Andrew Scott
E155
Lab 3
September 28, 2015

Lab 3: Keypad Scanner

Introduction

In this lab, I added a keypad to the breadboard circuit I had before. I then wrote Verilog code to scan the keyboard and see if any keys were pressed and then display the last two pressed keys on the seven segment display.

Design and Testing Methodology

The difficulty in this lab was mostly in the Verilog code, so there were little design choices to make for the hardware/breadboard. I added a reset switch to attach to the reset signal in my hardware (I didn't have a breadboard switch before). Also, I connected the column pins from my keypad to pull down resistors so that the utility board pins would see valid logic levels when the voltage on the keypad pins were low. I tested the keypad with a multimeter to check that the pins were becoming connected when a key was pressed, and then I tested the whole circuit by making sure everything worked after loading my Verilog code onto the FPGA.

The software side of this lab required many design decisions that were not easy. I decided to check which keys were pressed down by polling the rows one at a time and then checking the output from the columns to see if any of them were high. To accomplish this, I used a finite state machine whose states were the row currently being polled. The state machine then switched to the new state when a counter register filled up. I made the counter 13 bits long, which makes the row poller switch to a new row every 0.0002 seconds, which was plenty fast to catch any key presses. I then had to decide how to handle the key bounce problem. The solution I came up with was another finite state machine. This FSM has four states - no press, current press, recent press and ending press. The first two states are self explanatory. The recent press state is reached from the current press state when there's no key currently pressed, and this transition also resets a counter variable. The FSM then stays in the recently pressed state until the counter variable fills up, and then it goes to the ending press state. The ending press state is used to send a write enable signal to the registers holding the values that are displayed on the seven segment display, and then transitions back to the no press or currently pressed state depending on if a key is pressed. The counter to check when to leave the recent press state is 21 bits long. This means that if a key is not pressed down for just over 0.05 seconds (which is longer than any key bounce will last for) the next press will register as a new press. This time is short enough that nobody can press a key twice and have it register as only one press. The bounce checking FSM transitions between states every time an 11 bit counter fills up. This is shorter than the counter for the row poller FSM to ensure that multiple changes can happen while a single row is being pressed. Finally, I added a two register synchronizer to make sure that the column signal had time to resolve itself if a keypad button was released on a clock edge.

