Lab 04: The same testbench in UVM

In this lab you'll finish our port of the Lab 03 testbench to UVM. We wrote it from our own solution to Lab 03. As shipped, it watches only output 0, as Lab 03's testbench did, and it misses B06 and B13. You'll add a second `out_agent`, so that the testbench watches output 1, and change the scoreboard so that it checks output 1's beats. Then you'll write a test that stalls one output at a time, and make the scoreboard track output 1's level, which the new test needs. After that you'll write a packet class that goes to a disabled route one time in four, and a factory override that puts it in place of `router_pkt` without any edit to the sequence that makes the packets. With those pieces in place the testbench catches both bugs. At the end you'll count the lines in the port and in your Lab 03 testbench, and work out what the extra lines bought you.

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

Before you start

- Finish Lab 02 and Lab 03 on the Course path. Keep your Lab 03 files, because Part 5 counts them.
- Read The UVM skeleton, Sequences, sequencers and drivers, Monitors, analysis ports and scoreboards and The factory and config_db.
- Keep the router's specification open, or the router in one page. Parts 3 and 4 use the specification's Routing section, which the one-page version covers under Packets and routing.
- Copy this table into your notes and fill it in as you go:

	Build time	`make check-all`
The port as shipped (Part 1)		
After Part 2		
After Part 3		
After Part 4		

Time: about 2 h 30 min. On the course's 2 CPUs the first build took us 116 s, and the rebuilds after each part's edits took 30 s and 31 s in Part 2, 105 s and 32 s in Part 3, and 105 s in Part 4.

Part 1: Build the port

Does the port do what the Lab 03 testbench did, and which of this lab's bugs does it miss?

1. Go to the lab folder and start the build, because it's the slowest step in the lab:

```
cd labs/lab-04-the-same-testbench-in-uvm
make build
```

   The banner says a cold build takes 1.5 to 2.5 minutes on 2 CPUs, and ours took 116 s. It ends with:

```
- V e r i l a t i o n   R e p o r t: Verilator 5.050 2026-07-01 rev v5.050
- Verilator: Built from 48.745 MB sources in 415 modules, into 26.168 MB in 2960 C++ files
needing 156.586 MB
- Verilator: Walltime 116.774 s (elab=0.822, cvt=3.289, bld=111.863); cpu 5.279 s on 2
threads; allocated 339.555 MB
```

and then a line saying it built `build/sim` in 116 s, and a ccache line. Our ccache was empty (`Hits: 0 / 38`). Yours already holds the builds of Labs 01 to 03, so the counts on your ccache line will be higher. Write your build time in your table.

2. While it builds, open the files in `tb/`. Apart from the top module, `router_tb_top.sv`, every piece of the Lab 03 testbench is in one of these four files, and most of its classes are now UVM classes:

File	What's in it
`router_ifs.sv`	the three interfaces, with a driver's and a monitor's clocking block each
`router_agents_pkg.sv`	the packet, `router_pkt`, which is Lab 03's `rand_packet` with the same fields and the same constraints (`c_dst`, `c_kind`, `c_len`); an agent for the input, one for the APB port, and one output agent class, `out_agent`; the sequences
`router_env_pkg.sv`	the scoreboard, which is Lab 03's scoreboard and tracker in one class; the coverage, which is Lab 03's `cg_pkt` from our solution; and `router_env`, which builds and connects everything
`router_tests_pkg.sv`	`fill_plan`; the base test, with the steps Lab 03's `env` had (`write_routes`, `send_random`, `check_drops` and the rest) and `close_holes()`; and the tests, which are `router_class_test` (Lab 02's), `router_rand_test` and `router_fill_test` (Lab 03's), and the two that this lab adds, `router_stall_test` and `router_inject_test`

   Your Lab 02 and Lab 03 testbenches watched only output 0, and so does the port. `router_env` makes only one `out_agent`, the scoreboard queues only output 0's beats, and the scoreboard's `level()` takes output 1's level as 0, as Lab 03's tracker did. Find the `TODO (Part n)` comments. There are four in `router_env_pkg.sv` and three in `router_tests_pkg.sv`.

