**ECE 551 Homework #3**

**Due: Oct. 7th @ 11:00 am**

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

**[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*** operators and a *single continuous*

*assignment*, duplicate the functionality of the missing operators.

//Duplicates X = A && B

assign X = **(|(A)) & (|(B))**

//Duplicates Y = A || B

assign Y = **(|(A)) | (|(B))**

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@( **x, y, z, b, a**)

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@( **negedge preset\_n,**  **posedge clk**)

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.

**One way to infer a latch is to have an if that modifies a value, and not have an else/default that modifies the value otherwise. In the below example, c is given a specific value if the if is true. We should really have an else value.**

**Ex.**

**Always @(d) begin**

**If( d)**

**C <= 1’b0;**

**End**

**Another way to infer a latch is to have cases in a case statement. For the below case, we have multiple cases for we do not know what to do…**

**Ex.**

**Always @(\*) begin**

**Case (D)**

**3’b000: a = 1’b1;**

**3’b111: a = 1’b0;**

**endcase**

**end**

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

Using only *behavioral* Verilog,

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

**Attached as Figure 1: JKFF.v**

(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.

**Attached as Figure 2: priority\_encoder\_10\_to\_4\_74HC.v**

**[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 asynchronous. That is, data\_out should**

**change after you change rd\_addr, without waiting for a clock edge.**

**Attached as Figure 3: register\_file\_16.v**

(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.

**Attached as Figure 4: register\_file\_16\_tb.v**

**Attached as Figure 5: register\_file\_16\_transcript.txt**

**[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.

**Attached as Figure 6: ud\_counter.v**

(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).

**Attached as Figure 7: ud\_counter\_tb.v**

**Attached as Figure 8: ud\_counter\_tb\_transcript.txt**

**[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.

**Attached as Figure 9: prime\_detector\_behavioral.v**

(b) Create a testbench that instantiates both the Behavioral and Structural versions of the state

Machine.

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.

**Attached as Figure 10: prime\_detector\_behavioral\_structural\_tb.v**

**Attached as Figure 11: prime\_detector\_behavioral\_structural\_tb Wave**

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

Behavioral Verilog.

**Structural Verilog is very exacting, but tedious. Behavioral is very powerful and requires you to pay much closer attention. For this prime detector, it is much easier to program it in Behavioral Verilog. I think that unless you need your hardware to use very specific gates, Structural Verilog is unnecessary.**

**[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.

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)** and Updates (**UP**).

*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.

*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 |
| 0 | EVAL | 5 | 5 | a=0 b=0 c=0 d=0 e=0 s=0 | UP a=5 @ t=0 |
| 0 | UP | A= | 5 | a=5 b=0 c=0 d=0 e=0 s=0 | Always @(a,b) |
| 7 | EVAL | 10 | 10 | a=5 b=0 c=0 d=0 e=0 s=0 | UP D=10 @ t=27 |
| 7 | EVAL | 0 | 0 | a=5 b=0 c=0 d=0 e=0 s=0 | UP E=0 @ t=7 |
| 7 | UP | E= | 0 | a=5 b=0 c=0 d=0 e=0 s=0 |  |
| 10 | EVAL | 5+3 | 8 | a=5 b=0 c=0 d=0 e=0 s=0 | UP B=8 @ t=12 |
| 12 | UP | B= | 8 | a=5 b=8 c=0 d=0 e=0 s=0 | Always @(a,b) |
| 17 | EVAL | 0-8 | -8 | a=5 b=8 c=0 d=0 e=0 s=0 | UP C=-8 @ t=17 |
| 17 | UP | C= | -8 | a=5 b=8 c=-8 d=0 e=0 s=0 | Always @(c,d) |
| 17 | EVAL | 5+8-8+0+0 | 5 | a=5 b=8 c=-8 d=0 e=0 s=0 | UP S=5 @ t=20 |
| 19 | EVAL | 10 | 10 | a=5 b=8 c=-8 d=0 e=0 s=5 | UP D=10 @ t=39 |
| 19 | EVAL | 0 | 0 | a=5 b=8 c=-8 d=0 e=0 s=5 | UP E=0 @ t=19 |
| 19 | UP | E= | 0 | a=5 b=8 c=-8 d=0 e=0 s=5 |  |
| 20 | UP | S= | 5 | a=5 b=8 c=-8 d=0 e=0 s=5 |  |
| 27 | UP | D= | 10 | a=5 b=8 c=-8 d=10 e=0 s=5 | Always @(c,d) |
| 27 | EVAL | 5+8-8+10 | 15 | a=5 b=8 c=-8 d=10 e=0 s=5 | UP S=15 @t=30 |
| 30 | UP | S= | 15 | a=5 b=8 c=-8 d=10 e=0 s=15 |  |
| 39 | UP | D= | 10 | a=5 b=8 c=-8 d=10 e=0 s=15 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Table II: Event processing timeline for Code Listing II**