**Name:**

**CSCE A342**

**Assignment #3 (100 points)**

**Due 2/27/18 (Tuesday)**

**T-Bird Taillight on the BASYS3**

**The T-Bird taillight is a lab assignment in CSCE 241 that implements, via with chips and wires, a sequential taillight controller that may be used to control the taillights of a 1957 Ford Thunderbird. In this assignment, we will revisit that lab by recreating the T-Bird taillight controller in System Verilog and your BASYS3 board. Chips and wires no more!**

**As a reminder, here is a video that demos the sequential T-Bird taillights:**

[**https://www.youtube.com/watch?v=Qwzxn9ZPW-M**](https://www.youtube.com/watch?v=Qwzxn9ZPW-M)

**We will now describe the desired functionality of out T-Bird taillight circuit on the BASYS3**

The left turn signal should be assigned to the leftmost switch, sw[15]

The right turn signal should be assigned to the rightmost switch, sw[0]

The brake signal should be assigned to the center button, btnC

The reset signal should be assigned to the upper button, btnU

The three leftmost and three rightmost leds on the BASYS3 should behave as follows (where ON and OFF are denoted by white and black circles, respectively). The remaining leds should always remain off.

1. Brake off, no turn signals:

b. Brake on, no turn signals:

c. Left turn signal, brake off: (1)

(2)

(3)

(4)

and repeat…

d. Left turn signal, brake on: (1)

(2)

(3)

(4)

and repeat…

e. Right turn signal, brake off: (1)

(2)

(3)

(4)

and repeat…

f. Right turn signal, brake on: (1)

(2)

(3)

(4)

and repeat…

As you can see, a turn signal is indicated by a sequence of four states that turn on one, two, three, and zero lights in the direction of the turn. This sequence should not be interrupted by the brake signal, which should cause all of the taillights not involved in the turn signal to turn ON *asynchronously (i.e. immediately)*. On the other hand, a turn signal sequence should be suspended immediately if both turn signals are switched off.

In designing your circuit, you may assume that the LEFT and RIGHT turn signals will never be asserted simultaneously (i.e. sw[0] and sw[15] will never both be switched on simultaneously).

Before proceeding, take a few minutes and consider the modules you would make to create this in System Verilog.

**Step #1:**

The design sources for this project should consist of the following four files:

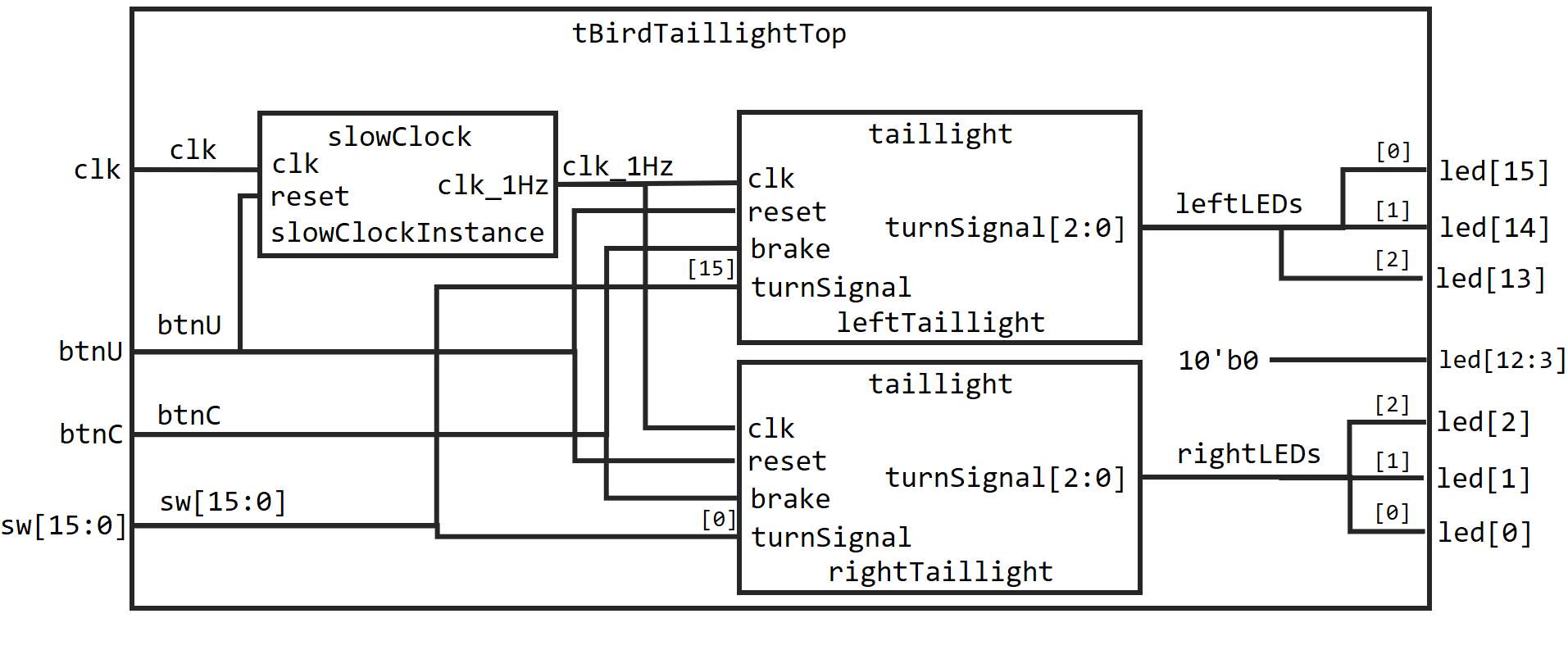
* tBirdTaillightTop.sv
* slowClock.sv
* taillight.sv
* blinkers.sv

tBirdTaillightTop.sv

A schematic of how the tBirdTaillightTop module should be designed is provided below. Each box is an instance of a module.

Note how the taillight module is instantiated twice, leftTaillight and rightTaillight. We will create the functionality for a taillight in the taillight module, and then use this module for both the left and right taillights.

The slowclock module is used to turn the 100 MHz clock into a 1 Hz signal.



Fill in the commented lines below to finish the tBirdTaillight module

module tBirdTaillightTop(

input logic clk,

input logic [15:0] sw,

input logic btnC,

input logic btnU,

output logic [15:0] led

);

logic clk\_1Hz;

logic [2:0] leftLEDs;

// declare a three-bit logic bus called rightLEDs (similar to leftLEDs)

slowClock slowClockInstance(.clk(clk), .reset(btnU), .clk\_1Hz(clk\_1Hz));

taillight leftTaillight(.clk(clk\_1Hz), .reset(btnU), .brake(btnC), .turnSignal(sw[15]), .taillights(leftLEDs));

// instantiate the rightTaillight, similar to the leftTaillight above

assign led[15:13] = {leftLEDs[0], leftLEDs[1], leftLEDs[2]};

assign led[12:3] = 10'b00000\_00000;

// assign led[2:0] to rightLEDs, similar to led[15:13] above

endmodule

slowClock.sv

We have not covered arithmetic operations yet, so I am giving you the code for generating a slow 1 Hz clock from a 100 MHz clock

Note how this clock can be asynchronously reset because posedge clk is in the sensitivity list

module slowClock( input logic clk,

input logic reset,

output logic clk\_1Hz);

logic [32:0] counter;

always\_ff @ (posedge reset or posedge clk)

begin

if (reset == 1'b1)

begin

clk\_1Hz <= 0;

counter <= 0;

end

else

begin

if ( counter == 25\_000\_000)

begin

counter <= 0;

clk\_1Hz <= ~clk\_1Hz;

end

else

counter <= counter + 1;