3. Run Lab 02's test, then the default test, then Lab 03's test with more packets:

```
make run UVM_TESTNAME=router_class_test
make run
make run UVM_TESTNAME=router_rand_test PLUSARGS=+N=400
```

   The default is `router_inject_test`, which for now does exactly what `router_rand_test` does. You should see, from the first run:

```
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(220)   @   7185000:
uvm_test_top.env.sb [SB] 100 packets in: 47 to output 0, 53 to output 1, 0 NOROUTE, 0 FULL;
the highest level a header saw was 1
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(222)   @   7185000:
uvm_test_top.env.sb [SB] beats checked: 208 on output 0, 0 on output 1; 1 DROP_CNT reads
checked
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(224)   @   7185000:
```

```
uvm_test_top.env.sb [SB] 185 beats went to output 1, which nothing watches
DD105: TEST PASSED
```

and from the second, Lab 03′s four phases and its `close_holes()`:

```
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)  @  185000:
uvm_test_top [PHASE] 1 of 4: routes 011-, 200 packets; output 0 ready 100%, output 1
ready 100%
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)  @  15295000:
uvm_test_top.env.sb [DROP] DROP_CNT 00000000 as expected
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)  @  15445000:
uvm_test_top [PHASE] 2 of 4: routes -110, 200 packets; output 0 ready 100%, output 1
ready 100%
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)  @  29235000:
uvm_test_top.env.sb [DROP] DROP_CNT 00000000 as expected
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)  @  29385000:
uvm_test_top [PHASE] 3 of 4: routes 01-0, 200 packets; output 0 ready 100%, output 1
ready 100%
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)  @  43715000:
uvm_test_top.env.sb [DROP] DROP_CNT 00000000 as expected
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)  @  43865000:
uvm_test_top [PHASE] 4 of 4: routes 110-, 200 packets; output 0 ready 100%, output 1
ready 100%
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)  @  57015000:
uvm_test_top.env.sb [DROP] DROP_CNT 00000000 as expected
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(243)  @  57185000:
uvm_test_top [HOLES] output 0 stalled and filled to 8 by packets of 3 4 1 beats, then 1
beats for DST 0 and 8 for DST 1
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(196)  @  57655000:
uvm_test_top.env.sb [DROP] DROP_CNT 00000001 as expected
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(220)  @  57655000:
uvm_test_top.env.sb [SB] 805 packets in: 353 to output 0, 451 to output 1, 1 NOROUTE, 0
FULL; the highest level a header saw was 9
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(222)  @  57655000:
uvm_test_top.env.sb [SB] beats checked: 1434 on output 0, 0 on output 1; 5 DROP_CNT reads
checked
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(224)  @  57655000:
uvm_test_top.env.sb [SB] 1799 beats went to output 1, which nothing watches
```

ending in `ok: 0 UVM_ERROR, 0 UVM_FATAL  1.48 s`. The routes are written as in Lab 03, with one character per route: `0` or `1` for the output, `-` for disabled. The `[DROP]` lines come from Lab 03′s `DROP_CNT` check, which the scoreboard now does. It compares every read of `DROP_CNT` with the drops it predicted, and prints a line when they agree. That `make run` took us 1.6 s in all, because nothing was rebuilt. `+UVM_TESTNAME` picks the test when the simulator starts, and the simulator reads the other plusargs then too, so the third run, with `+N=400` packets in each phase, needed no rebuild either. It sent 1605 packets in 3.03 s.

4. Look at the coverage of the runs you've made so far:

```
make cov
```

You should see `covergroup : 58.5% (24/41)`, and `41 covergroup bins, 24 hit at least once`, followed by the 17 empty bins by name. They fall into four groups:

- seven that no test can hit, as you found in Lab 03: `full_x_empty`, `full_x_low`, `full_x_at_8`, `out0_x_at_9`, `out0_x_high`, `out1_x_at_9` and `out1_x_high`;
- `cp_fate.full`, `full_x_at_9` and `full_x_high`: no packet was dropped as FULL;
- `out1_x_low` and `out1_x_at_8`: the scoreboard takes output 1's level as 0;
- `cp_level.high`, `noroute_x_empty`, `noroute_x_low`, `noroute_x_at_8` and `noroute_x_high`: the only NOROUTE drops are `close_holes()`'s packets for DST 1, one in each of the last two runs, and they met a level of 9.

Keep the second and third groups in mind for Part 3.

5. The port should catch what your Lab 02 and Lab 03 testbenches caught. Write down which of B05, B07, B08 and B09 you expect it to catch, then check:

```
make check-all BUG_LIST="B05 B07 B08 B09"
```

You should see `4 of 4 caught.`, with 50, 1, 2 and 50 `UVM_ERROR`s.

