Homework 2

CMPE 415 - Fall 2017
Prof. Ryan Robucci
Due Oct 3th

September 26, 2017

In this HW you should abide by the Section 5.0 of www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA.pdf with an exception in regard to Guideline #5 that I'll allow mixed assignments when all variables that are assigned using blocking statements are defined only locally within a named block.

Additional guidlines may be provided but where appropriate please note the following code grading guidelines.

- Correctness..
  - e.g Completeness of homework and matching specifications
  - Providing test-bench code and thoroughness of testing and reporting
  - e.g. Use of $finish in every testbench, which establishes when the simulation is complete
- Style..
  - e.g. Well commented code
  - Meaningful variable names
  - Meaningful file names
- Documentation..
  - e.g Student should clearly state that the code is complete and works or not (TAs don't want to have to guess if/why something isn't working or finished)
  - Students should clearly state special code implementation or usage instructions
  - Readable waveforms in report
- Submission: You use a unqiue subfolder for each question as procivided in the intial redo you can clone. Each folding should have with supporting source files and answers provided in a file named **report.pdf** or **report.md** (doc/docx files will not be read). To submit use the following git url substituting your username for <username>

  ssh://<username>@gl.umbc.edu/afs/umbc.edu/users/r/o/robucci/pub/cmpe415/submit/hw2/cmpe41

1. Consider the following 3-input,2-output truth table:

a	b	c	y	z
1	1	1	0	1
1	1	0	1	1
1	0	0	0	0
otherwise			0	**-**

   where **-** denotes don't care
   Implement the table as a module using the following variations. Synthesize each version and submit a screen-capture of the RTL view.

   (a) using behavioral code with if statements
   (b) using concurrent continuous assignment statement(s)
   (c) using behavioral code with case statements

2. Using synthesizable procedural code, create a module that with every clock cycle outputs a 1-bit value that represents the input 5 clock cycles prior. The module should also support an asyncronous reset that allows the initial outputs to be zero until the first input propogates to the output.

3. Create a synthesizable combinatorial module that accepts a 4-bit input **x** and and generates a single-bit ouput **y** based on **x** and function select signal **s**. The selector should select which reduction operator to be applied to **x** in order to produce **y** according to the following:
   - 0: and
   - 1: or
   - 2: xor
   - 3: nand
   - 4: nor

4. Create a synthesizable sequential module that accepts two 4-bit inputs **x** and **y** and generates a 4-bit ouput **q** based on **x** and a function selector **s**. The selector should select the bit-wise operator to be applied to **x** and **y** in order to produce **q.** The output **q** should be a registered output, and so the module should accept clk signal **clk**. There should be no unesssisary cycle delays from input to the output.
   - 0: and
   - 1: or
   - 2: xor
   - 3: nand
   - 4: nor

5. Leveraging the concatenation operator, create a module that accepts a 64-bit input **x**, and swaps pairs of even-and odd bytes to produce a 64-bit output **u**. The output should be registered using a clock signal **clk**.

6. Create a test bench to demonstrate the functionality of the following code in which **a** and **b** are 1-bit inputs. Note this example as provided breaks certain coding rules. The test bench should produce a table using the **$strobe** task. Your calls should print time using **%0t** and **$time**. Provide your tesbench and the output. Use the output to produce a condensed table with comments explaining the operation of the circuit.

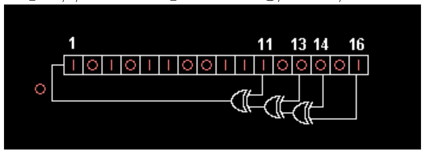
```
module DUT(y, z, a, b, clk );
input a, b, clk;
output reg[1:0] y, z;
always@ (posedge clk) begin
    y=2'b00;
    if ( a == 1 ) begin
      y[1] = 1'b1;
      y[0] = 1'bx;
      z = 2'b01;
    end else if (b == 1 ) begin
      y[0] = 1'b0;
      y[1] = 1'b0;
      z = 2'b10;
    end
  end
endmodule
```

7. Add **$display** calls to the above code after every assignment statement and resimulate to show that multiple assignments are observable in simulation whereas **$strobe** only prints one final result per change event. Both your **$display** and **$strobe** calls should print time using **%0t** and **$time**.

