks for no items. `SinkSeq` decides each item's values only after `start_item()` returns, which happens when the driver asks for the item, so in these tests it waits in `start_item()` and draws nothing. The new oversize kind has a weight of 0.

A2. Part 1: The Lab 04 testbench against this lab's bugs

By O8, a beat that goes into an empty output appears two cycles after its input handshake, so `80` is on the output from cycle 23. B10 loads `e5` over it as soon as `e5` gets there, two cycles after its own handshake, so `out0_data` first changes in cycle 26. The scoreboard sees only beats that are taken, and output 0′s sink took nothing until cycle 34, when it took `e5` where the scoreboard expected `80`. The router went wrong 8 cycles before the scoreboard could see it.

A3. Part 2: Assertions for the output rules

cocotb's top level is the router itself, so there is no SystemVerilog top module to put an instance of `router_sva` in, and editing the router's RTL to add one would change the design under test. `bind` puts an instance, `u_sva`, inside the router, wired to the router's own ports, from a file of its own. 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. Python reads the failure count through the router as `cocotb.top.u_sva.n_fail`, which is how `BaseTest` gets it.

A4. 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 `RouterTest` sends a packet longer than 8 beats yet, so on B11 the router never has a packet to cut.

B12 makes output 1′s buffer move on `out0_ready`. In phase 1 output 1′s sink is ready in every cycle, so it takes whatever beat is on output 1′s pins. Output 0 is ready in only 5% of cycles, so in most of those cycles the buffer behind output 1′s pins doesn't move on to the next beat, and the pins show a handshake with the same data again. In cycle 34 the scoreboard, which had already matched `c2`, saw the sink take `c2` again where `3b` was due. 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 check only cycles in which `out_ready` is 0, and `out1_ready` is 1 all through phase 1. The run stops at its 50th error in cycle 83, long before phase 2 stalls output 1.

A5. Part 3: A sink that answers each beat

The source driver can wait in `get_next_item()`, because nothing happens on the input until it has a packet. The sink driver can't wait there, because it drives `out_ready` in every cycle, and the router puts a beat on the output whether or not the sink has an item for it. `SinkSeq` is always waiting in `start_item()` with the next item, so the driver's `get_next_item()` returns in the same time step, and the loop goes on to drive the next cycle's `out_ready`. If no sequence were running on the sink, the driver would stop at that call for the rest of the test, `out_ready` would keep the last value it drove, and no beat on that output would be answered again.

A6. 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 packets land in `short_wait` and `long_wait` on both outputs. A sink that waits a cycle or more on about half its 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.

A7. Part 4: A virtual sequence

The base test starts each sink's `SinkSeq`, and it runs for the whole test, one item per beat. A virtual sequence steers the backpressure by changing `mode` and `ready_pct` in the sink's `OutCfg`. `SinkSeq` reads `mode` for every item, and the driver reads `mode` in every cycle, and `ready_pct` as well while the mode is `SNK_PCT`. Starting another sequence on a sink's sequencer would compete with the one already running there.

A8. 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 read the pins at the same falling edge, and cocotb runs them in no fixed order (which is also why `level()` adds back a beat taken in the header's own cycle), 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.

A9. Part 4: A virtual sequence

Each one should come out cut to 8 beats, with `out_last` on the 8th, and the router takes the rest of its beats from the input and throws them away (example E6 in the specification). Your sequence now reaches B11. Under 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.

A10. 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.

A11. 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 352, 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 `RouterTest`'s phase 1 the readies differ the other way round, with output 1 ready and output 0 stalled, so output 1′s sink takes the repeated beats and the scoreboard sees them at once.

A12. Finishing

In `RouterTest` the scoreboard reported errors from cycle 34, and O6 on output 1 failed from cycle 40. In `RouterLab05Test` O2 on output 1 failed from cycle 33 and O1 from cycle 38, the scoreboard's first error was in cycle 352 (Part 4, step 8), and O6 on output 1 failed in cycle
359. 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.

A13. Finishing

It would fail. A reset lowers `out_valid` at once, with no handshake (the specification's Valid/ready rules and example E8), but O1′s consequent needs `out_valid` to be 1 in the next cycle. Rules about normal operation don't hold across a reset, so their assertions are switched off during one.

VI 6. Registers

21 **Register models in pyuvm 171**
21 The example block 171
21 Fields, registers and the map 171
21 From a register operation to a transfer 173
21 Keeping the mirror 173
21 Kinds of prediction 174
21 Volatile fields 174
21 One operation per register at a time 175
21 Pieces pyuvm doesn't have 175
21 Check yourself 176
21 Further reading 176
21 Answers 176

22 **Regressions and seeds 177**
22 What a seed fixes 177
22 One more random call 177
22 Running many seeds 178
22 Corners no seed reaches 179
22 Check yourself 180
22 Further reading 180
22 Answers 180

23 **Lab 06: The register block 183**
23 Before you start 183
23 Part 1: The Lab 05 testbench against the registers 184
23 Part 2: The register model 185
23 Part 3: The predictor and the bit bash 187
23 Part 4: The addresses outside the map 189
23 Part 5: The interrupt 190
23 Finishing 192
23 Going further (optional) 192
23 Answers 192

21. Register models in pyuvm

This page answers: *what does a register model know about a design's registers, what does pyuvm's register layer do for you, and what do you have to write yourself?*

Up to the end of Lab 05, your testbench reached the router's registers with raw APB transfers. In the virtual sequence, `route()` and `ctrl()` write the routing table and `CTRL`, and `rd(DROP_CNT)` reads 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. Two of the five bugs planted in Lab 06 are of those kinds (B14 and B03), and your Lab 05 testbench misses all five. In that lab you'll give the testbench a **register model**. That is a set of pyuvm objects that describes every register and field the way the specification's Register map does, and it can turn an operation such as "write 1 to `CTRL.EN`" into an APB transfer.

pyuvm 4.0.1 has classes for the model, for the map that places its registers at their addresses, and for the adapter that turns a register operation into a bus item. Nothing in it keeps the model's copy of the registers up to date for you, though, and it has none of the ready-made register sequences that SystemVerilog UVM's library has. So in Lab 06 you also write a predictor, a reset sequence and a bit-bash sequence. We wrote a small register block for this page, `regblk`, and a pyuvm testbench for it with a register model, an adapter and a predictor. Both are in `guide-examples/register-models-in-pyuvm/`, and the sections below use them to show how the pieces work together, and what happens when one of them is missing.

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 `105B`
`0x4`	`CTRL`	RW	[0] `EN`, [3:1] `MODE`; resets to 0
`0x8`	`FLAGS`	W1C	[0] `OVF`, set by a pulse on `ev_ovf`; [1] `ERR`, set by a pulse on `ev_err`
`0xC`	`COUNT`	WC	[7:0], the number of events (cycles with a pulse on `ev_ovf` or `ev_err`) since the last write; any write clears it

Apart from a reset, only a write changes `CTRL`, so it holds whatever the testbench last wrote to it. `FLAGS` and `COUNT` change on their own whenever the design sees an event, the way the router's `STATUS` and `DROP_CNT` do. A register model has to treat the two kinds of register differently, as the sections on prediction and on volatile fields show.

Fields, registers and the map

Each register is a subclass of `uvm_reg`, and its `build()` creates the register's fields and configures each one. This is `CTRL`'s:

```
class CtrlReg(uvm_reg):
    def __init__(self, name="CTRL"):
```

```
        super().__init__(name, 16)

    def build(self):
        self.EN = uvm_reg_field("EN")
        self.EN.configure(self, 1, 0, "RW", False, 0)
        self.MODE = uvm_reg_field("MODE")
        self.MODE.configure(self, 3, 1, "RW", False, 0)
```

The 16 is the register's width in bits. A field's `configure()` takes six arguments: the register it belongs to, its width in bits, its lowest bit, its access policy, whether it's **volatile** (changed by the design on its own), and its reset value. Both fields of `FLAGS` are configured as `"W1C"` with volatile set to `True`, and the field in `COUNT` as `"WC"`, also volatile.

pyuvm checks each field against the one configured just before it, and raises `AssertionError` if the new field's lowest bit isn't above the previous field's highest bit. So configure the fields from bit 0 up. Lab 06, Part 2 shows the message you get when they're in the wrong order. pyuvm doesn't reject an access policy that it doesn't know, though. When we configured a field with `"W1c"`, a typo for `"W1C"`, pyuvm accepted it without a word and gave the field the policy `NOACCESS`.

The register block creates the four registers, gives each one its address, and adds each one to the block's map:

```
class RegBlk(uvm_reg_block):
    def __init__(self, name="regs"):
        super().__init__(name)
        self.map = uvm_reg_map("map")
        self.map.configure(self, 0)
        self.ID = self.add(IdReg("ID"), 0x0, "RO")
        self.CTRL = self.add(CtrlReg("CTRL"), 0x4)
        self.FLAGS = self.add(FlagsReg("FLAGS"), 0x8)
        self.COUNT = self.add(CountReg("COUNT"), 0xC)

    def add(self, reg, addr, rights="RW"):
        reg.configure(self, hex(addr), "", False, False)
        self.map.add_reg(reg, "0x0", rights)
        return reg
```

In pyuvm a register's address is a string, and you give it to the register's `configure()`, which also calls the register's `build()`. The map's `add_reg()` takes an offset, which is also a string. It adds the offset to the register's address and uses the sum, written the way Python's `hex()` writes it, as the register's key in the map. But when you read or write the register, the map looks it up by the string the register was configured with, so that string has to be the same as the key. With the offset `"0x0"` used here, the key for address 4 is `"0x4"`, and a register configured at `"0x04"` is never found. A register that we configured at `"0x00"` raised `KeyError: '0x00'` on its first write, before any transfer reached the bus. That is why `add()` gives every address as `hex(addr)`.

In a lab that has a register model, `make regcheck` builds the model outside the simulator. It stops with a message if the model doesn't build, as happens with overlapping fields, or if an address isn't written the way `hex()` writes it. The graders run the same check before any verdict. The listing shows each field's access policy, so a typo such as `"W1c"` shows up there as `NOACCESS`. For `regblk` it printed:

```
REGCHECK  test_regblk.py:RegBlk
    0x0  ID         ID[15:0] RO
    0x4  CTRL       EN[0:0] RW, MODE[3:1] RW
    0x8  FLAGS      OVF[0:0] W1C, ERR[1:1] W1C
    0xc  COUNT      COUNT[7:0] WC
  the model builds: 4 registers
```

For every field, the model keeps a **mirror**, which is the value that the model believes the design holds in that field right now. A register's `get_mirrored_value()` puts the mirrors of its fields together into one value.

From a register operation to a transfer

Suppose a test calls `regs.CTRL.write(0x0005, regs.map, path_t.FRONTDOOR, check_t.NO_CHECK)`. All four arguments are required, and the call returns a `uvm_resp_t`, either `PASS_RESP` or `ERROR_RESP`. (`read(map, path, check)` returns the status and the value read.) The map turns the call into a register operation, here a write of `0005` to the address `"0x4"`, and hands the operation to an **adapter**. The adapter's `reg2bus()` turns the 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 called `item_done()`, the map calls the adapter's `bus2reg()` with the same item, and `bus2reg()` fills in the operation's data and status from it. Here is the example's adapter:

```
class BusAdapter(uvm_reg_adapter):
    def reg2bus(self, rw):
        """rw.addr is the address as the string the register was configured with."""
        write = rw.kind == access_e.UVM_WRITE
        return BusItem("reg", write, int(rw.addr, 16), rw.data if write else 0)

    def bus2reg(self, t, rw):
        rw.kind = access_e.UVM_WRITE if t.write else access_e.UVM_READ
        rw.data = t.wdata if t.write else t.rdata
        rw.status = uvm_resp_t.PASS_RESP
```

pyuvm's registers and map don't look inside a bus item, so on the path from `write()` or `read()` to the bus, these two methods are the only code that depends on what your bus looks like. (The predictor later on this page reads bus items too, the ones the monitor reports.) The environment gives the map its sequencer and its adapter, with `map.set_sequencer()` and `map.set_adapter()`. In Lab 06, Part 2 you'll write the two methods for the router's APB agent, and there `bus2reg()` also has to report a transfer that ended with `PSLVERR` as `ERROR_RESP`.

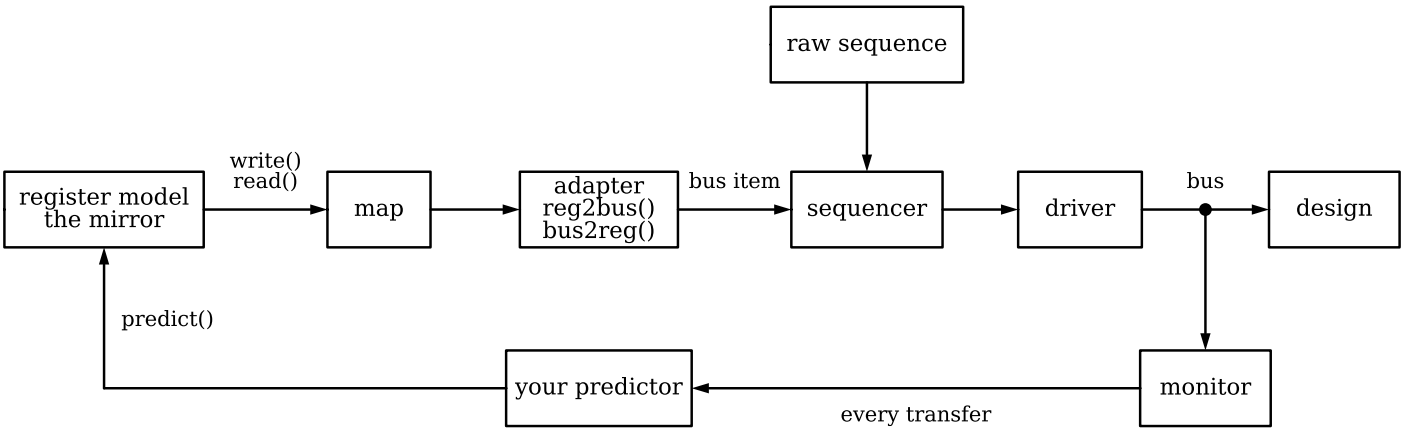


Figure 21.1: The register model, the path of its operations, and the predictor that keeps its mirror up to date

Keeping the mirror

The model's mirror follows the bus only if something tells the model about each transfer, by calling a register's `predict()` with the value written or read. In SystemVerilog UVM the map can do that itself after each operation it sends, which is called auto-predict. pyuvm 4.0.1's map has a switch for it, `enable_auto_predict`, and the switch is off. With the switch on, the map hands each operation to a predictor object after the operation completes, and that object has to be a `uvm_reg_predictor`, a class that pyuvm leaves empty, with no methods of its own. So in pyuvm the mirror changes only when your code calls `predict()`. The place to do that is a **predictor** of your own, a `uvm_subscriber` connected to the bus monitor's analysis port. It sees every transfer the monitor reports, whoever started it, and calls `predict()` on the register at that transfer's address.

Our example's `RegMirrorTest` 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 reads `CTRL` through the model and compares the value read with what the mirror held before the read. We ran it three times, with `+PREDICT=` choosing how the mirror is kept each time:

`+PREDICT=`	Mirror after the model's write	After the raw write	The read of `000b` against the mirror
`predictor`	`0005`	`000b`	match
`none`	`0000`	`0000`	`MISMATCH`
`auto`	the write raised `UVMError: map -- predictor Not set`		

With no predictor, even the model's own write left the mirror at its reset value, `0000`, and the read of `000b` didn't match. `regblk` ran with no bug switched on, so the design was right, and a comparison that fails on a correct design means the testbench is wrong. (`RegMirrorTest` only logs `MISMATCH`, so that run still ended `ok`.) In the `auto` run, the map's own auto-predict was switched on with no predictor object for it to call, so the model's first write raised `UVMError` and the test ended there; `make run` called the run BROKEN. So this testbench needs the predictor. In Lab 06, Part 3 you write one for the router and connect it to the APB monitor, which the scoreboard already listens to.

Kinds of prediction

`predict()` takes the value and a kind, which says what to do with the value. `UVM_PREDICT_WRITE` means the value was written, so each field takes what its access policy gives it. `UVM_PREDICT_READ` means the value was read. `UVM_PREDICT_DIRECT` means the mirror takes the value as it is. `RegPredictTest` calls `predict()` on the mirrors of all four of the example's registers, with no bus transfers at all. It printed:

```
[RAL] FLAGS (W1C), after the reset: 0x0
[RAL]   then predict(0x3, UVM_PREDICT_DIRECT)     0x3
[RAL]   then predict(0x1, UVM_PREDICT_WRITE)      0x2
[RAL]   then predict(0x2, UVM_PREDICT_READ)       0x0
[RAL]   then predict(0x2, UVM_PREDICT_DIRECT)     0x2
[RAL] COUNT (WC), after the reset: 0x0
[RAL]   then predict(0x2a, UVM_PREDICT_DIRECT)    0x2a
[RAL]   then predict(0x5, UVM_PREDICT_WRITE)      0x0
[RAL]   then predict(0x2a, UVM_PREDICT_DIRECT)    0x2a
[RAL]   then predict(0x2a, UVM_PREDICT_READ)      0x0
[RAL] CTRL (RW), after the reset: 0x0
[RAL]   then predict(0xb, UVM_PREDICT_WRITE)      0xb
[RAL]   then predict(0x5, UVM_PREDICT_READ)       0x5
[RAL] ID (RO), after the reset: 0x105b
[RAL]   then predict(0x1234, UVM_PREDICT_WRITE)   0x105b
```

The write predictions follow the access policies. Writing `0x1` to `FLAGS` clears only `OVF`, any write clears `COUNT`, a write to `CTRL` sets it to the value written, and `ID` ignores a write. After a read of `0x5`, `CTRL`'s mirror is `0x5`, as you'd expect, but after a read of `0x2` the mirror of `FLAGS` is 0, and after a read of `0x2a` the mirror of `COUNT` is 0. In pyuvm's source, a field's `predict_read()` returns the field's reset value for `W1C`, `WC` and twelve other policies, whatever the value read was, so after those two reads the mirror no longer holds what the design holds. Your predictor calls `predict()` after every read it sees, and Lab 06, Part 3 asks you which kind it should use.

Volatile fields

`FLAGS` is set by the design, so the model can't know its value until something reads it. In the run with the predictor, `RegMirrorTest` pulses the design's `ev_ovf` input, which sets `FLAGS.OVF`, and then reads `FLAGS` through the model:

```
[RAL] after an overflow event, the mirror of FLAGS holds 0000
[RAL] read FLAGS through the model: 0001; the mirror now holds 0001
```

The mirror said `0000` while the design held `0001`, and only the read brought it up to date. A mirror of a volatile field is out of date whenever the design has changed the field since the last read, so nothing should compare a read of a volatile field with its mirror. In pyuvm, nothing compares any read with the mirror unless you write the comparison yourself. `read()` takes a `check_t`, but pyuvm passes it on only to the auto-predict predictor, which is off, so `check_t.CHECK` compared nothing in our run. After a raw write of `0007` to `CTRL`, we set `CTRL`'s mirror to `0003` by hand and read the register with `CHECK`:

```
[RAL] read(CHECK) of CTRL returned PASS_RESP with 0007, against a mirror of 0003
```

and nothing else was logged. In Lab 06 your predictor does the comparing. It compares every read it sees with the mirror, outside the volatile fields. Every field of the router's `STATUS`, `DROP_CNT` and `LEVEL` is volatile, so in those three registers the predictor checks only the reserved bits. The Lab 06 scoreboard checks `STATUS` instead. It keeps its own copies of `STATUS` and `INT_EN`, updates both cycle by cycle the way the router does, and checks every `STATUS` read and the `irq` output against them.

