Handshakes and timing

This page answers: *every register, every flag and every byte of memory is correct. How can it still be broken?*

It can, because a block also has to produce its results at the agreed moment, and a register dump shows you only what the values are, not when they appeared.

Two of the thirteen bugs in this course are of that kind. Here is one of them run against the most thorough value-checking testbench in the repository:

```
      ok  the golden core                    3 tests, all passed             20.0s
  MISSED  B08_store_drops_done               3 tests, all passed             19.2s
----------------------------------------------------------------------
 VERDICT: FAIL — your testbench passes B08_store_drops_done too.
   That bug hides behind: checking values instead of the handshake
```

`B08_store_drops_done` changes one line so that `STORE` no longer pulses the `done` output. The byte still lands in memory, and the address, the data, the program counter, the flags and the cycle count are all right; the only difference is that the `done` pin stays still during a `STORE`.

The shipped six-test example also reports it as `MISSED`. The eleven-test reference suite catches it, but with exactly one test:

```
  caught  B08_store_drops_done               11 tests, 1 FAILED              18.2s
        R08_done_pulses_once_per_instruction
             AssertionError: 15 instructions retired in 60 clock cycles, but `done` pulsed
12 times.
```

Fifteen instructions gave twelve pulses because three of the fifteen were `STORE`. The other ten reference tests pass on this core. One of them steps a program and compares every register, the PC and both flags against a Python model after every instruction, and it still passes, because the model (`golden.py`) has no idea `done` exists.

> In software, "returned the right value" is usually the whole contract, and when it returned is a performance question. In hardware, the timing is part of the specification, and that part has a name: the handshake. A downstream block that is waiting for a pulse that never comes doesn't get a late answer; it gets no answer at all.

This matters more than it might seem, for the reason Where bugs hide gave: `done` is the signal a scoreboard triggers on, and a missing pulse doesn't make such a test fail. Instead the scoreboard stops scoring, and the test keeps reporting green about a stretch of the program it is no longer checking.

One cycle late

`B13_store_falls_to_idle` belongs to the same family, and its effect is even smaller. `STORE` drops back through the core's idle state on its way to the next instruction instead of going straight there. Every result and every byte of memory is identical, and the only cost is one extra clock cycle per `STORE`. Here it is against the example testbench:

```
  caught  B13_store_falls_to_idle            6 tests, 1 FAILED               5.8s
        t05_a_loop_and_its_cycle_count
            AssertionError: the program took 110 clock cycles, expected 109.
```

That one-cycle difference between 110 and 109 is the only symptom, so the test that catches it is the one that asserts a cycle count worked out by hand:

```
    setup    4+5+4+4+4+5+4               = 30
    loop     (ADD 4 + SUB 4 + BNZ 3) x 5 = 55
    finish   4+5+4+4+4+3                 = 24
                                           ---
                                           109
```

An assertion on a number derived from the datasheet costs one line, and it catches a class of bug that no value comparison can reach. The size of the difference also tells you where to look. If the answer had been 114 instead of 110, the difference of five would have pointed to a cycle lost once per loop iteration. A cycle count is a property of the design, and properties like that are cheap to assert.

The same idea on the wire

SimProc talks to the outside world over a UART. The harness in `simproc.py` contains a **bus functional model** (BFM), which is Python code that pretends to be the device at the other end of a wire; it knows the protocol and nothing else about the design. Its `send_byte()` sends a start bit, eight data bits and a stop bit, then holds the line idle for one more bit period, and a comment in the code says that gap is required.

`labs/common/test_handshake_timing.py` measures why the gap is needed. It has four tests and runs 90,910 ns of simulated time in under a second of wall clock, and you'll run it in Lab 05, Part 7.