end

end

endmodule

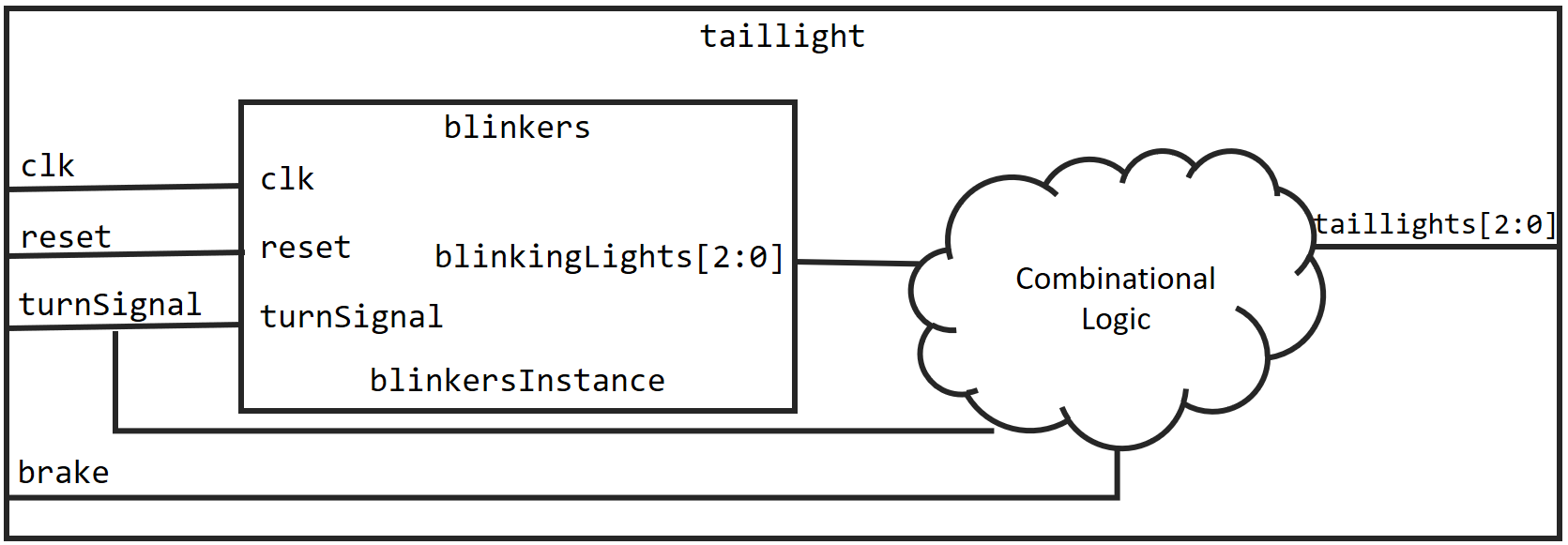
taillight.sv

The taillight module will accept a 1 Hz clock, reset, a brake signal, and a turn signal.

If the turn signal is on, then the taillight should blink using the output from the blinkers module

Otherwise, if the brake is on, all taillights should light up, and if it is not on, then all taillights should be off

The reset signal should be used to reset the blinkers state machine module.



Fill in the combinational logic in the commented section below to complete taillight.sv

module taillight(

input logic clk,

input logic reset,

input logic brake,

input logic turnSignal,

output logic [2:0] taillights

);

logic [2:0] blinkingLights;

blinkers blinkersInstance( .clk(clk), .reset(reset), .turnSignal(turnSignal), .blinkingLights(blinkingLights));

always\_comb

begin

// if turn signal is on, then use blinking lights for taillights

// otherwise

// if the brakes are on, light up the taillights

// otherwise, the taillights should be off

end

endmodule

blinkers.sv

The blinkers module is a finite state machine with a clock input, a reset, and a turn signal input (turn signal is a single bit)

The output of the blinkers module is a three bit bus of sequentially blinking lights.

The FSM has 4 states and a single input bit representing the turn signal being on/off. Each of these four states represents the various four phases of the light cycle as well as if the blinker is on.

Draw the state transition diagram for this FSM and label each state. Also label what the three bit output should be in each state.