One operation per register at a time

pyuvm allows one operation at a time on each register. When two coroutines wrote `CTRL` at once, the first write went through and the second raised an exception before it reached the bus:

```
[RAL] two writes to CTRL at once: the first PASS_RESP; the second UVMFatalError: CTRL
-- write cannot perform an operation while                    another is in progress
```

(The run of spaces is in pyuvm's message itself.) SystemVerilog UVM makes the second operation wait for the first instead. `UVMFatalError` isn't a check, so the course's grader calls a run that raises it BROKEN. Lab 06's helpers in `RouterBaseVseq` (`wr_reg()`, `rd_reg()`, `check_reg()` and `rd_all_regs()`) hold a cocotb `Lock` for the register while the operation runs, so that two parts of a virtual sequence that reach the same register take turns.

Pieces pyuvm doesn't have

The example's `RegQuirksTest` found that pyuvm 4.0.1 doesn't have these parts of SystemVerilog UVM's register layer:

- `uvm_reg` has no `mirror`, `update`, `needs_update`, `peek`, `poke`, `set` or `get_reset`. (A field has `get_reset()`.) In SystemVerilog UVM, `mirror()` reads a register and can compare the value with the mirror, `set()` records the value you want a register to have, `update()` writes the register only if that value differs from the mirror, and `peek()` and `poke()` are the back door.
- A back-door read, which reads a register through the signal that holds it instead of over the bus, raised `UVMNotImplemented: map -- BACKDOOR not implemented`. pyuvm's source raises the same for a back-door write.
- Of `uvm_reg_hw_reset_seq`, `uvm_reg_bit_bash_seq`, `uvm_reg_access_seq` and `uvm_reg_predictor`, pyuvm has only `uvm_reg_predictor`, and that class has no methods of its own.

So in Lab 06 you write three pieces that a SystemVerilog UVM student gets from the library: the predictor described above, a reset sequence and a bit-bash sequence. The **reset sequence** puts the mirror at the reset values (`map.reset("HARD")`). Then it reads every register that has at least one field that isn't volatile, and compares the value read with the register's reset value outside its volatile fields. On `regblk` it would read `ID` and `CTRL` and leave `FLAGS` and `COUNT` alone, because every field in them is volatile. It has to run while the design still holds its reset values, which in the labs means before anything has written a register.

The **bit-bash sequence** goes through every register, and every bit in it that isn't in a volatile field, one bit at a time. It writes the register with that bit flipped from the mirror's value, reads the register back and compares the value read with the mirror. Then it does the same for that bit a second time, so the bit is written both ways before the sequence moves on to the next one. It treats a bit with no field as read-only and expects it to read 0, so it checks the reserved bits too, such as bits 2 to 15 of `FLAGS`.

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 (Lab 06, Part 5).

Check yourself

1. A testbench has a register model and an adapter, and no predictor. A test writes `0x1` to `CTRL` with the model's `write()`, then reads `CTRL` and compares the value with what the mirror held before the read. What happens on the golden design?

   Answer: A1, in Answers at the end.

2. Why would a reset sequence on `regblk` read only `ID` and `CTRL`, and what would a bit bash still test in `FLAGS`?

   Answer: A2, in Answers at the end.

3. After `predict(0x2, UVM_PREDICT_READ)`, the mirror of `FLAGS` is `0x0`. Does the design hold 0 in `FLAGS`?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's register-layer pages (section 12) build one SystemVerilog UVM model for one small APB block, across several pages. pyuvm has the same structure of fields, registers, maps and adapters. The notes below say where pyuvm 4.0.1 differs, and each "Correction" is something that ChipVerify's page gets wrong.

- 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. pyuvm's `uvm_reg` has no `update()`, so this course uses only the mirror. 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 the choice between auto-predict and a predictor. In pyuvm auto-predict is off, and the predictor is yours.
- UVM Register Access Methods. Correction: it says a front-door `write()` or `read()` always updates the mirror. In SystemVerilog UVM that's true only with auto-predict on or a predictor connected, as the same page says further down. In pyuvm only a predictor that calls `predict()` updates the mirror.
- UVM Register Sequences: the built-in sequences, why the reset test goes before the bit bash, and how to leave a register out. pyuvm has none of these sequences.

Next: Regressions and seeds.

Answers

A1. Check yourself

The read returns `0x1` and the mirror still holds `0x0`, so the comparison fails although the design is right. pyuvm's auto-predict is off, so not even the model's own `write()` updates the mirror. Only `predict()` does, and with no predictor nothing calls it. Lab 06, Part 3, step 2 finds the same thing on the router.

A2. Check yourself

Every field of `FLAGS` and `COUNT` is volatile, so a read of either register has nothing that can be compared with a reset value, 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

No. The read returned `0x2`, so `FLAGS.ERR` is set, and reading a `W1C` field doesn't clear it. pyuvm's read prediction puts a `W1C` field at its reset value, whatever was read. A predictor that used it would leave the mirror of `FLAGS` wrong after every read that found a bit set, and code that used that mirror, for example to clear exactly the bits it last read, would go wrong.

22. 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` gives the simulator `COCOTB_RANDOM_SEED=1` unless you set `SEED=`, so every "You should see" in the labs is from seed 1, except in the steps that name another seed or run several. 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 5 the packet counts that `RouterTest` prints will change on the same seed, because the step you write there runs before Lab 04's and Lab 05's steps and draws values from `random`. In Lab 07, Part 3 you'll run ten tests on ten seeds each, and the lab asks whether 100 passing runs mean that the environment is finished. This page explains why a step added early changes the packets after it, and what a regression of many passing runs tells you. Most of its measurements come from a small example of our own in `guide-examples/regressions-and-seeds/`.

What a seed fixes

Before each test, cocotb seeds Python's `random` module from `COCOTB_RANDOM_SEED` and the test's name. Every random choice in the labs comes from `random`: the packets' payload bytes and the idle cycles between their beats, the sinks' coin flips, and the values that pyvsc's solver picks. pyvsc gives each object a random state of its own, and the first time the object is randomized it takes that state from `random`.

The example's `RandomTest` starts two coroutines in its run phase. Coroutine A draws four 32-bit values at time 0, and coroutine B draws four more a nanosecond later. On seed 1 the test printed

```
       1.00ns  INFO       ..ons-and-seeds/test_seeds.py(120)  [uvm_test_top]:  [RANDOM]
coroutine A: 8b673390 743e874a e7f16f4d a4fbbbf1
       1.00ns  INFO       ..ons-and-seeds/test_seeds.py(120)  [uvm_test_top]:  [RANDOM]
coroutine B: 50cb8999 ab2faa97 7cc9fa77 5f90ba32
```

On seed 1 again it printed the same eight values, and on seed 2 it printed eight different ones:

```
       1.00ns  INFO       ..ons-and-seeds/test_seeds.py(120)  [uvm_test_top]:  [RANDOM]
coroutine A: 4121bb9f 86c102eb 5dc7006b df91482a
       1.00ns  INFO       ..ons-and-seeds/test_seeds.py(120)  [uvm_test_top]:  [RANDOM]
coroutine B: 188403c9 f666f4ea 93964681 c1a41ad2
```

With the same code and the same seed, every random choice in every coroutine comes out the same, so you can repeat a run that failed exactly. `make run SEED=7` runs seed 7 and puts the run in a folder named after the test and 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 TEST=CountTest SEED=random` (`CountTest` is the example's counter test, described below), the run went into `build/runs/CountTest-s367546708`, so you could run that seed again by giving its number to `SEED=`.

cocotb 2.0.1 brings the test's name into the seed in its `regression.py`. It adds a hash of the test's full name to `COCOTB_RANDOM_SEED` and seeds `random` with the sum. The full name is the module and the class, `test_seeds.RandomTest` for the test above, so two tests with different names start from different seeds even when `COCOTB_RANDOM_SEED` is the same. The example's `RandomTwinTest` is `RandomTest`'s body under another name, and on seed 1 its coroutine A printed `d1d52011 163cde48 c7c1bb93 2d7507ca`, where `RandomTest`'s printed `8b673390 743e874a e7f16f4d a4fbbbf1`. This is why `RouterInjectTest` and `RouterRandTest` send different packets on the same seed, as you saw in Lab 04. It also means that a test you rename, or move to another module, gets different stimulus.

One more random call

The runs in this table are all `RandomTest` on seed 1, and in each one a plusarg makes coroutine A do one thing more before it draws:

Plusarg	What coroutine A does first	A's values	B's values
none	nothing	`8b673390 743e874a e7f16f4d a4fbbbf1`	`50cb8999 ab2faa97 7cc9fa77 5f90ba32`
`+EXTRA=draw`	one more 32-bit draw	`743e874a e7f16f4d a4fbbbf1 50cb8999`	`ab2faa97 7cc9fa77 5f90ba32 fb5670c4`
`+EXTRA=seq`	makes a pyuvm sequence	`8b673390 743e874a e7f16f4d a4fbbbf1`	`50cb8999 ab2faa97 7cc9fa77 5f90ba32`
`+EXTRA=start`	makes a pyuvm sequence and starts it	`8b673390 743e874a e7f16f4d a4fbbbf1`	`50cb8999 ab2faa97 7cc9fa77 5f90ba32`
`+EXTRA=vsc_new`	builds a pyvsc `randobj`	`8b673390 743e874a e7f16f4d a4fbbbf1`	`50cb8999 ab2faa97 7cc9fa77 5f90ba32`
`+EXTRA=vsc_rand`	builds a pyvsc `randobj` and calls `randomize()`	`e7f16f4d a4fbbbf1 50cb8999 ab2faa97`	`7cc9fa77 5f90ba32 fb5670c4 214cdde6`

All the coroutines in a test draw from the same `random` module, which hands out one sequence of values in order, whichever coroutine asks for the next one. The extra draw took the first value, so A got what had been the second to fifth values, and B, which draws after A, got the sixth to ninth. Every one of B's values moved, even though B's code didn't change and B draws a nanosecond after A has finished. Making a pyuvm sequence draws nothing, and neither does starting one, so those two rows match the first. Building a pyvsc object draws nothing either, but its first `randomize()` takes the object's random state from `random`, and on seed 1 that moved both coroutines along by two values.

One more random call, or one more pyvsc object randomized, anywhere in a test changes every random choice made after it, in other coroutines as well as its own. That is why Lab 05′s graded test, `RouterTest`, runs `RouterInjectTest`'s steps first and starts your Lab 05 sequence after them, so that Lab 04′s steps draw the same values as before. In Lab 05, Part 1, `RouterInjectTest`, run on its own, still prints Lab 04′s lines number for number, and the lab asks you why the new sink sequences and drivers didn't change them. Lab 06 can't put its sequence last. Its reset sequence compares what it reads with the registers' reset values, so `RouterTest` starts `RouterLab06Vseq` before anything else, while the router still holds those values. In Lab 06, Part 5 the step you add to that sequence sends a 9-beat packet, whose eight payload bytes are drawn from `random`, and those eight draws change every routing table and packet of Labs 04 and 05 that come after them. Lab 07 starts its new sequence, `RouterLab07Vseq`, last in `RouterTest`, so that nothing it draws changes what runs before it.

Three habits follow from this:

- Compare two runs on the same seed only when the code is the same, because one more draw anywhere in the testbench makes the same seed a different run.
- When a seed fails, write down the commit it failed on as well as the seed. Once you change the testbench, the failure may move to another seed or disappear without the bug being fixed.
- For a bug you need to keep reaching, write a directed step in a sequence that reaches it whatever the seed. You write steps like this in Lab 06, Part 5, for the interrupt, and in Lab 07, Part 5, for resets with traffic in flight.

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 raised `UVMFatalError` shows as BROKEN and isn't counted as a pass.

The example's `CountTest` drives a 4-bit counter that counts up on `inc`, goes back to 0 on `clr` and should stop at 15. In each of 300 cycles the test sets `inc` to 1 with a probability of 60% and `clr` to 1 with a probability of 15%, and it checks the count against a model in every cycle. Its coverage, collected with cocotb-coverage, has four bins: three for the count (0 to 7, 8 to 14, and 15) and one for an increment while the count is at 15. `make seeds TEST=CountTest SEEDS="1 2 3 4 5 6 7 8 9 10"` printed

```
     ok  CountTest                    seed 1      0 errors                      0.3s
     ok  CountTest                    seed 2      0 errors                      0.3s
     ok  CountTest                    seed 3      0 errors                      0.3s
     ok  CountTest                    seed 4      0 errors                      0.3s
     ok  CountTest                    seed 5      0 errors                      0.3s
     ok  CountTest                    seed 6      0 errors                      0.3s
     ok  CountTest                    seed 7      0 errors                      0.3s
     ok  CountTest                    seed 8      0 errors                      0.3s
     ok  CountTest                    seed 9      0 errors                      0.3s
     ok  CountTest                    seed 10     0 errors                      0.3s
----------------------------------------------------------------------------
  10 ok  (10 runs)
```

and then the coverage of all ten runs together:

```
  top                                          100.00 %   4 of 4 bins
    top.satcnt                                   100.00 %   4 of 4 bins
      top.satcnt.count                             100.00 %   3 of 3 bins
      top.satcnt.inc_at_top                        100.00 %   1 of 1 bins
```

Each run wrote its own `coverage.xml` into its run folder, and `regress.py` merged the ten files with cocotb-coverage's `merge_coverage()`, which is also what `make cov` uses. So that 100% counts a bin as hit when any one of the ten runs hit it. The `[COUNT]` line that each run prints at the end of the test says how many cycles that run's counter spent at 15, and how many times it was incremented there:

Seed	1	2	3	4	5	6	7	8	9	10
Cycles at 15	0	0	1	25	0	7	8	1	4	12
Incremented at 15	0	0	0	17	0	5	5	0	3	5
`BUG=wrap`	MISSED	MISSED	MISSED	caught	MISSED	caught	caught	MISSED	caught	caught

The last row is from the same regression with the planted bug `wrap` switched on, which makes the counter wrap from 15 to 0. That regression, `make seeds TEST=CountTest SEEDS="1 2 3 4 5 6 7 8 9 10" BUG=wrap`, ended with `5 MISSED, 5 caught  (10 runs)`. The test caught the wrap on exactly the five seeds that incremented the counter at 15, and passed on the other five. Seeds 3 and 8 got the counter to 15 for one cycle each, but neither incremented it there, so neither could see the bug. To reach 15 the counter needs fifteen increments with no clear between them. With `clr` set 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 unlucky ones.

A corner like this one, which the random stimulus reaches only by luck, gets one more chance with each seed you add to a regression. When you need a corner checked, it's better to write stimulus that reaches it on purpose than to wait for a seed to get there. In Lab 07, Part 6 you do that for the drop counters, with a sequence that drives both of them to 255.

Corners no seed reaches

More seeds help only with corners that the stimulus can reach. In Lab 07, Part 3, `make regress` runs the starter package's ten tests on ten seeds each. It printed `100 ok  (100 runs)`, and its merged coverage began

```
  top                                        81.67 %   49 of 60 bins
```

and ended

```
    top.router.corner                            0.00 %   0 of 4 bins
      top.router.corner.kind                       0.00 %   0 of 4 bins
       not hit: reset_with_beats, reset_mid_packet, noroute_saturated, full_saturated
```

The four bins of `router.corner` 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. No test in it tries to drive a counter to 255 either, and `RouterInjectTest` and `RouterLab07Vseq` both clear the counters at the end of each of their phases. The 100 passes mean that no check fired in 100 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, and the starter's `RouterTest` doesn't catch it. 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. If no test produces the situation that a rule needs, that rule's row has nothing under *Made to happen by*, and you can see this in the plan before you trust a regression's passes.

The test, the seed and the planted bug are all chosen when the simulator starts, so a regression never rebuilds, and all 100 runs used the one simulator that `make build` had made in 2.4 s. When we ran the regression for this page, it took 8 min 10 s on 2 CPUs, and nearly all of that was the runs themselves. A run of `RouterTest` took about 11 s, and a run of any of the other nine tests took 2 to 6 s.

Check yourself

1. You ran `make run SEED=4` yesterday and it failed. Today, after adding one `random.randrange()` 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 `4 of 4 bins`. Did any one seed hit all four bins, and does it matter?

   Answer: A2, in Answers at the end.

3. You rename `RouterTest` to `RouterGradedTest` and change nothing else. Does it send the same packets on seed 1?

   Answer: A3, in Answers at the end.

4. Lab 07's starter passed 100 runs and left `reset_with_beats` empty. Would 1,000 seeds have filled it?

   Answer: A4, 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. In this course the seed is `COCOTB_RANDOM_SEED`, which `make run SEED=n` sets.
- 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.
- 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 from this run. The extra call moves every value drawn after it, so today's seed 4 has different stimulus from yesterday's, and it may no longer reach the bug. Run yesterday's commit on seed 4 to confirm the failure, then write a directed step that reaches the bug 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. It matters because the other five seeds never incremented the counter at 15, and the test passed on all five of them with the wrap bug switched on, even though the merged coverage says `4 of 4 bins`. Merged coverage tells you that some run hit each bin, not that every run did.

A3. Check yourself

No. cocotb seeds `random` for each test from `COCOTB_RANDOM_SEED` plus a hash of the test's full name, so a test with a new name starts from a different seed, as `RandomTwinTest` showed. Every expected output in the labs is for the test names the labs use.

A4. 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 that a test picks, but the test's code fixes what it picks values for, so filling the bin takes a new sequence, which you write in Lab 07, Part 5.

23. Lab 06: The register block

In this lab you'll give the Lab 05 testbench a register model built on pyuvm's register classes. You write the fields of `STATUS`, `DROP_CNT`, `LEVEL` and the four `ROUTE` registers, and the adapter that turns a register operation into an APB transfer. You also write a reset sequence, a predictor and a bit-bash sequence. SystemVerilog UVM's library has all three ready to use, but pyuvm 4.0.1 doesn't have them. The reset sequence reads each register that the router doesn't change on its own and compares it with its reset value. The predictor keeps the model's copy of the registers in step with every transfer on the bus, and the bit bash writes each bit both ways and reads it back, except the bits that the router changes on its own. 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.

Package: `labs/lab-06-the-register-block/`

Before you start

- Finish Lab 05.
- Read Register models in pyuvm and Regressions and seeds.
- Keep the router specification open at its Register map, APB, STATUS, INT_EN and Reset sections.

The package is the Lab 05 testbench with our solution to Lab 05 in it: the three assertions, the sink driver's countdown and `RouterLab05Vseq.body()`. If your own Lab 05 printed `3 of 3 caught.`, your code did the same job, but use ours anyway, because this lab adds to the same files. What we added for you:

File	What Lab 06 adds
`tb/router_ral.py`	the register model, the adapter, the predictor and the two register sequences; seven registers' fields, the adapter and the reset sequence are yours (Part 2), and so are the predictor and the bit-bash sequence (Part 3)
`tb/router_agents.py`	`SysMonitor`, which reports `rst_n` and `irq` every cycle
`tb/router_env.py`	the environment makes the register model, the adapter and the predictor, and you connect the predictor (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); its Lab 05 error `no beat was checked` is now `nothing was checked`, and it fires only in a test that also reads no register
`tb/router_vseq.py`	helpers that go through the model (`wr_reg`, `rd_reg`, `check_reg`, `rd_all_regs`), and `RouterLab06Vseq`, with one `TODO` for each of Parts 2, 4 and 5 and two for Part 3
`tb/test_router.py`	`RouterTest` runs `RouterLab06Vseq` first, then everything Lab 05's `RouterTest` ran; `RouterLab06Test` runs the sequence alone
`src/predict.py`	`make predict`, which shows what `predict()` does to your model (Part 3)

Time: about 2 h 15 min. Every file you change is Python, so after the build in Part 1 nothing you change rebuilds the simulator. A run of `RouterTest` takes about 8 s, and `make check-all`, which runs it six times, about 45 s.