How long the receiver is busy

`t01` puts a monitor on `UART_RX1.state` and sends one byte at the default speed (`uio_in = 8`, so `clk_per_bit = 32`):

```
async def watch_rx(sp, cycles):
    """Record (state_name, how_many_clocks) for the receiver, for `cycles`."""
    rx = sp.sys.UART1.UART_RX1
    runs = []
    for _ in range(cycles):
        await RisingEdge(sp.dut.clk)
        name = RX_STATES[int(rx.state.value)]
        if runs and runs[-1][0] == name:
            runs[-1][1] += 1
        else:
            runs.append([name, 1])
    return [(n, c) for n, c in runs]
```

It prints:

```
clk_per_bit = 32; receiver states: START=16  RECEIVE=288  DONE=32
the frame on the wire is 320 clocks; the receiver is busy for 336 -- 16 clocks LONGER,
which is 0.50 of a bit period
```

You can work out each of those numbers from the RTL:

State	Clocks	Why
`START`	16	It leaves when `clkCount == (clk_per_bit-1)>>1` = 15, counting from 0, so it stays for 16 clocks. That is half a bit period, which puts the sampling point in the middle of the start bit
`RECEIVE`	288	9 × 32: the eight data bits, then one more bit period, which ends in the middle of the stop bit. The receiver's RTL keeps that ninth period as the slot for a parity bit, but SimProc's UART is built with no parity
`DONE`	32	One more bit period, during which it holds the byte before it looks at the line again

That adds up to 16 + 288 + 32 = 336 clocks. The frame on the wire was 10 bit periods (start, eight data, stop), which is 320 clocks, so the receiver is busy for exactly half a bit period longer than each frame it receives. The chip is ready again 16 clocks after your frame ends, rather than as it ends, and 16 is `clk_per_bit / 2`.

The test asserts the formula rather than just the number, so it still holds at other baud rates. Here it is swept across four settings:

`uio_in`	`clk_per_bit`	START	RECEIVE	DONE	Busy	Frame	Excess
4	16	8	144	16	168	160	8
8	32	16	288	32	336	320	16
12	48	24	432	48	504	480	24
16	64	32	576	64	672	640	32

The excess is `clk_per_bit / 2` in all four rows.

Sending without the gap

`t02` sends the same three bytes twice: once through the harness's `send_byte()`, which leaves the gap, and once through `send_frame_only()`, which is `send_byte()` with its last two lines deleted. That second version is the one almost everybody writes first:

```
async def send_frame_only(sp, value):
    """One 8N1 frame and NOTHING else -- no trailing idle."""
    frame = [0] + [(value >> i) & 1 for i in range(8)] + [1]
    for b in frame:
        sp.dut.ui_in.value = b
        await ClockCycles(sp.dut.clk, sp.clk_per_bit)
```

The command is `SET_PC 12`: three bytes, `0x99 0x0C 0x00`. The test prints:

```
sent    : 0x99 0x0C 0x00
received: 0x99 0x86 0x80
with the inter-frame gap: PC = 12.  Without it: PC = 134.
```

Nothing produced an error or a warning, and every bit was on the wire for exactly one bit period, but the program counter ended up at 134 instead of 12.

The corruption has a pattern, and you can read the cause straight from it:

- `0x99` arrived intact. The receiver was idle before it, so it caught the start bit on time.
- `0x0C` arrived as `0x86`. `0x0C` is `0000 1100` and `0x86` is `1000 0110`, which is `0x0C` shifted right by one with a `1` in the new top bit: `(0x0C >> 1) | 0x80 = 0x86`. Every bit was sampled one bit period late, so bit *i* of the received byte is bit *i+1* of the sent one, and the top bit came from the stop bit, which is always 1.
- `0x00` arrived as `0x80`, which is `(0x00 >> 1) | 0x80` by the same rule. For these three bytes the test pins the result exactly (`assert got == late and bad == late[1]`), because a prediction you can check is worth more than a shrug.
- `134` is `0x86`, the corrupted address byte, loaded into the program counter.

The receiver was still in `DONE` when the second start bit began, so it missed that falling edge. It picked up the next falling edge instead and never recovered, which is why every later frame is late by the same amount.

