The SimProc harness

This page answers: *what can I ask the harness for, and what does it already do for me?*

Everything on this page lives in `labs/common/`, and you can import it from any lab, because `sim.mk` puts both your lab directory and `common/` on `PYTHONPATH`.

Every worked example below is a real line from `test_api_tour.py`, and every output block is what that line printed. The whole tour is nine tests and 535,650 ns of simulated time, and it runs in about five seconds:

```
cd labs/common
make sim TB=test_api_tour
```

If a signature on this page stops matching the code, that file stops running, so the page and the code can't drift apart without someone noticing.

The five files

file	what it is
`simproc.py`	The harness: `SimProc`, `trace()`, `bit()`, and the command constants.
`simproc_asm.py`	The assembler: instruction helpers, `set_reg`, `assemble`, `disasm`, `listing`. It imports nothing, so it runs at a plain Python prompt.
`golden.py`	The reference model: the instruction set in about 120 lines of Python. It imports nothing.
`bugs.py` · `inject.py` · `check.py`	The bug catalogue, the injector that puts a bug into the core, and the checker that gives the PASS/FAIL verdict. You run these through `make` rather than calling them by hand.
`test_example.py` · `test_reference.py` · `test_api_tour.py`	Worked testbenches. Copy the first to start your own, read the second once you've written yours, and treat the third as the source of every example on this page.

The shortest complete test

```
import cocotb
from simproc import SimProc
from simproc_asm import ORI, ADD

@cocotb.test()
async def r1_gets_five(dut):
    sp = SimProc(dut)
    await sp.start()                              # clock, reset, blank memory
    cycles = await sp.run([ORI(5), ADD(1, 0)])    # r0 = 5 ; r1 = r1 + r0
    assert sp.reg(1) == 5, f"r1 is {sp.reg(1)}, expected 5"
    assert cycles == 9, f"took {cycles} cycles, expected 5 + 4"
```

```
cp test_example.py test_mine.py
make sim   TB=test_mine
make check TB=test_mine BUG=B01
```

`TB` takes the name of a Python module rather than a file name, so write `TB=test_mine`, never `TB=test_mine.py`.

Setup

`SimProc(dut, clock_ns=10, uio=8)`

`SimProc` wraps the top-level handle that cocotb gives you. Creating it doesn't touch the simulation; nothing happens until you call `start()`.

`clock_ns` is the clock period. `uio` sets the baud divider, with `clk_per_bit = uio * 4`, so the default of 8 gives 32 clocks per bit. That's fast enough for tests to finish quickly, and slow enough to keep the UART comfortably inside its spec.

The object also gives you hierarchical handles that reach past the pins and into the design: `sp.chip` (the TinyTapeout wrapper), `sp.sys` (`simproc_system`), `sp.cpu` (the core), `sp.ram` (the 64-byte array) and `sp.rf` (the four registers). A real chip has nothing like these, but a simulation does, and if you refuse to use them on principle, each bug will cost you an afternoon.

`await sp.start(fill=FILLER)`

Starts the clock, resets the chip, writes `fill` into all 64 bytes of RAM, and starts the background UART receiver. Always call it first.

```
sp = SimProc(dut, clock_ns=10, uio=8)
await sp.start()
```

```
clock period 10 ns; uio=8 so clk_per_bit=32
after start(): PC= 0  r0=  0 r1=  0 r2=  0 r3=  0  N=0 Z=0  IDLE
RAM filler is 0x0C, RAM depth is 64, the parking instruction is 0xF1 = JMP   -1
```

The filler matters because `0x0C` is an unimplemented opcode (it takes four cycles, advances the PC by one and changes no state), so a program that runs off its own end walks harmlessly through the filler. Without it, unwritten RAM reads `XXXXXXXX` under Icarus, the core decodes an X, and you get a hang with no error message at all.

