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.