6. Now check this lab's two bugs, B06 and B13:

```
make check-all
```

You should see:

```
      ok  the golden design                        0 UVM_ERROR, 0 UVM_FATAL        1.5s
-------------------------------------------------------------------------
  MISSED  tier 2  B06_disabled_route_forwards      0 UVM_ERROR, 0 UVM_FATAL        1.5s
  MISSED  tier 5  B13_admission_checks_other_output 0 UVM_ERROR, 0 UVM_FATAL        1.5s
-------------------------------------------------------------------------
 0 of 2 caught.
 Still hiding from this test:
   B06_disabled_route_forwards
      hides behind: never disabling a route
   B13_admission_checks_other_output
      hides behind: stalling both outputs together, or never letting one fill
```

followed by a note and an error line from `make`, because the verdict is a fail. B06 forwards a packet for a disabled route when its output has room. B13 judges whether an output has room from the level of the other output. Why does this test miss each of them?

Answer: A1, in Answers at the end.

Check: `4 of 4 caught.` for Lab 02's and Lab 03's bugs, and `0 of 2 caught.` for this lab's.

Part 2: Watch the second output

What does it take to watch output 1, when an output agent is a class you can make twice?

1. Read `out_agent`, `out_monitor` and `out_ready_driver` in `router_agents_pkg.sv`. Nothing in them says which output they're on. The monitor and the driver each get their interface (`"vif"`) and their knobs (an `out_cfg`, with `port` and `ready_pct`) from `config_db`. The top module sets the interfaces for `uvm_test_top.env.out0*`

and `uvm_test_top.env.out1*`, and `router_base_test` sets an `out_cfg` for each, so output 1's interface and `out_cfg` are already set for an agent named `out1`.

2. In `router_env`, do the two Part 2 TODOs: make `out[1]` with the name `"out1"`, and connect its monitor's `ap` to `sb.out_export`, as `out[0]` is. Leave the scoreboard alone for now. Before you build, write down what you expect `router_rand_test` to report on the golden router.

3. Rebuild and run:

```
make build
make run UVM_TESTNAME=router_rand_test
```

   The rebuild took us 30 s. The run should fail on the golden router, with 50 errors that start:

```
UVM_ERROR   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(163)   @   225000:
uvm_test_top.env.sb [OUT] output 1: got 80 in cycle 23, and no beat was expected
UVM_ERROR   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(163)   @   265000:
uvm_test_top.env.sb [OUT] output 1: got 24 in cycle 27, and no beat was expected
UVM_ERROR   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(163)   @   275000:
uvm_test_top.env.sb [OUT] output 1: got cf in cycle 28, and no beat was expected
```

   and `DD105: TEST FAILED (50 UVM_ERROR, 0 UVM_FATAL)`. The run stops at 50 because the base test sets UVM's quit count to 50. What is wrong, the router or the testbench?

   Answer: A2, in Answers at the end.

4. Do the Part 2 TODO in `router_scoreboard::write_in`: queue every forwarded beat on `expected[in_port]`. Rebuild (31 s for us) and run `router_rand_test` again. You should see:

```
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(216)   @   57695000:
uvm_test_top.env.sb [SB] 805 packets in: 353 to output 0, 451 to output 1, 1 NOROUTE, 0
FULL; the highest level a header saw was 9
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(218)   @   57695000:
uvm_test_top.env.sb [SB] beats checked: 1434 on output 0, 1799 on output 1; 5 DROP_CNT
reads checked
```

   and `ok`. The 1799 beats that went unwatched in Part 1 are now checked. The number in parentheses after each file name is a line in that file, so yours can differ by a few lines if your edit kept the TODO comment or added lines.

5. Predict whether `make check-all` catches B06 or B13 now, then run it.

   Answer: A3, in Answers at the end.

Check: `router_rand_test` passes with beats checked on both outputs, and `make check-all` still says `0 of 2 caught.`

Part 3: Stall one output at a time

What does B13 need from the stimulus, and does the scoreboard know enough to judge the result?

1. Read the Routing section of the specification, and `level()` and `write_in` in the scoreboard. Then write down the situation in which B13 does something different from the golden router, given that it judges an output's room from the other output's level.

Answer: A4, in Answers at the end.

2. Do the Part 3 TODO in `router_tests_pkg.sv`: in `router_stall_test`, override `set_sinks(k)` so that in phase `k` one output is ready in 5% of cycles and the other in 100%, output 0 stalling in phases 1 and 3 and output 1 in phases 2 and 4. `router_rand_test` calls `set_sinks(k)` at the start of each phase, with `k` from 1 to 4. Set `ocfg[p].ready_pct`. The test made those `out_cfg` objects and kept the handles, and each `out_ready_driver` reads its `ready_pct` again every cycle, so a change takes effect at the next clock edge. `ready_pct` is Lab 03's `sink.stall` grown into a percentage, where 0 is a stall and 100 is a sink that is always ready.

