Directed and random tests

This page answers one question: *should I write tests by hand, or generate them?*

You need both, because neither is enough on its own:

> Directed tests find what you thought of. Random tests find what you did not think of, as long as it lies inside the distribution you chose. Neither kind finds what neither covers, and that third category is where the expensive bugs live.

DD104 ships the tools to measure all three of those, so most of this page is measurements.

Directed tests

A directed test drives stimulus that you chose and checks the result against an answer you worked out. Nine of the eleven tests in the reference suite are directed, and none of them is clever: each one is a short program, a run and a comparison. What makes them work is the choice of stimulus.

Directed testing has one way of failing, which is that the test shares your blind spot. Because you choose the stimulus, an input class you never thought of is missing twice over: from the tests, and from your own idea of what needs testing.

The history of `B07`, told on Where bugs hide, is an example. The first `LOAD/STORE` test written for this core used `LOAD r1, r1`, so it passed on a core that takes the address from the wrong register field, and nobody found out until the bug was injected on purpose and the suite stayed green. That page describes the habit that prevents this. What matters here is that the fix was another directed test, written on purpose once somebody had named the input class. You can't generate your way out of a blind spot you haven't named.

Random tests

`R10_random_programs` generates eight programs of twenty instructions each, runs each one on the chip, and compares the final register file with the golden model. The whole test is about forty lines, generator included, and you couldn't work out a single one of its expected answers by hand.

It does find bugs. When we inject the bugs from the catalogue one at a time, it catches these three:

Bug	Random programs that disagreed with the model
`B01_sub_operands_swapped`	4 of 8
`B10_ori_adds_instead_of_ors`	5 of 8
`B05_branch_offset_zero_extended`	7 of 8

That is three of the thirteen bugs, found by stimulus nobody wrote and checked against answers nobody had to work out by hand, which is a good return on forty lines. Random testing doesn't replace thinking, and it isn't worthless either. The rest of this page measures what it didn't do.

What the generator reaches

The generator is designed around **windows**. A window is the input condition a bug needs before it can be seen at all. `SUB rX, rY` with `rX != rY` is the window for a swapped subtract, because outside it a backwards `SUB` still gives the right answer.

`gap.py` runs the same generator that `R10` uses (it imports it from the same file, so there is only one copy) through `golden.py`, and counts how often each of nine windows is entered. No simulator is involved, so it takes about a second. You'll run it as `make gap` in Lab 04, Part 6:

```
the eight seeds R10 actually runs
------------------------------------------------------------------------
window                                                entered   programs
  B01  SUB, operands differ by neither 0 nor 128            6      5 of 8
  B02  NAND with two different operands                    11      5 of 8
  B06  shift by 2 or 3                                     13      8 of 8
 *B05  branch with a negative displacement                  0      0 of 8
  B09  two branches off one comparison                      7      5 of 8
 *B07  any LOAD or STORE                                    0      0 of 8
  B03  non-zero result whose low nibble is zero             2      2 of 8
  B10  ORI that sets a bit r0 already had                  15      5 of 8
  B11  ORI executed while r0 is non-zero                   19      6 of 8
------------------------------------------------------------------------
8 programs, 160 instructions written, 2 of 9 windows never entered
```

Every number in that table is exact, because the generator is seeded and so writes the same 160 instructions on every machine.

The table shows three different situations, and the next three sections take them in turn.

Windows it enters constantly

`B06`'s window, a shift by 2 or 3, is entered thirteen times, in all 8 programs, and `B10`'s is entered fifteen times. Random stimulus covers these windows without anyone having thought about shift amounts at all. That is the case for random testing, and it is a strong one: it's cheap, and it doesn't depend on what you failed to imagine.

Windows it enters rarely, and one it enters and still misses

`B03`'s window is a flag-setting result that is non-zero but has a zero low nibble. The eight seeds enter it twice in 160 instructions. Over a thousand programs it is entered 338 times in 20,000 instructions, which is about 1.7 % of them.

Here is the reference suite run against `B03`:

```
      ok  the golden core                   11 tests, all passed        19.2s
  caught  B03_zero_flag_low_nibble          11 tests, 1 FAILED          16.2s
        R07_zero_and_negative_flags
            AssertionError: after r1 = 16 (0b00010000) the Z flag is 1.
```

One test of the eleven catches it, and it's `R07`, not the random one, even though the random test entered the window twice.

The reason it missed is the most useful point on this page. `R10` checks `model.snapshot()[0] != sp.regs()`, which compares the registers and nothing else. The flags are in the snapshot, but that comparison ignores them. A wrong `Z` can only reach a register if a later branch uses it and changes the control flow, and none of those eight programs did that.

> Coverage of your stimulus and coverage of your checks are two different questions, and you have to ask both. A generator can walk into a window and back out again, and if nothing in the checker can see the difference, the run exercised the design without testing it.