Part 1: The Lab 05 testbench against the registers

What does your Lab 05 testbench check about the registers?

1. Open a terminal on the workbench desktop, as in Lab 05, go to the lab folder, and build the simulator:

   ```
   . /foss/designs/common/.designinit   # in every new terminal
   mod dd105-b
   cd labs/lab-06-the-register-block
   make build
   ```

   It ends with:

   ```
        built     /foss/designs/modules/dd105-b/labs/lab-06-the-register-block/build/
   sim_build/Vtop  in  2.4  s  (2.1M  in  /foss/designs/modules/dd105-b/labs/lab-06-the-
   register-block/build/sim_build)
    ccache:  Hits: 41 / 56 (73.21%)  Misses: 15 / 56 (26.79%)
   ==========================================================================
    /foss/designs/modules/dd105-b/labs/lab-06-the-register-block/build/sim_build/Vtop
   is up to date.
   ```

   Our counts include the builds of Labs 01 to 05 and the rebuild in Lab 05's Part 2. Those builds came to 35 hits and 14 misses, so this build's own were 6 hits and 1 miss.

2. Run the graded test, `RouterTest`, and pull the scoreboard's summary out of the log:

   ```
   make run
   grep -o '\[SB\] [0-9a-z].*' build/runs/RouterTest-s1/run.log
   ```

   You should see `ok: 0 errors, 0 assertions   8.61 s` from `make run`, and from `grep`:

   ```
   [SB] 1123 packets in: 282 to output 0, 286 to output 1, 179 NOROUTE, 376 FULL; the
   highest level a header saw was 16
   [SB] beats checked: 1283 on output 0, 1254 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 `RouterInjectTest`'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. The first two lines are the ones `RouterTest` printed at the end of Lab 05 (Part 4, step 6), number for number. `RouterTest` now makes `RouterLab06Vseq` and starts it before anything else, and every packet comes from Python's `random`, one draw after another, so a single extra draw would change every packet after it. Why didn't the new sequence change any?

   Answer: A1, in Answers at the end.

4. `make check-all` runs `RouterTest` on the golden router and on this lab's five bugs. Write down how many you expect the starter to catch, then run it:

   ```
   make check-all
   ```

   You should see:

   ```
        ok  the golden design                    0 errors, 0 assertions         7.2s
   ---------------------------------------------------------------------------
   ```

```
 MISSED  tier 1  B01_status_write_clears_all     0 errors, 0 assertions        7.2s
 MISSED  tier 1  B02_irq_ignores_int_en         0 errors, 0 assertions        7.2s
 MISSED  tier 1  B03_route3_readback_swapped    0 errors, 0 assertions        7.2s
 MISSED  tier 1  B04_misaligned_address_accepted  0 errors, 0 assertions      7.3s
 MISSED  tier 6  B14_enable_resets_to_one       0 errors, 0 assertions        7.2s
-----------------------------------------------------------------------
 0 of 5 caught.
```

followed by each bug's "hides behind" hint. `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 a register model need to know about each register, and what does a reset sequence do with it?

1. Open `tb/router_ral.py` and read `IdReg`, `CtrlReg`, `IntEnReg` and `RouterRegs`. Each register is a `uvm_reg` subclass whose `build()` makes its fields and configures each one with

   ```
   field.configure(parent, size, lsb_pos, access, is_volatile, reset)
   ```

   that is, its width, its lowest bit, its access policy (`"RO"`, `"RW"`, `"W1C"`, `"WC"` and so on), whether the router changes it on its own (volatile), and its reset value. `RouterRegs` configures each register at its address and adds it to the map. pyuvm keys the map by the address as Python's `hex()` writes it, `"0x8"`, so the addresses are given as `hex(addr)`. The specification's Register map gives each field's policy and volatility, and Configuring a field, on the pyuvm register API page, gives the meaning of each argument and shows what pyuvm does with a field that's configured wrongly.

   The graders build the model on its own, outside the simulator, before every verdict. Run that build yourself:

   ```
   make regcheck
   ```

   You should see:

   ```
   REGCHECK  tb/router_ral.py:RouterRegs
       0x0  ID         ID[31:0] RO
       0x4  CTRL       EN[0:0] RW
       0x8  STATUS
       0xc  INT_EN     NOROUTE[0:0] RW, FULL[1:1] RW, OVERSIZE[2:2] RW
      0x10  DROP_CNT
      0x14  LEVEL
      0x20  ROUTE0
      0x24  ROUTE1
      0x28  ROUTE2
      0x2c  ROUTE3
     the model builds: 10 registers
   ```

   The seven registers with no fields listed are the ones you finish in the next step.

2. Write `build()` for `StatusReg`, `DropCntReg`, `LevelReg` and `RouteReg`, as their `TODO`s describe; the four `ROUTE` registers are all made from `RouteReg`. pyuvm checks each field against the one configured just before it, so configure them from bit 0 up. Then run `make regcheck` again. You should see every register with its fields:

```
     0x8  STATUS     NOROUTE[0:0] W1C, FULL[1:1] W1C, OVERSIZE[2:2] W1C
     0xc  INT_EN     NOROUTE[0:0] RW, FULL[1:1] RW, OVERSIZE[2:2] RW
    0x10  DROP_CNT   NOROUTE_CNT[7:0] WC, FULL_CNT[15:8] WC
    0x14  LEVEL      LEVEL0[4:0] RO, LEVEL1[12:8] RO
    0x20  ROUTE0     EN[0:0] RW, PORT[1:1] RW
```

with `ROUTE1` to `ROUTE3` like `ROUTE0`. When we configured `STATUS`'s fields from bit 2 down, `make regcheck` printed this instead, spaces and all:

```
   the model doesn't build: AssertionError: STATUS -- _add_field:             Fields
FULL overlap                with field                OVERSIZE
```

and `make check` stopped with `VERDICT: FAIL -- the register model doesn't build, so nothing was graded.` In a test, that `AssertionError` would have looked like a check that fired, which is why the graders build the model first.

3. Write the adapter's two methods, `ApbRegAdapter.reg2bus()` and `bus2reg()`. The map calls `reg2bus()` with each register operation and starts the `ApbItem` it returns on the APB sequencer. When the driver has finished the transfer, the map calls `bus2reg()` with the same item, and the status `bus2reg()` puts in `rw.status` is what the register's `read()` or `write()` returns. Give a transfer that ended with `PSLVERR` the status `uvm_resp_t.ERROR_RESP`. pyuvm passes the address as the string the register was configured with, so `reg2bus()` turns it back into a number with `int(rw.addr, 16)`.
4. Write `RegResetSeq.body()`. It puts the mirror (the model's copy of each register) at the reset values. Then it reads each register that has at least one field that isn't volatile, and compares the value read with the register's reset value outside its volatile fields. `reset_value(reg)` and `volatile_mask(reg)`, above the adapter, compute the reset value and the bits of the volatile fields for you. Then write the Part 2 step of `RouterLab06Vseq` in `tb/router_vseq.py`. That step starts the sequence, and its `TODO` gives the line.
5. Write down which of the ten registers you expect your sequence to read. Then run the test and look at the line the sequence logs when it ends:

```
make run
grep -o '\[RegResetSeq\].*' build/runs/RouterTest-s1/run.log
```

You should see `ok` and:

```
[RegResetSeq] 0 writes and 7 reads of 7 registers (ID, CTRL, INT_EN, ROUTE0, ROUTE1,
ROUTE2, ROUTE3); 0 mismatches
```

Right after the reset, `STATUS`, `DROP_CNT` and `LEVEL` hold their reset values too. Why does the sequence leave them out?

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 error, 0 assertions         7.2s
```

and the other four stay `MISSED`, for `1 of 5 caught.` The error is your reset sequence's:

```
grep ERROR build/grade/B14/run.log | grep -o '\[uvm_test_top[^]]*\]: .*'
```

```
[uvm_test_top.env]: [RegResetSeq] CTRL reads 0x1 after reset; its reset value is 0x0
```

(Your message may be worded differently; that one is ours.)

7. `RouterTest` runs `RouterLab06Vseq` 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 a bit-bash sequence test?

1. A predictor updates the mirror by calling a register's `predict()` with the value a transfer wrote or read, and the kind of prediction says what to do with that value: `UVM_PREDICT_WRITE` (it was written, so each field takes what its policy gives it), `UVM_PREDICT_READ` (it was read) or `UVM_PREDICT_DIRECT` (take it as it is). `make predict` builds your model and calls `predict()` on four of its registers. Read `src/predict.py`, write down what the mirror of `STATUS` should hold after each of its four calls, then run it:

   ```
   make predict
   ```

   You should see:

   ```
   STATUS (W1C), after the reset: 0x0
     then predict(0x7, UVM_PREDICT_DIRECT)     0x7
     then predict(0x1, UVM_PREDICT_WRITE)      0x6
     then predict(0x6, UVM_PREDICT_READ)       0x0
     then predict(0x6, UVM_PREDICT_DIRECT)     0x6
   DROP_CNT (WC), after the reset: 0x0
     then predict(0x302, UVM_PREDICT_DIRECT)   0x302
     then predict(0x5, UVM_PREDICT_WRITE)      0x0
     then predict(0x302, UVM_PREDICT_DIRECT)   0x302
     then predict(0x302, UVM_PREDICT_READ)     0x0
   CTRL (RW), after the reset: 0x0
     then predict(0x1, UVM_PREDICT_WRITE)      0x1
     then predict(0x1, UVM_PREDICT_READ)       0x1
   ID (RO), after the reset: 0xd1050001
     then predict(0x1234, UVM_PREDICT_WRITE)   0xd1050001
   ```

   A write of `0x1` to `STATUS` clears only bit 0, any write clears `DROP_CNT`, and `ID` ignores a write. Your predictor will call `predict()` after every read it sees. Which kind should it use?

   Answer: A5, in Answers at the end.

2. Lab 04's steps and your Lab 05 sequence set up the router with raw transfers, which go around the model, and `route()` and `ctrl()` in `RouterBaseVseq` send raw transfers too. Write the second Part 3 `TODO` of `RouterLab06Vseq`, "the router set up around the model": the five raw writes it lists, then `check_reg()` on `CTRL` and the four `ROUTE` registers. `check_reg()` reads a register through the model and compares the value read with what the mirror held before the read, as SystemVerilog UVM's `mirror()` does when it's told to check. Leave the first Part 3 `TODO`, the bit bash, for step 4.

   The predictor isn't written yet. Predict what `RouterTest` does on the golden router, then run it:

   ```
   make run
   grep ERROR build/runs/RouterTest-s1/run.log | grep -o '\[uvm_test_top[^]]*\]: .*'
   ```

   You should see `FAIL: 5 errors, 0 assertions  7.15 s` from `make run`, and from `grep`:

   ```
   [uvm_test_top.env]: [VSEQ] check_reg: CTRL read 0x1; the mirror held 0x0
   [uvm_test_top.env]: [VSEQ] check_reg: ROUTE0 read 0x1; the mirror held 0x0
   [uvm_test_top.env]: [VSEQ] check_reg: ROUTE1 read 0x3; the mirror held 0x0
   [uvm_test_top.env]: [VSEQ] check_reg: ROUTE2 read 0x2; the mirror held 0x0
   [uvm_test_top.env]: [VSEQ] check_reg: ROUTE3 read 0x3; the mirror held 0x0
   ```

   The test fails on the golden router. Which side is wrong, and would writing the five registers through the model, with `wr_reg()`, fix it?

Answer: A6, in Answers at the end.

3. Write the predictor, `RouterRegPredictor.write()` in `tb/router_ral.py`. The APB monitor calls it with every transfer it sees complete. Skip a transfer that ended with `PSLVERR`, or to an address with no register. Predict a write with `UVM_PREDICT_WRITE`. For a read, first compare the value read with the mirror outside the volatile fields, report a mismatch with `self.logger.error()`, and then predict the value read with the kind you chose in step 1. Count the reads, writes and mismatches as the `TODO` says, for the `[RAL]` line. Then write the Part 3 `TODO` in `RouterEnv.connect_phase()` in `tb/router_env.py`: give the predictor the model, and connect the APB monitor's analysis port to it, the way the coverage is connected earlier in the same method. Run the test again:

```
make run
grep -o '\[RAL\].*' build/runs/RouterTest-s1/run.log
```

You should see `ok: 0 errors, 0 assertions  7.17 s`, and:

```
[RAL] 18 register reads checked against the mirror, 39 writes predicted; 0 mismatches
```

The 18 reads are the reset sequence's 7, your 5 `check_reg()` reads and the 6 reads of `DROP_CNT` from Labs 04 and 05.

4. Write `RegBitBashSeq.body()` in `tb/router_ral.py`. For every bit that isn't in a volatile field, it does the same thing twice: it writes the register with that bit flipped from the mirror, takes the mirror, reads the register back, and compares the value read with the mirror it took. It has to take the mirror before the read, because once the read completes your predictor puts the value read in the mirror, and a sequence that compared with it then would be comparing the value read with itself. Then write the first Part 3 `TODO` of `RouterLab06Vseq`. It runs the bit bash, before the raw writes of step 2. Run the test and look at the lines of the two sequences and the predictor:

```
make run
grep   -o   '\[\(RAL\|RegResetSeq\|RegBitBashSeq\)\].*'   build/runs/RouterTest-s1/
run.log
```

You should see `ok` and:

```
[RegResetSeq] 0 writes and 7 reads of 7 registers (ID, CTRL, INT_EN, ROUTE0, ROUTE1,
ROUTE2, ROUTE3); 0 mismatches
[RegBitBashSeq] 582 writes and 582 reads of 10 registers (ID, CTRL, STATUS, INT_EN,
DROP_CNT, LEVEL, ROUTE0, ROUTE1, ROUTE2, ROUTE3); 0 mismatches
[RAL] 600 register reads checked against the mirror, 621 writes predicted; 0
mismatches
```

Every field of `STATUS` is volatile, and the bit bash skips the bits of volatile fields. What does it test in `STATUS`?

Answer: A7, in Answers at the end.

5. Run `make check-all`. You should see `caught` on B03 (4 errors) and B14 (2), and the other three `MISSED`, for `2 of 5 caught.` B14 has one error more than in Part 2, because your predictor now checks the reset sequence's read of `CTRL` as well. Look at B03's errors:

```
grep ERROR build/grade/B03/run.log | grep -o '\[uvm_test_top[^]]*\]: .*'
```

```
[uvm_test_top.env.predictor]: [RAL] ROUTE3 read 0x2 in cycle 3663; the mirror holds
0x1
[uvm_test_top.env]: [RegBitBashSeq] writing a 1 to bit 0 of ROUTE3 (was 0x0) read
back 0x2, not 0x1
[uvm_test_top.env.predictor]: [RAL] ROUTE3 read 0x2 in cycle 3677; the mirror holds
0x1
```

```
[uvm_test_top.env]: [RegBitBashSeq] writing a 0 to bit 1 of ROUTE3 (was 0x3) read
back 0x2, not 0x1
```

When we took the mirror after the read instead, our bit bash reported nothing on B03, and only the predictor's two errors caught it. Where did the 1 written to `EN` come back, and why did your Lab 05 testbench never notice B03?

Answer: A8, 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 of `RouterLab06Vseq`: read every bad address with `rd(a)`, write each with `wr(a, 0xFFFF_FFFF)`, then read every register with `rd_all_regs()`. `good_addr(a)` is true 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 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 check-all
   grep -o '\[SB\] [0-9a-z].*' build/grade/golden/run.log
   ```

   You should see B04 `caught` with `50 errors` (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 ERROR build/grade/B04/run.log | grep -o '\[uvm_test_top[^]]*\]: .*' | head -2
   ```

   ```
   [uvm_test_top.env.sb]: a read of 01, completing in cycle 4143: pslverr is 0, but 01
   is a bad address
   [uvm_test_top.env.sb]: a read of 01, completing in cycle 4143 returned d1050001; a
   read of a bad address returns 0
   ```

   `d1050001` is `ID`'s value. Why did a read of `0x01` return it?

   Answer: A9, in Answers at the end.

4. To see these reads in a waveform, record `RouterLab06Test`, which runs your sequence alone, on the golden router and on B04. It runs cycle for cycle like the first part of `RouterTest`, so the read of `01` completes in cycle 4143 there too:

   ```
   make waves TEST=RouterLab06Test
   make waves TEST=RouterLab06Test BUG=B04
   surfer build/runs/RouterLab06Test-s1-B04/waves.fst &
   ```

   Add `clk`, `psel`, `penable`, `pwrite`, `paddr`, `pready`, `pslverr` and `prdata` from `router`, and `bad_addr` from `router.u_regs`, and look at 41340 to 41520 ns. Cycle n ends at the rising edge at $10n$ ns, so the read

of `01` takes cycles 4141 to 4143, from 41400 to 41430 ns. The first error in the B04 run's log is printed at 41425 ns, the falling edge of cycle 4143, where the APB monitor sees the read complete.

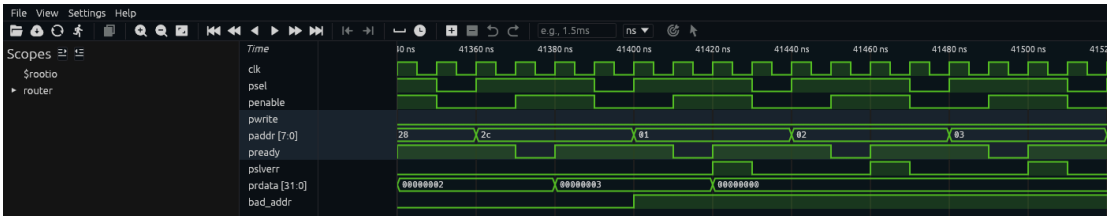


Figure 23.1: Reads of 2c, 01, 02 and 03 on the golden router, in Surfer

The golden router, 41340 to 41520 ns. The read of `2c` is the `check_reg()` of `ROUTE3` from Part 3, and it returns `00000003`. `bad_addr` rises with `paddr` at `01`. `pslverr` is 1 in the last cycle of each of the reads of `01`, `02` and `03`, and each of those reads returns 0.

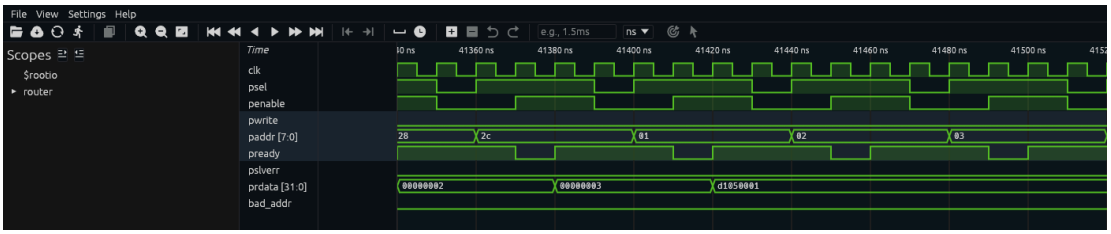


Figure 23.2: The same reads with B04 switched on, in Surfer

With B04, `bad_addr` stays 0, `pslverr` never rises, and `prdata` is `d1050001` for the reads of `01`, `02` and `03`.

How many cycles does each read take, and in which of them does the router decide what `prdata` will be?

