Lab 06: The register block

In this lab you'll give the Lab 05 testbench a register model. You write the fields of four of the router's registers and the adapter that turns register operations into APB transfers, and run UVM's built-in reset sequence through the model and your adapter. Then you run UVM's bit-bash sequence and connect a predictor, which updates the model from every transfer on the bus. After that you test the 246 addresses that the model can't reach, and make the scoreboard check `irq` against `STATUS` and `INT_EN` in every cycle. The Lab 05 testbench misses all five of this lab's planted bugs (B01, B02, B03, B04 and B14), and with your additions it catches all five by the end of Part 5. Part 6 shows what peek and poke do on this router.

Package: `labs/lab-06-the-register-block/`

Before you start

- Finish Lab 05.
- Read Register models and Regressions and seeds.
- Keep the router specification open at its Register map, APB, INT_EN and Reset sections, and the UVM register layer page beside it, which lists the arguments of a field's `configure()`, the access policies, and the messages the register layer prints.

The package is the Lab 05 testbench with our solution to Lab 05 filled in: the three assertions, the sink driver's answer to each beat, and the body of `router_lab05_vseq`. If your own Lab 05 printed `3 of 3 caught.`, your code did the same job, but keep ours, because this lab's additions are in the same files you edited. Here is what we added for you:

File	What Lab 06 adds
`tb/router_ral_pkg.sv`	the register model and its adapter; four registers' fields and the adapter are yours (Part 2)
`tb/router_ifs.sv`, `tb/router_tb_top.sv`, `tb/router_agents_pkg.sv`	`router_sys_if` and `sys_monitor`, which report `rst_n` and `irq` every cycle
`tb/router_env_pkg.sv`	the environment makes the register model, the adapter and the predictor, and you write the predictor's connections (Part 3); the scoreboard models `STATUS` and `INT_EN`, checks `STATUS` reads, `PSLVERR` and reads of bad addresses, and has an `irq` check for you to write (Part 5)
`tb/router_vseq_pkg.sv`	helpers that go through the model (`wr_reg`, `rd_reg`, `check_reg`, `rd_all_regs`), `router_lab06_vseq` with one `TODO` per part, and `router_backdoor_vseq`
`tb/router_tests_pkg.sv`	`router_test` runs `router_lab06_vseq` first, then everything Lab 05's `router_test` ran; `router_lab06_test` and `router_backdoor_test`
`tb/router_regs.vlt`	the five register signals the back door reaches (Part 6)

Time: about 2 h. You'll build the simulator seven times. On the course's 2 CPUs our first build took 127 s, and the rebuilds after each part's edits took 107 s (Part 2), 108 s and 113 s (Part 3), 35 s (Part 4), and 105 s and 107 s (Part 5), which comes to about 12 minutes of building. Each `make check-all` runs six simulations, each taking about 5 s or less.

Part 1: The Lab 05 testbench against the registers

What does your Lab 05 testbench check about the registers?

1. Go to the lab folder and start the build:

   ```
   cd labs/lab-06-the-register-block
   make build
   ```

   Ours took 127 s. Near the end it prints Verilator's report:

   ```
   - V e r i l a t i o n   R e p o r t: Verilator 5.050 2026-07-01 rev v5.050
   - Verilator: Built from 63.624 MB sources in 449 modules, into 29.574 MB in 3440 C++ files
   needing 213.243 MB
   - Verilator: Walltime 126.831 s (elab=0.857, cvt=3.307, bld=121.710); cpu 5.400 s on 2
   threads; allocated 369.230 MB
   ```

2. Run the graded test, `router_test`, and pull the scoreboard's summary out of the log:

   ```
   make run
   grep -o '\[SB\] [0-9a-z].*' build/runs/router_test-s1/run.log
   ```

   You should see `ok: 0 UVM_ERROR, 0 UVM_FATAL  3.44 s` from `make run`, and from `grep`:

   ```
   [SB] 1123 packets in: 287 to output 0, 355 to output 1, 192 NOROUTE, 289 FULL; the highest
   level a header saw was 16
   [SB] beats checked: 1281 on output 0, 1552 on output 1; 6 DROP_CNT reads checked
   [SB] registers: 6 reads, 0 of STATUS checked; 0 transfers to bad addresses; irq checked in
   0 cycles, wrong in 0
   ```

   The third line is new in this lab. The six reads are `router_inject_test`'s five `DROP_CNT` reads and the one at the end of your Lab 05 sequence. Nothing checks `irq` yet, so the line says it was checked in 0 cycles.