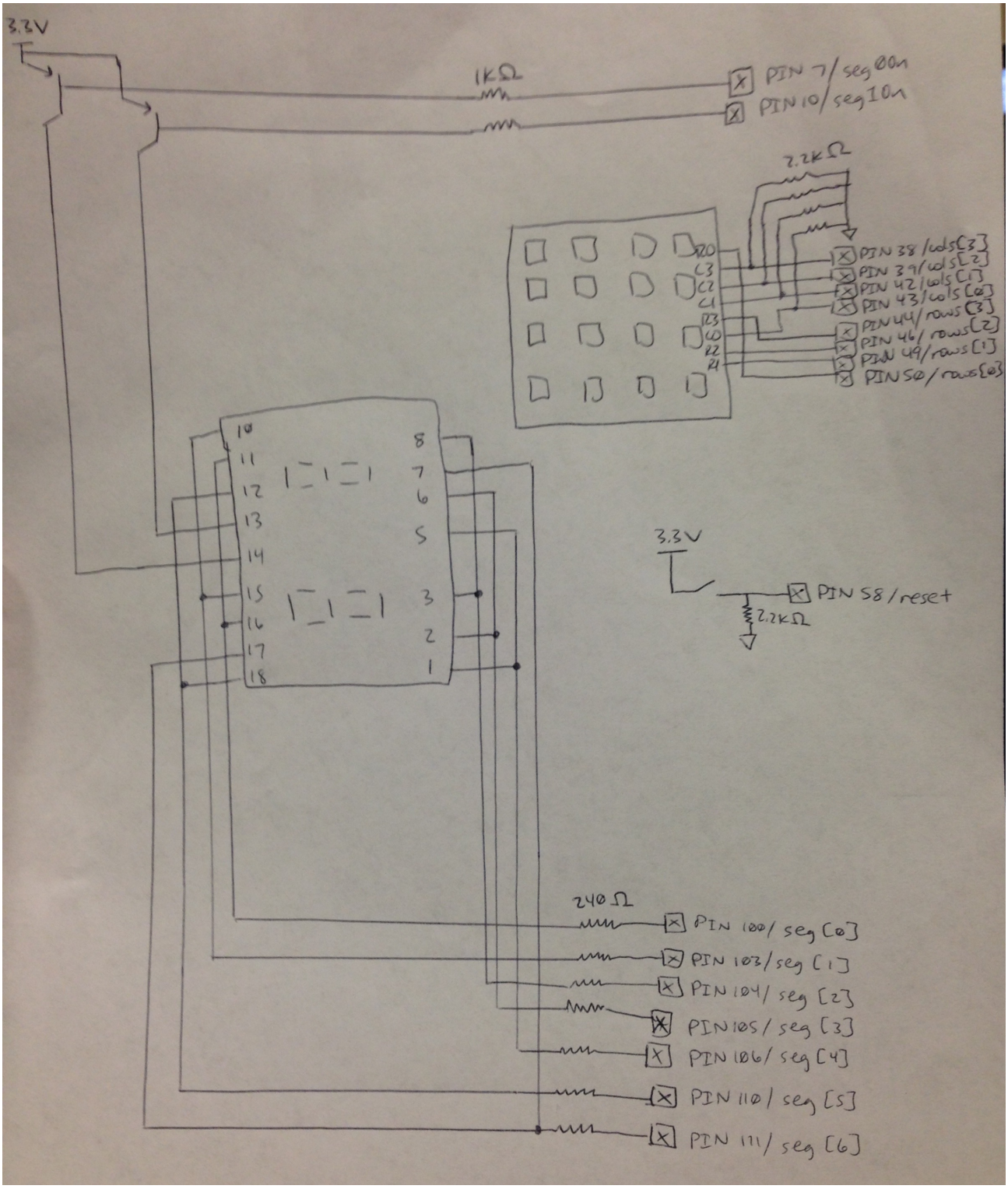
Results and Discussion

I was able to finish this entire lab, and everything appears to be working as it is supposed to. I am happy that I simulated my hardware before testing on the breadboard because I noticed some timing issues in my circuit that I was able to fix.

Conclusion

I spent about 9 hours on this lab.