Answer: A10, 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. `irq()` combines the two registers into the value that `irq` should have. Write the two Part 5 `TODO`s in `RouterScoreboard.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 run `make check-all` and look at B02's error:

```
make check-all
grep ERROR build/grade/B02/run.log | grep -o '\[uvm_test_top[^]]*\]: .*'
```

You should see `4 of 5 caught.`, with B01 still `MISSED` and B02 caught by one error. You chose the wording of your message; in ours it is:

```
[uvm_test_top.env.sb]: irq is 1 from cycle 5927, 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 13632 cycles, wrong in 0`. Why does B02 show up in Lab 04's traffic, and why does B01 still hide?

Answer: A11, in Answers at the end.

3. Write the Part 5 step of `RouterLab06Vseq`:

   - 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 `0b111`, then clear the `STATUS` bits one at a time by writing `0b001`, `0b100` and `0b010`, 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 predictor doesn't compare a read of `STATUS`, because its fields are volatile, but the scoreboard checks every one.

4. Run `make check-all`. You should see:

```
     ok  the golden design                      0 errors, 0 assertions         8.1s
-------------------------------------------------------------------------------
  caught  tier 1  B01_status_write_clears_all     3 errors, 0 assertions        7.9s
  caught  tier 1  B02_irq_ignores_int_en          1 error, 0 assertions         7.9s
  caught  tier 1  B03_route3_readback_swapped     4 errors, 0 assertions        8.0s
  caught  tier 1  B04_misaligned_address_accepted  50 errors, 0 assertions       2.5s
  caught  tier 6  B14_enable_resets_to_one        2 errors, 0 assertions        8.0s
-------------------------------------------------------------------------------
 5 of 5 caught.
 Every bug in the list, each one caught by a check that fired.
```

   Look at B01's errors:

```
grep ERROR build/grade/B01/run.log | grep -o '\[uvm_test_top[^]]*\]: .*'
```

   With our wording they are:

```
[uvm_test_top.env.sb]: irq is 0 from cycle 5955, but STATUS = 110 and INT_EN = 111
make it 1
[uvm_test_top.env.sb]: a read of 08, completing in cycle 5958 returned 00000000; it
should return 00000006, STATUS in cycle 5957
[uvm_test_top.env.sb]: a read of 08, completing in cycle 5965 returned 00000000; it
should return 00000002, STATUS in cycle 5964
```

   The first write of `0b001` 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 5906`, in your own step. Why doesn't Lab 04's traffic catch B02 any more?

   Answer: A12, in Answers at the end.

5. Look at the golden run's summary again:

```
grep -o '\[SB\] [0-9a-z].*' build/grade/golden/run.log
```

   You should see:

```
[SB] 1135 packets in: 311 to output 0, 288 to output 1, 177 NOROUTE, 359 FULL; the
highest level a header saw was 16
[SB] beats checked: 1423 on output 0, 1285 on output 1; 40 DROP_CNT reads checked
```

```
[SB] registers: 870 reads, 64 of STATUS checked; 492 transfers to bad addresses; irq
checked in 13918 cycles, wrong in 0
```

Up to Part 4, the packet and beat counts in the first two lines were Lab 05's: 1123 packets, 376 of them FULL. The total is now 12 more, which are the 12 packets your step sent. Only one of those 12 was a FULL drop, but there are 359 FULL drops where Lab 05 had 376. Why did Lab 04's and Lab 05's packets change now, and not in Parts 2 to 4?

Answer: A13, in Answers at the end.

6. 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: A14, in Answers at the end.

Check: `5 of 5 caught.`, and `irq checked in 13918 cycles, wrong in 0` in the golden run.

Finishing

You're done when `make check-all` prints `5 of 5 caught.` with the register fields, the adapter, the reset and bit-bash sequences, the predictor and its connections, the `irq` check and the `RouterLab06Vseq` 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 to submit.

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"
```

Our solution catches 14 of the 15, in 88 s. These are the last two rows of its table, with the count under them:

```
  caught  tier 6  B14_enable_resets_to_one        2 errors, 0 assertions         8.0s
  MISSED  tier 6  B15_output_valid_survives_reset  0 errors, 0 assertions         8.0s
-------------------------------------------------------------------------------
 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

`RouterLab06Vseq` does nothing yet, and making and starting a pyuvm sequence draws nothing from `random`. The other things this lab added (the sys monitor, the scoreboard's model of `STATUS` and `INT_EN`, the register model and the predictor) don't call `random` at all. We also ran our finished `RouterTest` with `RouterLab06Vseq` left out, on seeds 1 to 5, and it printed Lab 05's lines every time. Part 5 comes back to this, at the step where the numbers change.

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. In that window a read of `CTRL` would return 1, and a packet offered then would be taken, although `in_ready` should stay 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, because the only register reads it checks

are those of DROP_CNT and STATUS. Nothing checks that in_ready stays 0 until CTRL.EN is set. B02 hides because nothing checks irq yet, as the third [SB] line says.

A3. Part 2: The register model

Every field in those three is volatile, because the router changes it without any write. The router sets a STATUS bit on a drop or an oversize packet, counts drops in DROP_CNT, and changes LEVEL with every beat. A register model doesn't compare a volatile field, because the design may have changed it since the model last looked, and a reset sequence skips a register in which nothing would be compared. SystemVerilog UVM's built-in reset sequence skips the same three. If your sequence read more than seven registers, one of your fields in those three isn't marked volatile.

A4. Part 2: The register model

It would report errors although the router is right. The sequence puts the mirror at the reset values and compares every read with them, but the router is reset only once, at time 0, and Lab 04′s steps leave CTRL.EN at 1 and some of the routes set. When we ran it after RouterInjectTest's steps, whose last routing table is 0---, it reported CTRL and ROUTE0, both reading 0x1. 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

UVM_PREDICT_DIRECT. After a read of 0x6, UVM_PREDICT_READ left the mirror of STATUS at 0, and after a read of 0x302 it left DROP_CNT's at 0, although the registers held the values that were read. In pyuvm 4.0.1, a read prediction puts a field whose policy is W1C or WC, or one of twelve other policies, back at its reset value instead of the value read; an RW or RO field keeps the value read. UVM_PREDICT_DIRECT takes the value read for every policy. No check in this testbench would notice the difference, because the W1C and WC fields here are all volatile, and the mirror is never compared in a volatile field. With UVM_PREDICT_READ our solution still passed the golden router and still caught all five bugs. Code that uses the mirror of STATUS, for example to clear exactly the bits it last read, would go wrong.

A6. Part 3: The predictor and the bit bash

The mirror is wrong, and the router returned what was written. The mirror changes only when something calls predict(), and so far no code in the testbench calls it. The raw writes went around the model, and pyuvm doesn't predict after the model's own operations either, because its auto-predict is off. We wrote the five registers with wr_reg() instead, and the same five errors came back. The testbench needs a predictor that updates the mirror from every transfer on the bus, whoever started it.

A7. Part 3: The predictor and the bit bash

Bits 3 to 31, which belong to no field. A bit with no field is read-only and reads 0, and the mirror has 0 there, so both writes write a 1 to the bit, and each read must still return 0 there. That way the bit bash tests the reserved bits of every register that has them. In STATUS, DROP_CNT and LEVEL it tests only the reserved bits, with two writes and two reads for each: 29 bits of STATUS, 16 of DROP_CNT and 22 of LEVEL. With all 32 bits of ID, CTRL, INT_EN and each ROUTE register, that is 291 bits and 582 writes.

A8. 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 ROUTE register, so no read ever came back with the fields swapped. The check_reg() you wrote in step 2 reads ROUTE3 after route(3, 1, 1), and 0x3 swapped is still 0x3. The bit bash reads the register back after every write, and each write changes one bit, so the two fields hold different values at some point.

A9. 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.

A10. 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, at 41420 ns for the read of 01. 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 rising edge before it starts each transfer.

A11. Part 5: The interrupt

Your RouterLab06Vseq sends no packets yet, so the drop that set STATUS.NOROUTE was in Lab 04′s traffic. Every phase of RouterInjectTest disables a route and sends packets to it (in RouterTest's first phase the routing table is --01). 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 the reset, when every bit is 0.

A12. Part 5: The interrupt

Your step ends with `INT_EN` at `0b111`, 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` is `STATUS`, so B02 makes no difference to `irq`. That is the situation the catalogue 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.

A13. Part 5: The interrupt

The 9-beat packet's 8 payload bytes come from `random`, and every packet after them is drawn from a different place in the sequence of random numbers, so the routing tables, the packets and the sinks' coin flips of Labs 04 and 05 all differ. Parts 2 to 4 drew nothing. With Part 4′s step written, your sequence ran for about 5,900 cycles before Lab 04′s steps began, and every packet after it was still Lab 05′s, so the extra cycles on their own don't change the packets. We also counted the draws, by saving `random`'s state before `RouterLab06Vseq` and putting it back after the sequence. The sequence had drawn 8 values, and with the state put back, every routing table and random packet after it was Lab 05′s again, as the `[PHASE]` and `[HOLES]` lines showed.

A14. 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 falling edge of 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 falling edge, so it would get the old or the new `INT_EN` depending on which monitor cocotb ran first. That is the race 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.

A15. Finishing

It added stimulus made from the register description. The reset sequence read every register it could check and caught B14, and the bit bash wrote and read back every bit it could check and caught B03, without a step written for either bug. It also added access by name (`wr_reg(self.regs.INT_EN, ...)`), and a mirror that the predictor compares with every read of a field the router doesn't change on its own. It left raw transfers and the scoreboard to cover 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). In pyuvm it also left you the predictor and both sequences, which SystemVerilog UVM's library would have given you.

VII 7. Capstone

24 Verification plans and sign-off 197
24 One row for each rule 197
24 Bugs that hid behind an empty column 198
24 Coverage against the plan 198
24 Sign-off criteria 199
24 A quiet hunt 199
24 Check yourself 199
24 Further reading 200
24 Answers 200

25 Writing the verification report 201
25 A short report for the counter 201
25 Numbers and the commands behind them . . . 202
25 The sign-off sentence 202
25 Section 7: the limits 203
25 The hunt table 203
25 Check yourself 203
25 Further reading 203
25 Answers 204

26 Lab 07: Capstone 205
26 Before you start 205
26 Part 1: Your repository and the first build 206
26 Part 2: The verification plan 207
26 Part 3: A seed regression 208
26 Part 4: Mystery hunts 208
26 Part 5: Reset with traffic in flight 210
26 Part 6: Counters at 255 211
26 Part 7: Sign-off 211
26 Part 8: The report 213
26 Finishing 213
26 Going further (optional) 214
26 Answers 214

24. 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 from our solution to Lab 06. Lab 07's package adds a reset driver, a scoreboard that can follow a reset while traffic is in flight, and a coverage group, `router.corner`, with a bin for each of four corner cases that the scoreboard counts. On that package, `make regress` runs ten tests, each on seeds 1 to 10, and prints

```
  100 ok  (100 runs)
```

`make check-all` runs the graded test, `RouterTest`, 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 errors, 0 assertions        11.7s
```

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 100 runs passed. `make check-all` shows that the same graded test also passes on a router with B15 switched on. On that router, a beat that was waiting on a stalled output when `rst_n` fell is still on the output after the reset, and that breaks rule O4 of the specification. The line `100 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 a row for every rule in the router specification, and a few rows hold two or three rules that go together, such as I1 and I2. 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 (`a_02`), one of the scoreboard's `write_` methods, the predictor's comparison of a read with the mirror, a register sequence, or "nothing".
- **Made to happen by**: the sequence, and the step in it, that creates the situation the rule describes.
- **Seen in coverage**: the bin in `router.pkt`, `router.wait` or `router.corner` 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: `RouterTest`'s `long_wait` and `stalled`	`router.wait.x_done_wait`	done

Id	The rule	Checked by	Made to happen by	Seen in coverage	Status
			phases, and `backpressure()` in `RouterLab05Vseq`	on both outputs	

Bugs that hid behind an empty column

You can call a rule verified only when it has a check that would fire if the router broke it, a step in the stimulus that creates the situation that the rule needs, and evidence from a run that the situation happened, such as a coverage bin's count or a count in the scoreboard's summary at the end of the log. Each of these three bugs got past the testbench that its lab starts from, 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 `router.pkt.len`'s `oversize` bin was empty
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. Assertions for interface rules has an example on a one-beat valid/ready stage. Its `drop` bug throws away a beat that has waited four cycles, which breaks the stage's rule S1, and with the sink ready in every cycle no beat ever waits, so the assertion for S1 passes.

B02 is a less obvious case. The check is there and so is the traffic, but the traffic runs in a state in which the bug changes nothing. In Lab 06, Part 5, the `irq` check caught B02 as soon as you wrote it, in Lab 04′s traffic. `INT_EN` was still 0 there, so the first NOROUTE drop set a `STATUS` bit whose enable was 0. A correct router keeps `irq` at 0 when that happens, and a router with B02 raises it. The step of `RouterLab06Vseq` that you write next, in Part 5, step 3, ends with `INT_EN` at `0b111`, and `RouterTest` runs it before Lab 04′s traffic starts. With every enable on, `STATUS & INT_EN` equals `STATUS`, so a router that ignores the enables raises `irq` exactly when a correct one does. In Lab 06′s `RouterTest`, once that step is in place, only the cycles of the step itself before it turns the enables on 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 coverage that the regression merges at the end. For the starter's regression, the merged coverage began

```
   top                                      81.67 %   49 of 60 bins
```

and named the eleven bins it never hit. Seven of them are bins of `router.pkt.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', 'low')`,

`('out0', 'high')` or the other five. The other four are the bins of `router.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 ten tests did.

No test can hit those seven bins, so the regression can't reach 100%, and the target is instead 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, our regression of twelve tests printed `120 ok  (120 runs)` and

```
  top                                          88.33 %   53 of 60 bins
```

with only the seven bins of `router.pkt.x_fate_level` still empty, which meets that target. cocotb-coverage's percentage for a group is the share of its bins that were hit, crosses included, so a bin that the specification rules out stays in the count, and 53 of 60 bins (88.33 %) is the most that any regression of this environment can reach. The total can't tell you which empty bins are ruled out and which are holes, so read the list of bins as well.

You can check the plan and the coverage against each other. In Lab 07 you find the gaps in the reset and saturation rows twice, once in Part 2 by reading the sequences and once in Part 3 in the four empty bins of `router.corner`. 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 that the design does what the plan says, under the conditions that 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 version of the testbench 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 merged coverage shows which situations happened, with each empty bin named. `make check-all` on more than one seed shows that the checks can fail, and Lab 07, Part 7 runs it on seeds 1, 2 and 3. 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. Catching all fifteen tells you only 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 we dealt B15 on the starter package, with `make new BUGPICK=B15`, the hunt printed

```
   quiet  the mystery bug                       0 errors, 0 assertions        11.0s
-------------------------------------------------------------------------------
 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.
```

You follow up a quiet hunt in your plan. The rows that could hide a bug from `RouterTest` 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*: `RouterTest`; *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 that an industrial project clears before sign-off. Lab 07′s criteria are a small version of them. The page's first gate is code coverage, which this course's builds don't collect (they collect only cocotb-coverage bins). Its functional-coverage gate is a percentage, and our percentage counts the seven bins that the specification rules out, so judge that gate by the list of bins that `make cov` prints.

Next: Writing the verification report.

Answers

A1. Check yourself

Nothing shows that `RouterTest` 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 count above 0 in `router.pkt.len`'s `oversize` bin is the evidence the row needs.

A2. Check yourself

Seven of the bins are bins of the cross `router.pkt.x_fate_level` 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 bin is hit except the ones that the specification rules out, and that the plan gives the reason for each of those. Any other empty bin is a hole for new stimulus to close.

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 version of the testbench the claim is about, which "the regression passes" leaves open.

25. 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. This course's capstone ends with a **verification report**, which claims that a whole design follows its specification in every situation that a written plan describes and that your environment checks. 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, `CountTest`, 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. CountTest 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 incremented the counter at 15, so the rule about 15 was tried on those
five only.

## 4. Regression

make seeds TEST=CountTest SEEDS="1 2 3 4 5 6 7 8 9 10"

    10 ok  (10 runs)

## 5. Coverage

    top                                          100.00 %   4 of 4 bins

The four bins are the merge of all ten runs. Run by run (the [COUNT] line at the end
of each log), the counter was incremented at 15 on seeds 4, 6, 7, 9 and 10, and never
on seeds 1, 2, 3, 5 and 8.

## 6. Bugs

make seeds TEST=CountTest SEEDS="1 2 3 4 5 6 7 8 9 10" BUG=wrap

    5 MISSED, 5 caught  (10 runs)

caught on seeds 4, 6, 7, 9 and 10, MISSED on seeds 1, 2, 3, 5 and 8. The first error
on seed 4:
```

```
       1675.00ns ERROR    ..ions-and-seeds/test_seeds.py(69) [uvm_test_top]: [COUNT]
count is 0, expected 15

## 7. What this doesn't show

- On half the seeds the counter was never incremented at 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 an increment at 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 CountTest
on seeds 1 to 10, which incremented it at 15 on seeds 4, 6, 7, 9 and 10. To check it:

    make seeds TEST=CountTest 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 the example has no written plan. A plan for it would have three rows, one for each rule in the summary.

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 and that all ten passed. `make seeds` marks a run that crashed, or that raised an exception that isn't a check, as BROKEN and counts it separately on the same line, so the line also tells you that none of the ten runs crashed. 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"). So that you have those lines when you write the report, Lab 07 has you save the output of the runs that the report quotes as you go. In Part 6 you send the regression's output through `tee` into `results/`, in Part 7 you do the same with the three `make check-all` runs, and in Part 8 you copy that folder into your repository beside the report. 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 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 draw anywhere in a test changes every random choice made after it, in any coroutine, and that renaming a test changes its stimulus too.)

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 `CountTest` on seeds 1 to 10, and only five of those seeds incremented the counter at 15, so the report's sentence names the test, the seeds and those five. Every condition that your evidence depends on has to go into the sentence, 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. 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 was never incremented at 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 nothing checks, or that nothing makes happen, 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 seeds you didn't run;
- anything that depends on X, which Verilator never shows, because it has only two states; for example, the specification says that the router's state before its first reset is undefined, but on Verilator the router 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 so that it records how well a failure message from your environment led you to the fault. It's also practice for the times when there is no `make verdict` to ask, and the log is all you have to work from.

A hunt that comes back `quiet` gets a row too. `quiet` means that nothing `RouterTest` 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; in this course the seed is `COCOTB_RANDOM_SEED`, which `make run SEED=n` sets.
- 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

Any three of: 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 incremented it at 15.

A2. Check yourself

Any two of: the plan's open items (rows that nothing checks or nothing makes happen), 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.

26. Lab 07: Capstone

In this lab you take the Lab 06 testbench to sign-off. You write a verification plan from the specification, then run every test on ten seeds and read the merged coverage. You also hunt three mystery bugs, each dealt from all fifteen bugs in the catalogue. As shipped, the package's graded test catches fourteen of the fifteen. You catch the one it misses by writing the 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 write a sequence that takes both drop counters to 255, which fills the last two coverage bins that a test can reach, and you 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.
- 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.

The package is the Lab 06 testbench with our solution to Lab 06 in it. That solution has the seven registers' fields, the adapter, the reset and bit-bash sequences, the predictor and its connections, the `irq` check and `RouterLab06Vseq`'s steps. If your own Lab 06 printed `5 of 5 caught.`, your code did the same job, but use ours anyway, 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 to each file for this lab.