3. At the end of Lab 05 (Part 4, step 6), `router_test` printed `1122 packets in: 240 to output 0, 270 to output 1, 194 NOROUTE, 418 FULL` on the same seed. Your Lab 05 code hasn't changed, and `router_lab06_vseq`, which `router_test` now starts first, does nothing yet. Why are the numbers different?

   Answer: A1, in Answers at the end.

4. `make check-all` runs `router_test` on the golden router and on the five bugs. Run `python3 ../common/bugs.py`, which lists all 15 bugs, find this lab's five in the list, and read their "hides behind" hints. Write down how many you expect the starter to catch, then run it:

   ```
   make check-all
   ```

   You should see:

   ```
         ok  the golden design                        0 UVM_ERROR, 0 UVM_FATAL        3.6s
   ------------------------------------------------------------------------
     MISSED  tier 1  B01_status_write_clears_all      0 UVM_ERROR, 0 UVM_FATAL        3.6s
     MISSED  tier 1  B02_irq_ignores_int_en           0 UVM_ERROR, 0 UVM_FATAL        3.6s
     MISSED  tier 1  B03_route3_readback_swapped      0 UVM_ERROR, 0 UVM_FATAL        3.6s
   ```

```
  MISSED  tier 1  B04_misaligned_address_accepted  0 UVM_ERROR, 0 UVM_FATAL        3.6s
  MISSED  tier 6  B14_enable_resets_to_one        0 UVM_ERROR, 0 UVM_FATAL        3.6s
-------------------------------------------------------------------------
 0 of 5 caught.
```

`make check`, which grades the first bug in the list, says `VERDICT: FAIL -- your test passes B01_status_write_clears_all too.`

B14 makes `CTRL.EN` reset to 1 instead of 0. Why doesn't your testbench see it?

Answer: A2, in Answers at the end.

Check: `0 of 5 caught.`, and `irq checked in 0 cycles`.

Part 2: The register model

What does UVM need to know about each register, and what does the reset sequence do with it?

1. Open `tb/router_ral_pkg.sv` and read `router_reg_ctrl`, `router_reg_int_en` and `router_reg_block::build()`. Each register's `build()` creates its fields and configures each one with a width, a position, an access policy, whether it's volatile (changed by the router on its own), and a reset value. The block's `build()` adds each register to a map at its address. The specification's Register map section gives each field's policy and volatility, and the register layer page explains each argument of `configure()`.
2. Write `build()` for `router_reg_status`, `router_reg_drop_cnt`, `router_reg_level` and `router_reg_route`, as their `TODO`s describe. Pass `is_rand` as 1 only for an `RW` field.
3. Write the adapter's two functions at the bottom of the file. `reg2bus()` fills in an `apb_item` from a register operation, and `bus2reg()` does the reverse for a finished transfer. Give a transfer that ended with `PSLVERR` the status `UVM_NOT_OK`.
4. In `tb/router_vseq_pkg.sv`, write the Part 2 step of `router_lab06_vseq`: create a `uvm_reg_hw_reset_seq`, set its `model` to `regs`, and start it with `start(null, this)`. It reads every register it can check, through the model and so through your adapter, and compares each with its reset value. Run `make lint`, which takes seconds and prints any syntax error with its line; ours printed `0 warnings`. Then run `make build` (107 s for us).
5. Write down which of the ten registers you expect the reset sequence to read. Then run the test and list the ones it did:

```
make run
grep -o 'Verifying reset value of register [^ ]*' build/runs/router_test-s1/run.log
```

You should see `ok` and seven lines:

```
Verifying reset value of register regs.ID
Verifying reset value of register regs.CTRL
Verifying reset value of register regs.INT_EN
Verifying reset value of register regs.ROUTE0
Verifying reset value of register regs.ROUTE1
Verifying reset value of register regs.ROUTE2
Verifying reset value of register regs.ROUTE3
```

Why are `STATUS`, `DROP_CNT` and `LEVEL` missing?

Answer: A3, in Answers at the end.

6. Run `make check-all`. B14's row becomes

```
  caught  tier 6  B14_enable_resets_to_one         1 UVM_ERROR                    3.4s
```

and the other four stay `MISSED`, for `1 of 5 caught.` That error comes from the register model, and this prints it:

```
grep '^UVM_ERROR' build/grade/B14/run.log | grep -o '\[[A-Za-z_]*\] .*'
```

```
[RegModel] Register "regs.CTRL" value read from DUT (0x0000000000000001) does not match
mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000001)
```

`router_test` runs `router_lab06_vseq` before Lab 04's steps. What would the reset sequence report on the golden router if it ran after them?

