The golden model

This page answers: *the input space is too big to walk. What do I compare against instead?*

In Lab 03 you drove all 393,216 legal vectors through SimProc's ALU in about twenty seconds, and afterwards there was no unexplored corner left for a bug. That is the best position verification ever puts you in, and you should take it whenever a block is small enough.

Now count the processor: 554 bits of architectural state, and a program is 64 bytes, so the space of programs alone is 256^{64}, and testing all of it is out of the question.

So instead of checking every input, you check every instruction as it runs, by comparing the design against a second opinion.

What a golden model is

A **golden model**, or reference model, is a second implementation of the same specification, written in a language where you can see what it does. You run the design and the model on the same stimulus and compare them after every step. When they disagree, one of them is wrong.

DD104's golden model is `labs/common/golden.py`, about 120 lines of plain Python with no imports and no simulator behind it. You ran it on its own in Lab 02, Part 4, and it printed this:

```
16 instructions in memory, 28 executed, 109 clock cycles
registers: r0=32 r1=15 r2=1 r3=32
mem[0x20] = 15   (all 28 instructions executed)
```

That's a whole SimProc program running with no Verilog involved. It runs in no measurable time, and because it's plain Python you can read it and put a `print` in it.

The interface is deliberately small:

```
def snapshot(self):
    """(regs, pc, N, Z) — the same shape `SimProc.snapshot()` returns, so
    the two can be compared with `==` and nothing else."""
    return (tuple(self.regs), self.pc, self.n, self.z)
```

Because both sides return the same tuple, comparing them takes a single `==`. You don't check field by field or keep a list of things to remember to look at, so the comparison also checks the registers you weren't thinking about.

> The rule that `spec()` followed in Lab 03 applies here too: write the model from the ISA, not from the Verilog. A model written by reading the RTL agrees with the design about everything, including its bugs, so it passes forever and proves nothing. Lab 03 called this the most common way a verification effort wastes six months, and it's easy to do by accident, so it's worth writing at the top of the file.
>
> `golden.py` is written from the instruction set. Where the ISA was ambiguous, somebody had to make a decision, and the end of this page describes one.

Using it in four lines

```
await sp.load(prog)
model = Golden(prog + [PARK])
steps, problem = await trace(sp, 29, model=model)
assert problem is None, f"{problem}\n\n{listing(prog)}"
```

`trace()` single-steps the real design, steps the model alongside it, and compares the full snapshot after every instruction. The program here is the one in `test_example.t06`: thirty instructions, of which 29 execute, because a taken branch skips one. So those four lines check all four registers, the PC and both flags, twenty-nine times over, and that includes everything you didn't think to check.

`trace()` doesn't look at memory, so the shipped test follows those four lines with one call to `compare_memory(model, sp)`, which compares all 64 bytes. When both pass, this one line is all it reports:

```
29 instructions, every register / PC / flag / byte of memory agreed with golden.py
```

When it fails, you get a message like this one, from `test_example.t06` run on the core whose `SUB` computes `B - A`:

```
AssertionError: instruction 19, at address 19 (0x66 = SUB   r1, r2), left the machine in a
different state:
        model: r=(48, 195, 29, 48) PC=20 N=1 Z=0
          DUT: r=(48, 61, 29, 48) PC=20 N=0 Z=0

the program:
  00: 06   SUB   r0, r0
  01: C7   ORI   24
  ...
  12: 64   ADD   r1, r2
  13: 66   SUB   r1, r2
```

You didn't write any of that message yourself, and every part of it is useful:

- which instruction failed: number 19, counting from 0, rather than "somewhere in the program";
- where that instruction lives: address 19;
- what it was: `0x66`, already disassembled to `SUB r1, r2`, so you don't have to decode the hex;
- the state of the whole machine on both sides. Here $195 + 61 = 256$, which is the signature of a negated result, so you know the operands went in backwards before you open anything;
- the flags as well: `N=1` against `N=0`, which the same comparison caught without any extra assertion.

What a model gives you that a hand-written expectation can't

There's nothing wrong with directed assertions. `assert sp.reg(1) == 5` is fine, and you should write plenty of them. A model, though, gives you three things that a hand-written expected value can't.

1. It checks the registers you were not looking at

A directed test asserts on the register that the instruction names, and says nothing about the other three. "The answer is correct" and "nothing else moved" are different claims, and a directed test only makes the first one.

On the same broken core, the directed test in the same file reports:

```
AssertionError: r1 is 199, expected 247.
```

That message is true and useful, and it points you at `r1`, where the answer was supposed to go. But one bug in the catalogue writes the right value into the wrong register, and against that bug a directed assertion on the destination reports a symptom in a register that was never the problem. The model comparison catches it immediately, because a bystander register that changed fails `==` just as loudly as a destination register that didn't.

Check the state that should not have changed. It costs nothing when you compare snapshots, and it is almost impossible to remember when you don't.

2. It names the first disagreement, not the last

Once a processor's state has diverged, everything downstream is wrong too. A test that checks state at the end of a program hands you the symptom (a wrong byte, forty instructions later, in a register nobody cared about) and leaves you to walk it back by hand.