`await sp.reset()`

Pulses `rst_n` low for 10 cycles. That clears the registers, the PC and the flags, but not the RAM: `dp_ram` has no reset, so whatever you loaded into it survives.

```
sp.poke(0, 0x5A)
await sp.reset()
```

```
after reset(): PC= 0  r0=  0 r1=  0 r2=  0 r3=  0  N=0 Z=0  IDLE, and mem[0] is still 0x5A
```

Because reset leaves memory alone, you can load a program, reset the core, and then run the program from a known state.

`await sp.set_baud(uio)`

Changes the divider in the middle of a test, and `clk_per_bit` becomes `uio * 4`. The chip reads the divider from `uio_in` on every cycle, so the change takes effect immediately, and the harness's own receiver switches to the new rate along with it.

```
await sp.set_baud(16)                    # clk_per_bit = 64
```

```
set_baud(16): clk_per_bit is now 64; uart_ping() -> 0xAA
```

If you only ever test a parametric design at its default parameter, you've tested one point of a range. One of the thirteen bugs is invisible only at the default setting.

The backdoor

`poke`, `peek` and `load` write and read the RAM array directly, in zero simulated time. The UART does the same job through the chip's real pins, at 1,104 clock cycles per command. Use the backdoor for setting up a test, and use the UART when the UART is the thing you're testing.

`sp.poke(addr, value)` · `sp.peek(addr)`

These move one byte each way. `peek` returns an `int`, or raises `ValueError` if the byte contains an X. Both raise `IndexError` for an address outside 0–63.

```
sp.poke(0x30, 0xA5)
dut._log.info(f"... -> 0x{sp.peek(0x30):02X}   <- the OLD value")
await RisingEdge(dut.clk)
dut._log.info(f"one clock later, peek(0x30) -> 0x{sp.peek(0x30):02X}")
```

```
poke(0x30, 0xA5), peek(0x30) in the same instant -> 0x0C   <- the OLD value
one clock later, peek(0x30) -> 0xA5
```

> Watch out for this. `poke()` writes a simulator signal, and a signal written at time T can't be read back until the simulator has stepped forward. If you read it in the same instant, you get the old value, with no error and no warning. That's why `load()` ends with two clock cycles, whereas a bare `poke()` doesn't wait at all. Put a trigger (an `await` such as `RisingEdge(dut.clk)`) between a poke and the peek that checks it.

An address out of range gets this message:

```
poke(64, 0) raises: address 64 (0x40) is outside SimProc's 64-byte RAM. The hardware
addresses it with 8 bits but only has 64 entries, so on real silicon this read returns X
and this write is dropped. Verilator's linter says so statically; Icarus only shows it at
run time. See Lab 02.
```

`await sp.load(program, at=0, park=True)`

Writes a list of instruction bytes into memory and returns the parking address.

```
park = await sp.load([ORI(5), ADD(1, 0)])
```

```
load() put the program at 0..1 and the parking loop at 2 (0xF1 = JMP   -1)
```

Unless you pass `park=False`, it also writes one `JMP -1`, a jump to itself, right after your last instruction. You need that, because there's no way to stop this processor after exactly *N* instructions: `HALT` arrives over the UART and takes 1,104 clock cycles to get there, and the core keeps running the whole time. So programs park themselves, the way real bare-metal test code does, and `run()` watches for the parking address.

`load()` doesn't reset the core and doesn't start it, and it takes 2 clock cycles.

`sp.memory(lo=0, hi=64)`
Returns the RAM contents as a dict, for dumping when something has already gone wrong. A byte containing an X comes back as the string `'XXXXXXXX'` instead of raising an exception, because a dump you print after a failure shouldn't itself crash on the entry you most want to see.

```
memory(0, 8) minus the filler: 0:0x2F 1:0x44 2:0xF1
```

Reading the machine

All of these are synchronous, so you call them without `await` and they take no simulated time.

call	returns	example
`sp.reg(i)`	one register as an `int`	`200`
`sp.regs()`	all four as a tuple	`(200, 200, 0, 0)`
`sp.pc()`	the program counter	`6`
`sp.flags()`	`(N, Z)`	`(1, 0)`
`sp.fsm()`	the core's state, as a name	`IDLE`
`sp.halt`	the `halt` pin, as a character	`'1'`
`sp.done`	the `done` pin, as a character	`'0'`
`sp.snapshot()`	`(regs, pc, N, Z)`	`((200, 200, 0, 0), 6, 1, 0)`
`sp.describe()`	one line of everything	`PC= 6  r0=200 r1=200 r2=  0 r3=  0  N=1 Z=0  IDLE`