Windows it can never enter

The two rows marked `*` are still at zero after a thousand programs and twenty thousand instructions. That isn't bad luck, and more seeds won't change it, because the generator never writes those instructions at all. Its docstring says why:

```
def random_program(seed, n=20):
    """Arithmetic, shifts, ORI, and forward branches that cannot loop.

    No LOAD or STORE: a random register value is a random address, and half of
    them are outside the 64-byte RAM. No backward branches: a random loop is an
    infinite loop. Both of those are worth testing — with stimulus you designed
    rather than stimulus you rolled."""
```

Both exclusions are good engineering, and nobody made a mistake here. A random 8-bit register value is an address outside the 64-byte RAM more often than not, and a randomly placed backward branch is an infinite loop that turns your regression into a timeout.

Look at what those constraints cost, though. `B07` lives in `LOAD`, and `B05` lives in backward branches. The generator's necessary and sensible exclusions mark out exactly the directed tests you still have to write, and the reference suite's `R06` and `R04` are those two tests.

> A constrained random generator's constraints describe what it will never find, so write them down and treat them as a to-do list.

Why `R10` catches `B05`

`R10` does fail on `B05`, even though the table says the generator never emits a backward branch. Both of those are true, and it's worth two minutes to see why.

The harness's `load()` appends a parking loop, a single `JMP -1` (byte `0xF1`), after every program, because this machine has no `HALT` instruction and a program has to stop itself. On a core that zero-extends branch displacements instead of sign-extending them, that byte decodes as `JMP +15`, which doesn't loop at all.

So the program runs off its own end and through the filler, its 8-bit PC wraps around the 64-byte RAM, and it executes a second time on top of its own results. Here is the result on a three-instruction program:

	`r1`
Model	10
Chip	20

The random test catches `B05` because the harness added one backward branch to every program and the bug broke it. Nothing the generator chose had anything to do with the catch. That leaves you with two things to keep in mind:

1. This is real coverage and a real catch, so take the failure seriously; it is telling you the truth about the core.
2. You can't report it as evidence that your random testing covers backward branches, and you can't rely on it. If someone changes the parking idiom, the catch disappears and the bug stays.

Being able to say why a test failed, and not only that it failed, is what separates a verification result from a lucky one.

The gap

Putting the three together gives this:

	Finds	Misses	Costs
Directed	Exactly what you named	Everything you did not name, including whatever your model of the design gets wrong	One test per input class, written by someone who understands the machine
Random	Things nobody named, inside the distribution you chose	Anything the constraints exclude; anything your checker cannot observe	A generator, a model, and honesty about what it did not reach
The gap		What neither covers, which is where the bugs that reach silicon live	A written list

The only thing that closes the gap is a list of questions that you maintain by hand. That list is what functional coverage means. It isn't a tool, and Coverage as a question shows what one looks like. For this machine, the list starts like this:

The question	Why it is on the list
Have I executed every opcode, including the three unimplemented ones?	
For every instruction, `rA == rB` and `rA != rB`?	Four bugs live here
All four shift amounts, both directions?	Shifting by 1 is what everybody writes
All sixteen branch displacements, taken and not taken?	Half of the displacements are negative, and a bug can hide in that half
Two branches off one comparison?	Flags are supposed to survive a branch
An `ORI` whose immediate overlaps bits `r0` already has?	`OR` and `ADD` agree until the bits overlap
A second `ORI`, once `r0` is no longer zero?	After reset, several wrong machines look right
`done` counted, `halt` counted, cycle counts asserted?	Two bugs disturb nothing else
`clk_per_bit` swept, not sampled?	One bug does not exist at the default

You can answer every line today with a `for` loop and a set, without any tool. Writing the list is where the engineering goes; the counting is clerical work.

Making the generator better

If you want to improve `R10` rather than replace it, there are three separate improvements you can make, and each one fixes a different problem:

1. Constrain instead of excluding. `LOAD` and `STORE` are excluded because a random address is usually out of range. If you force the pointer register into `0..63` before the access, they can come back. That is what "constrained random" means, and it reaches the window where `B07` lives. Backward branches come back the same way, as a loop with a counter that the generator sets up.

2. Widen the checker rather than the stimulus. Compare the whole snapshot (registers, PC, `N` and `Z`) instead of only the register tuple. That is a one-line change, from `model.snapshot()[0] != sp.regs()` to `model.snapshot() != sp.snapshot()`, and it makes every flag window the generator already enters count.
3. Report what you reached. `gap.py` is the smallest possible coverage model: a list of named windows, a counter for each one, and a `*` on the ones that stayed cold. Without something like it, "we ran ten thousand random programs" tells people more about your CPU budget than about your design.

Do (2) first, because it's the cheapest, and it turns stimulus you already have into checking you don't have yet.

> After every random run, ask what it could not have reached, and write the answer next to the seed count. "Ten thousand programs, none of which contained a memory access" is a useful sentence. "Ten thousand programs" on its own isn't a result, for the same reason that a green tick is not a result.

Check yourself

1. A random generator never emits `STORE`. Which bugs in the catalogue can it never reach directly, and what directed test would you write?
2. `R10` entered `B03`'s window twice and still passed. What one change to its checker fixes that?

Next: Lab 04: Your first bug hunt.