`trace()` stops at instruction 19 of 29, while the machine's state is still only one instruction away from correct. Everything before that instruction is known to be good, so you start debugging at instruction 19 instead of walking back from the end.

3. It lets you run stimulus you can't work out by hand

You can't write down the expected final register file for thirty random instructions, but the model can, and that opens up a whole category of test. The limits of that kind of test are the subject of Directed and random tests.

What a model cannot see

A model checks only what it models, so write down what it leaves out next to the model itself, rather than discovering it at midnight. `golden.py` models architectural state: four registers, the PC, two flags and 64 bytes of RAM. It has no concept of:

`done`	The one-cycle retire pulse
`halt`	The pin that says the core is in `IDLE`
Cycle counts	It publishes them, but nothing compares them by default
The UART	Baud rate, framing, the debug protocol
`X`	A model has no undefined value

A core that computes every answer correctly but gets its handshake wrong passes a model comparison, because a correct answer and a correct handshake are different kinds of property. Two bugs in the catalogue are exactly that kind, and finding them is the work of Lab 05. Running more vectors won't help with them; you need a different kind of check.

When the model is the one that's wrong

Everyone gets caught out by a wrong model the first time, so here is a real example from this repository.

The unimplemented opcodes, `0xC`, `0xD` and `0xE`, take four cycles each, and `golden.py` used to say they took three. What they do is easy to describe: the PC advances, nothing else changes, and `done` never pulses. The model gave them three cycles, the same as every branch and `STORE`, which is what the RTL looks as though it should do. This is what we measured on four filler bytes:

Clock cycles	16, four for each byte
`done` pulses	0
Cycles with `halt` high	4

These opcodes match no clause in `CYCLE_3`, so the FSM takes its default exit to `IDLE`, spends a cycle there, and is sent back to `CYCLE_1` by `run`, which is still asserted. That adds up to four cycles, one of them with `halt` high in the middle of a running program.

There are three lessons in that:

1. The model was the one that was wrong. It's a second implementation written by a person, not an oracle. "The DUT disagrees with the model" is a statement about the pair of them, so the next question is always which one is wrong. Answer it from the specification or, when the specification is silent, by measuring the hardware and writing down what it does.
2. The model was still right about what it was mostly used for. Its architectural state was correct all along, and the number it got wrong was a timing figure that it also happened to publish. A model can be trustworthy in one dimension and wrong in another, so know exactly which dimensions yours covers.
3. The mistake was found by measuring a case that nobody tests. `0x0C` is filler, and no program runs it on purpose, so it's exactly the kind of "obviously nothing happens here" behaviour that never gets an assertion. That is the subject of Where bugs hide, applied here to the testbench instead of the design.

When the specification has not decided

SimProc's `ORI` has no register field (all five spare bits are the immediate), so the register it accumulates into is implicit in the hardware. For a year the RTL carried a comment saying it might be the other register.

We built and simulated four variants to settle it, and found that `r0` and `r1` are exactly isomorphic in this machine, meaning that nothing treats them differently: a register index never leaves the instruction word, `r0` is not hardwired to zero, and no flag, addressing mode, reset value or debug command treats either one specially. The same program with `r0` and `r1` swapped produces the same memory and mirrored registers.

Since the hardware can't tell the two registers apart, no measurement can break the tie. Somebody has to decide, write the decision down and make the model match, and from then on the decision is part of the specification rather than an accident of the RTL. DD104 specifies `r0`.

Which register you pick is a matter of taste, but the two lines that implement the decision must change together. If you apply it to only one of them, you get "OR the immediate into one register, write the answer to a different one", which is an instruction nobody specified. That is one of the three tier-5 bugs, the hardest in the catalogue to think of. It isn't hard to catch, though: after a reset both candidate registers hold zero, so the first `ORI` of a program puts the right value into the machine, but in the wrong register, and any test that checks `r0` afterwards finds zero there. Lab 02′s one-`ORI` test catches it, and so do seven of the eleven reference tests in Lab 06.

When the model and the design disagree

1. Read the failing instruction rather than the final state; `trace()` has already told you which instruction it is.
2. Decide which side is wrong from the specification, not by siding with whichever one you wrote.
3. If the specification is silent, measure the hardware, write down what it does, and say in the model that this records behaviour rather than a requirement.
4. If the specification is ambiguous rather than silent, the ambiguity is a hazard in its own right, so decide it, document it, and check that everything implementing the decision changed together.
5. Then ask what a model comparison could not have seen on this design, and write that test separately.

> Compare snapshots, not values. If both sides can produce the same tuple, compare the tuples: it is less code than a hand-written expectation, it checks state you were not thinking about, and it fails with a location instead of a symptom.
>
> Keep the model's blind spots written down next to it. A model that everyone trusts and nobody has bounded is worse than no model, because it turns "we did not check that" into "it passed".

Check yourself

1. A `trace()` failure says the model has `r2 = 10` and the DUT has `r2 = 10`, but `r3` differs. The instruction was `ADD r2, r1`. What kind of bug do you suspect?
2. Name two things a SimProc core can get wrong that `golden.py` will never notice.

Next: Where bugs hide.