Draw state transition diagram here:

Complete taillight.sv using the state transition diagram you just drew as a guide:

module blinkers( input logic clk,

input logic reset,

input logic turnSignal,

output logic [2:0] blinkingLights );

// typedef statements

statetype state, nextstate;

always\_ff @( posedge clk, posedge reset)

begin

if( reset )

// set state to turn signal off state

else

state <= nextstate;

end

always\_comb

begin

case(state)

// logic for determining next state

endcase;

end

// Assign blinkingLights based on the current state

// the most significant bit of blinkingLights should light first

assign blinkingLights[2] = // fill in remainder of line

assign blinkingLights[1] = // fill in remainder of line

assign blinkingLights[0] = // fill in remainder of line

endmodule

**Step #2: Simulate**

Simulate the taillight.sv module using the following testbench

module blinkersTestbench();

logic clk, reset, turnSignal;

logic [2:0] blinkingLights , blinkingLightsExpected;

logic [31:0] vectornum, errors;

logic [4:0] testvectors[10000:0];

blinkers blinkersInstance( .clk(clk), .reset(reset), .turnSignal(turnSignal), .blinkingLights(blinkingLights) );

always

begin

clk=0; #5; clk=1; #5;

end

initial

begin

$readmemb("blinkers.tv", testvectors);

vectornum = 0; errors = 0;

end

//apply test vector inputs on falling edge of clock

always @(negedge clk)

begin

#4;

{reset,turnSignal,blinkingLightsExpected} = testvectors[vectornum];

end

//check results after rising edge of clock

always @(posedge clk)

begin

#1; //wait slight after rising edge

if( blinkingLights !== blinkingLightsExpected ) //check result

begin

$display("Error: inputs=%b", {reset,turnSignal});

$display(" outputs=%b%b%b (%b%b%b expected)", blinkingLights[2], blinkingLights[1], blinkingLights[0], blinkingLightsExpected[2], blinkingLightsExpected[1], blinkingLightsExpected[0]);

errors = errors + 1;

end

vectornum = vectornum + 1;

if( testvectors[vectornum] === 5'bx) //check for end of file

begin

$display("%d tests completed with %d errors", vectornum, errors);

$finish;

end

end

endmodule

Task 1: Finish writing the blinkers.tv test vector file so that every transition from the state diagram has occurred.

In each line, the first bit corresponds to the reset bit, and the second bit corresponds to the turnSignal bit.

The last three bits correspond to the blinkingLights output.

blinker.tv

10\_000 // reset blinkers FSM, all bits of blinkingLights turn off

01\_100 // turn signal on, so most significant bit of blinkingLights turns on

01\_110 // turn signal remains on, two most significant bits of blinkingLights are on

01\_111 // turn signal remains on, all bits of blinkingLights are on

01\_000 // turn signal remains on, all bits of blinkingLights turn off

01\_100 // turn signal on, so most significant bit of blinkingLights turns on

//continue checking the FSM so that \*all\* the transitions have been enumerated, the above is just to help you get a start

Task 2: Simulate the blinkers module with your test vector. Does it behave as expected? Take a screenshot of your simulation.

**Step #3: Synthesize and Implement**

Download the Basys3\_Master.xdc constraint file and add it to your project.

Uncomment the lines corresponding to clk, sw[15:0] (all switch lines), led[15:0] (all leds), btnC (center button), and btnU (upper button). Beyond uncommenting, you do not need to change the contents of this file

**Synthesize and implement your design. Does it behave as expected?**

**What to turn in:**

* completed tBirdTaillightTop.sv
* completed taillight.sv
* completed blinkers.sv
* completed blinkers.tv
* screenshot of your simulation

These files can be turned in via the blackboard system.

* blinkers FSM state transition diagram

This can be turned in on paper or via the blackboard system.

I also need to verify the implementation, so please bring your FPGA with the T-bird controller to the 2/27 lecture

**Ungraded bonus:**

**Augment your design so that one of the switches is an “emergency signal” that causes both taillights blink in some way of your choosing.**