Breadboard Schematic



Verilog Code

```
/*
Andrew Scott
ascott@hmc.edu
Created on September 25, 2015

This is the top level module for e155 lab 3. It scans a keypad by polling the
rows of the keypad one at a time and reading the columns, and then outputs the
last two pressed keys read to be displayed on a seven segment display.
*/
module lab3_as(input logic clk, reset,
            input logic [3:0] cols,
            output logic [3:0] rows,
            output logic [6:0] seg,
            output logic seg0On, seg1On);

logic newPress;
logic [3:0] newPressNumber, newNumber, oldNumber;
logic [4:0] keyInput;
input_scanner scanner(clk, reset, cols, rows, keyInput);
bounce_checker bounceChecker(clk, reset, keyInput, newPress, newPressNumber);
werFlop #(4) newNumReg(clk, reset, newPress, newPressNumber, newNumber);
werFlop #(4) oldNumReg(clk, reset, newPress, newNumber, oldNumber);

seven_seg_controller segController(clk, reset, oldNumber, newNumber, seg,
                    seg0On, seg1On);
endmodule


/*
Andrew Scott
ascott@hmc.edu
Created on September 25, 2015

Testbench module that instantiates a clock. The circuit can then be tested
by forcing cols to different values.
*/
module testbench();
logic clk, reset, seg0On, seg1On;
logic [3:0] rows, cols;
logic [6:0] seg;

lab3_as labSimulator(clk, reset, cols, rows, seg, seg0On, seg1On);

// generate clock with 5 10 period
always
begin
  clk <= 1; # 5; clk <= 0; # 5;
end
endmodule


/*
Andrew Scott
ascott@hmc.edu
Created on September 25, 2015

Module that polls the rows one at a time and checks the columns to check if they
are high. It also has a decoder to figure out what key is pressed if a column
is high.
*/
module input_scanner(input logic clk, reset,
                    input logic [3:0] cols,
                    output logic [3:0] rows,
                    output logic [4:0] keyInput);
```

```
row_poller poller(clk, reset, rows);
input_decoder decoder(clk, reset, cols, rows, keyInput);
endmodule


/*
Andrew Scott
ascott@hmc.edu
Created on September 21, 2015

Decoder to determine which key is currently being pressed. If one row bit and
come column bits are high, bit 4 is set high (meaning valid) and the lower bits
hold the largest value from the keypad in that row that is high (if 1 and 2 are
pressed at the same time, the system will read this as 2 being pressed). If no
keys are pressed, the output will be all zeros.
*/
module input_decoder(input logic clk, reset,
                     input logic [3:0] cols, rows,
                     output logic [4:0] keyInput);
logic [3:0] cols1, cols2, rows1, rows2;
rFlop #(4) rowReg1(clk, reset, rows, rows1);
rFlop #(4) rowReg2(clk, reset, rows1, rows2);
rFlop #(4) synchReg1(clk, reset, cols, cols1);
rFlop #(4) synchReg2(clk, reset, cols1, cols2);
always_comb
casez({rows2, cols2})
    // top row
    8'b00010001 : keyInput = 5'b10001;  // 1
    8'b0001001? : keyInput = 5'b10010;  // 2
    8'b000101?? : keyInput = 5'b10011;  // 3
    8'b00011??? : keyInput = 5'b11010;  // A
    // second row
    8'b00100001 : keyInput = 5'b10100;  // 4
    8'b0010001? : keyInput = 5'b10101;  // 5
    8'b001001?? : keyInput = 5'b10110;  // 6
    8'b00101??? : keyInput = 5'b11011;  // B
    // third row
    8'b01000001 : keyInput = 5'b10111;  // 7
    8'b0100001? : keyInput = 5'b11000;  // 8
    8'b010001?? : keyInput = 5'b11001;  // 9
    8'b01001??? : keyInput = 5'b11100;  // C
    // last row
    8'b10000001 : keyInput = 5'b11110;  // E
    8'b1000001? : keyInput = 5'b10000;  // 0
    8'b100001?? : keyInput = 5'b11111;  // F
    8'b10001??? : keyInput = 5'b11101;  // D

    8'b????0000 : keyInput = 5'b00000;  // nothing pressed
    default : keyInput = 5'b01111;      // uh oh... row poller is failing
endcase
endmodule


/*
Andrew Scott
ascott@hmc.edu
Created on September 25, 2015

Finite state machine that puts one row pin high at a time, cycling through the
rows.
*/
module row_poller(input logic clk, reset,
                  output logic [3:0] rows);
logic[3:0] state, next;
logic [12:0] count;

// state count register (we update the state when count gets to a certain value)
always_ff @(posedge clk, posedge reset)
```

```
    if (reset)
        count <= 13'b0;
    else
        count <= count + 1'b1;

// state register (updates when count is all 1's)
always_ff @(posedge clk, posedge reset)
    if (reset)
        state <= 4'b0;
    else if (&count)
        state <= next;


// logic to derive next state
always_comb
    case(state)
        4'b0001 : next = 4'b0010;
        4'b0010 : next = 4'b0100;
        4'b0100 : next = 4'b1000;
        4'b1000 : next = 4'b0001;
        default : next = 4'b0001;
    endcase

// output logic
assign rows = state;
endmodule


/*
Andrew Scott
ascott@hmc.edu
Created on September 21, 2015