Answer: A4, in Answers at the end.

Check: B14 is `caught`, and the other four are `MISSED`.

Part 3: The predictor and the bit bash

How does the register model keep its mirror in step with the router, and what does the bit-bash sequence test?

1. Write the first half of the Part 3 step: run a `uvm_reg_bit_bash_seq` the way you ran the reset sequence. The register layer page says what it writes and reads for each bit, and what it skips.
2. Write the second half. Lab 04 and Lab 05 set up the router with raw APB transfers, which go around the model, and the `route()` and `ctrl()` you use here send raw transfers too. After the five writes that the `TODO` lists, read `CTRL` and the four `ROUTEn` with `check_reg()`, which compares each read with the model's mirror.
3. The predictor isn't connected yet, and the model's auto-predict is off (`set_auto_predict(0)` in `router_env::connect_phase`). Predict what `router_test` does on the golden router, then build (108 s for us) and run it:

```
make build
make run
grep '^UVM_ERROR' build/runs/router_test-s1/run.log | grep -o '\[[A-Za-z_]*\] .*'
```

You should see `FAIL: 5 UVM_ERROR  4.33 s` from `make run`, and from `grep`:

```
[RegModel] Register "regs.CTRL" value read from DUT (0x0000000000000001) does not match
mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000001)
[RegModel] Register "regs.ROUTE0" value read from DUT (0x0000000000000001) does not match
mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000003)
[RegModel] Register "regs.ROUTE1" value read from DUT (0x0000000000000003) does not match
mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000003)
[RegModel] Register "regs.ROUTE2" value read from DUT (0x0000000000000002) does not match
mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000003)
[RegModel] Register "regs.ROUTE3" value read from DUT (0x0000000000000003) does not match
mirrored value (0x0000000000000000) (valid bit mask = 0x0000000000000003)
```

The test fails on the golden router. Which side is wrong, and would turning auto-predict on fix it?

Answer: A5, in Answers at the end.

4. Write the Part 3 `TODO` in `router_env::connect_phase`: give the predictor the map and the adapter, connect the APB monitor's analysis port, `apb.mon.ap`, to its `bus_in`, and turn on the map's check-on-read, so the predictor compares every read it sees with the mirror. A predictor is usually connected to the bus monitor, because the monitor sees every transfer on the bus, whoever started it. The scoreboard gets its APB transfers from the same port.

5. Build (113 s for us) and run:

```
make build
make run
grep -o 'Verifying bits in register [^ ]*' build/runs/router_test-s1/run.log
```

   You should see `ok: 0 UVM_ERROR, 0 UVM_FATAL  4.45 s`, and ten lines, from `Verifying bits in register regs.ID` to `Verifying bits in register regs.ROUTE3`. The list includes `STATUS`, although every field of `STATUS` is volatile and the bit bash skips the bits of volatile fields. What does it test in `STATUS`?

   Answer: A6, in Answers at the end.

6. Run `make check-all`. You should see `caught` on B03 (4 `UVM_ERROR`s) and B14 (2), and the other three `MISSED`, for `2 of 5 caught.` B14 has one error more than in Part 2, because the predictor now checks the reset sequence's read of `CTRL` as well. Look at B03's first two errors:

```
grep '^UVM_ERROR' build/grade/B03/run.log | grep -o '\[[A-Za-z_]*\] .*' | head -2
```

```
[RegModel] Register "regs.ROUTE3" value read from DUT (0x0000000000000002) does not match
mirrored value (0x0000000000000001) (valid bit mask = 0x0000000000000003)
[uvm_reg_bit_bash_seq] Writing a 1 in bit #0 of register "regs.ROUTE3" with initial value
'h0000000000000000 yielded 'h0000000000000002 instead of 'h0000000000000001
```

   Where did the 1 written to `EN` come back, and why did your Lab 05 testbench never notice B03?

   Answer: A7, in Answers at the end.

Check: B03 and B14 are `caught`, and B01, B02 and B04 are `MISSED`.

Part 4: The addresses outside the map

Which addresses can the register model never test?