File	What Lab 07 adds
`tb/router_agents.py`	`ResetDriver` and `ResetSeq`, which let the testbench reset the router, in a sys agent beside Lab 06's sys monitor; `ResetState`, through which the reset driver warns the other drivers a cycle before each reset; a source driver that abandons its packet when a reset comes (rule I3); and an APB driver that keeps `psel` at 0 during a reset and starts the interrupted transfer again after it (rule A4)
`tb/router_env.py`	`reset_model()` in the scoreboard, which at a reset throws away every beat the scoreboard expected and puts the scoreboard's 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 of each kind with its counter already at 255); and the coverage group `router.corner`, with a bin for each corner
`tb/router_sva.sv`, `tb/router_sva_bind.sv`	the APB port and `irq` as inputs of the bound module, and a failure counter for each rule about reset; you write the assertions (Part 5)

File	What Lab 07 adds
`tb/router_vseq.py`	`reset()` in the base sequence; `RouterResetVseq` and `RouterSaturateVseq`, whose bodies you write (Parts 5 and 6); `RouterLab07Vseq`, four phases of random traffic, in which you write the resets (Part 5)
`tb/test_router.py`	`RouterTest` runs `RouterLab07Vseq` last; three new tests, `RouterLab07Test`, `RouterResetTest` and `RouterSaturateTest`
`Makefile`	`RouterLab07Test` in `TESTS`; `BUG_LIST` with all fifteen bugs, B15 first, because `make check` grades the first bug in the list
`PLAN-template.md`, `REPORT-template.md`	the plan and the report you fill in

Time: about 3 h 30 min, of which about 30 minutes is waiting for the machine. We measured the runs in the course image with 2 CPUs. The two regressions took 8 min 27 s and 9 min 21 s, each of the three runs of `make check-all` took 114 to 118 s, and a run of `RouterTest` takes about 12 s. The assertions are SystemVerilog, so `make build` rebuilds the simulator after you write them, which took us 2.5 s. The sequences are Python and the two tests you add are names in the Makefile's `TESTS`, so neither needs a rebuild. 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. Open a terminal on the workbench desktop, as in Lab 06, go to the lab folder, and build the simulator:

   ```
   . /foss/designs/common/.designinit   # in every new terminal
   mod dd105-b
   cd labs/lab-07-capstone
   make build
   ```

   It ends with:

   ```
    built /foss/designs/modules/dd105-b/labs/lab-07-capstone/build/sim_build/Vtop in
   2.4 s (2.1M in /foss/designs/modules/dd105-b/labs/lab-07-capstone/build/sim_build)
    ccache:  Hits: 47 / 63 (74.60%)  Misses: 16 / 63 (25.40%)
   ==========================================================================
    /foss/designs/modules/dd105-b/labs/lab-07-capstone/build/sim_build/Vtop is up to
   date.
   ```

   The ccache counts on that line include our builds of Labs 01 to 06, which came to 41 hits and 15 misses, so this build's own were 6 hits and 1 miss.

2. 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 into a `projects` folder beside the course, and tell git who you are, and to remember your token, in that repository:

   ```
   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-b/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 the repository's `.git/config`, and `/foss/designs` is on your own disk, so they're still there after a restart. 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. Run the graded test, `RouterTest`, and pull the scoreboard's summary out of the log:

   ```
   make run
   grep -o '\[SB\] [0-9a-z].*' build/runs/RouterTest-s1/run.log
   ```

   You should see `ok: 0 errors, 0 assertions  12.54 s` from `make run`, and from `grep`:

   ```
   [SB] 1735 packets in: 447 to output 0, 479 to output 1, 245 NOROUTE, 564 FULL; the
   highest level a header saw was 16
   [SB] beats checked: 2038 on output 0, 2139 on output 1; 45 DROP_CNT reads checked
   [SB] registers: 884 reads, 65 of STATUS checked; 492 transfers to bad addresses; irq
   checked in 18584 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 `RouterTest` checked `irq` in 13918 cycles. This one checks it in 18584. What ran in the other 4666 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". The environment has three checkers. The scoreboard is `RouterScoreboard` in `tb/router_env.py`, and its checks are in `write_in`, `write_out`, `write_apb` and `write_sys`. The predictor is `RouterRegPredictor` in `tb/router_ral.py`, which you wrote and connected in Lab 06. The third is the assertions in `tb/router_sva.sv`. The reset and bit-bash sequences also compare what they read, but only while they run. The docstring or comment at the top of each file says what it covers.

3. Run `grep -n TODO tb/*.py tb/*.sv Makefile` 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. The comments in `tb/router_vseq.py` say what each step of Lab 05's and Lab 06's sequences sets up, the comment above `RouterLab07Vseq` describes its phases, and the docstring of `RouterTest` in `tb/test_router.py` lists what the graded test runs.

4. Mark the rows you think `RouterTest` never reaches. Then look at what the scoreboard says about resets in your Part 1 run:

   ```
   grep -o '\[SB\] resets.*' build/runs/RouterTest-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, then merges the coverage of those runs. 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 8 min 27 s. You should see `100 ok  (100 runs)`, and then the merged coverage of the 100 runs:

   ```
     top                                          81.67 %   49 of 60 bins
       top.router                                   81.67 %   49 of 60 bins
         top.router.pkt                               83.33 %   35 of 42 bins
           top.router.pkt.len                           100.00 %   9 of 9 bins
           top.router.pkt.dst                           100.00 %   4 of 4 bins
           top.router.pkt.fate                          100.00 %   4 of 4 bins
           top.router.pkt.level                         100.00 %   5 of 5 bins
           top.router.pkt.x_fate_level                   65.00 %   13 of 20 bins
               not hit: ('full', 'empty'), ('full', 'low'), ('full', 'at_8'), ('out0',
   'at_9'), ('out0', 'high'), ('out1', 'at_9'), ('out1', 'high')
         top.router.wait                              100.00 %   14 of 14 bins
           top.router.wait.port                         100.00 %   2 of 2 bins
           top.router.wait.wait                         100.00 %   4 of 4 bins
           top.router.wait.x_port_wait                  100.00 %   8 of 8 bins
         top.router.corner                            0.00 %   0 of 4 bins
           top.router.corner.kind                       0.00 %   0 of 4 bins
                    not hit:  reset_with_beats,  reset_mid_packet,  noroute_saturated,
   full_saturated
   ```

3. `router.corner` has a bin for each of the four corners named near the top of `tb/router_env.py`, and the scoreboard counts each corner and sends it to the corner coverage 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/merged.xml` 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: `100 ok  (100 runs)` and `81.67 %  49 of 60 bins`, 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 `RouterTest` 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 12 s. A bug that piles up 50 errors early takes less, because the scoreboard stops a run at its 50th error. On one of our deals, the hunt printed:

```
  caught  the mystery bug                        1 error, 0 assertions         11.3s
       RouterTest: AssertionError: 1 error logged, the first: irq is 1 from cycle
5906, but STATUS = 001 an
```

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 names the rule that 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. 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 23 s in all. For the hunt above you should see:

```
     ok  1. the golden design  must PASS         0 errors, 0 assertions        11.3s
 caught  2. the mystery bug    must FAIL         1 error, 0 assertions         11.3s
```

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 errors, 0 assertions        11.5s
------------------------------------------------------------------------------
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 `RouterTest`. `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 `RouterScoreboard`. 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 2.5 s. You should see `ok: 0 errors, 0 assertions  11.50 s` from `make run`, and the same `[SB]` lines as in Part 1. None of the assertions fails. Seven of the eight check the router's own rules, which the golden router keeps, and `a_A4` checks a rule of the APB master's, which our APB driver keeps. 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, which lasts two cycles and comes when the router holds nothing.

4. Write `RouterResetVseq.body()` in `tb/router_vseq.py`. 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. Run the reset beside the traffic with `par()`, the way `RouterLab05Vseq` runs its route writes beside its random packets. After each reset, let the sinks take beats, call `setup()` and send more packets. `reset()` warns the other drivers at the next rising edge and pulls `rst_n` low at the one after, so a reset starts two cycles after you call it.

5. In `RouterLab07Vseq`, at the `TODO (Parts 5 and 6)`, write the line that runs your sequence (Part 6 adds the line for the other one). Then write the two `TODO (Part 5)`s in the random phases, which set `do_reset` and add the thread that resets the router, and add `RouterResetTest` 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 run it:

   ```
   make run TEST=RouterResetTest
   grep -E 'reset from|abandoned|\[SB\] resets' build/runs/RouterResetTest-s1/run.log
   ```

   With the reference solution's sequence, the first lines are from the E8 reset:

   ```
      275.00ns INFO     ..7-capstone/tb/router_env.py(369) [uvm_test_top.env.sb]: [SB]
   reset from cycle 28: 4 expected beats are lost, 2 beats of the packet coming in
      275.00ns INFO     ..apstone/tb/router_agents.py(270) [uvm_test_top.env.src.drv]:
   [SRC] reset: abandoned dst 0, 8 beats: 01 15 74 d5 6f da 03 ae after 2 beats
   ```

   then, in the reset during register traffic,

   ```
     1390.00ns INFO     ..apstone/tb/router_agents.py(582) [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, 14 expected beats lost; corners hit: reset_with_beats=3 reset_mid_packet=1 noroute_saturated=0 full_saturated=0` and `ok`, in 2.29 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 take 7

cycles together, counting the idle cycle that the APB driver leaves before each one. After `cycles(8)`, `reset(1)` warned the APB driver 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: `RouterResetTest` 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 `RouterSaturateVseq.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 by the time the read completes, but a read returns the counter as it was a cycle earlier. 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 `RouterLab07Vseq`, add `RouterSaturateTest` to `TESTS`, and run the test:

   ```
   make run TEST=RouterSaturateTest
   ```

   With the reference solution you should see `noroute_saturated=5  full_saturated=5` in the `[SB] resets` line, and `ok`, in 2.47 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 9 min 21 s. You should see `120 ok  (120 runs)` and:

   ```
     top                                          88.33 %   53 of 60 bins
   ```

   with only the seven bins of `router.pkt.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: `120 ok  (120 runs)` and `88.33 %   53 of 60 bins`.

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 `RouterTest` 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 114 to 118 s. With the reference solution, seed 1 printed:

   ```
        ok  the golden design                     0 errors, 0 assertions        11.7s
   ------------------------------------------------------------------------------
     caught  tier 6  B15_output_valid_survives_reset  50 errors, 37 assertions     11.3s
     caught  tier 1  B01_status_write_clears_all     3 errors, 0 assertions        11.7s
     caught  tier 1  B02_irq_ignores_int_en          4 errors, 0 assertions        11.8s
   ```

```
  caught  tier 1  B03_route3_readback_swapped     5 errors, 0 assertions        11.8s
  caught  tier 1  B04_misaligned_address_accepted  50 errors, 0 assertions       2.5s
  caught  tier 2  B05_dst_field_off_by_one        50 errors, 0 assertions        2.9s
  caught  tier 2  B06_disabled_route_forwards     50 errors, 0 assertions        3.1s
  caught  tier 2  B07_full_counted_on_disabled_route 6 errors, 0 assertions      11.9s
  caught  tier 3  B08_admission_off_by_one        50 errors, 0 assertions        3.4s
  caught  tier 3  B09_write_pointer_power_of_two   50 errors, 1 assertion        2.9s
  caught  tier 4  B10_output_reloads_while_stalled 50 errors, 368 assertions     3.5s
  caught  tier 4  B11_cut_packet_has_no_last      50 errors, 45 assertions       11.6s
  caught  tier 5  B12_out1_uses_out0_ready        50 errors, 3 assertions        2.7s
  caught  tier 5  B13_admission_checks_other_output 50 errors, 0 assertions      2.9s
  caught  tier 6  B14_enable_resets_to_one        3 errors, 20 assertions        11.6s
-------------------------------------------------------------------------------
 15 of 15 caught.
```

Seeds 2 and 3 printed `15 of 15 caught.` too, with B15 at `27 errors, 31 assertions` and `50 errors, 31 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. Your reset sequence caught it in Lab 06, Part 2, and your predictor from Part 3 on. Which check fires first on it now?

```
grep -m1 -E '%Error|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 TEST=RouterResetTest
make waves TEST=RouterResetTest BUG=B15
surfer build/runs/RouterResetTest-s1/waves.fst &
surfer build/runs/RouterResetTest-s1-B15/waves.fst &
```

   Add `clk`, `rst_n`, `in_valid`, `in_ready`, `in_data`, `out0_valid`, `out0_ready`, `out0_data` and `level0` from `router`, and `n_O4_rst` and `n_O4_after` from `router.u_sva.u_out0`, and look at 200 to 340 ns:

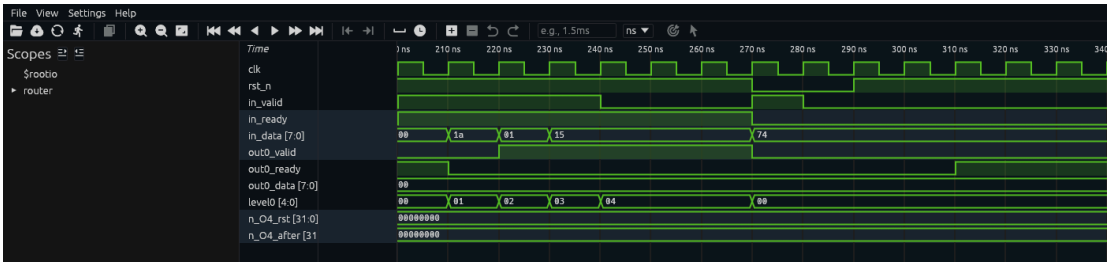


Figure 26.1: The E8 reset on the golden router, in Surfer

The golden router, with the reference solution's sequence. Output 0 has been stalled since 210 ns. The 2-beat packet `00 1a` went in first, and its header `00` has been waiting on output 0 since 220 ns, when `out0_valid` rose. The next packet's first two beats, `01 15`, went in after it, so `level0` is 4. (From 240 ns `in_valid` is 0, because that packet's third beat waits out a 3-cycle gap.) `rst_n` falls at 270 ns, and `out0_valid`, `in_ready` and `level0` fall with it, without waiting for a clock edge. The source driver offers the third beat, `74`, in the first reset cycle, then lowers `in_valid` at 280 ns and drops the rest of the packet. `rst_n` rises at 290 ns, the sink is ready from 310 ns, and nothing comes out, as example E8 says.

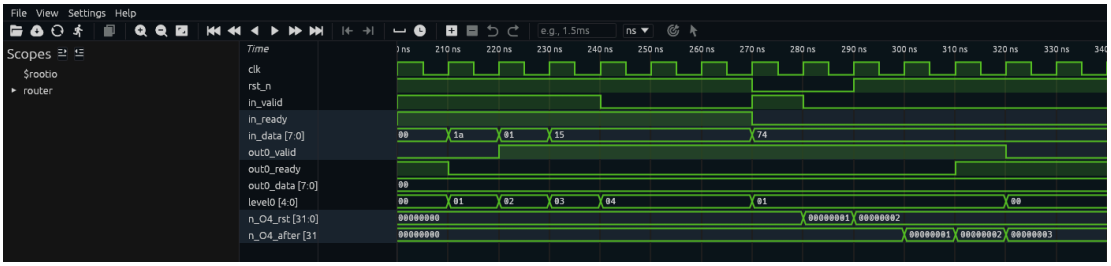


Figure 26.2: The same reset with B15 on, in Surfer

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 the rising edges that end the two reset cycles (280 and 290 ns), and* `a_O4_after` *at the next three (300, 310 and 320 ns). The sink takes* `00` *at the edge at 320 ns, and the scoreboard reports it:* `output 0: got 00 in cycle 32, 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 where a reader who has to rely on your claim finds 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/*.py tb/*.sv $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 100 runs out of 100. 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 would give the `in_ready` rows a checker, if you keep a copy of `CTRL.EN` beside them, updated from the APB writes. A check of `LEVEL` reads against the levels that 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`. In Part 6′s regression the twelve tests took about 56 s a seed, so 50 seeds take about 47 minutes.
- 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

`RouterLab07Vseq`, which `RouterTest` now starts after everything Lab 06′s `RouterTest` ran. Its four phases of random traffic, and its reads of every register at the end, 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 yet run the reset and saturation sequences that you write in Parts 5 and 6. It goes last because every random value in the testbench comes from Python's `random`, one after another, so a sequence that ran earlier would change every packet after it. In Lab 06, Part 5, the payload of your 9-beat packet changed the packets of Labs 04 and 05 in just that way, because they ran after it. We ran `RouterTest` with `RouterLab07Vseq` left out, on seeds 1 to 5, and it printed Lab 06′s lines number for number, with the new `[SB] resets` line as well. On seed 1 those lines include `1135 packets in: 311 to output 0, 288 to output 1, 177 NOROUTE, 359 FULL` and `irq checked in 13918 cycles`. So nothing else that we added changes what runs before `RouterLab07Vseq`.

A2. Part 2: The verification plan

Four rows need a reset with traffic in flight, because at time 0 the router holds nothing. They are RS2 (a reset with beats in a buffer), RS3 (a reset in the middle of an input packet), O4, and the source's rule I3. I7, O4, the first half of A4 and RS1 have no assertion yet; theirs go where the `TODO (Part 5)` comments are, in `tb/router_sva.sv`. The scoreboard checks `irq` during a reset, but nothing checks RS1′s other three values. DR2, the rule that each counter stops at 255, needs 255 drops with no clear or reset between them, but `RouterInjectTest` and `RouterLab07Vseq` both clear `DROP_CNT` at the end of each of their phases.

Nothing in the environment compares `in_ready` with what the specification says it should be (I4, I5, I6, RT5); the source driver reports only an `in_ready` that stays 0 for 1000 cycles with a beat on offer. Nor does anything check when a beat appears (O8), whether the outputs come from flip-flops (O3), what `CTRL.EN` does (EN1), whether the APB master keeps its rules (A1 to A3), when `pready` is 0 (A5, A6) or what `LEVEL` reads (RG4). Nothing in this lab adds a check for any of these rows, so they 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 `router.corner`. The specification rules out the seven in `router.pkt.x_fate_level`, as you found in Lab 03, because a packet is a FULL drop only at a level of 9 or more, and is written to an output only at a level of 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 bins point at them too. 100 passes mean that no check fired in 100 runs. The four empty bins are situations that the stimulus never made, so no check ever saw them, and a regression that passes on every seed doesn't tell you that the environment is finished.

A4. Part 4: Mystery hunts

The rule is IRQ1, that `irq` is the OR of `STATUS & INT_EN`. In cycle 5906 `STATUS.NOROUTE` is set and its enable is 0, and the router raises `irq` anyway, so it isn't masking `STATUS` with `INT_EN`. The interrupt is in `router_regs.sv`, with the registers. The hunt's line cuts the message short. The full log from `make hunt VERBOSE=1` has all of it: `irq is 1 from cycle 5906, but STATUS = 001 and INT_EN = 000 make it 0`. In that log the error comes between `starting RouterLab06Vseq` and `RouterLab06Vseq 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. We took the call to `reset_model()` out of `write_sys()` and ran the reference solution's `RouterResetTest`, and it failed the golden router with 56 errors. The register model's mirror still held the values from before the E8 reset, so the predictor failed the first reads after it (`[RAL] CTRL read 0x0 in cycle 57; the mirror holds 0x1`). The beats the scoreboard expected were still queued, so the first beats out of output 0 after the reset were compared with beats from before it (`output 0: got 02 in cycle 120, expected 00`). The scoreboard's routing table, levels and drop counts would also still hold their values from before the reset, so it would decide the fates of later packets wrongly. `reset_model()` throws all of that away in the first reset cycle.

Lab 06's drivers needed changing too. We ran the same test again, with `reset_model()` back in, using Lab 06's source and APB drivers in place of this lab's. After the E8 reset the source driver went on offering the third beat of the packet that the reset had cut. `CTRL.EN` was 0 again, so nothing took the beat, and 1000 cycles later the driver gave up with `in_ready stayed 0 for 1000 cycles with beat 2 of dst 0, 8 beats: 01 15 74 d5 6f da 03 ae on offer; dropping the rest of it`. In a later reset the APB driver kept `psel` at 1, which `a_A4` reported as `A4: psel is 1 during reset`.