Three of these need a note:

`halt` and `done` are characters rather than integers: `'0'`, `'1'` or `'x'`. We did that on purpose. In cocotb 2.x, `int()` on a value containing an X raises `ValueError`, and a test that dies with a stack trace has told you something real in the least helpful way it could.

`snapshot()` returns the same shape that `Golden.snapshot()` returns, so you can compare the two with a plain `==`. That comparison is the whole interface to model-based checking.

`describe()` belongs in every failure message you write, because one call gives you every register, both flags, the PC and the state.

`bit(signal, index)`
`bit` is a free function rather than a method of `SimProc`, and it returns one bit of any signal as the character `'0'`, `'1'`, `'x'` or `'z'`. Use it instead of `(int(sig.value) >> i) & 1` on any signal that a broken design might drive to X.

```
dut._log.info(f"bit(dut.uo_out, 0) -> '{bit(dut.uo_out, 0)}'")
```

```
bit(dut.uo_out, 0) -> '1'  (the UART TX pin, idle high)
```

Making it go

SimProc has two ways to execute a program, and you can't swap one for the other. `RUN` free-runs until you send `HALT`, which is fast, but it gives you no way to stop after exactly *N* instructions. `STEP` executes one instruction and drops back to IDLE, which is exact, but it costs a whole UART command for every instruction.

`await sp.run(program=None, at=0, timeout_cycles=20000)` → cycles

This is the call you'll use most. It loads the program, adds the parking loop, starts the core, and watches the fetch address until it reaches the parking loop. Pass `program=None` to run something you loaded earlier.

```
cycles = await sp.run([ORI(5), ADD(1, 0)])
```

```
run([ORI(5), ADD(1,0)]) -> 9 cycles, r1 = 5   (ORI is 5, ADD is 4)
```

The returned count is exact, and you can check it by hand. It's the number of clocks between fetching your first instruction and fetching the parking loop, which is exactly the sum of your program's instruction times.

On a timeout it raises an exception whose message is a diagnosis rather than a bare number:

```
the program did not reach its parking loop at address N within 20000 clock cycles.
  IDLE means the core never started. A PC far past your program means it ran off the end —
  check your branch displacements. A PC that is not moving means an infinite loop you did
  not intend.
```

`await sp.step(n=1)`

Executes exactly `n` instructions, sending one `STEP` command for each, and the core comes to rest in IDLE after every one. Use this mode when you want to inspect the state after each instruction.

```
await sp.load([ORI(1), ORI(2), ORI(4)])
await sp.set_pc(0)
await sp.step(1)
await sp.step(2)
```

```
after step(1):  r0 = 5, PC = 1, state IDLE
after step(2) more: r0 = 7, PC = 3
```

(`r0` starts at 5, left over from the previous `run()`. `step()` doesn't reset anything, and that's deliberate, since the point of stepping is to carry on from the state you're in.)

`await sp.run_cycles(n)`

Free-runs while counting `n` clock cycles of execution. The count starts at the first instruction fetch, not when the `RUN` command was sent. Use it when your question is about time rather than about the answer, for example *"after 40 clocks, how far has it got?"*

```
await sp.load([ORI(1), ORI(2), ORI(4), ORI(8)])
await sp.set_pc(0)
await sp.run_cycles(10)
```

```
after run_cycles(10): r0 = 15, PC = 4
```

`run_cycles()` counts `n` cycles, but it doesn't freeze the machine after them. When the count is reached the harness sends `HALT`, and `HALT` is a UART command that takes time to arrive, so the core keeps executing until it does. Here is what we measured on a program of 40 unimplemented opcodes with no parking loop:

```
same call on a program that does NOT park: PC = 20.
```

At the instant the count reached ten, the PC was 3, and by the time the core came to rest it was 20. If you need the state at exactly cycle *n*, read it inside a watcher of your own (copy `_go()`), or give the program a parking loop and use `run()`. The test `a04` in `test_api_tour.py` measures both.