1. The APB address has 8 bits, so there are 256 addresses, and the map has a register at ten of them. A transfer to any of the other 246 completes with `PSLVERR`; a bad read returns 0 and a bad write changes nothing. The model can't send a transfer to an address it has no register for, so you have to reach these with raw transfers. Write the Part 4 step: read every bad address with `rd(a)`, write each with `wr(a, 32'hFFFF_FFFF)`, then read every register with `rd_all_regs()`. `good_addr(a)` is 1 for the ten good addresses. The scoreboard already checks `PSLVERR` on every transfer and the data of every bad read.

2. Write down what the golden router should return for a read of `0x01`, and what `pslverr` should be. Then build (35 s for us), run `make check-all`, and look at the scoreboard's summary in the golden run, whose log `make check-all` keeps in `build/grade/golden/`:

```
make build
make check-all
grep -o '\[SB\] [0-9a-z].*' build/grade/golden/run.log
```

You should see B04 `caught` with `50 UVM_ERROR` (the run stops at 50) and `3 of 5 caught.`, and the last of the three `[SB]` lines should be:

```
[SB] registers: 856 reads, 59 of STATUS checked; 492 transfers to bad addresses; irq checked
in 0 cycles, wrong in 0
```

3. Look at B04's first two errors:

```
grep '^UVM_ERROR' build/grade/B04/run.log | grep -o '\[[A-Za-z_]*\] .*' | head -2
```

```
[APB] a read of 01, completing in cycle 4145: pslverr is 0, but 01 is a bad address
[APB] a read of 01, completing in cycle 4145 returned d1050001; a read of a bad address
returns 0
```

`d1050001` is `ID`'s value. Why did a read of `0x01` return it?

Answer: A8, in Answers at the end.

4. To see these reads in a waveform, record `router_lab06_test`, which runs your sequence alone, on the golden router and on B04. It runs cycle for cycle like the first part of `router_test`, so the read of `01` completes in cycle 4145 there too:

```
make waves UVM_TESTNAME=router_lab06_test
make waves UVM_TESTNAME=router_lab06_test BUG=B04
surfer build/runs/router_lab06_test-s1-B04/waves.fst &
```

Add `clk` from `router_tb_top`, `psel`, `penable`, `pwrite`, `paddr`, `pready`, `pslverr` and `prdata` from `apb`, and `bad_addr` from `dut.u_regs`. Cycle n ends at the rising edge at 10n − 5 ns, so the read of `01` takes cycles 4143 to 4145, from 41415 to 41445 ns. The first error in the B04 run's log is printed at 41445000 ps, the edge at which that read completes.

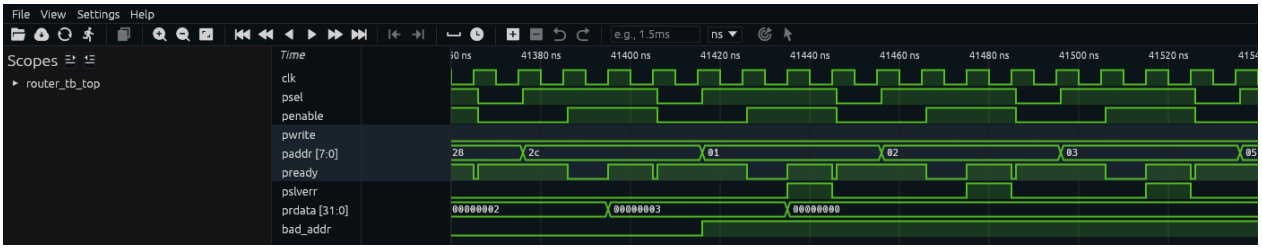


Figure 1: Reads of 2c, 01, 02 and 03 on the golden router, in Surfer

The golden router, 41360 to 41540 ns. `pslverr` is 1 in the last cycle of the reads of `01`, `02` and `03`, and each of those reads returns 0.