3. Rebuild with `make build`. It took us 105 s, against 30 s and 31 s in Part 2, because the Part 2 edits were inside existing function bodies and this one adds a function to a class. That changes most of the C++ that Verilator generates, as adding members to a class did in Lab 01, Part 5. Before you run it, look at `level()` again. Phase 2 stalls output 1. What will the scoreboard predict for the packets routed there?

4. Run the stall test on the golden router:

```
make run UVM_TESTNAME=router_stall_test
```

   You should see phase 1 end with its `DROP_CNT` check, and phase 2 fail:

```
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(192)   @   15395000:
uvm_test_top.env.sb [DROP] DROP_CNT 00002500 as expected
UVM_INFO  labs/lab-04-the-same-testbench-in-uvm/tb/router_tests_pkg.sv(273)  @  15545000:
uvm_test_top [PHASE] 2 of 4: routes -110, 200 packets; output 0 ready 100%, output 1 ready 5%
UVM_ERROR  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(165)   @   19275000:
uvm_test_top.env.sb [OUT] output 1: got a1 in cycle 1928, expected 4d
UVM_ERROR  labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(165)   @   19325000:
uvm_test_top.env.sb [OUT] output 1: got 94 in cycle 1933, expected db
```

   `FULL_CNT` is bits 15:8 of `DROP_CNT`, so phase 1 dropped 0x25, or 37, packets as FULL, and the scoreboard predicted every one of them. The grader can't use this test yet, because it fails on the golden router. `make check BUG=B13 UVM_TESTNAME=router_stall_test` stops after the golden run:

```
    FAIL  the golden design                          50 UVM_ERROR                    0.8s
        UVM_ERROR [OUT] output 1: got a1 in cycle 1928, expected 4d
        UVM_ERROR [OUT] output 1: got 94 in cycle 1933, expected db
        UVM_ERROR [OUT] output 1: got 68 last in cycle 1936, expected 25
--------------------------------------------------------------------------------
 VERDICT: FAIL -- your test does not pass the GOLDEN design.
```

   Answer: A5, in Answers at the end.

5. Do the Part 3 TODO in `level()`, so it gives output 1's level the way it gives output 0's. Rebuild (32 s for us) and run the stall test again. You should see:

```
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(209)   @   58095000:
uvm_test_top.env.sb [SB] 805 packets in: 190 to output 0, 241 to output 1, 1 NOROUTE, 373
FULL; the highest level a header saw was 16
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(211)   @   58095000:
uvm_test_top.env.sb [SB] beats checked: 753 on output 0, 988 on output 1; 5 DROP_CNT
reads checked
```

and `ok`, with 373 FULL drops where the random test had none. The four phases' `DROP_CNT` reads were `00002500`, `00007700`, `00007e00` and `00005b00`, which are 37, 119, 126 and 91 FULL drops, and the scoreboard predicted each one.

6. Ask for the verdict on B13:

   ```
   make check BUG=B13 UVM_TESTNAME=router_stall_test
   ```

   You should see:

   ```
        ok  the golden design                          0 UVM_ERROR, 0 UVM_FATAL        1.5s
    caught  B13_admission_checks_other_output         50 UVM_ERROR                     0.1s
         UVM_ERROR [OUT] output 0: got e5 in cycle 142, expected 3f
         UVM_ERROR [OUT] output 0: got 90 in cycle 172, expected 73
         UVM_ERROR [OUT] output 0: got 28 in cycle 195, expected f6
  ----------------------------------------------------------------------
   VERDICT: PASS
   ```

   Phase 1 stalls output 0 and routes `DST` 1 and 2 to output 1 (`011-`). The first error is on output 0 in cycle 142, where `e5` came out and the scoreboard expected `3f`. Which of B13's two wrong decisions from step 1 puts a wrong beat on output 0?

   Answer: A6, in Answers at the end.

Check: `router_stall_test` passes on the golden router with 373 FULL drops, and `make check BUG=B13 UVM_TESTNAME=router_stall_test` prints `VERDICT: PASS`.

Part 4: A factory override for disabled routes

How do you make a test send some packets to disabled routes, without editing the sequence that makes them or the driver that sends them?

1. Read `router_rand_seq` in `router_agents_pkg.sv`. It makes each packet with `router_pkt::type_id::create("pkt")`, which asks the factory for a `router_pkt`. If a test has told the factory to substitute a subclass of `router_pkt`, the sequence gets the subclass without knowing it. Then read `router_pkt`'s `c_dst` constraint, which is Lab 03's.