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 errors, 14 assertions`. The scoreboard you already had compares every `DROP_CNT` read with its own counts, which stop at 255, so once the new sequence reads `DROP_CNT` past 255, the scoreboard 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 errors, 0 assertions        11.5s
```

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's log. In our seed 3 run the first failure came in the E8 step of the reset sequence:

```
[140020000]      %Error:      router_sva.sv:83:      Assertion      failed      in
router.u_sva.u_out0.a_O4_rst: O4: out_valid is 1 during reset
```

`a_O4_rst` failed again at the next rising edge and `a_O4_after` at the two after that, and then the scoreboard reported the beat that came out:

```
140055.00ns ERROR    ..7-capstone/tb/router_env.py(181) [uvm_test_top.env.sb]: output
0: got 00 in cycle 14006, and no beat was expected
```

A8. Part 7: Sign-off

In the package as shipped, the first check was the predictor's, in cycle 10, on the reset sequence's read of `CTRL` in `RouterLab06Vseq`: `[RAL] CTRL read 0x1 in cycle 10; the mirror holds 0x0`, followed by `RegResetSeq`'s own error. Now the first is your assertion `a_I7`. It fires during the reset at time 0, at the rising edge at 10000 ps that ends cycle 1: `[10000] %Error: router_sva.sv:148: Assertion failed in router.u_sva.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 predictor's error comes nine cycles later and tells you which register is wrong, while `a_I7`'s names the rule that 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 four cycles later, and only because the sink took the beat. Its message, though, 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, your plan named no assertion for I7, O4, the first half of A4 or RS1 and nothing that made RS2, RS3 or DR2 happen, and it had 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 do you need to know about the router to write a test for it, before you've read the whole specification?*

The router's specification is the page The packet router, and you write every router testbench in this course from it. It describes the router's pins cycle by cycle, which makes it long. This page covers the same design with only the detail that your tests depend on, and each section links to the parts of the specification 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` prints during a run of the RTL, so each one is a record of what the RTL did.

The blocks

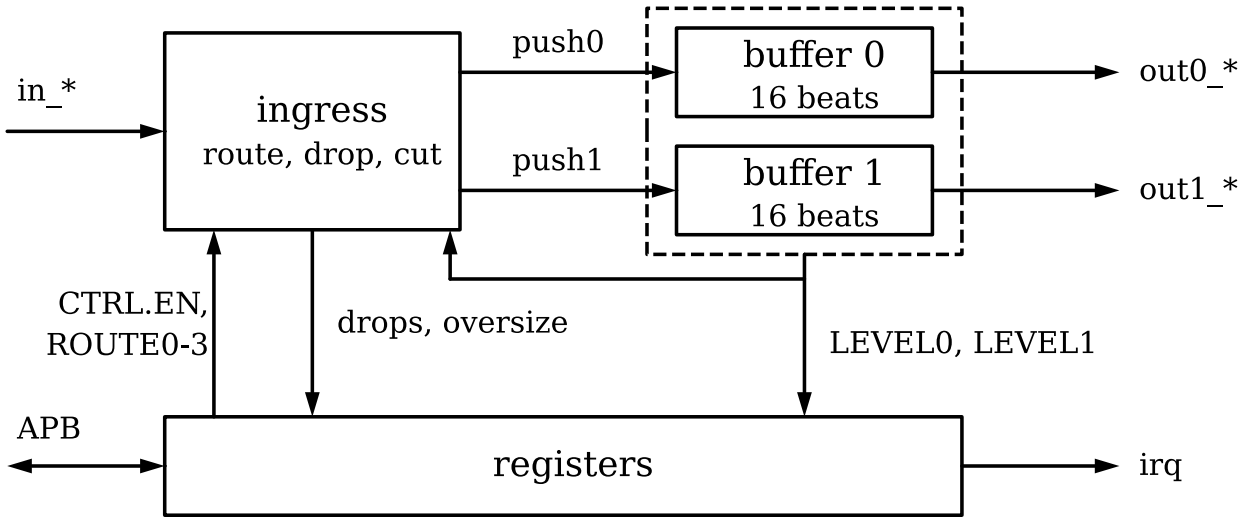


Figure A.1: The router's four blocks. Arrows show which way the data goes, and each stream also has a `ready` going the other way.

The RTL is in `labs/common/rtl/`, 730 lines in five files. One file is a package of sizes and register offsets, three files hold the three kinds of block (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 the RTL, and from Lab 02 on you'll be testing faulty builds of it that you can't trust anyway, but this is how the blocks divide the work:

- The **ingress** takes packets from the input stream. When a packet's header arrives, the ingress looks up the routing table and the level of the output that the route names, and decides once, for the whole packet, whether to write it into that output's buffer or to drop it. It also cuts a packet that is longer than 8 beats, and it reports every drop and every oversize packet to the registers.
- Each **buffer** holds up to 16 beats, 15 in a small RAM and one in the register that drives the output pins, so `out0_valid`, `out0_data` and `out0_last` come straight from flip-flops. A buffer's **level** is the number of beats it holds, including the one on the pins.
- The **registers** are the APB slave. They hold the configuration that the ingress uses (`CTRL.EN` and the four `ROUTE` entries), record drops in `STATUS` and `DROP_CNT` and oversize packets in `STATUS` alone, read back both levels in `LEVEL`, and drive `irq` from `STATUS` and `INT_EN`.

The sizes are small on purpose, so that a short simulation can reach every corner of the design. Beats are 8 bits, there are 4 routes and 2 outputs, packets are 1 to 8 beats long, each buffer holds 16 beats, and the drop counters stop at 255 (Sizes).

Ports

Everything happens on the rising edge of `clk`, the APB port included, and the router samples its inputs there (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 Python testbenches change the router's inputs just after a rising edge and sample its outputs at the falling edge, in the middle of the cycle, so no signal changes at the moment it's sampled.

Verilator has only the values 0 and 1, so a `data` signal whose `valid` is low still holds some value, usually the last beat it carried. Never compare `out0_data` or `out0_last` in a cycle where `out0_valid` is 0, because what you'd be comparing is most likely left over from an earlier beat.

Packets and routing

A packet is 1 to 8 beats on the input stream. 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 labs' testbenches put the packet's sequence number there. A delivered packet comes out exactly as it went in, except that an oversize packet is cut to 8 beats, as the end of this section describes (Packets).

The router decides what happens to a packet once, in the cycle its header is accepted. It uses `ROUTE[DST]` and the level of the output that the entry's `PORT` names, both as they are in that cycle (Routing):

`ROUTE[DST].EN`	Level of output `PORT`	The packet is	Counted in
0	anything	dropped	`NOROUTE_CNT`, `STATUS.NOROUTE`
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` while a packet is coming in, or either level changes, only later packets are affected. A header accepted in the cycle after the 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.

These three details of admission matter when you write a scoreboard:

- A packet is admitted only at a level of 8 or less, and it's at most 8 beats long, so an admitted packet always fits in the 16-beat buffer. With output 0 stalled, 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 a tenth is dropped.
- The level in the header's cycle counts every beat written into the output in earlier cycles, minus every beat taken by an output handshake in earlier cycles. Neither the header itself nor a handshake on the output in the same cycle counts. The beats written into an output are the input beats of the packets that your testbench predicts go 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 its `PORT` names, and it's never counted as FULL as well.

At the edge that ends the header's cycle, a drop adds one to its counter, unless the counter is already 255, and sets its `STATUS` bit (Dropped packets).

A packet is **oversize** if its 8th beat doesn't have `in_last` = 1. In the cycle the router accepts that beat, it sets `STATUS.OVERSIZE`, which 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 from the input and throws it away, up to and including the beat with `in_last` = 1. An oversize packet that is also dropped is counted once, as a drop, and sets `STATUS.OVERSIZE` as well. `DROP_CNT` has no counter for oversize packets (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 in which it's taken, where t_k is the cycle of its input handshake	rule O8
A stalled output	holds its beat, unchanged, until the sink takes it, while the other output and the input carry on	rules O1, O2, O7
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	for a drop, its `STATUS` bit and its counter change from the cycle	E4, E5, E6

What	When	In the specification
	after the header; for an oversize packet, `STATUS.OVERSIZE` changes from the cycle after the 8th beat	
`irq`	changes 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. They are `0x18`, `0x1C`, everything from `0x30` to `0xFF`, and every address that isn't a multiple of 4. A transfer to a bad address takes the same number of cycles as any other 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 write is ignored.

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` recording events; it only decides which ones 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 the counter reads 1 afterwards (E9, DROP_CNT).
- `PORT` keeps its value while `EN` is 0, but has no effect then (ROUTE0 to ROUTE3).

Reset

`rst_n` is asynchronous. When it falls, the whole router resets at once, without waiting for a clock edge, and the router runs normally again from the first rising edge after `rst_n` returns to 1. While it's low, `in_ready`, both `out_valid`s, `irq`, `pslverr` and `prdata` are 0 and `pready` is 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 comes out afterwards, not even a beat that was waiting on a stalled output (E8). The router's state is undefined until its first reset, so hold `rst_n` low through at least one rising edge at the start of every simulation (Reset).

The rules

The table below gives each numbered rule from the specification's Valid/ready rules and APB rules sections in one line. The rules marked *testbench* are for your stimulus to keep, and the router assumes them without checking them. The specification also says how to check each kind of rule. Most rules 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, 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; a reference model checks them, and so does a test that changes an input in the middle of a cycle.

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 `rst_n` falls.
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.
O4	router	`out_valid` is 0 during reset, and after it until a beat accepted after the reset reaches the output.
O5	router	An output delivers its packets' beats in the order the router accepted them, unchanged except for a cut packet's `out_last`, and never interleaved.

Rule	Whose	What it says
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`, and nothing about output 1 depends 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 end of a read's first access cycle and when `rst_n` falls.
A9	router	A good write takes effect at the end of its access cycle, by the register's access type.

Check yourself

1. `ROUTE0` is `0x1`, `CTRL.EN` is 1, and output 0 is stalled with seven 1-beat packets in its buffer. 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. `ROUTE1` is 0. Your test writes `0x3` to it, and the write completes in the same cycle that a header with `DST` 1 is accepted. 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 its 8 beats take the level to 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` is set from the next cycle.

A2. Check yourself

`NOROUTE_CNT`, and only that one. A packet for a disabled route is a NOROUTE drop whatever the level of the output its `PORT` names.

A3. Check yourself

Nowhere. It's dropped as NOROUTE, because a header accepted in the cycle that a write completes uses the register's old value, and the new value applies only from the next cycle.

B. The packet router

This page is the specification of the packet router, which is the design you verify in this course. It describes everything the router does at its pins, cycle by cycle, so you shouldn't need to read the RTL to write a testbench for it. From Lab 02 on, your tests run on a faulty build of the router with bugs planted in its RTL, so the RTL wouldn't be a reliable guide to what the router should do anyway. If this page and the router ever disagree, one of them has a bug, and we'd like you to 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

We kept the sizes small on purpose, so that you can reach every corner of the design in a short simulation:

- A beat is one byte, so the header fits in one beat, and you can read a packet straight off a waveform in hex.
- With four routes and two outputs you can cover the routing table completely, because there are 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.
- A legal packet is 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.
- Each output's buffer holds exactly two maximum-length packets. A packet is admitted only while its output holds 8 beats or fewer, so when an output is stalled and packets keep arriving for it, the first packet to arrive after it holds more than 8 beats is dropped. Getting there takes between 9 beats of traffic (nine 1-beat packets) and 16 (two 8-beat packets).
- The drop counters saturate at 255. A stream of 1-beat packets for a disabled route drops one packet per cycle, so you can saturate the NOROUTE counter in 255 cycles.
- The APB address space has 256 byte addresses, and 246 of them are bad (not in the register map). A read takes 3 cycles, so a test can read all 256 in 768 cycles.

The directed test we wrote for the router covers every section of this page and runs for 6,852 clock cycles.

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, and so has no X to mark a value as unknown. A data signal whose `valid` is low still holds an ordinary-looking 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 refer to clock cycles by number. **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 the router's own flip-flops, so they change just after a rising edge or when `rst_n` falls (see Reset). The exceptions are `pready`, which also depends on `psel`, `penable` and `pwrite` in the same cycle, and `pslverr`, which depends on those three and on `paddr`. Sample those two at the end of the cycle, as you would for any APB slave.

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. This course's Python testbenches change inputs right after a rising edge and look at outputs at the falling edge in the middle of the cycle, so no signal is ever sampled at the moment it changes.

Packets

A legal packet is 1 to 8 beats on the input stream. Its first beat is the **header**, and the beat with `in_last` = 1 ends it, so 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`. Every beat of a delivered packet, the header included, goes to the output unchanged and in the order it arrived, so the packet that comes out is identical to the one 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. The level ranges from 0 to 16, and it's what the `LEVEL` register reads. It includes the beat on the output's pins as well as the ones waiting behind it. In example E2, for instance, the level in cycle 4 is 2, because `05` is on `out0_data` and `11` is behind it.

A testbench can track both levels exactly, and that's enough to predict every FULL drop. The beats written into an output are the input beats of the packets that your testbench predicts are routed there (only the first 8 beats 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 it counts neither the header itself nor a handshake on the output in that same cycle.

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. With 1-beat packets, nine 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, and it's never counted as FULL as well.

The decision holds for the whole packet. If software writes `ROUTE[DST]` or `CTRL` while a packet is coming in, or either output's level changes, only later packets are affected. A register write applies to headers accepted from the cycle after the write completes; a header accepted in the same cycle that the write completes still uses the old value.

The router still takes a dropped packet from the input, with `in_ready` at 1 for all of its beats, and throws the beats away. Because the input never waits for a full output, a stalled output doesn't hold up 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. Cutting a packet isn't a drop, so a forwarded oversize packet isn't counted in `DROP_CNT`. A packet that is dropped and is also oversize is counted once, as a drop, and sets `STATUS.OVERSIZE` as well. 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, but 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, so beats already in the buffers keep flowing while it's 0. It resets to 0, so the router takes nothing from the input 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.

To work out 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 that no beat covers this way. For the first beat after reset, $a_k = t_k + 2$. In particular:

- 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 the sink doesn't take stays on the output until it's taken, and the next beat appears in the cycle after that (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. The router drops the ones for output 0 once output 0 is 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, with `disable iff (!rst_n)`. The rules about reset itself work 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 others need more than a 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 so far, 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, and you can't see that in any single trace. 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 that the source was offering when `rst_n` fell is abandoned, and the first beat the source offers after the reset 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, and handles whichever beats it took.

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; only a falling `rst_n` changes them between edges.)

- **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 that the router accepted them, unchanged except for the `out_last` of a cut packet. One packet's beats are never interleaved with another's.
- **O6**. Every packet on an output is 1 to 8 beats long and ends with the only beat that has `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 rules to follow. Its `out_ready` may be high before `out_valid`, may change in any cycle, and may stay low for as long as the sink 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.

In the first access cycle of a read, the router copies the register into an output register, and in the second it drives `prdata` from that copy. So a read returns the register's value as it is in the first access cycle. For a register that changes on its own, like `LEVEL`, this means that a beat written into a buffer at the edge ending the first access cycle isn't 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, isn't an error. The transfer completes with `pslverr` = 0, and the router ignores the write. (`DROP_CNT` is read-only too, but a write to it isn't ignored, because any write 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, so the bus is either 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.

When you model the registers in pyuvm's register layer in Lab 06, these are the access policies to give the fields:

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 it stays set until software writes a 1 to it or the router is reset. 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`, so 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

$$\texttt{irq} = \bigvee_{i=0}^{2} \left(\texttt{STATUS}[i] \wedge \texttt{INT_EN}[i] \right) \tag{2.2}$$

It's a plain combination of the two registers with no extra delay, so `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 the write that clears the bit completes in cycle 7, so both are 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

`DROP_CNT` holds 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.

Any write to `DROP_CNT` clears both counters, whatever value it writes. 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

`LEVEL0` and `LEVEL1` are the number of beats that output 0's and output 1's buffers hold, 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, because 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 at which `rst_n` is 1.

Change `rst_n` just after a rising edge, both when you pull it low and when you release it, 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, but don't rely on that.

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 that 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 that the value doesn't matter in that cycle, and the table shows a dash in place of whatever is left on the wires. The tables use a dash for

- `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) show a value inside the router, which is what a read whose first access cycle is that cycle would return. Hex values are written without `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	-	-	-	-	-	-	-

cycle	1	2	3	4	5	6	7	8	9
`pready`	1	1	1	0	1	1	0	1	1
`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, with the same packet, and `out0_ready` low in cycles 5 to 7 and in cycle 9. Beat `11` is on output 0 from cycle 5 to cycle 8 and is taken in cycle 8. Beat `22` is on it in cycles 9 and 10 and is taken in cycle 10. Neither beat 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	0	1	2	2	2	2	2	1	1	0	0	0
`out0_valid`	0	0	0	1	1	1	1	1	1	1	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` are 1 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. The router takes beats `08` and `09` and discards them.

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	1	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. The reset clears everything at once, including the beat 01 waiting on output 0, so nothing comes out when the sink becomes ready in cycle 6. After the 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 in which 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 builds pass `--timescale 1ns/1ps`, and our testbenches 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 of `+BUG` 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. If you add `+BUG_ECHO`, it prints one line at time 0 that names the bug, or says that none is on.

C. The pyuvm register API

Use this page to look things up while you work through Lab 06. It gives the arguments of each `configure()` and what `predict()` does under each access policy. It also shows how `read()` and `write()` reach the bus, and what SystemVerilog UVM has that pyuvm 4.0.1 doesn't. The router's own register map is in the specification, with each field's policy and whether the field is volatile.

Every fact here about pyuvm comes from pyuvm 4.0.1 in the course image or from a run of the six tests in `guide-examples/pyuvm-register-api`. pyuvm's source is in `/usr/local/lib/python3.12/dist-packages/pyuvm/`, and a reference like `s19:70` means line 70 of `s19_uvm_reg_field.py` in that folder. The one quotation, under What pyuvm 4.0.1 lacks, is from pyuvm's README. The six tests run on the golden router, with the reference environment's APB agent, register model and adapter (`labs/ref/tb`) on the bus side. They print every result on a line that starts with `REGAPI`, and the outputs below are those lines, as `grep REGAPI` prints them.

The classes

You import all of these from `pyuvm`.

Class	File	What it is
`uvm_reg_block`	`s18_uvm_reg_block.py`	the register model: it holds the registers and the map
`uvm_reg`	`s20_uvm_reg.py`	one register: its fields, its address, and `read()`, `write()` and `predict()`
`uvm_reg_field`	`s19_uvm_reg_field.py`	one field: its width, position, access policy, volatility and reset value, and the model's copy of its value
`uvm_reg_map`	`s21_uvm_reg_map.py`	the registers by address, and the sequencer and adapter that their operations go through
`uvm_reg_adapter`	`s25_uvm_adapter.py`	the base class of your adapter, whose `reg2bus()` and `bus2reg()` you write
`uvm_reg_bus_op`	`s24_uvm_reg_includes.py`	the operation that the map passes to `reg2bus()` and `bus2reg()`
`uvm_reg_predictor`	`s26_uvm_predictor.py`	a class with nothing in it but a constructor (see What pyuvm 4.0.1 lacks)
`uvm_mem`	`s22_uvm_mem.py`	raises `UVMNotImplemented` as soon as you make one

The enumerations you pass and get back:

Enumeration	Values	Where it's used
`path_t`	`FRONTDOOR`, `BACKDOOR`, `USER_FRONTDOOR`	the `path` argument of `read()` and `write()`

Enumeration	Values	Where it's used
`check_t`	`CHECK`, `NO_CHECK`	the `check` argument of `read()` and `write()`
`uvm_resp_t`	`PASS_RESP`, `ERROR_RESP`	the status that `read()` and `write()` return, and that your `bus2reg()` sets
`access_e`	`UVM_READ`, `UVM_WRITE`	the `kind` of a `uvm_reg_bus_op`
`uvm_predict_e`	`UVM_PREDICT_DIRECT`, `UVM_PREDICT_READ`, `UVM_PREDICT_WRITE`	the `kind` argument of `predict()`
`uvm_door_e`	`UVM_FRONTDOOR`, `UVM_BACKDOOR`, `UVM_PREDICT`, `UVM_DEFAULT_DOOR`	the `path` argument of `predict()`

`path_t` and `uvm_door_e` both have a front door, but the two aren't interchangeable. The sections on predict() and read() and write() say what happens when you pass the wrong one.

Configuring a field

A register's `build()` makes each field and configures it (`s19:70-78`):

