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_O2`), 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: `RouterInjectTest`'s `wait` phases, and `backpressured()` in `RouterLab05Vseq`	`router.wait.x_port_wait`: done on both outputs	done

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.