2. Do the first Part 4 TODO: write `router_noroute_pkt`, which extends `router_pkt` and sends one packet in four to a disabled route. Give it its own constraint named `c_dst`. A constraint with the same name as one in the parent class replaces the parent's, and Verilator 5.050 does that as the standard says. Decide which packets go to a disabled route with a non-random bit that `pre_randomize()` sets with `$urandom_range`, not with a `dist`. On this simulator a `dist` draws its value first, and `randomize()` fails when another constraint rules that value out (Lab 03, Part 3, line 3 of the experiments, and Constrained random on Verilator). Every routing table that `random_routes()` writes has at least one disabled route, so the constraint can always be met.

3. Do the second Part 4 TODO: in `router_inject_test`, add a `build_phase` that calls `router_pkt::type_id::set_type_override(router_noroute_pkt::get_type())` and then `super.build_phase(phase)`. Nothing else in the test changes, so it's still the stall test, with different packets.

   The override reaches every packet the factory makes, wherever it's made. `close_holes()` picks the `DST` of its two random packets itself, and switches `c_dst` off for the one it sends to `DST` 1, so it needs both to be plain `router_pkt`s. It makes them with `new()`, in `new_packet()`, where the override can't reach them. When we made them through the factory instead, `router_inject_test` failed on 4 of 5 seeds with `UVM_FATAL [RAND]`.

The packet for `DST` 1 was then a `router_noroute_pkt`, and when `close_holes()` switched `c_dst` off through a `router_pkt` handle, the subclass's `c_dst` stayed on.

4. Write down what you expect the golden run's `[SB]` line to say about NOROUTE drops. Then rebuild (105 s for us) and run the default test:

```
make build
make run
```

With the reference solution you should see:

```
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(209)   @   61465000:
uvm_test_top.env.sb [SB] 805 packets in: 144 to output 0, 190 to output 1, 194 NOROUTE, 277
FULL; the highest level a header saw was 16
UVM_INFO   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(211)   @   61465000:
uvm_test_top.env.sb [SB] beats checked: 648 on output 0, 799 on output 1; 5 DROP_CNT
reads checked
```

and `ok`. You didn't have to change the scoreboard for this. It was written from the specification's Routing table, so it already predicts a NOROUTE drop for a packet to a disabled route; until now, no test had sent one while its output had room. Which packets go to disabled routes depends on how your `pre_randomize()` draws its bit. A version of ours that wrote the constraint differently but drew the bit the same way gave the same counts, so if yours differ, compare your draw with the reference solution's once step 5 says `2 of 2 caught.`

5. Ask for the lab's verdict:

```
make check-all
```

You should see:

```
      ok  the golden design                       0 UVM_ERROR, 0 UVM_FATAL        1.6s
--------------------------------------------------------------------------------
  caught  tier 2  B06_disabled_route_forwards     50 UVM_ERROR                    0.4s
  caught  tier 5  B13_admission_checks_other_output 50 UVM_ERROR                   0.1s
--------------------------------------------------------------------------------
 2 of 2 caught.
```

6. You can also apply a factory override from the command line, without a test class. Apply yours to Lab 03's test that way, on the golden router and then on B06, and time the first run:

```
time                make                run                UVM_TESTNAME=router_rand_test
PLUSARGS=+uvm_set_type_override=router_pkt,router_noroute_pkt
make                            run                             UVM_TESTNAME=router_rand_test
PLUSARGS=+uvm_set_type_override=router_pkt,router_noroute_pkt BUG=B06
```

The first prints `[UVM_CMDLINE_PROC] Applying type override from the command line: +uvm_set_type_override=router_pkt,router_noroute_pkt`, then `194 NOROUTE, 0 FULL` and `ok`, in 1.70 s with no rebuild. The second fails, and its first error is:

```
UVM_ERROR   labs/lab-04-the-same-testbench-in-uvm/tb/router_env_pkg.sv(152)   @   705000:
uvm_test_top.env.sb [OUT] output 0: got c6 in cycle 71, and no beat was expected
```

Without the plusarg, the same test on B06 says `MISSED`.