8. Repeat (7) with with blocking statements converted to non-blocking statements. Discuss your observation.

9. Now, perform a post-synthesis simulation using the module in (6). Instructions are provided in the appendix. By observing the times printed using a **$monitor** call in your testbench, show that the output y and z change with some delay after the clk edge. Be sure to create a 50 MHz clock from your testbench according to a time unit convention you document. Provide your output and discuss what other change you see in the output as compared to what you saw in the presynthesis simulation. WARNING: **USE EXPLICIT PORT MAPPING IN YOUR TESTBENCH SINCE SYNTHESIS SOMETIMES REORDERS THE PORTS.**

10. Implement the linear feedback shift register shown on https://en.wikipedia.org/wiki/Linear_feedback_shift_register



The core 16-bit register must be implemented with a 16-bit signal. The the shift should be implemented using a concatenation operator to right-shift with the left-most bit loaded from the output of the xor gates. Verify that linear-feedback shift-register produces all possible states of a 16-bit register excluding the zero state before repeating its seqeunce.

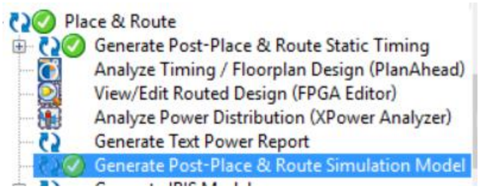
To do this, use a Verilog Test Bench to save the output to a file every clock cycle for at least two full cycles of the pattern. Write C/Python/Java code to verify that the pattern repeats identically at least twice and that no number is repeated or missed during one cycle of the pattern. You should submit all code and output files.

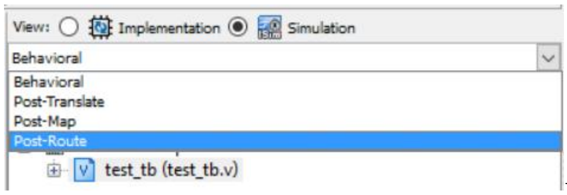
11. [1] Implement as a strictly combinatorial module ALU that will add or subtract two 5-bit two-signed values **a** and **b** based on a control signal **sub**. The outputs should be a 5-bit two's-complement result **y** and an over-/under-flow bit **c**. Implement two variations of this module using

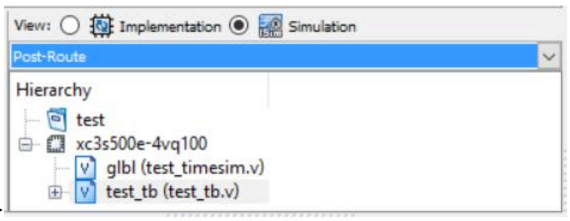
   (a) structurally using full 1-bit adders that are in turn built from Verilog primatives
   (b) using behavioural code with arithmetic operators

Appendix – Post-Synthesis Simulation

- Perform post Place and Route Simulation using steps as follows
- After successfully Implement Design, explore place and route option
  - Click on Generate Post-Place & Route Simulation Model, it should now run post Place and route simulation
  - If successful you should see something similar to screen capture shown here:

    * 
    Place & Route
    Generate Post-Place & Route Static Timing
    Analyze Timing / Floorplan Design (PlanAhead)
    View/Edit Routed Design (FPGA Editor)
    Analyze Power Distribution (XPower Analyzer)
    Generate Text Power Report
    Generate Post-Place & Route Simulation Model
  - Go to simulation radiobutton/tab and select Post Route simulation from the dropdown, as shown here:

    * 

    $\rightarrow$ 

[1] Q11 is are taken directly or modified from Chapter 4 problems of Michael D. Ciletti, Modeling, Synthesis, and Rapid Prototyping with the VERILOG (TM) HDL 1st Edition, Prentice Hall