`await sp.set_pc(addr)` · `start_running()` · `stop_running()`

These are the three raw controls. `set_pc` only takes effect while the core is in IDLE. `start_running` sends `RUN`, and `stop_running` sends `HALT`, which takes effect when the current instruction retires.

If you write your own free-running measurement, start your monitor before you call `start_running()`. The chip acts on `RUN` 57 clock cycles before `uart_cmd` returns (we measured this in `test_handshake_timing.py`), so a monitor started afterwards has already missed about fourteen instructions. That's why every counting call in the harness goes through one four-line helper, `_go()`, and why you should copy it rather than write your own.

Protocol counters

These check the handshake rather than the data. A core can compute every answer correctly and still get its handshake wrong, and two of the thirteen bugs do exactly that.

`await sp.count_done_pulses(cycles)` → count

Runs the loaded program and counts rising edges of `done` over the first `cycles` clocks of execution. `done` should pulse once per retired instruction.

```
await sp.load([ORI(1), ADD(1, 0), SUB(2, 2), NAND(3, 3)])
await sp.set_pc(0)
pulses = await sp.count_done_pulses(17)          # 5 + 4 + 4 + 4
```

```
count_done_pulses(17) over 4 instructions -> 4
```

Unimplemented opcodes never pulse `done`, so keep them out of a program whose pulses you're counting, or put them in on purpose to check that they don't:

```
two unimplemented opcodes, 6 clocks -> 0 done pulses: 0x0C retires nothing
```

If your scoreboard is triggered by `done`, a missing pulse doesn't make your test fail. Instead your test silently stops scoring, with no message at all, which is worse than a failure.

`await sp.count_halt_cycles(cycles)` → count

Counts how many of the first `cycles` clocks of execution had `halt` high. While `RUN` is asserted and the program runs only implemented opcodes, the answer must be zero:

```
count_halt_cycles(17) -> 0   (implemented opcodes never pass through IDLE)
```

Unimplemented opcodes are the exception here too. Each one spends a cycle in `IDLE` with `halt` high, so four filler bytes (`0x0C`) give 4 halt cycles in 16 clocks, as the instruction set reports. Keep them out of a program whose halt cycles you're counting, or expect one per unimplemented opcode.

A core that blips back into IDLE between instructions can still compute every answer correctly, but it's breaking the handshake, and this counter is how you catch it.

The wire

The harness contains a bus functional model (BFM): Python that plays the device at the other end of the serial line. It knows the protocol (8N1: one low start bit, eight data bits sent least significant first, and one high stop bit), and it knows nothing else about the design.

The receiver runs as a background task, which `start()` launches, and it watches `uo_out[0]` from before the first command goes out. It has to be running that early, because the chip can begin its reply while the harness is still finishing the last stop bit of the command, and a receiver that only starts looking afterwards gets a plausible but wrong byte instead of an error.

call	what it does	replies?
`await sp.uart_ping()`	`PING`, returns the reply	yes, `0xAA`
`await sp.uart_read(addr)`	`READ`, returns the byte	yes
`await sp.uart_write(addr, data)`	`WRITE`	no
`await sp.uart_get_pc()`	`GET_PC`, returns the PC	yes
`await sp.uart_cmd(cmd, addr=0, data=0)`	any command, no reply waited for	—
`await sp.send_byte(value)`	one raw 8N1 frame plus the mandatory idle gap	—
`await sp.recv_byte(timeout_cycles=20000)`	the next byte the chip sent	—

```
dut._log.info(f"uart_ping()      -> 0x{await sp.uart_ping():02X}")
await sp.uart_write(0x21, 0x3C)
dut._log.info(f"... uart_read(0x21) -> 0x{await sp.uart_read(0x21):02X}")
await sp.set_pc(9)
dut._log.info(f"set_pc(9) then uart_get_pc() -> {await sp.uart_get_pc()}")
```

```
uart_ping()      -> 0xAA
uart_write(0x21, 0x3C) then uart_read(0x21) -> 0x3C
set_pc(9) then uart_get_pc() -> 9
three raw send_byte() calls -> recv_byte() 0xAA
```

Four things about the UART will each cost you an hour if you don't know them:

Every command is three bytes (`cmd`, `addr`, `data`), even when the last two are ignored. If you send only two, the chip is still waiting for the third.