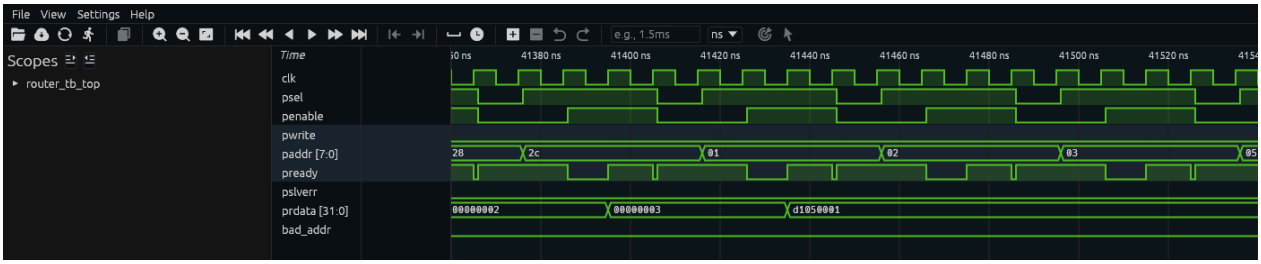


Figure 2: The same reads with B04 switched on, in Surfer

With B04, `bad_addr` stays 0, `pslverr` never rises, and the reads of `01`, `02` and `03` all return `d1050001`.

How many cycles does each read take, and in which of them does the router decide what `prdata` will be?

Answer: A9, in Answers at the end.

Check: B03, B04 and B14 are `caught`, and B01 and B02 are `MISSED`.

Part 5: The interrupt

Does `irq` follow `STATUS` and `INT_EN` in every cycle?

1. The specification's INT_EN section defines `irq` as the OR, over the three bits, of `STATUS[i] & INT_EN[i]`, with no delay. The scoreboard already models both registers cycle by cycle. Each drop, oversize packet, or write to `STATUS` or `INT_EN` waits in `changes` until `advance()` reaches the cycle in which it takes effect, as route writes waited in `pending` in Lab 05, and the function `irq()` combines the two registers. Write the two Part 5 `TODO`s in `router_scoreboard::write_sys()`, which the sys monitor calls every cycle: in a reset cycle `irq` must be 0, and in every other cycle it must equal `irq()`.

2. You haven't written any stimulus for `irq` yet. Write down which of B01 and B02 the check will catch with the stimulus you already have, then build (105 s for us), run `make check-all` and look at B02's error:

   ```
   make build
   make check-all
   grep '^UVM_ERROR' build/grade/B02/run.log | grep -o '\[[A-Za-z_]*\] .*'
   ```

   You should see `4 of 5 caught.`, with B01 still `MISSED` and B02 caught by one `UVM_ERROR`. You chose the wording of your message; in the reference code's wording it is:

   ```
   [IRQ] irq is 1 from cycle 5921, but STATUS = 001 and INT_EN = 000 make it 0
   ```

   In the golden run, the scoreboard's summary (the `grep` in Part 4, step 2) now says `irq checked in 14681 cycles, wrong in 0`. Why does B02 show up in Lab 04's traffic, and why does B01 still hide?

   Answer: A10, in Answers at the end.

3. Write the Part 5 step of `router_lab06_vseq`:

   - disable `DST` 1 with `route(1, 0, 0)` and make both sinks fast with `sink(p, SNK_FAST)`;
   - with `INT_EN` still 0, set all three `STATUS` bits, sending the packets with `burst()`: one 1-beat packet to `DST` 1, which is a NOROUTE drop, and one 9-beat packet to `DST` 0, which is OVERSIZE; then `drain()`, stall output 0 with `stall(0)`, and send ten 1-beat packets to `DST` 0, the last of which is dropped as FULL;
   - read `STATUS`;
   - turn each `INT_EN` bit on by itself, waiting two cycles after each write;

- set `INT_EN` to `'b111`, then clear the `STATUS` bits one at a time by writing `'b001`, `'b100` and `'b010`, and read `STATUS` after each write;
- `drain()`, then `rd_all_regs()`.

Write `INT_EN` and `STATUS` with `wr_reg()` and read them with `rd_reg()`. The model doesn't check a read of `STATUS`, because its fields are volatile, but the scoreboard checks every one.

4. Build (107 s for us) and run `make check-all`. You should see:

```
      ok  the golden design                        0 UVM_ERROR, 0 UVM_FATAL        4.8s
-------------------------------------------------------------------------
  caught  tier 1  B01_status_write_clears_all      3 UVM_ERROR                     4.8s
  caught  tier 1  B02_irq_ignores_int_en           1 UVM_ERROR                     4.7s
  caught  tier 1  B03_route3_readback_swapped      4 UVM_ERROR                     4.7s
  caught  tier 1  B04_misaligned_address_accepted  50 UVM_ERROR                    1.4s
  caught  tier 6  B14_enable_resets_to_one         2 UVM_ERROR                     4.7s
-------------------------------------------------------------------------
 5 of 5 caught.
 Every bug in the list, each one caught by a check that fired.
```

   B01's first two errors, with the reference code, are:

```
[IRQ] irq is 0 from cycle 5969, but STATUS = 110 and INT_EN = 111 make it 1
[STATUS] a read of 08, completing in cycle 5972 returned 00000000; it should return 00000006,
STATUS in cycle 5971
```

   The first write of `'b001` cleared `NOROUTE` and, with B01, `FULL` and `OVERSIZE` too. B02 is still caught by one error, but the error is now `irq is 1 from cycle 5909`, in your own step. Why doesn't Lab 04's traffic catch B02 any more?