7. Record both runs from step 6 and look at the first packet that B06 forwards:

```
make                          waves                          UVM_TESTNAME=router_rand_test
PLUSARGS=+uvm_set_type_override=router_pkt,router_noroute_pkt
make                          waves                          UVM_TESTNAME=router_rand_test
PLUSARGS=+uvm_set_type_override=router_pkt,router_noroute_pkt BUG=B06
surfer build/runs/router_rand_test-s1/waves.fst &
surfer build/runs/router_rand_test-s1-B06/waves.fst &
```

Add `valid`, `ready`, `data` and `last` from `in_s` and `out0`, and `route_en_q`, `status_q` and `cnt_noroute_q` from `dut.u_regs`, and look at 650 to 850 ns:

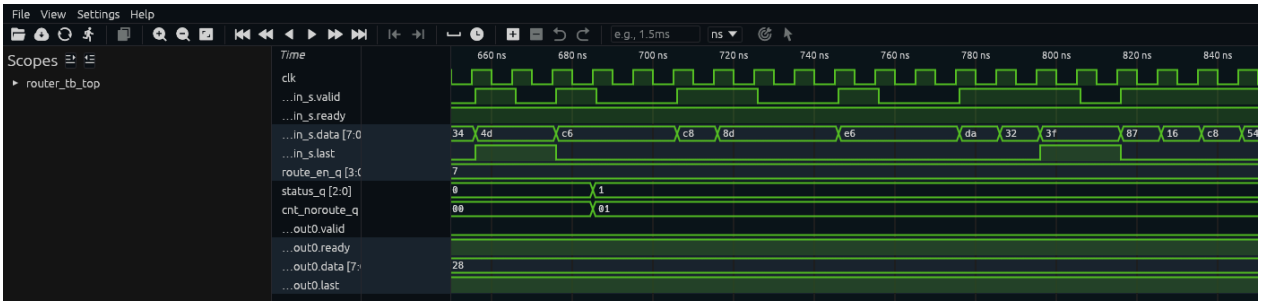


Figure 1: Packets for a disabled route on the golden router, in Surfer

The golden router. The header `c6` (`DST` 3) is accepted at the edge at 685 ns, while `route_en_q` is `7` (routes 0, 1 and 2 enabled). `STATUS.NOROUTE` and `NOROUTE_CNT` change at that edge, and `out0_valid` stays low.

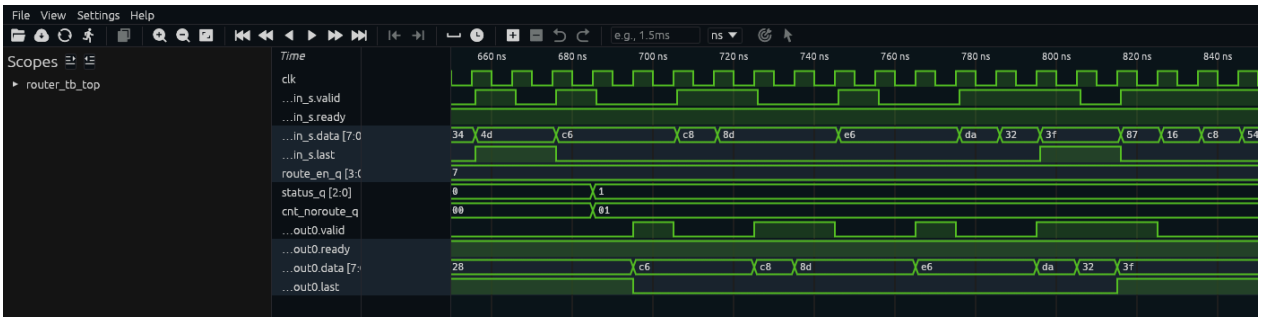


Figure 2: The same packet with B06 on, in Surfer

B06. The router counts the same NOROUTE drop, and then puts `c6 c8 8d e6 da 32 3f` on output 0, starting in the cycle that ends at 705 ns. That's two cycles after the header's cycle, which is when rule O8 says a forwarded beat first appears on the output.

Both pictures have the same input and the same register values, because the seed is the same and this is the first packet for a disabled route in the run, so B06 has changed nothing before it.

8. Run every test on five seeds and read the coverage:

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

You should see `25 ok (25 runs)` and `covergroup : 82.9% (34/41)`. The stall test's FULL drops hit the three bins of Part 1's second group: `cp_fate.full`, `full_x_at_9` and `full_x_high`. The two bins of the third group, `out1_x_low` and `out1_x_at_8`, are hit because `level()` now gives output 1's real level. The five bins of the fourth group are hit too, now that `router_inject_test` sends packets to disabled routes while one output is stalled. The seven bins left empty are the seven that no test can hit, so this is as far as the percentage goes. Read the coverage that `make regress` prints, not `make cov`'s. `make regress` reports only its own runs, while `make cov` merges every run in `build/runs`, and the runs you made before an edit moved the covergroup's lines show each bin a second time.

Check: `2 of 2 caught.` from `make check-all`, and `cp_fate.full` hit.