`WRITE`, `RUN`, `HALT`, `STEP` and `SET_PC` don't reply. If you wait for a reply to one of those, you wait forever, and `recv_byte` says so in its timeout message.

`send_byte()` leaves one bit period of idle line after the frame, and that gap is required, because the chip's receiver stays busy for half a bit period longer than the frame it just received. If you send the next start bit immediately, the receiver misses it, and instead of an error you get a byte shifted by one. Handshakes and timing measures all of this.

A working `PING` doesn't prove that the link works. `0xF0` already has its top bit set and `0xAA` is a fixed pattern, so a receiver whose bit period is slightly wrong still gets both of them right.

The assembler — `simproc_asm.py`

The assembler imports nothing, so it runs at a plain Python prompt, and `python3 simproc_asm.py 0x2F 0x44` disassembles bytes from the command line.

helper	form	byte	disassembles as
`ORI(5)`	`R0 |= imm5`, always r0	`0x2F`	`ORI  5`
`ADD(1, 0)`	`rA += rB`	`0x44`	`ADD  r1, r0`
`SUB(2, 3)`	`rA -= rB`	`0xB6`	`SUB  r2, r3`
`NAND(1, 1)`	`rA = ~(rA & rB)`	`0x58`	`NAND  r1, r1`
`SHL(0, 3)`	`rA <<= amt`, amt 0–3	`0x3B`	`SHL  r0, 3`
`SHR(0, 1)`	`rA >>= amt`	`0x0B`	`SHR  r0, 1`
`LOAD(0, 1)`	`rA = mem[rB]`	`0x10`	`LOAD  r0, r1`
`STORE(1, 2)`	`mem[rB] = rA`	`0x62`	`STORE r1, r2`
`JMP(-1)`	signed 4-bit displacement	`0xF1`	`JMP  -1`
`BZ(2)`	branch if `Z`	`0x2A`	`BZ  +2`
`BNZ(-3)`	branch if not `Z`	`0xD9`	`BNZ  -3`
`BPZ(1)`	branch if `N == 0`	`0x15`	`BPZ  +1`
`UNIMP()`	unimplemented opcode	`0x0C`	`(unimplemented)`

`set_reg(reg, value)`

Returns the instructions that put an arbitrary 0–255 byte in a register. You need a helper for this because `ORI` is the only way to get a constant into this machine, and it has three limits: it ORs rather than loads, its immediate is only 5 bits, and it only ever writes `r0`. So for any other register, `set_reg` builds the value in `r0` first and then copies it across:

```
listing(set_reg(1, 0xB7))
```

```
  00: 06   SUB   r0, r0
  01: B7   ORI   22
  02: 3B   SHL   r0, 3
  03: 3F   ORI   7
  04: 56   SUB   r1, r1
  05: 44   ADD   r1, r0
```

It's always 4 instructions for `r0` and 6 for the other registers, so the cycle count of a test that uses it stays predictable.

> `set_reg` clobbers `N` and `Z`, because `SUB`, `ORI` and `SHIFT` all set flags. Set your registers up before a compare, never between the compare and the branch. Getting this wrong is the most common way for a flag test to end up testing `set_reg` instead of the design.

