ECE 551 Homework #2

Due: Sept. 23rd @ 11:00 am

This homework assignment is to be completed individually.

Please Note: To receive full credit, you should use the following best practices in your homework assignments**:**

- Use meaningful names for modules, ports, and wires when possible.
- Use underscores to break up long binary and hex numbers into groups of 4 digits to make them more readable. Ex: 12'b1001_0011_1101
- Show your work if you wish to receive partial credit.
- Make port connections *by name* unless using gate primitives.
- Explicitly declare any nets you use.
- All figures & graphs should include a number and caption (handwritten is OK).
- When printing waveforms, ensure that the relevant signal transitions are clearly visible.
- *All code should be typed*. Please do not submit handwritten code.
- **You must always turn in a printout of all Verilog code used with each problem unless explicitly told not to do so.**

[1] *(6pts)*

(a) Give two reasons why delays are used in Verilog.

(b) Give the code needed to instantiate an array of 10 XOR gates, each with a delay of 5 time units, using a single structural Verilog statement.

(c) Give one advantage and one disadvantage of using structural Verilog compared to behavioral Verilog.

[2] *(20pts)*

In this problem, you will implement a module that performs a combinational *modulo-6* operation in *structural* Verilog. The module should have the following interface:

```
module modulo_6 (input [2:0] original_value, output [2:0] mod6_value);
```

The module will compute *mod6_value = original_value mod 6*. Treat both input and output values as unsigned.

(a) Create a schematic for a circuit that implements a 3-bit modulo 6 operation (*neat* hand-drawn schematics are acceptable). Show any work you did to derive the schematic (truth table, K-map, etc).

(b) Implement the circuit you designed in part (a) using *structural* Verilog.

(c) Create an *exhaustive* testbench to exercise the module designed in part (b) (*Note*: the testbench is not required to be in purely structural Verilog). Simulate the testbench in ModelSim and submit a printout of the Wave window showing the output of your module. Submit both the testbench source code and the printout.

[3] *(16pts)*

Using structural Verilog, implement a Moore-style finite state machine that detects whether the last three input bits form a prime number in unsigned binary (Note that 1 is not considered prime). If the previous three inputs form a prime number, your state machine should output a 1 on the next clock cycle. Your state machine should use binary encoding (i.e not one-hot-state encoding).

Sample output:

Cycle	Input	Last 3 inputs (decimal value)	Output
1	0	xx0 (X)	0
2	1	x01 (X)	0
3	0	010 (2)	0
4	0	100 (4)	1
5	1	001 (1)	0
6	1	011 (3)	0
7	1	111 (7)	1
8	...	...	1

Table I: Sample output for prime number detector

Your module should use the following interface:

```
module prime_detector (input in_bit, clk, rst, output prime);
```

Use the following module to instantiate the necessary flip-flops in your structural Verilog:

```
module dff_async_rst(input clk, d, rst, output reg q);

     always@(posedge clk, posedge rst)
     begin
           if(rst == 1'b1)
                 q <= 1'b0;
           else
                 q <= d;
     end

endmodule
```

Listing 1: D-Flip Flop with active-high, asynchronous reset.

Show your work for the derivation of the Next State and Output combinational logic and include a state diagram (neat hand-drawn diagrams are acceptable).

[4] *(8pts)*

Give the result of the following operations in Verilog binary number format, with the proper number of bits:

(a) 4'b0101 | 4'b0011

(b) 4'b0101 * 4'b0011

(c) 4'b1100 ~^ 4'd7

(d) {4{3'b101}}

(e) ^8'b0010110

(f) 6'b100000 && 6'b111111

(g) !4'b0101

(h) 6'b010110 >> 2

[5] *(6pts)*

Implement the body of the following modules using **a single RTL continuous assignment statement**.

(a)

```
module endian_converter (input [15:0] little_endian, output [15:0] big_endian);

//Switch the order of the upper and lower bytes of the input and assign it to the output.

endmodule
```

(b)

```
module parity_generator (input [11:0] in_number, output is_even);

//set output to 1'b1 if input contains an even number of bits set to 1.

endmodule
```

(c)

```
module duplicator (input [1:0] a, b, output [511:0] r);

//set the output = abababab.......ab (512 bits). Hint: Use the replication operator

endmodule
```

[6] *(16pts)*

(a) Using RTL Verilog, recreate the functionality of the 74139 2-to-4 line decoder (see Learn@UW for a datasheet showing the 74139's truth table). Your module should use the following interface:

```
module decoder_2_to_4_139(input E, A0, A1, output O0, O1, O2, O3);
```

(b) Using structural Verilog and the 2-to-4 decoder you implemented in part (a), implement a 3-to-8 decoder. Unlike the 2-to-4 decoder, this decoder should use an active-high enable signal. Interface:

```
module decoder_3_to_8(input enable_hi, A0, A1, A2, output O0, O1, O2, O3, O4, O5, O6, O7)
```

[7] *(12 pts)*

Use RTL continuous assignment statements and the DFF module from Listing 1 to implement a 4-bit partial counter with *synchronous active-high* reset (You must make sure your reset is synchronous, not asynchronous! Do not modify the DFF module to achieve this.). The partial counter should reset to an output of 4'd3, count up to a value of 4'd12, and then go back to 4'd3 on the next cycle.

Write a testbench that shows your counter transitioning through all states. Use a $monitor statement to print the simulation time and current state for each transition. Make sure your $monitor statement is formatted to be clearly readable. Turn in the output generated by your testbench and a printout of the waveform along with all Verilog code.