Part 5: Count what the port cost

Where did the lines go, compared with your Lab 03 testbench, and what did they buy you?

1. Count both testbenches. `FILES` takes the paths of your Lab 03 files, which are in Lab 03's `tb/` with the covergroup, the constraints and the `close_holes()` you wrote:

   ```
   make loc FILES="../lab-03-constraints-and-coverage/tb/*.sv ../lab-03-constraints-and-coverage/tb/*.svh"
   ```

   With the reference solution's edits, the UVM part prints:

   ```
   This lab's UVM testbench:
     file                      lines    code
     tb/router_ifs.sv             85      54
     tb/router_agents_pkg.sv     495     359
     tb/router_env_pkg.sv        305     214
     tb/router_tests_pkg.sv      396     266
     tb/router_tb_top.sv          64      49
     total                      1345     942
   ```

   As shipped it was 1330 lines and 921 code lines, so the reference solution's edits added 21 code lines. With our Lab 03 solution in Lab 03's `tb/`, the `FILES:` part totals 1206 lines and 789 code lines. Write your two totals, for this lab's testbench and for your Lab 03 testbench, in your notes.

2. For each piece of your Lab 03 testbench, find the UVM piece that does its job, and write down whether its code moved across nearly unchanged or changed shape. Then list the UVM pieces that have no Lab 03 counterpart at all.

   Answer: A7, in Answers at the end.

3. Using Parts 2 to 4, what did the extra lines buy you in this lab?

   Answer: A8, in Answers at the end.

Check: your notes have a line count for both testbenches, and your table has a build time and a `make check-all` result for each part.

Finishing

You're done when `make check-all` prints `2 of 2 caught.` with the `router_stall_test`, `router_noroute_pkt` and `router_inject_test` you wrote.

Questions:

1. The first B13 error came in phase 1, where output 1 is always ready and the old `level()` shortcut gave the right answer. Why did you still have to fix `level()` before `make check` would give a verdict on B13?

   Answer: A9, in Answers at the end.

2. In Part 4, step 7, why does the B06 run's `NOROUTE_CNT` agree with the golden run's?

   Answer: A10, in Answers at the end.

Submit: nothing for this lab.

Next on the Course path: the reading on assertions, reactive agents and virtual sequences, then Lab 05: Protocol rules and backpressure.

Answers

A1. Part 1: Build the port

`router_pkt`'s `c_dst` constraint sends random packets only to enabled routes. The only packet the test sends to a disabled route is `close_holes()`'s packet for DST 1. It arrives when output 0 already holds 9 to 16 beats (9 on this seed: the fill of 3, 4 and 1 beats, then a 1-beat packet for DST 0), so there's no room, and B06 drops it just as the golden router does.

B13 changes a packet's fate only when one output holds more than 8 beats and the other 8 or fewer. In the random phases both outputs are always ready, so neither holds more than a beat or two (Lab 02′s test reports a highest level of 1). In `close_holes()` output 0 fills, but the packet for DST 0 arrives at exactly 8, which B13 admits too because output 1 is empty, and the last packet goes to a disabled route, which is a NOROUTE drop either way. No packet for an enabled route met an output holding more than 8 beats, which is also why `cp_fate.full` was never hit.

A2. Part 2: Watch the second output

The testbench. The new monitor reports every beat on output 1, but `write_in` still queues only output 0′s beats (that's the scoreboard's Part 2 TODO), so the scoreboard has nothing to compare them with. When a check fires on the golden router, look for the bug in the check first.

A3. Part 2: Watch the second output

Neither. `make check-all` says `0 of 2 caught.` again. The scoreboard now sees both outputs, but the stimulus hasn't changed, so the router never gets a packet that either bug would treat differently.

A4. Part 3: Stall one output at a time

When a header arrives, one output has to hold more than 8 beats and the other 8 or fewer. A packet for the output with room is then dropped as FULL, because B13 looks at the full one, and a packet for the full output is admitted, because B13 looks at the one with room.

A5. Part 3: Stall one output at a time

`level()` returns 0 for output 1, a shortcut that was safe while output 1 was always ready, as it was in every test before this one. With output 1 ready in only 5% of cycles it fills past 8, and the router drops its next packets as FULL, as the specification says, but the scoreboard still expected them. The packet starting `4d` (`DST` 1, routed to output 1 by `-110`) was dropped, so the next header to come out of output 1, `a1` (`DST` 2, also routed to output 1), was compared with `4d`. Output 0′s level was right all along, which is why phase 1 passed.

