Unit Testing for Hardware Description Languages

”Test Driven LED blinking”

Maxence Caron–Lasne

LSE Lightning Talks, October 15, 2019

Introduction

- Unit testing require a lot of tests.
- In some cases, unit testing is hard/cumbersome.
- The quicker tests are written, the quicker code can be written.
- Let’s see how to do Test Driven Development with an Hardware Description Language.

What is an Hardware Description Language?

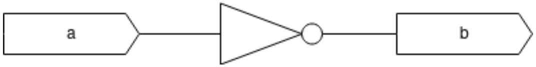
Definition from Wikipedia

Language used to describe the structure and behavior of digital logic circuit.

- Different from a programming language.
- Timing is explicit.
- Synthesize into netlist.

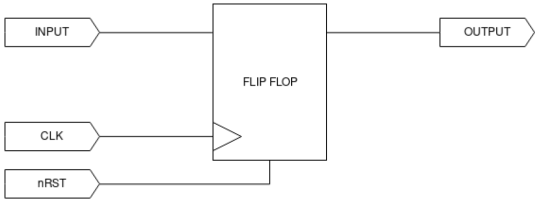
HDL: an example

```
ENTITY not_example IS PORT(
    a: IN  STD_LOGIC;
    b: OUT STD_LOGIC);
END not_example;

ARCHITECTURE behavioral OF
not_example IS
BEGIN
    b <= NOT a;
END behavioral;
```



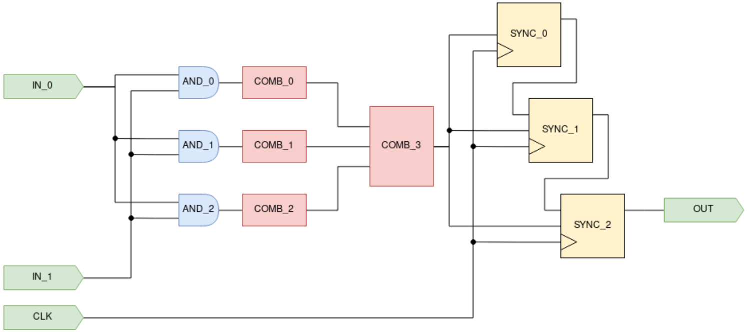
HDL: a synchronous example

```
ENTITY FlipFlop IS PORT(
    Clk    : IN STD_LOGIC;
    nRst   : IN STD_LOGIC;
    Input  : IN STD_LOGIC;
    Output : OUT STD_LOGIC);
END FlipFlop;

ARCHITECTURE rtl OF FlipFlop IS
BEGIN
    PROCESS(Clk, nRst) IS
    BEGIN
        IF nRst = '0' THEN
            Output <= '0';
        ELSIF rising_edge(Clk) THEN
            Output <= Input;
        END IF;
    END PROCESS;
END ARCHITECTURE;
```



What is a module?

- Mean of abstraction.
- Inputs
- Outputs
- Wires or registers.
- Logic
- Submodules.



A module in VHDL

```vhdl
ENTITY servo_ctl IS PORT (
    clk, rst: IN STD_LOGIC;
    degrees:  IN INTEGER range 0 to 180;
    gpio:     OUT STD_LOGIC);
END ENTITY servo_ctl;

ARCHITECTURE rtl OF servo_ctl IS
    SIGNAL inner_clk : STD_LOGIC;
    SIGNAL inner_cnt : INTEGER range 0 to 20*180;
BEGIN
    servo_clk : ENTITY work.servo_clk
        PORT MAP (rst => rst,
                  clk => clk,
                  clk_out => inner_clk);

    PROCESS (clk, rst) BEGIN
        -- some logic
    END PROCESS;
END ARCHITECTURE;
```

- Inputs/Outputs
- Signals
- Submodule
- Logic

A module in Migen

Migen is a Python library used to generate Verilog.

```
class ServoCtl(Module):
    def __init__(self):
        self.i_degrees = Signal(max=180)
        self.o_gpio = Signal()

        inner_clk = Signal()
        inner_cnt = Signal(max=20*180)

        servo_clk = ServoClock(inner_clk)
        self.submodules += servo_clk

        self.sync += [
            # Some logic here
        ]
```

- Inputs/Outputs
  (no clock, no reset)
- Signals
- Submodule
- Logic

What means ”unit testing” when working with HDL?

- Testing a single module.
- May mock submodules.
- Two cases: asynchronous and synchronous modules.
- Asynchronous: kind of ”pure”.
- Synchronous: timing is important.

