ECE 551   Homework #3

Due: Oct. 7th @ <u>11:00 am</u>

(Note: No late submissions after noon Oct. 8th for this homework)

<u>This homework assignment is to be completed individually.</u>

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

- Follow specific coding guidelines given in lecture (no #0 delays, no mixed blocking/non-blocking, etc.).
- When designing a non-testbench module, your code should be synthesizable.
- Use meaningful names for modules, ports, and wires when possible.
- Use underscores to break up long binary and hex numbers for readability.
- 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, and label them if necessary.
- *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] Improvising with Operators - *(4pts)*

The economic downturn has forced your employer to buy a discount Verilog simulator that does not support the logical operators "&&" and "||". Using ***bit-wise*** and/or ***reduction*** operators and a *single continuous assignment*, duplicate the functionality of the missing operators.

//Duplicates X = A && B

assign X =

//Duplicates Y = A || B

assign Y =

You should assume that X/Y are one-bit wires and A/B are multiple-bit vectors.

[2] Fun with Sensitivity Lists - *(10pts)*

(a) Fill in the sensitivity list for the following combinational logic (do not use *).

```
always@(                          )
begin
        case(a)
                2’b00 : d = x & y;
                2’b01 : d = x | z;
                default: d = ~b[3];
        endcase
end
```

(b) Fill in the sensitivity list for the following flip-flop with active-high synchronous Enable and active-low asynchronous Preset.

```
always@(                          )
begin
        if(preset_n  == 1’b0)
                q <= 1’b1;
        else if(enable == 1’b1)
                q <= d;
end
```

(c) Draw the hardware that might be inferred from the following description (neat hand-drawn circuits are acceptable). You may draw larger structures like muxes without showing their internal gate-level implementation. You may assume all signals are 1-bit wide.

```
always@(a,b,e)
begin
        d = ~e;
        if(a)
                c = ~b;
end
```

(d) Draw the hardware that might be inferred from the following description:

```
always@(a,b,e)
begin
        d = ~e;
        if(a)
                c = ~b;
        else
                c = e;
end
```

(e) Describe two different types of mistakes that can cause a latch to be unintentionally inferred when using Behavioral Verilog and give an original (i.e. not copied from homework or lecture slides) example of each.

[3] Basic Combinational and Sequential Logic (Behavioral Style) - *(12pts)*

Using only *behavioral* Verilog,

(a) Implement a JK flip-flop with asynchronous active-low reset.

(b) Implement the functionality of a 74HC147 priority encoder (see course website for datasheet with truth table) using the **casex** statement. Any ports that are active-low in the datasheet's truth table must also be active-low in your module. Use the port names shown in the datasheet.

[4] Synchronous Logic (Behavioral Style) – *(18pts)*

(a) Implement a register file consisting of 16 entries with an 8-bit register size, a single read port, and dual write ports. Your register file should have the following ports:

input clk	//Clock signal
input wr_en_1	//Active-high write enable for WR_ADDR_1
input wr_en_2	//Active-high write enable for WR_ADDR_2
input rst	//Asynchronous active-low reset (clears entire file)
input [7:0] data_in_1	//Input data for WR_ADDR_1
input [7:0] data_in_2	//Input data for WR_ADDR_2
input [2:0] wr_addr_1	//Write Address 1 can write to entries 0 ~ 7 of reg file
input [2:0] wr_addr_2	//Write Address 2 can write to entries 8 ~ 15 of reg file
input [3:0] rd_addr	//Read address can read entries 0 ~ 15
output reg [7:0] data_out	//Output data from RD_ADDR.

Your register's read function should be <u>asynchronous</u>. That is, data_out should change after you change rd_addr, without waiting for a clock edge.

(b) Create a testbench for the module designed in part (a). This testbench need not be exhaustive, but should demonstrate correct performance of all major functions (reading/writing to each entry, using write enable, etc). Use $display, $monitor, and/or $strobe commands to print the output of your testbench. You must also print messages labeling each test in the testbench.

Ex:

```
$display("Testing write enable:");
```

```
// Sequence of assignments/display statements to test write enable on/off

$display("Testing reset:");

// Sequence to test reset

//etc.
```

[5] FSM Control Logic (Behavioral Style) - *(20pts)*

(a) Using behavioral Verilog, implement a programmable up/down 3-bit counter. Your module should use the following interface:

```
module ud_counter(input clk, rst_count, load_en, mode, input [2:0]
upper_limit, output reg [2:0] count);
```

This module must contain a 3-bit register called `count_limit`. If `mode = 0`, your counter should count up to `count_limit`, go back to 0, and repeat. If `mode` = 1, it should count down to 0, go back to `count_limit`, and repeat. Changing `mode` should not reset your current count value.

