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` passes `+verilator+seed+1` unless you give it `SEED=`, so every "You should see" in the labs is from seed 1, except in the steps that run several seeds. 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 1 every packet count that `router_test` prints changes, on the same seed, although none of your code has changed. In Lab 07, Part 3 you'll run eleven tests on ten seeds each and be asked whether 110 passing runs mean that the environment is finished. This page explains both, with measurements from a small example of our own in `guide-examples/regressions-and-seeds/`.

What a seed fixes

The example's `random_test` forks two threads in its run phase. Thread A draws four values with `$urandom` at time 0, and thread B draws four more a nanosecond later. On seed 1 the test printed

```
UVM_INFO guide-examples/regressions-and-seeds/seeds_tb.sv(143) @ 1000: uvm_test_top [RANDOM]
thread A: 0bbddfb0 daff8fa1 bfb2d373 4a0f49df
UVM_INFO guide-examples/regressions-and-seeds/seeds_tb.sv(144) @ 1000: uvm_test_top [RANDOM]
thread B: b0011fed 80a529cd c8e18eaf d407ae8b
```

On seed 1 again it printed the same eight values, and on seed 2 it printed eight different ones:

```
UVM_INFO guide-examples/regressions-and-seeds/seeds_tb.sv(143) @ 1000: uvm_test_top [RANDOM]
thread A: 6d5ed04a 79212f85 cac574d1 3eae46dd
UVM_INFO guide-examples/regressions-and-seeds/seeds_tb.sv(144) @ 1000: uvm_test_top [RANDOM]
thread B: b1403e12 ca4b19dc 91ea6ece 9a0aa6d2
```

Given the same simulator, the same plusargs and the same seed, every random choice in every thread comes out the same, so you can repeat a run that failed exactly. `make run SEED=7` reruns seed 7 and puts the run in a folder named after 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 UVM_TESTNAME=count_test SEED=random`, the run went into `build/runs/count_test-s16616597`, so you can run that seed again by giving its number to `SEED=`.

One more random call

By "the same simulator" we mean the same build of the same code, and the runs in this table show why. They are all on seed 1, and in each one a plusarg makes thread A do one thing more before it draws:

Plusarg	What thread A does first	Thread A's values	Thread B's values
none	nothing	`0bbddfb0 daff8fa1 bfb2d373 4a0f49df`	`b0011fed 80a529cd c8e18eaf d407ae8b`
`+EXTRA_DRAW`	one more `$urandom`	`daff8fa1 bfb2d373 4a0f49df 5f2fa507`	`2be22380 36bdaeeb b33e045a 78e0e61e`
`+EXTRA_NEW`	`new()` of a plain class with a `rand` member	`0bbddfb0 daff8fa1 bfb2d373 4a0f49df`	`b0011fed 80a529cd c8e18eaf d407ae8b`
`+EXTRA_SEQ`	`type_id::create()` of an empty UVM sequence	`03c400f3 7d776947 f2c8eda9 d78d5d6e`	`cca11dc6 fee8bd5a b5a1ea57 fd5d592f`

Plusarg	What thread A does first	Thread A's values	Thread B's values
`+EXTRA_START`	`type_id::create()` of that sequence, then `start()`	`e1b55258 15eacca0 ec923ff4 eb3f888a`	`291de991 9e00215f 4e3d822b 6575033c`

The extra draw moved thread A along by one value, as you'd expect, so A's first three values are the original's second to fourth. It also changed all four of thread B's values, although B's code is the same and B draws a nanosecond after A has finished. Making a UVM sequence changed every value in both threads, and starting it as well changed them again. A plain `new()` changed nothing. (The SystemVerilog standard says otherwise on two points. Under the standard each thread has its own generator, so B's values shouldn't have moved, and `new()` seeds the new object's generator from its thread's, so A's should have. The table shows what Verilator 5.050 does, and that is the simulator your runs use.)

On this simulator, then, one more random call or one more sequence anywhere in a test can change the random choices made after it, in other threads as well as in its own.

You'll be asked about this in Lab 06, Part 1. It is also why Lab 07 starts its new sequence, `router_lab07_vseq`, last in `router_test`. When we left that sequence out, Lab 07's `router_test` printed Lab 06's `[SB]` lines number for number.

Three habits follow from this:

- Compare two runs on the same seed only when the code is the same. A seed names a run of one build; on a different build the same seed is a different run, and its values can all change.
- When a seed fails, write down the commit it failed on as well as the seed. After you change the testbench, that seed is a different run, and the failure may move or disappear without the bug being fixed.
- If a failure matters, write a directed step in a sequence that reaches the bug whatever the seed. Labs 06 and 07 have you write one for each bug you need to reach.

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 stopped on a `UVM_FATAL` shows as BROKEN and isn't counted as a pass.