```
self.NOROUTE = uvm_reg_field("NOROUTE")
self.NOROUTE.configure(self, 1, 0, "W1C", True, 0)
```

Argument	Meaning
`parent`	the register the field belongs to, which is `self` in the register's `build()`
`size`	the field's width in bits
`lsb_pos`	the position of its lowest bit in the register
`access`	the access policy, one of the 26 names under Access policies; the router's fields use `"RO"`, `"RW"`, `"W1C"` and `"WC"`
`is_volatile`	`True` if the design can change the field on its own
`reset`	its value after reset

SystemVerilog UVM's `configure()` has nine arguments, and DD105-A's register layer page lists them. pyuvm's has the first six. It has nothing in place of the other three, `has_reset`, `is_rand` and `individually_accessible`.

`configure()` checks the field's size and reset value itself (`s19:93-116`). Then it adds the field to its register, and the register checks that the widths of its fields add up to no more than its own width, and that the new field doesn't overlap the one configured before it (`s20:144-179`). `RegPolicyTest` makes five mistakes on purpose. The first, a policy name in lower case, passes without a message, and each of the other four raises an exception:

```
REGAPI configure(..., "rw", ...): get_access() is 'NOACCESS'; after predict(0x1,
kind=UVM_PREDICT_WRITE) the mirror is 0x0
REGAPI a 1-bit field with reset value 2: raised UVMError: F -- Reset value for REG
field : F is [2]                  is beyond the MAX valoue given                  by
the size [1]
REGAPI a field of size 0: raised UVMFatalError: F -- Size of a filed
cannot be 0 MINIMUM allowed is 1
REGAPI a 40-bit field in a 32-bit register: raised AssertionError: WIDE -- _add_field:
Fields F                      doesn't fit into a 32 bits register
```

```
REGAPI STATUS's fields configured from bit 1 down: raised AssertionError: STATUS --
_add_field:            Fields NOROUTE overlap            with field            FULL
```

- A policy name that isn't one of the 26, such as `"rw"` in lower case, becomes `"NOACCESS"` without a message (`s19:100-101`). A `NOACCESS` field ignores every predicted write, so the mirror of a field that you meant to be read-write doesn't change when a write is predicted.
- The overlap check compares a field's lowest bit with the highest bit of the field configured just before it (`s20:169-179`), so you have to configure a register's fields from bit 0 up. When `FULL` (bit 1) of `STATUS` is configured before `NOROUTE` (bit 0), pyuvm reports an overlap, although the two fields don't share a bit.
- The last two mistakes raise `AssertionError`. Inside a test, an `AssertionError` counts as a check that fired, so a register model that raises one would be graded as a caught bug. That's why the graders build your model on its own, outside the simulator, before every verdict. You run the same build with `make regcheck` in Lab 06, Part 2. In the simulation's log, each of the two `AssertionError`s has a second line, `assert False`, which `grep REGAPI` leaves out.

Configuring a register and adding it to the map

The signatures (`s20:27`, `s20:57-64`, `s21:53`, `s21:100`):

```
uvm_reg(name, reg_width=32)
reg.configure(parent,       address,       hdl_path,       throw_error_on_read=False,
throw_error_on_write=False)
reg_map.configure(parent, base_addr)
reg_map.add_reg(reg, offset="0x0", rights="RW")
```

Call and argument	Meaning
`uvm_reg(name, reg_width)`	your register class calls this from its `__init__()`; the width is in bits
`reg.configure(parent, ...)`	`parent` is the register block. `configure()` calls the register's `build()` and adds the register to the block (`s20:75-79`)
`address`	a string, written the way Python's `hex()` writes it (see Addresses are strings)
`hdl_path`	pyuvm stores it, and nothing in pyuvm reads it (pyuvm has no back door); Lab 06 passes `""`
`throw_error_on_read`, `throw_error_on_write`	copied into each field as it is configured; Lab 06 passes `False, False`
`reg_map.configure(parent, base_addr)`	`parent` is the block. pyuvm stores `base_addr`, but no register's address uses it
`add_reg(reg, offset, rights)`	the map files the register under `hex(int(offset, 16) + int(address, 16))` (`s21:100-104`). `rights` must be one of the 26 policy names, or `add_reg()` raises `UVMFatalError`. pyuvm stores `rights`, and only `get_access_policy()` reads it back

```
REGAPI  add_reg(reg,  "0x0",  "XX"):  raised  UVMFatalError:  R  --  given  access
policy not correct: XX
```

Addresses are strings

The map files each register under the `hex()` form of its address plus the offset, but `read()` and `write()` look the register up by the string that it was configured with (`s21:325`, `s21:379`). The two strings are the same only when the register's address is written the way `hex()` writes it and the offset is zero. That's why

Lab 06 configures every register with `hex(addr)` and adds it with `"0x0"`. `RegAddrTest` configures three registers on a map whose base address is `0x200`:

```
REGAPI map.configure(block, 0x200); the map's keys: ['0x0', '0x104', '0x8']
REGAPI Z.read(...), Z configured at "0x00" and added at offset "0x0": raised KeyError:
'0x00'
REGAPI O.read(...), O configured at "0x4" and added at offset "0x100": raised KeyError:
'0x4'
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x8', data 0x0, n_bits
32, byte_en True
REGAPI   bus, cycle 8: APB R 08 00000000
REGAPI  S.read(...),  S  configured  at  "0x8"  and  added  at  offset  "0x0":  returned
(PASS_RESP, 0x0)
REGAPI map.get_reg_by_offset("0x8").get_name(): S
REGAPI map.get_reg_by_offset("0x08").get_name(): raised KeyError: '0x08'
```

The read of `S` went to address `08` on the bus, so the map's base address, `0x200`, wasn't added to it. The address reaches your adapter as the same string, `'0x8'`, which is why Lab 06's `reg2bus()` turns it back into a number with `int(rw.addr, 16)`. `make regcheck`, which runs the same check that the graders run before every verdict, reports a register configured at an address that `hex()` would write differently, such as `"0x00"` (`labs/common/README.md`).

Access policies

pyuvm accepts 26 policy names (`s17:16-47`). `predict()` with `UVM_PREDICT_WRITE` applies the policy's write rule (`predict_write()`, `s19:421-452`), and with `UVM_PREDICT_READ` its read rule (`predict_read()`, `s19:486-512`). `RegPolicyTest` gives a 4-bit field each policy in turn, with a reset value of `0x5`. Before each prediction it puts the mirror at `0xc` with `UVM_PREDICT_DIRECT`. In the table, then, `0xc` means that the mirror didn't change, `0xa` that it took the value written, `0x3` that it took the value read, and `0x5` that it went back to the reset value:

```
REGAPI   policy    WRITE of 0xa    READ of 0x3
REGAPI   RO          0xc             0x3
REGAPI   RW          0xa             0x3
REGAPI   RC          0xc             0x0
REGAPI   RS          0xc             0xf
REGAPI   WRC         0xa             0x0
REGAPI   WRS         0xa             0xf
REGAPI   WC          0x0             0x5
REGAPI   WS          0xf             0x5
REGAPI   WSRC        0xf             0x0
REGAPI   WCRS        0x0             0xf
REGAPI   W1C         0x4             0x5
REGAPI   W1S         0xe             0x5
REGAPI   W1T         0x6             0x5
REGAPI   W0C         0x8             0x5
REGAPI   W0S         0xd             0x5
REGAPI   W0T         0x9             0x5
REGAPI   W1SRC       0xe             0x0
REGAPI   W1CRS       0x4             0xf
REGAPI   W0SRC       0xd             0x0
REGAPI   W0CRS       0x8             0xf
REGAPI   WO          0xa             0x5
REGAPI   WOC         0x0             0x5
REGAPI   WOS         0xf             0x5
REGAPI   W1          0x6             0x5
REGAPI   WO1         0xa             0x5
REGAPI   NOACCESS    0xc             0x5
```

The router's four policies behave like this:

Policy	Router fields	A predicted write	A predicted read
RO	ID, LEVEL0, LEVEL1	leaves the mirror as it was	takes the value read
RW	CTRL.EN, INT_EN's three, ROUTEn.EN and ROUTEn.PORT	takes the value written	takes the value read
W1C	STATUS's three	clears the bits written as 1 (0xc and 0xa give 0x4)	goes back to the reset value
WC	NOROUTE_CNT, FULL_CNT	clears every bit	goes back to the reset value

In two places these results differ from what SystemVerilog UVM does. For SystemVerilog UVM we read uvm_reg_field.svh in DD105-A's copy of the library, which is UVM 2020-3.2.

- In pyuvm, UVM_PREDICT_READ puts a field back at its reset value, whatever was read, under 14 policies: WO, WOC, WOS, WO1, NOACCESS, W1, W1T, W0T, WC, WS, W1C, W1S, W0C and W0S (s19:494-511). Under the first five of those, SystemVerilog UVM leaves the mirror as it was, and under the other nine it takes the value read (uvm_reg_field.svh, lines 985 to 1031). The policies of the router's STATUS (W1C) and DROP_CNT (WC) are among the nine, so in pyuvm a predictor that predicts each read with UVM_PREDICT_READ leaves both registers at their reset value, 0, after every read. UVM_PREDICT_DIRECT takes the value read under every policy, and it's the kind that Lab 06's predictor uses after a read. Part 3, step 1 shows the difference on your own model, with make predict.
- A W1 field should take the first value written after a reset and ignore later writes. pyuvm's own comment on the policy says so (s19:416-417), and SystemVerilog UVM does it (uvm_reg_field.svh, line 893). In pyuvm, under UVM_PREDICT_WRITE, a W1 field toggles the bits written as 1 instead, as W1T does, so 0xc and 0xa give 0x6 where they should give 0xa. The test for W1T in predict_write() is field_access in ("W1T") (s19:438). Python reads ("W1T") as the string "W1T" rather than as a tuple, so the test is true for a W1 field too, because "W1" is a substring of "W1T". The router has no W1 field.

predict()

```
reg.predict(value, be=-1, kind=uvm_predict_e.UVM_PREDICT_DIRECT,
            path=uvm_door_e.UVM_FRONTDOOR, map=None, fname="", lineno=0)
```

A register's predict() (s20:193-212) hands each field its own bits of value through the field's do_predict() (s20:214-233). The field works out the predicted value from kind, path and its policy, and sets both its mirror and its desired value to it (s19:568-570). A field has a predict() of its own, with the same arguments (s19:575-594). predict() returns True unless the operation's status is UVM_NOT_OK.

Pass kind and path by name. kind is the third parameter, after be, the byte enables, so predict(value, kind) passes the kind as be. The register then raises TypeError when it shifts be for its first field (s20:232), before any field's mirror has changed. path takes a uvm_door_e. If you pass path_t.FRONTDOOR instead, the field skips its policy (s19:534-539) and takes the value as it is, as if kind were UVM_PREDICT_DIRECT. RegPolicyTest predicts a write on STATUS each of the three ways:

```
REGAPI STATUS at 0x7, then predict(0x1, kind=UVM_PREDICT_WRITE): 0x6
REGAPI STATUS at 0x7, then predict(0x1, kind=UVM_PREDICT_WRITE, path=path_t.FRONTDOOR):
0x1
REGAPI  STATUS  at  0x7,  then  predict(0x1,  UVM_PREDICT_WRITE):  raised  TypeError:
unsupported operand type(s) for >>: 'uvm_predict_e' and 'int'
REGAPI   STATUS's mirror after that: 0x7
```

The other methods that read or set the model's values:

Method	What it does
reg.get_mirrored_value()	the mirror: every field's mirror, shifted into its place (s20:236-241)

Method	What it does
`reg.get_desired()`, `reg.set_desired(value)`	the desired value. `set_desired()` applies each field's write rule to it (`s20:139-141`, `s19:302-340`), and `write(None, ...)` writes it
`reg.reset()`, `reg_map.reset("HARD")`	every field's mirror back to its reset value (`s19:262-265`). The map takes `"HARD"` or `"SOFT"`, which do the same, and raises `UVMNotImplemented` for anything else
`field.get_value()`, `field.get()`, `field.get_reset()`	a field's mirror, its desired value and its reset value
`field.get_access()`, `field.is_volatile()`, `field.get_lsb_pos()`, `field.get_n_bits()`, `field.get_msb_pos()`	what `configure()` was given, except that `get_access()` returns `"NOACCESS"` for a policy name that isn't one of the 26, and `get_msb_pos()` works out the field's highest bit from its position and width
`reg.get_fields()`, `reg.get_address()`, `reg.get_reg_size()`	the register's fields in the order they were configured, which is from bit 0 up; its address string; its width
`reg_map.get_registers(as_dict=True)`	the map's registers, keyed by the `hex()` strings

```
REGAPI map.reset("POWER"): raised UVMNotImplemented: map -- reset -- POWER is not
mapped as type of reset                                    available values are
['SOFT', 'HARD']
```

read() and write()

```
status = await reg.write(value, reg_map, path, check)
status, value = await reg.read(reg_map, path, check)
```

`write()` returns a `uvm_resp_t`, and `read()` returns a `uvm_resp_t` and the value read (`s20:276-358`). Neither has default arguments.

Argument	Meaning
`value`	the value to write, or `None` to write the register's desired value (`s20:304-307`)
`reg_map`	the map that holds the register; Lab 06′s is `regs.map`
`path`	use `path_t.FRONTDOOR`. `path_t.BACKDOOR` and `path_t.USER_FRONTDOOR` raise `UVMNotImplemented`. Any other value, such as `uvm_door_e.UVM_FRONTDOOR`, matches none of the three, so `write()` returns `None` without a transfer on the bus, and `read()` raises `TypeError`
`check`	used only by auto-predict, which is off (see What pyuvm 4.0.1 lacks), so `read(..., check_t.CHECK)` compares nothing

For a front-door operation, the map builds a `uvm_reg_bus_op` and hands it to your adapter's `reg2bus()`. It starts the item that `reg2bus()` returns on the map's sequencer and waits for the driver to finish it. Then it calls `bus2reg(item, rw)` and returns the status that `bus2reg()` put in `rw.status`, with `rw.data` for a read (`s21:299-406`). A read whose status is `ERROR_RESP` returns 0 as its value (`s20:346-347`). Neither the

map nor the register changes the mirror. `RegOpsTest` reads and writes registers through the map, on the golden router:

```
REGAPI ID's mirror before the read: 0xd1050001
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x0', data 0x0, n_bits
32, byte_en True
REGAPI   bus, cycle 8: APB R 00 d1050001
REGAPI  ID.read(map,  path_t.FRONTDOOR,  check_t.NO_CHECK):  returned  (PASS_RESP,
0xd1050001)
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x1, n_bits
32, byte_en True
REGAPI   bus, cycle 11: APB W 04 00000001
REGAPI CTRL.write(0x1, map, path_t.FRONTDOOR, check_t.NO_CHECK): returned PASS_RESP
REGAPI CTRL's mirror after the write: 0x0
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x4', data 0x0, n_bits
32, byte_en True
REGAPI   bus, cycle 14: APB R 04 00000001
REGAPI CTRL.read(map, path_t.FRONTDOOR, check_t.CHECK): returned (PASS_RESP, 0x1)
REGAPI CTRL's mirror after the read: 0x0; errors logged so far: 0
REGAPI INT_EN.set_desired(0x5): get_desired() is 0x5, get_mirrored_value() is 0x0
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0xc', data 0x5, n_bits
32, byte_en True
REGAPI   bus, cycle 17: APB W 0c 00000005
REGAPI INT_EN.write(None, map, path_t.FRONTDOOR, check_t.NO_CHECK): returned PASS_RESP
REGAPI CTRL.write(0x0, map, uvm_door_e.UVM_FRONTDOOR, check_t.NO_CHECK): returned None
REGAPI   transfers on the bus during that write: 0
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x4', data 0x0, n_bits
32, byte_en True
REGAPI   bus, cycle 21: APB R 04 00000001
REGAPI CTRL.read(map, path_t.FRONTDOOR, check_t.NO_CHECK): returned (PASS_RESP, 0x1)
REGAPI ROUTE0.read(map, uvm_door_e.UVM_FRONTDOOR, check_t.NO_CHECK): raised TypeError:
cannot unpack non-iterable NoneType object
REGAPI  ROUTE0.read(map,  path_t.FRONTDOOR,  check_t.NO_CHECK)  after  that:  raised
UVMFatalError: ROUTE0 -- read cannot perform an operation while                another
is in progress
```

`ID`'s mirror is already `0xd1050001` before the read, because the test's environment calls `map.reset("HARD")` when it builds the model. `CTRL`'s mirror stays at `0x0` after the write of `0x1` and after the read, and the read with `check_t.CHECK` logs no error, although it read `0x1` and the mirror held `0x0`. `CTRL` reads `0x1` after the write of `0x0` with `uvm_door_e.UVM_FRONTDOOR`, because that write never reached the bus. The last line comes from the rule in the next section.

One operation per register at a time

A register sets a flag, `_op_in_progress`, for the whole of a `read()` or `write()`, and an operation that starts while the flag is set raises `UVMFatalError` at once (`s20:296-316`, `s20:340-356`). SystemVerilog UVM would make the second operation wait. Operations on two different registers can overlap. `RegTwoOpsTest` overlaps two operations on `CTRL` twice, and then an operation on `CTRL` with one on `INT_EN`:

```
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x1, n_bits
32, byte_en True
REGAPI CTRL.write(0x0, ...), started in the same time step: raised UVMFatalError: CTRL
-- write cannot perform an operation while                    another is in progress
REGAPI   bus, cycle 6: APB W 04 00000001
REGAPI CTRL.write(0x1, ...), started first: returned PASS_RESP
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x0, n_bits
32, byte_en True
REGAPI CTRL.read(...), one cycle into that write: raised UVMFatalError: CTRL -- read
cannot perform an operation while                      another is in progress
REGAPI   bus, cycle 8: APB W 04 00000000
REGAPI CTRL.write(0x0, ...): returned PASS_RESP
```