`assemble(text, base=0)`

Lets you write the program as text, with labels, so that the assembler does the displacement arithmetic for you:

```
prog = assemble("""
        SUB   r1, r1
top:    ORI   1
        BZ    top
        SUB   r3, r3
""")
```

```
  00: 56   SUB   r1, r1
  01: 0F   ORI   1
  02: EA   BZ    -2
  03: F6   SUB   r3, r3
```

`BZ top` became `BZ -2` without you having to count anything. Branch displacements are 4-bit and signed, and when a label is out of reach, `assemble` raises an error instead of truncating the displacement:

```
ValueError line 12: 'BZ top': 'top' is -11 instructions away and a SimProc branch only
reaches -8..+7. Move the label closer, or JMP to a JMP.
```

`disasm(word)` · `listing(prog, base=0)`

`disasm` turns one byte back into text, and `listing` produces the block of address, hex and mnemonic shown above. If you put a `listing()` in a log line when a test fails, you'll save yourself twenty minutes. If you put a `disasm()` in the failure message, whoever reads it doesn't have to look anything up: `expected 0x14, got 0x24` tells nobody anything, while *"at PC 6 the core ran `ORI 4` and the model ran `ADD r0, r1`"* tells them where to look.

The golden model — `golden.py`

The golden model is a second implementation of the same specification, written in a language where you can see what it does. It imports nothing, and `python3 golden.py` runs a demo program on its own.

call	what it does
`Golden(memory, pc=0)`	A machine holding a program. It pads memory to 64 bytes with the same `0x0C` filler the DUT gets.
`g.step()`	Runs one instruction and returns its cycle count, or a string if it can't continue.
`g.run(n, stop_on_reason=True)`	Runs `n` steps and returns `(cycles, reason_or_None)`.
`g.snapshot()`	`(regs, pc, N, Z)`, the same shape that `sp.snapshot()` returns
`g.memory(lo, hi)`	a dict, like `sp.memory()`
`compare_memory(model, sp, lo=0, hi=64)`	every address where they disagree, as `(addr, model, dut)`

```
prog = set_reg(1, 20) + set_reg(2, 6) + [SUB(1, 2), STORE(1, 2)]
g = Golden(prog + [PARK])
cycles, reason = g.run(len(prog))
await sp.run(prog)
diffs = compare_memory(g, sp)
```

```
Golden.run(14) -> 59 cycles, reason=None
Golden.snapshot() -> ((6, 14, 6, 0), 14, 0, 0)
SimProc.snapshot() -> ((6, 14, 6, 0), 14, 0, 0)
compare_memory() -> []   (an empty list is the good answer)
```

We made `step()` return a string rather than raise an exception on purpose, so that the caller can print the model's reason next to the DUT's state instead of having to catch an exception.

The model's limitation is stated at the top of its own file: it has no concept of `done`, `halt`, cycle counts or the UART, so a bug in any of those is invisible to it. That isn't a flaw in model-based testing, but it is the reason a test suite needs more than one kind of test.

`await trace(sp, n_instr, model=None)` → `(steps, problem)`

Single-steps `n_instr` instructions and records the state after each one. `steps` is a list of `(index, address, instruction_word, snapshot)` tuples, and `problem` is either `None` or a string describing the first place the DUT disagreed with the model. This example, from `a09_trace_against_the_model` in `test_api_tour.py`, traces a seven-instruction program on a fresh model:

```
prog = set_reg(0, 9) + [ADD(1, 0), SHL(1, 1), NAND(2, 2)]
g = Golden(prog + [PARK])
await sp.load(prog)
await sp.set_pc(0)
steps, problem = await trace(sp, len(prog), model=g)
```

```
trace() ran 7 instructions; problem = None
   0  @00  0x06 SUB   r0, r0     regs=(0, 0, 0, 0) PC=1 N=0 Z=1
   1  @01  0x0F ORI   1          regs=(1, 0, 0, 0) PC=2 N=0 Z=0
```

```
  2  @02  0x3B SHL   r0, 3      regs=(8, 0, 0, 0) PC=3 N=0 Z=0
  3  @03  0x0F ORI   1          regs=(9, 0, 0, 0) PC=4 N=0 Z=0
  4  @04  0x44 ADD   r1, r0     regs=(9, 9, 0, 0) PC=5 N=0 Z=0
  5  @05  0x6B SHL   r1, 1      regs=(9, 18, 0, 0) PC=6 N=0 Z=0
  6  @06  0xA8 NAND  r2, r2     regs=(9, 18, 255, 0) PC=7 N=1 Z=0
```