A6. Part 3: Stall one output at a time

B13 admits a packet for output 0 while output 0 holds more than 8 beats, because it judges output 0′s room from output 1, which is nearly empty. In the waves of this run, the header `0d` (`DST` 0) arrived in cycle 114, when output 0

held 15 beats and output 1 none. The golden router dropped it as FULL. B13 admitted it and wrote its six beats into a buffer that holds 16 and already held 15, so they overran the buffer and wrote over beats that were still waiting in it. `e5` is the second beat of that packet.

B13 had made the other wrong decision first. From cycle 100 on it dropped packets for output 1 as FULL (the first was `8a`, `DST` 2), because output 0 was above 8. Those packets never came out, and output 1 sent nothing else before cycle 142, so the first beat to disagree with the scoreboard was on output 0.

A7. Part 5: Count what the port cost

Lab 03	Lab 04	What happened
`packet` and `rand_packet`	`router_pkt`	moved: the fields and the constraints `c_dst`, `c_kind` and `c_len` are the same; it gained `` `uvm_object_utils `` and is made through the factory
`generator` and the mailbox to the driver	`router_rand_seq` on the input agent's sequencer	changed shape: `start_item`/`finish_item` instead of a mailbox `put`
`driver`	`src_driver`	moved: `get_next_item`/`item_done` replace the mailbox `get`, and the idle gaps and the clocking-block drives are the same
`sink` and its `stall` knob	`out_ready_driver` and `out_cfg.ready_pct`	moved: the stall grew into a percentage, which the driver reads every cycle
`monitor` and the mailboxes to the scoreboard	`src_monitor`, `out_monitor` and their analysis ports	changed shape: each beat goes to `ap.write()` as it moves, instead of a whole packet into a mailbox
`scoreboard` and `tracker`, each with its own `run()` loop	`write_in`, `write_out`, `write_apb`, `level()`	changed shape: one class, which the monitors call, with no thread; it predicts FULL drops as well, from the tracker's levels
`router_coverage` and `cg_pkt`	`router_coverage` as a subscriber	moved: the same covergroup, sampled from the scoreboard's `fate_ap`
`apb_master` with `write()` and `read()`	`apb_agent`, `apb_rw_seq`, `write_reg`, `read_reg`	grew: a driver, a monitor and a sequence, and the monitor reports every transfer to the scoreboard
`env` and its steps (`write_routes`, `send_random`, `check_drops`, ...)	`router_env`, and the steps as tasks of `router_base_test`	changed shape: `build_phase` and `connect_phase` instead of a constructor that makes the mailboxes
`fill_plan` and `close_holes()`	the same, in `router_tests_pkg.sv`	moved: the same steps in the same order, now sent through the agents' sequences

Lab 03	Lab 04	What happened
`router_cfg` and its plusargs	plusargs read in the tests	moved, except `+SHOW`, which is gone
`$error` and `n_errors`	`` `uvm_error ``	changed: UVM counts the errors and stops the run at 50
`router_base_test` and its three tests	`router_base_test` and five tests	moved: the three tests' `body()`s call the same steps, and two tests are new; `+UVM_TESTNAME` picks one at run time, as it did in Lab 03
nothing	the agent classes, `out_cfg` and `config_db`, the factory registration lines	new

The code that touches the router (the drives, the sampling, the prediction) does the same work in both testbenches. UVM changed how the pieces find each other and pass data between them: through `config_db` and the factory instead of the environment's constructor, and through sequencers and analysis ports instead of mailboxes.

A8. Part 5: Count what the port cost

Watching output 1 took two lines in `router_env` and one in the scoreboard, which Lab 03 had written for output 0 alone. Two lines were enough in `router_env` because the output agent's driver and monitor get from `config_db` everything that differs between the two outputs. Stalling one output took a short function in a new test, which you pick at run time, and it works because the test holds the `out_cfg` handles that the drivers read. The factory override changed every packet that the sequence made, in every phase, without an edit to the sequence or the driver. You could apply it from a test class, or from the command line, where it took 1.70 s with no rebuild; any edit to the testbench cost 30 to 105 s of rebuilding. Lab 03's testbench could have done each of these, but each would have meant editing the classes that already worked.

A9. Finishing

`make check` runs the golden router first and stops if your test fails it (Part 3, step 4), because a test that fails on a correct design tells you nothing about a broken one. The shortcut predicted the wrong fate for output 1's packets in phases 2 and 4, where output 1 stalls, so the scoreboard reported errors on the golden router.

A10. Finishing

B06 changes only whether the packet is written to its output, and the router still counts it as a NOROUTE drop. So at this point the counters agree, and the bug shows as beats on an output that should have stayed idle, which the scoreboard checks.