   Answer: A11, in Answers at the end.

5. The register model has a mirror of `INT_EN` and `STATUS` as well. Why does the scoreboard keep its own copy of both, instead of reading the mirror?

   Answer: A12, in Answers at the end.

Check: `5 of 5 caught.`, and `irq checked in 14840 cycles, wrong in 0` in the golden run.

Part 6: The back door

What do peek and poke do on this router?

1. A back-door access reads (`peek`) or writes (`poke`) a register through the signal that holds it, with no bus transfer and no simulation time. Read the HDL paths at the end of `router_reg_block::build()`, `tb/router_regs.vlt` (the register layer page says how they fit together), and `router_backdoor_vseq` in `tb/router_vseq_pkg.sv`.

2. The sequence writes `CTRL` with a raw transfer and peeks it twice, then peeks `STATUS` and `DROP_CNT` after a NOROUTE drop. Then it pokes `CTRL`, `INT_EN` and `DROP_CNT` in turn, and peeks each one at once, one rising edge later and two rising edges later. Write down what you expect each peek to return, then run the test and pull out its lines:

```
make run UVM_TESTNAME=router_backdoor_test
sed -n 's/.*@ \([0-9]*\): .*\[BACKDOOR\] /\1 ps  /p' build/runs/router_backdoor_test-s1/
run.log
```

You should see `ok: 0 UVM_ERROR, 0 UVM_FATAL  0.09 s` and:

```
65000 ps  peek CTRL as the write of 1 returns:   1     (UVM_IS_OK)
75000 ps  peek CTRL one rising edge later:       1     (UVM_IS_OK)
125000 ps  peek STATUS after a NOROUTE drop:      1     (UVM_IS_OK)
125000 ps  peek DROP_CNT:                         1     (UVM_IS_OK)
125000 ps  poke CTRL with                         0     (UVM_IS_OK)
125000 ps    peek at once:                        0     (UVM_IS_OK)
135000 ps    peek one rising edge later:          0     (UVM_IS_OK)
145000 ps    peek two rising edges later:         0     (UVM_IS_OK)
145000 ps  poke INT_EN with                       6     (UVM_IS_OK)
145000 ps    peek at once:                        6     (UVM_IS_OK)
155000 ps    peek one rising edge later:          6     (UVM_IS_OK)
165000 ps    peek two rising edges later:         6     (UVM_IS_OK)
165000 ps  poke DROP_CNT with                     505   (UVM_IS_OK)
165000 ps    peek at once:                        505   (UVM_IS_OK)
175000 ps    peek one rising edge later:          505   (UVM_IS_OK)
185000 ps    peek two rising edges later:         505   (UVM_IS_OK)
```

Each poked value is still there two rising edges later. We measured that only on these three registers of this router, so before you rely on a poke in another design, peek the register after a rising edge, as this test does. `CTRL` and `INT_EN` change only on a write or a reset, but the router assigns both `DROP_CNT` counters at every rising edge. Why does the poked `505` survive?

Answer: A13, in Answers at the end.

3. Before the test reads anything through the front door, it pokes each register back to the value it had. `+BACKDOOR_READBACK` adds one front-door read of `DROP_CNT` before that, while `DROP_CNT` still holds the poked `505`:

```
make   run   UVM_TESTNAME=router_backdoor_test   PLUSARGS="+verilator+error+limit+100000
+BACKDOOR_READBACK"
```

The `sed` command now prints one more line,

```
225000 ps  front-door read of DROP_CNT:          505   (UVM_IS_OK)
```

and `make run` prints:

```
 FAIL: 1 UVM_ERROR   0.09 s   log: build/runs/router_backdoor_test-s1/run.log
   UVM_ERROR [DROP_CNT] read 00000505, expected 00000001 (FULL_CNT 0, NOROUTE_CNT 1)
```

The router returned what was poked. Why does the scoreboard call it wrong?

