EMBEDDED SYSTEMS < HTTPS://BLOGS.PLYMOUTH.AC.UK/EMBEDDED-SYSTEMS/>

Just another Plymouth University Blogs site

Part 3 – Automatic Testing with 'assert'

prev < https://blogs.plymouth.ac.uk/embedded-systems/fpga-and-vhdl/test-benches/part-2-multiple-architectures/>

In part 2, we looked at home to compare two architectures of the same component visually using ModelSim waveforms. This is fine for such a small design, but as an approach, it does not scale well as projects become more complex.

Interpreting timing diagrams is repetitive, tedious and error prone. Surely it would be far batter if the tools could check the outputs? Well, the good news is this is precisely what you can do. In software engineering, it's known as **unit testing**. The principles are the same.

- The compiler is only capable of detecting syntax errors, not logical errors – it cannot know what your code is supposed to do.
- Testbench code is additional code written by the developer and performs run-time tests, applying pre-defined inputs and checking outputs for expected results.

Engineers are very aware how implementing changes to code or hardware introduces risk, even bug fixes bring their own risk and can introduce new logical

errors (new bugs for old!). Again, this is where writing automated testbenches (or units tests in software) can help.

Every time you make a change to your component (hardware or software), you can rerun the tests to confirm behaviour has not adversely changed.

Do not be surprised if you write longer test code than the component under test itself!

Using assert

Let’s look at a simple example. Based on the **previous page < https://blogs.plymouth.ac.uk/embedded-systems/fpga-and-vhdl/test-benches/part-2-multiple-architectures/>** , we are going to add a simple test to ensure both architectures produce the ‘same’ output. We need to qualify what me mean by same. In this case, I am only testing for valid input states and (for now) ignoring any timing differences.

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

entity comp1_test is
end comp1_test;

architecture inst1 of comp1_test is

        --Declare signals
        signal AA : std_logic;
        signal BB : std_logic;
        signal CC : std_logic;
        signal Y1 : std_logic;  --output for version 1
        signal Y2 : std_logic;  --output for version 2

begin
        --Instantiate v1 of the component type comp1
        u1: entity work.comp1(v1) PORT MAP (
                A => AA,
                B => BB,
                C => CC,
                Y => Y1
        );

        --Instantiate v2 of the component type comp1
        u2: entity work.comp1(v2) PORT MAP (
                A => AA,
                B => BB,
                C => CC,
                Y => Y2
        );

        comp1_process:
        process
                variable count : std_logic_vector(2 downto 0);
        begin
                --Exhaustive test
                for idx in 0 to 7 loop
                        --Convert integer idx to std_logic
                        count := std_logic_vector(to_unsigned(idx,3)

                        --Assign internal signals to the individual
```

```
                        AA <= count(0);
                        BB <= count(1);
                        CC <= count(2);

                        --Wait for 20ns (also prompts signals to upd
                        wait for 20 ns;
                        assert(Y1 = Y2) report "Mismatch" severity e
                end loop;

                --End of test is to wait forever
                wait;
        end process;

end inst1;
```

The only change is the highlighted line:

```
assert(Y1 = Y2) report "Mismatch" severity error;
```

The assert command takes a boolean (true or false). If it is false, the report string it written to the console output. The severity it typically `error` or `warning`. Note that this statement follows the wait statement. Thinking ahead, this would allow 20ns for outputs to settle.

Let's reintroduce the original error in the component `comp1`.

```
architecture v2 of comp1 is
begin
        y <= ((not A) and (not B) and  (not C));
        --y <= not ((not A) and (not B) and  (not C));
end v2;
```

If we simply run the testbench, and without using the wave view, the following is displayed in the terminal:

```
run -all
# ** Error: Mismatch
#    Time: 20 ns  Iteration: 0  Instance: /comp1_test
# ** Error: Mismatch
#    Time: 40 ns  Iteration: 0  Instance: /comp1_test
# ** Error: Mismatch
#    Time: 60 ns  Iteration: 0  Instance: /comp1_test
# ** Error: Mismatch
#    Time: 80 ns  Iteration: 0  Instance: /comp1_test
# ** Error: Mismatch
#    Time: 100 ns  Iteration: 0  Instance: /comp1_test
# ** Error: Mismatch
#    Time: 120 ns  Iteration: 0  Instance: /comp1_test
# ** Error: Mismatch
#    Time: 140 ns  Iteration: 0  Instance: /comp1_test
# ** Error: Mismatch
#    Time: 160 ns  Iteration: 0  Instance: /comp1_test
```

It is clear the two architectures do not produce the same result. If the bug is fixed, the output is blank.

There is more you can do with assert. We will meet more examples in following sections. This example only confirms the two architectures are the same. What we've still not confirmed is whether the output is logically correct! Furthermore, we've only performed functional testing. To bring in timing, there is more work to do.

Next < https://blogs.plymouth.ac.uk/embedded-systems/fpga-and-vhdl/test-benches/part-4-testing-with-simulated-timing/> – Part 4 – Testing with simulated timing

UNIVERSITY OF PLYMOUTH **< HTTPS://BLOGS.PLYMOUTH.AC.UK/EMBEDDED-SYSTEMS/>**

Proudly powered by **WordPress < https://en-gb.wordpress.org/>** .