`PING` without the gap

`t03` sends `PING` the same broken way:

```
gapless PING answered 0xAA -- a green light on a link that just corrupted two of its three bytes
```

`PING` is `0xF0`, and it ignores its address and data bytes, so the two frames that get mangled are the two that nothing reads, and the reply is a fixed `0xAA` either way. Your link test passes while every command that carries an argument is broken.

> A link test that passes over a broken link is worse than having no link test. It has the same flaw as the shipped test that asserts the processor is not running: it is a green light that was never wired to the thing it claims to check.

Two numbers your testbench already depends on

`t04` measures the two constants the harness is built on, and checks each against arithmetic you can do on paper:

```
one 3-byte command costs 1104 clock cycles = 3 x 11 x 32 (frame + gap) + 32 + 16 = 1104
the core fetched its first instruction 57 clock cycles BEFORE uart_cmd(RUN) returned
```

The 1,104 is three frames of eleven bit periods each (ten on the wire plus the gap), plus the 48 clocks the harness waits at the end of a command: 3 × 11 × 32 = 1,056, and 1,056 + 48 = 1,104. This is why you can't stop this processor after exactly *N* instructions: `HALT` takes 1,104 cycles to arrive, and the core keeps running the whole time. It is also why `run()` appends a parking loop.

The 57 is the core's head start. The chip acts on `RUN` before the last stop bit has finished going out, so a monitor that you start after `start_running()` returns has already missed about fourteen instructions. For that reason every counting call in the harness (`run()`, `run_cycles()`, `count_done_pulses()`, `count_halt_cycles()`) starts its watcher first. If you write your own counting call, do the same; the harness's `_go()` is the four lines to copy.

Other bytes

`COMMAND` is one line at the top of the test file, and nothing in `t02` is a constant, so you can change the data and watch the corruption change its pattern. Here are four runs, all of which pass:

Address, data	Received	The "one period late" prediction	PC
`0x0C, 0x00` (shipped)	`0x99 0x86 0x80`	`0x99 0x86 0x80` ✓	134
`0x34, 0x12`	`0x99 0x9A 0x89`	`0x99 0x9A 0x89` ✓	154
`0x0F, 0x33`	`0x99 0x87 0xE6`	`0x99 0x87 0x99` ✗	135
`0x55, 0xFF`	`0x99 0xAA`, two bytes only	`0x99 0xAA 0xFF` ✗	0

They all pass because `t02` asserts only what is true for any three bytes: the first frame arrives intact, something later is corrupted, and the command did not do what you asked.

The last two rows show that the clean one-bit-period slip is only the common case, and you can't rely on it.

`0x0F, 0x33` slips cleanly on the second frame and then lands somewhere else on the third. The receiver resynchronised onto a different edge, and `0xE6` is neither the byte you sent nor what a uniform slip would give.

`0x55, 0xFF` is worse, because only two bytes arrive. After its start bit, `0xFF` is a solid run of ones, with no falling edge for a resynchronising receiver to latch onto. The receiver never sees a third byte, the chip is left waiting for it, and the program counter stays at 0. A lost frame and a corrupted frame look completely different in a log, even though they come from the same defect.

> When a protocol has a recovery or turnaround period, find its length in the RTL, make your BFM wait it out, and write a test that measures it, so that you find out when someone changes it. When you pick test data, avoid bytes that are fixed points of the corruption you are hunting. `0x00`, `0xFF` and `0xAA` survive more transformations than any other bytes, and they happen to be the three everybody reaches for first.

Check yourself

1. A core gets every register right and pulses `done` once for every two instructions. Which of `golden.py`, a cycle-count assertion and a `done` counter will notice?
2. At `clk_per_bit = 48`, how many clocks after the end of a frame is the receiver ready again?
3. Why does `PING` make a poor link test?

Next: Lab 05: Every answer is right.