Answer: A14, in Answers at the end.

Check: the sixteen lines above and `ok` from the shipped test, and the one `[DROP_CNT]` error with `+BACKDOOR_READBACK`.

Finishing

You're done when `make check-all` prints `5 of 5 caught.` with the register fields, adapter, predictor connections, `irq` check and `router_lab06_vseq` steps you wrote in `tb/`, and `make seeds SEEDS="1 2 3 4 5"` prints `5 ok (5 runs)`.

Questions:

1. Before this lab, the scoreboard checked one register, `DROP_CNT`. What did the register model add, and what did it leave for the scoreboard and for raw transfers?

   Answer: A15, in Answers at the end.

Submit: nothing for this lab.

Going further (optional)

Run your testbench against every bug in the catalogue:

```
make check-all BUG_LIST="B01 B02 B03 B04 B05 B06 B07 B08 B09 B10 B11 B12 B13 B14 B15"
```

The reference solution catches 14 of the 15, and its output ends with:

```
  caught  tier 6  B14_enable_resets_to_one         2 UVM_ERROR                    4.8s
  MISSED  tier 6  B15_output_valid_survives_reset  0 UVM_ERROR, 0 UVM_FATAL       4.9s
---------------------------------------------------------------------------
 14 of 15 caught.
```

It misses only B15, which hides behind "resetting only at time 0, when every buffer is already empty". You'll meet B15 in Lab 07.

Next on the Course path: the reading on verification plans and sign-off and on writing the verification report, then Lab 07: Capstone.

Answers

A1. Part 1: The Lab 05 testbench against the registers

Making a sequence and starting it both draw from the random state of the thread that does it. In `router_test` that is the test's own thread, which then goes on to pick Lab 04's routing tables and to start the sequences that randomize Lab 04's packets. So all of those random choices come out differently, even though `router_lab06_vseq` does nothing. That is why Lab 05's `router_test` made and started its own sequence only after Lab 04's steps. We also ran this lab's `router_test` with `router_lab06_vseq` left out, and it printed Lab 05's two lines number for number, so nothing else this lab added changes the stimulus, and the checks are the same either way.

A2. Part 1: The Lab 05 testbench against the registers

A wrong reset value shows only between the reset and the first write to the register. A read of `CTRL` in that window would show it, and so would a packet offered then, because `in_ready` stays 0 after reset until software sets `CTRL.EN`. Lab 04's steps write `CTRL` before the first packet and never read it, and the scoreboard wouldn't check a read of `CTRL` anyway, since the only register reads it checks are those of `DROP_CNT` and `STATUS`. B02 hides because nothing checks `irq` yet, as the third `[SB]` line says.

A3. Part 2: The register model

Every field in them is volatile. UVM doesn't compare a volatile field when it reads one, since the design may have changed it since the model last looked, and the reset sequence skips a register in which no field would be compared. If your run printed more than seven lines, one of your fields in those three registers isn't marked volatile.

A4. Part 2: The register model

It would report errors, although the router is right. The sequence resets the model to the reset values and then compares every read with them, but the router is reset only once, at time 0, and Lab 04′s steps and your Lab 05 sequence leave `CTRL.EN` at 1 and the routing table set. So `CTRL` and at least some of the `ROUTEn` would read back what those steps wrote, and the sequence would call each of them wrong. A reset sequence has to run while the registers still hold their reset values, which in this testbench means first.

A5. Part 3: The predictor and the bit bash

The mirror is wrong; the router returned what was written. The mirror changes only when something tells the model about a transfer. Auto-predict updates it after each operation the model itself sends, but `route()` and `ctrl()` sent raw transfers, so with auto-predict on the mirror would still hold the reset values and the five errors would stay. Since the golden router is right, the bug is in the testbench, which has no predictor to update the mirror from every transfer on the bus, whoever started it.

A6. Part 3: The predictor and the bit bash

It tests bits 3 to 31, which belong to no field. The bit bash treats a bit with no field as read-only. The model expects it to read 0, so both writes set it to 1, and after each one the bit bash checks that it still reads 0. That way it tests the reserved bits of every register that has them. In `STATUS`, `DROP_CNT` and `LEVEL`, whose fields are all volatile, the reserved bits are the only ones it tests.

