**ECE 551 Homework #2**

**Due: Sept. 23rd @ 11:00 am**

***This homework assignment is to be completed individually.***

**[1]** *(6pts)*

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

**Delays allow us to make a simulation behave more like we expect it to in real life.**

**1. They represent propagation time of signal from inputs to outputs.**

**2. They help to model propagation delays of electrical elements.**

(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.

**XOR #5 XOR\_10[9:0] (out, in1, in2);**

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

**Advantage: Structural Verilog is more predictable. Because you specify the gates to use precisely, we can be more sure of the way the synthesizer will interpret modules.**

**Disadvantage: Because we have to describe our hardware in a more exacting way, it can be tedious, and involves more derivation of Boolean equations**.

**[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).

**Attached as Figure 1: Problem 2 – Part A – Schematic**

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

**Attached as Figure 2: Problem 2 – Part B – modulo\_6.v**

(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.

**Attached as Figure 3: Problem 2 – Part C – modulo\_6\_tb.v**

**Attached as Figure 4: Problem 2 – Part C – modulo\_6\_tb.v Wave**

**[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).

|  |  |  |  |
| --- | --- | --- | --- |
| Cycle | Input | Last 3 Inputs | Output |
| 1 | 0 | XX0 | 0 |
| 2 | 1 | X01 | 0 |
| 3 | 0 | 010 | 0 |
| 4 | 0 | 100 | 1 |
| 5 | 1 | 001 | 0 |
| 6 | 1 | 011 | 0 |
| 7 | 1 | 111 | 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).

**Attachments**

**Figure 5: Problem 3 – Finite State Machine Diagram**

**Figure 6: Problem 3 – State Equations**

**Figure 7: Problem 3 – Kmaps and Reduction**

**Figure 8: Problem 3 – prime\_detector.v**

**Figure 9: Problem 3 – prime\_detector\_tb.v**

**Figure 10: Problem 3 – prime\_detector\_tb.v Wave**

**[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 = 4'b0111

(b) 4’b0101 \* 4’b0011 = 8'b0000\_1111

(c) 4’b1100 ~^ 4’d7 = 4'b1100 ~^ 4'b0111 = 4'b0100

(d) {4{3’b101}} = 12'b1011\_0110\_1101

(e) ^8’b0010110 = 1'b1

(f) 6’b100000 && 6’b111111 = 1'b1

(g) !4’b0101 = 1'b0

(h) 6’b010110 >> 2 = 6'b000101

**[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.

assign big\_endian = {little\_endian[3:0],

little\_endian[7:4],

little\_endian[11:8],

little\_endian[15:12]};

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.

assign is\_even = ~^ in\_number

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

assign r = {256{a,b}};//concatenate a,b and replicate it 256 times for 512 bits

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);

**Attachments**

**Figure 11: Problem 6 – Part A – Boolean Logic**

**Figure 12: Problem 6 – Part A – decoder\_2\_to\_4\_139.v**

**Figure 13: Problem 6 – Part A – decoder\_2\_to\_4\_139\_tb.v**

**Figure 14: Problem 6 – Part A – decoder\_2\_to\_4\_139\_tb.v Wave**

(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);

**Attachments**

**Figure 15: Problem 6 – Part B – Truth Table and Schematic**

**Figure 16: Problem 6 – Part B – decoder\_3\_to\_8.v**

**Figure 17: Problem 6 – Part B – decoder\_3\_to\_8\_tb.v**

**Figure 18: Problem 6 – Part B – decoder\_3\_to\_8\_tb.v Wave**

**[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.

**Attachments**

**Figure 19: Problem 7 – FSM and Output Equations**

**Figure 20: Problem 7 – Next State Logic**

**Figure 21: Problem 7– partial\_counter.v**

**Figure 22: Problem 7 – partial\_counter \_tb.v**

**Figure 23: Problem 7 – partial\_counter\_tb.v Wave**

**Figure 24: Problem 7 – partial\_counter\_tb.v Monitor Output**