The example's other test, `count_test`, drives a 4-bit counter that counts up on `inc`, goes back to 0 on `clr` and should stop at 15. For 300 cycles the test sets `inc` at random in 60% of cycles and `clr` in 15%, and it checks the count against a model in every cycle. `make seeds UVM_TESTNAME=count_test SEEDS="1 2 3 4 5 6 7 8 9 10"` printed

```
      ok  count_test                  seed 1      0 UVM_ERROR, 0 UVM_FATAL        0.0s
      ok  count_test                  seed 2      0 UVM_ERROR, 0 UVM_FATAL        0.0s
      ok  count_test                  seed 3      0 UVM_ERROR, 0 UVM_FATAL        0.0s
      ok  count_test                  seed 4      0 UVM_ERROR, 0 UVM_FATAL        0.0s
      ok  count_test                  seed 5      0 UVM_ERROR, 0 UVM_FATAL        0.0s
      ok  count_test                  seed 6      0 UVM_ERROR, 0 UVM_FATAL        0.0s
      ok  count_test                  seed 7      0 UVM_ERROR, 0 UVM_FATAL        0.0s
      ok  count_test                  seed 8      0 UVM_ERROR, 0 UVM_FATAL        0.0s
      ok  count_test                  seed 9      0 UVM_ERROR, 0 UVM_FATAL        0.0s
      ok  count_test                  seed 10     0 UVM_ERROR, 0 UVM_FATAL        0.0s
----------------------------------------------------------------------------
  10 ok  (10 runs)
```

and then the coverage of all ten runs together, which included

```
verilator_coverage: covergroup : 100.0% (4/4)
```

All ten passed, and the merged coverage hit all four bins: the count in 0 to 7, in 8 to 14 and at 15, and an increment while it was at 15. That 100% is the union of the ten runs. The `[COUNT]` line that each run prints at the end of the test says how often its own counter was at 15, and how often it was incremented there:

Seed	1	2	3	4	5	6	7	8	9	10
Cycles at 15	0	6	20	0	0	0	0	4	3	24
Incremented at 15	0	2	13	0	0	0	0	2	2	19
`BUG=wrap`	MISSED	caught	caught	MISSED	MISSED	MISSED	MISSED	caught	caught	caught

The last row is the same test with the planted bug `wrap` switched on, which makes the counter wrap from 15 to 0. We ran it one seed at a time, with `make run SEED=n BUG=wrap`, because `make seeds` doesn't pass `BUG` on to its runs. The test caught the wrap on exactly the five seeds that incremented the counter at 15, each time with `[COUNT] count is 0, expected 15`, and passed on the other five. To reach 15 the counter needs fifteen increments with no clear between them, and with a clear 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 five unlucky ones.

Each extra seed in a regression is another chance at a corner that the stimulus reaches only by luck. When a corner matters, it's better not to leave it to luck at all, which is why Lab 07, Part 6 has you write a sequence that drives both drop counters to 255 on purpose.

Corners no seed reaches

More seeds help only with corners the stimulus can reach. In Lab 07, Part 3, the starter's regression of eleven tests on ten seeds printed

```
 110 ok  (110 runs)
verilator_coverage: covergroup : 81.7% (49/60)
```

and listed eleven coverage bins that no run hit. Four of them 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. Nothing in it sets out to drive a counter to 255 either, and every phase of `router_inject_test` clears the counters. The 110 passes mean that no check fired in 110 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. 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. A situation that no test produces then has nothing in its row to make it happen, and you can see that in the plan before you trust a regression's passes.

All 110 runs used the one simulator that `make build` had made. On 2 CPUs the whole regression took 3 min 57 s, about two seconds a run, and a build takes about two minutes. That is why the course chooses the test, the seed and the bug with plusargs when you run, rather than with anything that needs a rebuild.

Check yourself

1. You ran `make run SEED=4` yesterday and it failed. Today, after adding one `$urandom_range` 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 `covergroup : 100.0% (4/4)`. Did any one seed hit all four bins, and does it matter?

   Answer: A2, in Answers at the end.

3. Lab 07's starter passed 110 runs and left `reset_with_beats` empty. Would 1,000 seeds have filled it?

   Answer: A3, 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. On Verilator the run-time option is `+verilator+seed+N`.
- 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.
- SystemVerilog Command Line Arguments: `$test$plusargs` and `$value$plusargs`, which every test in the course uses to read its knobs.
- 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. On Verilator 5.050 one extra random call changes the values drawn after it, so seed 4 is now a different run with different stimulus, and it may simply no longer reach the bug. Run yesterday's commit on seed 4 to confirm the failure, then turn it into a directed step that fails 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. The other five seeds matter more. They never got the counter to 15, and on those five the wrap bug passed. Merged coverage tells you that some run hit each bin, not that every run did.

A3. 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 a test picks, but which choices the test makes at all is fixed by its code, so filling the bin takes a new sequence, which you write in Lab 07, Part 5.