Writing testbenchs in VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity tb_servo_clk is
end tb_servo_clk;

architecture tb of tb_servo_clk is

  component servo_clk
    port (rst     : in std_logic;
          clk     : in std_logic;
          clk_out : out std_logic);
  end component;

  signal rst     : std_logic;
  signal clk     : std_logic;
  signal clk_out : std_logic;

  constant TbPeriod : time := 4 ns;
  signal TbClock : std_logic := '0';
  signal TbSimEnded : std_logic := '0';
```

```vhdl
begin
  dut : entity work.servo_clk
    port map (rst     => rst,
              clk     => clk,
              clk_out => clk_out);

  TbClock <= not TbClock after TbPeriod/2 when TbSimEnded /= '1' else '0';

  clk <= TbClock;

  stimuli : process
  begin
    rst <= '1';
    wait for 100 ns;
    rst <= '0';
    wait for 100 ns;

    wait for 50000 * TbPeriod;

    TbSimEnded <= '1';
    wait;
  end process;

end tb;
```

Writing testbenchs in Migen

```python
dut = ORGate()

def testbench():
  yield dut.a.eq(0)
  yield dut.b.eq(0)
  yield
  assert (yield dut.x) == 0

  yield dut.a.eq(0)
  yield dut.b.eq(1)
  yield
  assert (yield dut.x) == 1

run_simulation(dut, testbench())
```

- `yield dut.a.eq(0)` is for assigning values to signals.
- A single `yield` is for passing a clock cycle.
- `yield dut.x` is for accessing a signal's value.

Using Python `unittest` library with Migen

```python
class TestOneHotToBinary(unittest.TestCase):
    def setUp(self):
        i_onehot = Signal(8)
        o_result = Signal(8)
        self.mod = implications.OneHotToBinary(
            i_onehot, o_result)

    def test_zero(self):
        self.id = "zero"
        self.data = [{
            "onehot": 0b00000000,
            "result": 0b00000000
        }]

    # ...

    def tearDown(self):
        def sim(mod, data):
            for d in self.data:
                yield mod.i_onehot.eq(d["onehot"])
                yield
                output = yield mod.o_result
                self.assertEqual(output, d["result"])
        run_simulation(self.mod, sim(self.mod, self.data))
```

- `setUp()` instantiates the module under test.
- `test*()` declares the tests inputs and expected outputs.
- `tearDown()` runs the simulation.
- `self.data` is a list of dictionaries.
- Each dictionary is for one clock cycle (”before” and ”after”).
- Still quite cumbersome.

Generic tests for HDL

```json
{
    "id": "implication_detector",
    "inputs": {
        "i_clause_entry": {"size": 8}
    },
    "outputs": {
        "o_impl_vector": {"size": 4, "display": "hex"}
    },
    "tests": [
        {
            "tag": "zero",
            "yields_after_inputs": 1,
            "yields_after_outputs": 0,
            "rounds": [
                {
                    "inputs": {
                        "i_clause_entry": "0b00000000"
                    },
                    "outputs": {
                        "o_impl_vector": "0b0000"
                    }
                }
            ]
        }
    ]
}
```

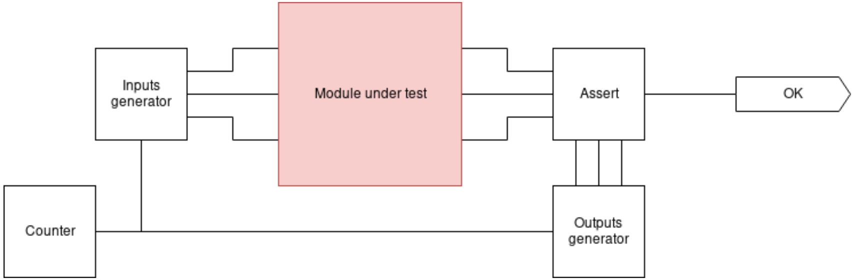
- Test declarations are test engine agnostic (kind of).
- Custom test engine can be used without duplicating code or breaking previous test engines.

From simulation to LED blinking

- Test declarations are agnostic.
- Can we build hardware testbenchs?

Dynamic supermodules

- A module dynamicly built to run the module under test.
- In charge of input application, output gathering and counting cycles.
- Can print some values on seven segments or LEDs.
- Gets module under test and I/O signals from init arguments.



`TestSuite` class

- Gets test declarations and module *class* from init arguments.
- Builds I/O signals and test signals from test declaration.
- Instantiate the module under test with correct test signals.
- The supermodule is ready to be converted to Verilog.

TODOs

- Better test declarations.
- Agglomerate all unit tests into a single module.
- Refactor.
- Maybe make this public.

Conclusion

- Describing tests rather than ”programming” them gives to the user some liberty.
- Meta-programmation is easier when working with a Python library.
- Meta-programmation permits dynamic module building.
- Dynamic modules permits generic and synthesizable test modules.