COMP4300

Lab Exercise One, FA 24

Objective

This lab is aimed at getting you familiar with using the ModelSim simulator. This lab requires you to implement a two-bit counter circuit with a single-bit input, two-bit internal state, and one two-bit output.

Instructions

Write a VHDL program consisting of a single entity whose architecture is a single process that implements a two-bit counter. The input is a onebit signal **increment,** and the two-bit output is **count.** The count output always reflects the internal value of the counter The **increment** signal causes the internal value and the output signal to go up by 1 at time 10ns after **incremen**t transitions from 1 to 0, or if count is "11" it should go to "00". Note that bit_vector constants are in double quotes, while single bit constants are in single quotes. Behavior is undefined if you change the increment signal more rapidly than every 10 ns. We will test it by changing increment at a speed of 50ns per transition (100ns clock). You do not need to initialize the counter (see below for extra credit).

This should take very few lines of code. If your solution is much longer than the examples we did in class, you are probably doing something wrong – come see me! The main purpose of this exercise is to get you familiar with using the ModelSim tool chain and simulator.

You should implement the counter's architecture as a single VHDL process that is sensitive to the one input signal, and use an internal variable of type **bit_vector(1 downto 0)** to store the state. Note that you cannot use the normal addition operator '+' for integers on bit vector variables. In later labs we will use a library to do bit vector math, but for now you can use a 4-way if-then-else to increment the counter.

The entity declaration is given below:

```
entity twobitcounter is
begin
        generic(prop_delay: Time := 10ns);
        port(increment: in bit;
              count: out bitvector(1 downto 0)
        );
end entity twobitcounter;
```

You should test enough cases to convince the TA that your circuit works as it should. How many is "enough" is a decision you will have to make. We expect to see normal incrementing behavior 00 to 01 to 10 to 11, and then wrapping back around to 00.

Extra credit: In real life no one would waste circuitry to cause this counter to initialize to zero (have to detect a power on condition – that would take way more circuitry than the counter itself). They would let it initialize to garbage and use an external signal **reset** set the internal state to "00". This would be a savings because you could detect power one in one place and then distribute the reset signal to everything that needed to be reset on power-up. So for 25 points extra credit you can add that single-bit **reset** input to your circuit. On the falling edge of **reset**, the internal state should go to zero "00" and the output **count** should go to "00" 10 ns later. The **reset** signal should override the **increment** signal if they both go from 1 to 0 within 10 ns of each other, even if reset goes to zero first. Be sure to include screenshots showing the testing of your reset signal. Note – you can use the Boolean value reset'event to check if the reason the code is running is because the reset signal changed. This exists for any signal, so you can also use inrcrement'event if you need it. So a test like

```
if increment'event and event = '0' then

end if
```

could be used to detect if the code is running because a change to the increment signal is the reason the code is running, and that it just changed to 0.

Deliverables: Please turn in the following things for this lab:

- The file with your VHDL code.
- A screen shot (one or more) of your simulation