Module that takes input from the input_decoder and then uses a finite state
machine to check if a pressed key is a new press or is part of a bounce from
a previous press.
*/
module bounce_checker(input logic clk, reset,
                      input logic [4:0] keyInput,
                      output logic newPress,
                      output logic [3:0] newPressNumber);
typedef enum logic[1:0]{NoPress = 2'b00, CurrentPress = 2'b01,
RecentPress = 2'b10, EndingPress = 2'b11} statetype;
statetype state, next;
logic [20:0] bounceCount;
logic [10:0] stateCount;
logic bounceCountReset;

// bounce count register
always_ff @(posedge clk, posedge reset)
    if (reset)
        bounceCount <= 0;
    else if (bounceCountReset)
        bounceCount <= 0;
    else
        bounceCount <= bounceCount + 1'b1;

// state count register (we update the state when count gets to a certain value)
always_ff @(posedge clk, posedge reset)
    if (reset)
        stateCount <= 0;
    else
        stateCount <= stateCount + 1'b1;

// state register (updates when count is all 1's)
always_ff @(posedge clk, posedge reset)
    if (reset)
        state <= NoPress;
```

```
    else if (&stateCount)
        state <= next;

// logic to derive next state
always_comb
    case(state)
        NoPress : next = keyInput[4] ? CurrentPress : NoPress;
        CurrentPress : next = keyInput[4] ? CurrentPress : RecentPress;
        RecentPress : next = keyInput[4] ? CurrentPress : (bounceCount[21] ? EndingPress : RecentPress);
        EndingPress : next = keyInput[4] ? CurrentPress : NoPress;
        default : next = NoPress;
    endcase

// output logic
assign newPress = state == EndingPress && stateCount == 11'b0;
werFlop #(4) keyInputReg(clk, reset, keyInput[4], keyInput[3:0], newPressNumber);
assign bounceCountReset = state == CurrentPress;
endmodule


/*
Andrew Scott
ascott@hmc.edu
Created on September 21, 2015

A simple write-enable, asynchronously resettable flip flop. This means that
whenever reset is high the flop's value will go to 0, and otherwise it will write
to the register on the rising edge of the clock whenever the enable signal is high.
*/
module werFlop #(parameter WIDTH = 8)
                (input logic clk, reset, enable,
                input logic [WIDTH - 1:0] d,
                output logic [WIDTH - 1:0] q);
always_ff @(posedge clk, posedge reset)
    if (reset)
        q <= 0;
    else if (enable)
        q <= d;
endmodule


/*
Andrew Scott
ascott@hmc.edu
Created on September 21, 2015

A simple  asynchronously resettable flip flop. This means that
whenever reset is high the flop's value will go to 0, and otherwise it will write
to the register on the rising edge of the clock.
*/
module rFlop #(parameter WIDTH = 8)
                (input logic clk, reset,
                input logic [WIDTH - 1:0] d,
                output logic [WIDTH - 1:0] q);
always_ff @(posedge clk, posedge reset)
    if (reset)
        q <= 0;
    else
        q <= d;
endmodule


/*
Andrew Scott
ascott@hmc.edu
Created on September 25, 2015

This module controls a two digit seven segment display with a single seven
```

```
segment display module, switching back and forth between which digit is being
controlled quicker than the human eye can detect.
*/
module seven_seg_controller(input logic clk, reset,
                            input logic [3:0] oldNumber, newNumber,
                            output logic [6:0] seg,
                            output logic seg0On, seg1On);

logic [17:0] count;
logic switchSelect;
logic [3:0] switchIn;

// time mux counter
always_ff @(posedge clk, posedge reset)
    if (reset)
        count <= 0;
    else
        count <= count + 1'b1;

assign switchSelect = count[17];
assign switchIn = switchSelect ? oldNumber : newNumber;
assign seg0On = switchSelect;
assign seg1On = ~switchSelect;
seven_seg_disp display(switchIn, seg);
endmodule


/*
Andrew Scott
ascott@hmc.edu
Created on September 14, 2015

This module contains the logic for controlling a common anode seven segment
display with a four bit input. The module uses a truth table  to determine when
to light up each segment of the display.
*/
module seven_seg_disp(input logic [3:0] s,
                      output logic [6:0] seg);

always_comb
case(s)
    4'b0000 : seg = 7'b1000000;
    4'b0001 : seg = 7'b1111001;
    4'b0010 : seg = 7'b0100100;
    4'b0011 : seg = 7'b0110000;
    4'b0100 : seg = 7'b0011001;
    4'b0101 : seg = 7'b0010010;
    4'b0110 : seg = 7'b0000010;
    4'b0111 : seg = 7'b1111000;
    4'b1000 : seg = 7'b0000000;
    4'b1001 : seg = 7'b0011000;
    4'b1010 : seg = 7'b0001000;
    4'b1011 : seg = 7'b0000011;
    4'b1100 : seg = 7'b0100111;
    4'b1101 : seg = 7'b0100001;
    4'b1110 : seg = 7'b0000110;
    4'b1111 : seg = 7'b0001110;
    default : seg = 7'b0000000;
endcase
endmodule
```