When the model and the DUT disagree, `problem` tells you exactly which instruction went wrong, instead of leaving you with *"something is wrong somewhere"*:

```
instruction 19, at address 19 (0x66 = SUB   r1, r2), left the machine in a different state:
      model: r=(...) PC=... N=... Z=...
        DUT: r=(...) PC=... N=... Z=...
```

Stepping is slower than free-running, because every instruction costs a whole UART command. Use `run()` when you want the answer, and `trace()` when you want to know where it went wrong.

Constants

```
from simproc import (MEM_DEPTH, FILLER, PARK, STATE_NAMES,
                     CMD_READ, CMD_WRITE, CMD_PING, CMD_RUN,
                     CMD_HALT, CMD_STEP, CMD_SET_PC, CMD_GET_PC)
```

	value	
`MEM_DEPTH`	`64`	`dp_ram` is 64 bytes, so addresses 64–255 read X
`FILLER`	`0x0C`	an unimplemented opcode: 4 cycles, PC+1, no `done`
`PARK`	`0xF1`	`JMP -1`, the parking loop
`CMD_READ` `CMD_WRITE`	`0xA3` `0x5C`	`READ` replies, `WRITE` doesn't
`CMD_PING` `CMD_GET_PC`	`0xF0` `0xB2`	both reply
`CMD_RUN` `CMD_HALT` `CMD_STEP` `CMD_SET_PC`	`0x3F` `0xC7` `0x6D` `0x99`	none of them reply
`STATE_NAMES`	`IDLE, CYCLE_1 … CYCLE_5`	the list `fsm()` looks names up in

Timing you can assert against

These are counts of simulated clock cycles, not wall-clock times, so they're deterministic and you can check each of them in a testbench.

instruction	cycles
`ORI`	5
`ADD` `SUB` `NAND` `SHL` `SHR` `LOAD`	4
`STORE` `JMP` `BZ` `BNZ` `BPZ`	3
an unimplemented opcode (`0xC`, `0xD`, `0xE`)	4, no `done` pulse, and one cycle of `halt`

the UART, at the default `clk_per_bit = 32`	cycles
one 3-byte command	1,104 = 3 × 11 × 32 + 32 + 16
a command with a reply (`PING`)	1,353
loading a 64-byte program over the wire	70,656
the same program through the backdoor	2
`RUN`'s head start: the core is executing this long before `uart_cmd` returns	57

Two things that are not obvious

`load()` writes a `JMP -1` after your program. Nothing on this hardware can stop a program after *N* instructions, so programs park themselves, the way real bare-metal test code does.

A reset clears the registers but not the RAM. That's what lets you load a program, reset the core and start from a known state, and it's why the random-program test in `test_reference.py` can call `sp.reset()` between programs and still compare memory afterwards.

Make targets

target	what it does	wall clock
`make`	`sim`, then `check`	20 s
`make sim`	your testbench against the golden core	~6 s
`make waves`	the same run, dumping `waves.vcd` (3,007,592 bytes)	~6 s
`make check`	the verdict: PASS on the good core, FAIL on `BUG`	~12 s
`make check-all`	your testbench against all thirteen broken cores	~77 s
`make bugs`	prints the bug catalogue: names, tiers, and what each one teaches	instant
`make model`	runs `golden.py` on its own, with no simulator involved	instant
`make verify`	confirm every catalogue anchor still matches `rtl/`	instant
`make shuttle`	the test the chip shipped with, against everything	~30 s

```
make sim   TB=test_mine          # your module instead of test_example
make check BUG=B07               # which broken core to check against
make sim   SIM=verilator         # the strict simulator
```

The wall-clock figures are ballpark and move by a second or two between runs. The simulated-time numbers don't move at all, on either simulator; see Two simulators for more on that.

For the warnings all of this prints along the way, see Output that is not an error.