```
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x1, n_bits
32, byte_en True
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0xc', data 0x7, n_bits
32, byte_en True
REGAPI   bus, cycle 11: APB W 04 00000001
REGAPI CTRL.write(0x1, ...): returned PASS_RESP
REGAPI   bus, cycle 14: APB W 0c 00000007
REGAPI INT_EN.write(0x7, ...), started in the same time step: returned PASS_RESP
REGAPI   reg2bus() got a uvm_reg_bus_op: kind UVM_READ, addr '0x4', data 0x0, n_bits
32, byte_en True
REGAPI   bus, cycle 19: APB R 04 00000001
REGAPI CTRL.read(...) after all that: returned (PASS_RESP, 0x1)
```

In each pair on `CTRL`, the operation that started second raises, and the first one completes. The flag is cleared only when an operation returns normally, so an exception raised inside `read()` or `write()` leaves it set, and every later operation on that register raises `UVMFatalError` too. That happened to `ROUTE0` in `RegOpsTest` above, and to `CTRL` after a back-door read in `RegBackdoorTest`, below. Lab 06's register helpers are in `tb/router_vseq.py`. `wr_reg` and `rd_reg` hold a cocotb `Lock` for the register while the operation runs, and `check_reg` and `rd_all_regs` go through `rd_reg`, so two parts of a sequence that use one register take turns.

Your adapter

Your adapter derives from `uvm_reg_adapter`, whose `reg2bus()` and `bus2reg()` raise `UVMNotImplemented` until you override them (`s25:32-43`).

Method	The map calls it	Yours does
`reg2bus(rw)`	with each front-door operation, before anything reaches the bus	returns a new bus item (an `ApbItem` in Lab 06) made from `rw`
`bus2reg(item, rw)`	with the same item, once the driver has finished it	sets `rw.kind`, `rw.data` and `rw.status` from the item; `rw.status` is what `read()` or `write()` returns

`rw` is a `uvm_reg_bus_op` (`s24:162-174`):

Field	In `reg2bus()`
`kind`	`access_e.UVM_READ` or `access_e.UVM_WRITE`
`addr`	the register's address, as the string it was configured with (`'0x4'`)
`data`	the value to write, or 0 for a read
`n_bits`	the register's width
`byte_en`	the adapter's `byte_enable`, which is `True` unless you change it
`status`	`uvm_resp_t.PASS_RESP`, for `bus2reg()` to change

What pyuvm 4.0.1 lacks

pyuvm 4.0.1's `uvm_reg_predictor` has no `predict()`, and pyuvm has no reset or bit-bash sequence, so Lab 06 has you write all three. The first column of this table is SystemVerilog UVM's register layer as DD105-A's register layer page describes it, and the second is what pyuvm 4.0.1 has in its place.

SystemVerilog UVM	pyuvm 4.0.1	Shown by
`uvm_reg_predictor`, which updates the mirror from a bus monitor's transfers	a class of that name with nothing in it but a constructor, so it has no analysis export and no `predict()`. pyuvm's README describes it as "Main register predictor, should be disabled if auto_prediction is not set"	`s26:6-9`; `RegPolicyTest` and `RegAutoPredictTest`, below
auto-predict, which updates the mirror after each operation the model sends	off, with no method to turn it on (see below)	`s24:125`, `s21:8`, `s21:345-347`; `RegAutoPredictTest`
check-on-read in the map, `mirror(UVM_CHECK)` on a register	neither; `read(..., check_t.CHECK)` compares nothing	`RegPolicyTest`; `RegOpsTest`, above
`mirror()`, `update()`, `needs_update()`, `peek()`, `poke()`, `set()`, `get()` and `get_reset()` on a register	none of them. A field has `get()` and `get_reset()`, and a register has `set_desired()` and `get_desired()`. The to-do list at the top of the block's file includes `needs_update`, `update` and `mirror`	`RegPolicyTest`; `s18:31-33`
`uvm_reg_hw_reset_seq`, `uvm_reg_bit_bash_seq`, `uvm_reg_access_seq`, `uvm_reg_mem_built_in_seq`	none of them. The enumeration `uvm_reg_mem_tests_e` has a value for running each of the first three (`UVM_DO_REG_HW_RESET`, `UVM_DO_REG_BIT_BASH`, `UVM_DO_REG_ACCESS`), and no file in pyuvm 4.0.1 defines the sequences	`RegPolicyTest`; `s17:149-164`
the back door, and HDL paths	`path_t.BACKDOOR` raises `UVMNotImplemented`, and so does `add_hdl_path()` on a block	`RegBackdoorTest`; `s21:316-317`, `s21:370-371`, `s18:77-78`
a user-defined front door	`path_t.USER_FRONTDOOR` raises `UVMNotImplemented`	`RegBackdoorTest`; `s21:318-319`, `s21:372-373`
`uvm_mem`	raises `UVMNotImplemented`	`RegPolicyTest`; `s22:7-10`
a second operation on a register waits for the first	it raises `UVMFatalError`	`RegTwoOpsTest`, above

What `RegPolicyTest` found:

```
REGAPI add_hdl_path("dut.u_regs") on the block: raised UVMNotImplemented: router_regs
--
REGAPI uvm_mem("mem"): raised UVMNotImplemented: mem uvm_mem not implemented
REGAPI uvm_reg_predictor derives from uvm_object and defines ['__init__']
REGAPI uvm_reg has read, write, predict, get_mirrored_value, get_desired, set_desired,
reset; it has no mirror, update, needs_update, peek, poke, get_reset, set, get
REGAPI uvm_reg_field has predict, get_value, get_reset, get, field_set, reset,
is_volatile, get_access; it has no set
REGAPI uvm_reg_map has set_sequencer, set_adapter, set_predictor, reset, get_registers,
get_reg_by_offset; it has no set_auto_predict, set_check_on_read
REGAPI uvm_reg_block has get_registers, get_reg_by_name; it has no mirror, update, reset
REGAPI in pyuvm: uvm_reg_predictor; not in pyuvm: uvm_reg_hw_reset_seq,
```

```
uvm_reg_bit_bash_seq, uvm_reg_access_seq, uvm_reg_mem_built_in_seq
REGAPI enable_auto_predict: False in s21_uvm_reg_map, False in s24_uvm_reg_includes
```

The back door, in `RegBackdoorTest`:

```
REGAPI CTRL.read(map, path_t.BACKDOOR, check_t.NO_CHECK): raised UVMNotImplemented:
map -- BACKDOOR not implemented
REGAPI CTRL.read(map, path_t.FRONTDOOR, check_t.NO_CHECK) after that: raised
UVMFatalError: CTRL -- read cannot perform an operation while another
is in progress
REGAPI INT_EN.write(0x1, map, path_t.BACKDOOR, check_t.NO_CHECK): raised
UVMNotImplemented: map -- BACKDOOR not implemented
REGAPI ROUTE0.read(map, path_t.USER_FRONTDOOR, check_t.NO_CHECK): raised
UVMNotImplemented: map -- USER_FRONTDOOR not implemented
REGAPI transfers on the bus: 0
```

Auto-predict

A module-level variable, `enable_auto_predict`, turns auto-predict on and off, and it's `False` in `s24_uvm_reg_includes` (`s24:125`). The map's module imports it by name (`s21:8`), so `s21_uvm_reg_map` gets its own copy of the value when pyuvm is imported, and setting the variable in `s24_uvm_reg_includes` afterwards doesn't change the map's copy. Set to `True` in `s21_uvm_reg_map`, it makes the map call `predict()` on its predictor after each operation (`s21:345-347`, `s21:398-400`). But `set_predictor()` accepts only a `uvm_reg_predictor` (`s21:122-130`), and that class has no `predict()`. `RegAutoPredictTest` sets the variable in each module in turn, and in `s21_uvm_reg_map` tries it without a predictor and then with one:

```
REGAPI reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x4', data 0x1, n_bits
32, byte_en True
REGAPI bus, cycle 5: APB W 04 00000001
REGAPI with s24_uvm_reg_includes.enable_auto_predict = True: CTRL.write(0x1, ...):
returned PASS_RESP
REGAPI CTRL's mirror after it: 0x0
REGAPI reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0xc', data 0x1, n_bits
32, byte_en True
REGAPI bus, cycle 8: APB W 0c 00000001
REGAPI with s21_uvm_reg_map.enable_auto_predict = True, no predictor:
INT_EN.write(0x1, ...): raised UVMError: map -- predictor Not set
REGAPI reg2bus() got a uvm_reg_bus_op: kind UVM_WRITE, addr '0x20', data 0x1, n_bits
32, byte_en True
REGAPI bus, cycle 12: APB W 20 00000001
REGAPI the same, with map.set_predictor(uvm_reg_predictor("predictor")):
ROUTE0.write(0x1, ...): raised AttributeError: 'uvm_reg_predictor' object has no
attribute 'predict'
REGAPI mirrors after the three writes: CTRL 0x0, INT_EN 0x0, ROUTE0 0x0; transfers on
the bus: 3
```

Each of the three writes reached the router, and none of them changed a mirror. In Lab 06 your predictor is a `uvm_subscriber` on the APB monitor, and it calls `predict()` itself. It sees every transfer on the bus, including the raw ones that go around the model.

D. Getting the tools outside the course image

If you want to run the labs somewhere other than the course image, this page lists the tools they use, with the version of each that the image has, and says where each tool comes from and what works differently once you're outside the image.

The course image is IIC-OSIC-TOOLS 2026.08, and it's what the workbench you set up in IC101 runs. Every number and every output on this course's pages comes from a run in it. The image already has Verilator, cocotb and pyuvm. From Lab 03 on, the labs also need pyvsc and cocotb-coverage, which aren't in the image, so `make venv` installs them into a Python virtual environment that these pages call the course venv. We checked every fact on this page on November 17, 2026, either in the image or against the source it names. The commands for installing the tools on other systems come from each project's own documentation, except Verilator's, which we copied from DD105-A's page, where they're quoted from Verilator's install guide. We haven't run any of these commands, so 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 the design and any bound assertions into a simulator, through `labs/common/pyuvm.mk` and cocotb's own Verilator makefile
g++	13.3.0	compiles the C++ that Verilator writes
mold	2.30.0	the linker that the image's Verilator was configured to use (`-fuse-ld=mold` in its `verilated.mk`)
ccache	4.9.1	keeps compiled C++ between builds, so that a file that hasn't changed isn't compiled again; the image's `verilated.mk` has `OBJCACHE ?= ccache`
GNU make, bash, GNU coreutils, GNU awk	make 4.3, bash 5.2.21, coreutils 9.4, gawk 5.2.1	run `pyuvm.mk`
Python	3.12.3, as `/usr/bin/python3`, with `/usr/lib/x86_64-linux-gnu/libpython3.12.so.1.0`	the testbenches, which run inside the simulator, and the graders in `labs/common`
cocotb	2.0.1, with find_libpython 0.5.1	starts Python inside the simulator and runs the tests; its Verilator makefile does the build
pyuvm	4.0.1	the UVM classes, in Lab 01 and from Lab 04 on

Tool	In the course image	What the labs use it for
pytest	9.1.1	if it is installed, cocotb uses it to print more detailed `AssertionError` messages
pyvsc	1.1.0, in the course venv	constrained random, from Lab 03 on
cocotb-coverage	2.0, in the course venv	functional coverage, from Lab 03 on
Surfer	0.7.0	the waveform viewer the labs use
`vcd2fst`	from GTKWave, in `/foss/tools/bin`	`make waves` converts the simulator's VCD to FST with it

cocotb 2.0.1 won't build with a Verilator older than 5.036. Line 25 of its Verilator makefile (`cocotb_tools/makefiles/simulators/Makefile.verilator`) sets `VLT_MIN := 5.036`, and with an older version the build stops with an error message that begins `cocotb requires Verilator 5.036 or later`. Every output in the labs comes from 5.050, so any other version, even one that cocotb accepts, may print different numbers.

The graders in `labs/common` use only Python's standard library, but two of their jobs need a Python that has pyuvm or cocotb-coverage installed. The check of the register model imports your model, and pyuvm along with it, and `make cov` merges coverage with cocotb-coverage. The next section says which Python each of them uses.

The course image

The image is Ubuntu 24.04.4 LTS. 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
```

On our machine, `docker image inspect` reports the image under that digest as `linux/amd64`, with 61 layers and 11.7 GB unpacked. DD105-A pins the same digest, and its page gives the download as 3.76 GB. The course's CI (`.github/workflows/labs.yml`) runs every lab in this 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. If you have Docker and the image on your machine, these two commands, run from the top of a clone, build and run Lab 01's pyuvm testbench with the same image and limits that the CI uses:

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

In our run the test passed, and its verdict line read `ok: 0 errors  1.10 s`. `in-image.sh` puts `--skip` in front of your command. That tells the image's entrypoint to run your command instead of starting its desktop, and the entrypoint prints `[INFO] SKIPPING UI STARTUP` when it does so.

How `pyuvm.mk` finds Python

The tests run in a Python interpreter that the simulator embeds. Activating a venv in your shell doesn't change which interpreter that is, because cocotb starts the one named by the environment variable `PYGPI_PYTHON_BIN` (`labs/common/README.md`). So `pyuvm.mk` gives each Python it uses a make variable of its own:

Variable	Default	What uses it
`SYSTEM_PY`	`/usr/bin/python3`	the build: `pyuvm.mk` finds cocotb's makefiles

Variable	Default	What uses it
		with `$(SYSTEM_PY) -m cocotb_tools.config --makefiles` and passes `PYTHON_BIN=$(SYSTEM_PY)` to them. After the build it writes `LIBPYTHON_LOC`, from `$(SYSTEM_PY) -m cocotb_tools.config --libpython`, into `build/sim.env`
`VENV_DIR`	`/foss/designs/.venvs/dd105-b` if `/foss/designs` is writable, otherwise `.venv` at the top of the clone	where `make venv` makes the course venv
`PYTHON_BIN`	the course venv's `python` if `$(VENV_DIR)/bin/python` exists, otherwise `$(SYSTEM_PY)`	the simulator's Python: every run gets `PYGPI_PYTHON_BIN=$(PYTHON_BIN)`. `make help` prints it
`PYTHON`	`python3`	the graders, and `make regcheck`, which builds the register model with it. The register-model check that `make check` and the other graders run before a verdict uses `PYTHON_BIN` instead, and so does `make cov`

`make venv` runs `$(SYSTEM_PY) -m venv --system-site-packages $(VENV_DIR)` and then installs `labs/common/requirements.txt` into the venv. That file pins `pyvsc==1.1.0` and `cocotb-coverage==2.0`. Installing them needs the network, and if pip fails, `make venv` deletes the venv and asks whether the network is up. In a bare container, with the clone mounted at `/w` and no writable `/foss/designs`, it made `/w/.venv`:

```
========================================================================
 VENV  /w/.venv
   from /w/labs/common/requirements.txt
========================================================================
 imports: vsc, cocotb_coverage, cocotb 2.0.1 pyuvm 4.0.1
 created in 9.9 s, 37M
cocotb                          2.0.1
cocotb-coverage                 2.0
pyuvm                           4.0.1
pyvsc                           1.1.0
```

The `--system-site-packages` option lets the venv see a cocotb and a pyuvm that are installed for the system's Python. In the image, though, any venv sees the image's cocotb, pyuvm and pytest, whether it was made with that option or not. The image sets `PYTHONPATH` to a list of folders that includes `/usr/local/lib/python3.12/dist-packages`, where those three packages are, and `pyuvm.mk` passes your `PYTHONPATH` on to the simulator after the lab's own folders. We made a venv without `--system-site-packages` and installed cocotb, pyuvm and the course's two packages into it. `pip` listed the image's pytest 9.1.1 in that venv, although we hadn't installed pytest, and a run's log said `Initialized cocotb v2.0.1 from /usr/local/lib/python3.12/dist-packages/cocotb`, so the run used the image's copy of cocotb even though the venv had its own.

Everything in one venv

By default `pyuvm.mk` expects cocotb and pyuvm to be installed for the system's Python, `/usr/bin/python3`. If you install them with `pip` into a venv instead, the labs still run, as long as you point `SYSTEM_PY` and `PYTHON` at the venv's `python` and put the venv where `VENV_DIR` expects it, so that `PYTHON_BIN` is the

venv's `python` too. We checked this in the course image, from the top of a clone mounted at `/w`, with `PYTHONPATH` unset so that the image's own cocotb and pyuvm were out of reach:

```
unset PYTHONPATH
python3 -m venv .venv
.venv/bin/python -m pip install -q cocotb==2.0.1 pyuvm==4.0.1 -r labs/common/
requirements.txt
cd labs/lab-06-the-register-block
make regcheck SYSTEM_PY=/w/.venv/bin/python PYTHON=/w/.venv/bin/python
make run SYSTEM_PY=/w/.venv/bin/python PYTHON=/w/.venv/bin/python
make check SYSTEM_PY=/w/.venv/bin/python PYTHON=/w/.venv/bin/python
```

`make regcheck` printed `the model builds: 10 registers`, and `make run` printed `ok: 0 errors, 0 assertions 5.78 s`. The log of that run had these lines:

```
Initialized cocotb v2.0.1 from /w/.venv/lib/python3.12/site-packages/cocotb
pytest not found, install it to enable better AssertionError messages
[SB] 1123 packets in: 282 to output 0, 286 to output 1, 179 NOROUTE, 376 FULL; the
highest level a header saw was 16
[SB] beats checked: 1283 on output 0, 1254 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 three `[SB]` lines are the same as the ones in Lab 06, Part 1, and `make check` gave the verdict that part shows for the starter, `VERDICT: FAIL -- your test passes B01_status_write_clears_all too.` In an earlier run of the same commands, which used `/usr/bin/python3` and `pip`'s `--disable-pip-version-check`, the venv took 14.0 s to make and came to 102 MB. The venv has no pytest, because nothing the course installs requires it, which is why cocotb printed `pytest not found` in the log.

Installing on other systems

pyuvm's README is also its PyPI description, and the image's copy of the package includes it. It gives this command, and says that the command also installs cocotb, which pyuvm requires (*untested by us*):

```
% pip install pyuvm
```

cocotb-coverage's README gives this one (*untested by us*):

```
pip install cocotb-coverage
```

Neither cocotb's README nor pyvsc's gives an install command. Each points to its project's documentation instead, at https://docs.cocotb.org/en/stable/ and https://dvkit.org/fvutils/pyvsc/.

The metadata of the four packages, read from the copies in the image and in the venv that `make venv` made, says which Python each one needs and what else it requires:

Package	Python	Other requirements
cocotb 2.0.1	3.6.2 or later	find_libpython
pyuvm 4.0.1	3.6 or later	cocotb 1.6.0 or later, and earlier than 3.0
pyvsc 1.1.0	3.10 or later	dv-solve, pyucis and toposort; on Linux x86-64 and macOS, also pyboolector
cocotb-coverage 2.0	3.11 or later	cocotb 2.0 or later, python-constraint, pyyaml

pyvsc needs pyucis, and pyucis pulls in an MCP server stack (mcp, starlette, uvicorn, httpx2 and more), none of which the course uses (`labs/common/README.md`). Neither install command above names a version. The labs were measured with cocotb 2.0.1, pyuvm 4.0.1, pyvsc 1.1.0 and cocotb-coverage 2.0, which are the versions that the command under Everything in one venv pins.

For Verilator, DD105-A's page (Getting the tools outside the course image) quotes these steps from Verilator 5.050's install guide (`docs/guide/install.rst` at tag `v5.050`), for building it from Git on Ubuntu. We copied them from DD105-A's page, not from the guide itself (*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`, without the `#`. DD105-A's page also quotes the guide's note that ccache and mold are "If present at build, needed for run", which means that if they were installed when you configured Verilator, they have to be installed wherever you use it.

Other differences outside the image

What	In the image	Elsewhere
The machine	every build and run time on these pages was measured with 2 CPUs and 3 GB, and `pyuvm.mk` builds with two g++ jobs at a time (`JOBS=2`)	set `JOBS` to the number of g++ jobs you want to run at once
Shell tools	GNU make 4.3, bash, GNU coreutils	`pyuvm.mk` uses `timeout`, `stdbuf`, `date +%s.%N`, `seq`, `awk`, `du`, `cmp` and bash's `printf %q`; we haven't run it on a system where these work differently, such as macOS
Python	3.12.3, with cocotb, pyuvm and pytest in `/usr/local/lib/python3.12/dist-packages`	3.11 or later, because cocotb-coverage needs it; we have run only 3.12.3
The course venv	in `/foss/designs/.venvs/dd105-b` in the workspace, or `.venv` in the clone	`.venv` in the clone, unless you set `VENV_DIR`
ccache	in use, because the image's `verilated.mk` sets `OBJCACHE ?= ccache`	look for `OBJCACHE` in your Verilator's `verilated.mk`. The build times on these pages as-

What	In the image	Elsewhere
		sume ccache is in use. Lab 01′s pyuvm build, for example, took 1.7 s because six of its seven files came from the cache, and 7.3 s with an empty cache
Waves	the simulator's VCD is converted to FST with `vcd2fst`	`make waves` needs `vcd2fst` on your `PATH`