A7. Part 3: The predictor and the bit bash

It came back in bit 1, `PORT`, because B03 swaps `ROUTE3`′s two fields when it's read. The route itself works, because B03 is in the read path only, so every packet still went where `ROUTE3` sent it. Lab 04′s steps and your Lab 05 sequence never read a `ROUTEn`, so no read ever came back with the fields swapped. A test that never reads a register can't see a bug in its read path. The bit bash reads the register back after every write, and each write changes one bit.

A8. Part 4: The addresses outside the map

The registers are 4 bytes apart, so the router finds the register from `paddr[7:2]`, and for `0x01` that's 0, the word where `ID` lives. The router calls an address bad if no register is at its word or if `paddr[1:0]` isn't `00`. B04 drops the second test, so a misaligned address reaches the register below it, for reads and writes alike.

A9. Part 4: The addresses outside the map

Each read takes three cycles: a setup cycle (`psel` 1, `penable` 0), a first access cycle with `pready` 0, and a second with `pready` 1, in which the read completes. The router copies the register into `prdata` at the edge that ends the first access cycle (rule A8), so `prdata` changes in the middle of each read. `pslverr` is 1 only in the cycle in which a bad transfer completes (rule A7). The idle cycle with `psel` 0 between reads comes from the APB driver, which waits for a clock edge before it starts each transfer.

A10. Part 5: The interrupt

Your `router_lab06_vseq` sends no packets yet, so the drop that set `STATUS.NOROUTE` was in Lab 04′s traffic. Every phase of `router_inject_test` disables a route and sends packets to it (in the first phase the routing table is `010-`). `INT_EN` stays 0 all through that traffic, because Lab 04′s steps and your Lab 05 sequence never write it, and the bit bash leaves each of its bits the way it found it. So `irq` must stay 0, but B02, which ignores `INT_EN`, raises it at the first NOROUTE drop. Nothing after that drop clears `STATUS`, so `irq` stays wrong and the check reports it once.

B01 shows only when something writes `STATUS` while a bit that the write shouldn't clear is set. The only writes to `STATUS` so far are the bit bash's, right after reset, when every bit is 0.

A11. Part 5: The interrupt

Your step ends with `INT_EN` at `'b111`, and nothing writes it again, so Lab 04's traffic and your Lab 05 sequence run with every enable on. With every enable on, `|(STATUS & INT_EN)` equals `|STATUS`, so B02 makes no difference to `irq`. That is the situation `bugs.py` means when it says that B02 hides behind "turning on every interrupt enable before looking at irq". Your step catches B02 earlier, after the NOROUTE drop and while `INT_EN` is still 0.

A12. Part 5: The interrupt

The mirror of `STATUS` is out of date most of the time, because the router sets its bits on its own and the model learns their value only when something reads the register. The mirror of `INT_EN` changes at the wrong moment for this check. The predictor updates it when the monitor reports the write, at the edge that ends the cycle in which the write completes, and the router uses the new value from the next cycle. The scoreboard checks `irq` for a cycle at that same edge, so it would get the old or the new `INT_EN` depending on which monitor the simulator woke first. That is the race that your Lab 05 scoreboard avoided by stamping route writes with their cycle. To check `irq` exactly in every cycle, the scoreboard has to apply each change to the registers in the cycle in which the router applies it, so it keeps its own copy of both.

A13. Part 6: The back door

`router_regs.sv` computes each counter's next value from its present one, in the function `bump()`, which clears the counter only on a write to `DROP_CNT` and adds one only on a drop. With neither, the next value is the present value, and after a poke the present value is the poked one.

A14. Part 6: The back door

The scoreboard works out what `DROP_CNT` should hold from the drops it sees on the input and the writes it sees on the bus, and a poke is neither. It has seen one NOROUTE drop and no write to `DROP_CNT`, so as far as it knows `DROP_CNT` holds `00000001`. A testbench that sets registers through the back door has to tell every model that keeps its own copy of them. `peek` is the safer of the two. It shows you `STATUS` and `DROP_CNT` without a bus transfer, and since it doesn't change the router, no model needs to hear about it.

A15. Finishing

It added stimulus made from the register description. The reset sequence read every register it could check, and the bit bash wrote and read back every bit it could check, which caught B14 and B03 without a step written for either bug. It also added access by name (`wr_reg(regs.int_en, ...)`), a mirror that the model compares with every read of a register that doesn't change on its own, and the back door. It left the 246 bad addresses (B04), the volatile registers, and what the registers do to the rest of the router, such as `irq` (B01 and B02), to raw transfers and the scoreboard.