If `load_en` = 1, your module should synchronously update the value of `count_limit` with the value of the `upper_limit` input. You may assume that `upper_limit` is unsigned and will never be set to 0.

`rst_count` is a synchronous, high-active reset signal that resets the current count to 0. It does not reset the value of `count_limit`.

(b) Write a testbench that demonstrates correct performance of your counter. Use a $monitor statement to print output. Appropriately label sections of your output (either using $display or by hand).

[6] Elements of Style - *(18pts)*

(a) Using Behavioral Verilog, re-implement the prime detector state machine you designed in Homework #2. Your Behavioral design should use **if/else** or **case** statements for the output and next state logic rather than attempting to duplicate the design on a gate level. You must also use **parameters** to name your states.

(b) Create a testbench that instantiates both the Behavioral and Structural versions of the state machine (If your Structural version from HW2 didn't work properly, you can use the design from the HW2 solution instead).

In your testbench include the following:

```
wire error;
assign error = (behavioral_output != structural_output);
```

Turn in a copy of a test waveform demonstrating that the two modules produce the same output. If there are any discrepancies between the output of the two modules, explain them.

(c) Briefly comment on your experience designing control logic with Structural Verilog versus Behavioral Verilog.

[7] The Horrors of the Stratified Event Queue - *(12pts)*

In this problem, we will examine the interaction of blocking and non-blocking assignment event timing within the Verilog Stratified Event Queue (If you haven't read the Cumming's SNUG2000 paper yet, you may find it worthwhile to do so).

Your job is to fill in the values in Table II. This table has six columns:

Time – The current simulation time.

Type – The type of event being processed by the simulator. We will be breaking events up into two types: Evaluations (**EVAL) –** evaluating the RHS of a Verilog statement, and Updates (**UP**) – changing the value of a variable on the LHS of a Verilog statement.

Statement – The Verilog statement that triggered the current event.

Result – The result of an evaluation if EVAL type or the value being assigned if UP type.

Values – The current value of all variables

Scheduled Events – Any new events that are added to the queue as the result of the current event. For example, every EVAL event should schedule a corresponding UP event. If an UP event triggers an always block, you should also list that here.

Table I gives the example data for Listing I. You must fill in Table II with the corresponding data for Listing II. **You should note that due to the nature of the Verilog Event Queue standard, you may encounter cases for which the order of events is non-deterministic.** In these cases, you are free to choose any event ordering that conforms to the standard.

Code Listing I: Sample code for Table I

For this sample, assume all variables start with an initial value of 0 rather than X.

```
initial begin
        x = 1;
        #3
        y <= #5 x;
        x = 2;
end
```

```
always@(x,y)
begin
        z = #1 2*x + y;
end
```

Time	Type	Statement	Result	Values	Scheduled Events
0	EVAL	1	1	x=0, y=0, z=0	UPDATE x=1 @ t=0
0	UP	x =	1	x=1, y=0, z=0	always@(x,y)
0	EVAL	2*x + y	2	x=1, y=0, z=0	UPDATE z=2 @ t=1
1	UP	z =	2	x=1, y=0, z=2	
3	EVAL	x	1	x=1, y=0, z=2	UPDATE y=1 @ t=8
3	EVAL	2	2	x=1, y=0, z=2	UPDATE x=2 @ t=3
3	UP	x =	2	x=2, y=0, z=2	always@(x,y)
3	EVAL	2*x + y	4	x=2, y=0, z=2	UPDATE z=4 @ t=4
4	UP	z =	4	x=2, y=0, z=4	
8	UP	y =	1	x=2, y=1, z=4	always@(x,y)
8	EVAL	2*x + y	5	x=2, y=1, z=4	UPDATE z=5 @ t=9
9	UP	z =	5	x=2, y=1, z=5	

Table I: Sample data for Listing I

Code Listing II: Sample code for Table II

For this sample, assume all variables start with an initial value of 0 rather than X.

```
initial begin
        a = 5;
        #10
        b = #2 a + 3;
        #5
        c <= d - b;
end

always@(a,b)
begin
        #7
        d <= #20 (2 * a);
        e = d;
end

always@(c,d)
         s <= #3 a + b + c + d + e;
```

Time	Type	Statement	Result	Values	Scheduled Events
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	
				a= b= c= d= e= s=	

Table II: Event processing timeline for Code Listing II