EGR 224: Introduction to Digital System Design

Lab Section 901

Fall 2020

Instructor: Prof. Zuidema

Final Project

Digital Lock

Dustin Matthews

12/11/2020

Contents

List of Figures

1. Objectives

The objective of the final project is to implement a digital lock design using the PMOD keypad and LCD with the BASYS3 board, which requires integrating several modules together into one project and communicating between them.

2. Equipment

Part	Description	Model	Measured Value	Notes
Xilinx Vivado 2019.2	Digital circuit design software	n/a	n/a	n/a
Digilent BASYS-3 FPGA Board	Circuit Development Board	BASYS-3	n/a	n/a
PmodLCD	16x2 LCD	n/a	n/a	n/a
PmodKYPD	4x4 Keypad	n/a	n/a	n/a

3. Introduction

To mimic a digital lock, numbers from the keypad are logged and compared to a stored PIN value, which if matching will display “Door is unlocked” if the PIN matches or “Invalid PIN” if the PIN does not match. The key pressed is additionally displayed on the seven-segment LEDs of the Basys, though this is not entirely necessary and was included largely for troubleshooting purposes.

4. Procedure

Module Development

Several new modules were developed and some were modified from earlier iterations to complete this project. The overall schematic can be viewed in Appendix A.

4.1 Top Level Module

The top level module hosts all the modules required for the design as well as connecting wires and registers. It is essentially the communication hub between all lower level modules. See TOP LEVEL MODULE in Appendix B. The entire top module was copied with short dummy values for delays and the clock dividers were commented out in order to verify that the LCD and Keypad modules would communicate properly.

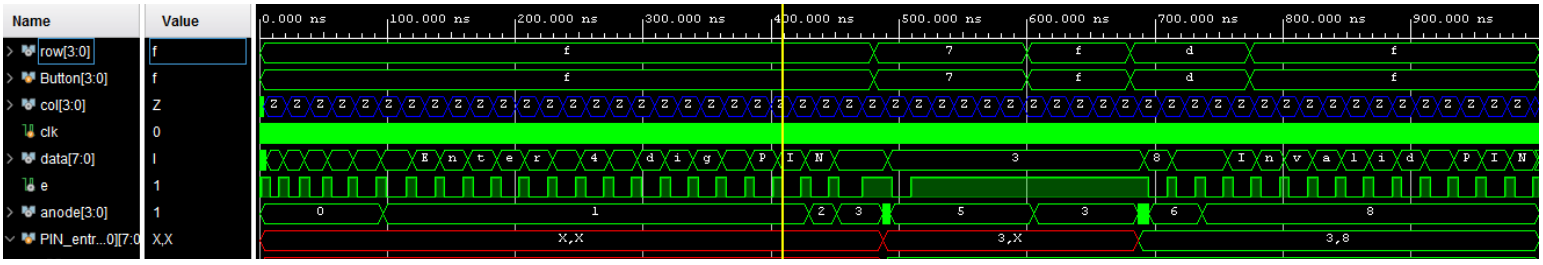


Figure 1 Top Level Simulation

4.2 Clock Dividers

Two clock dividers were used for the project- a 1KHz divider and a 1MHz divider. The 1KHz is solely used for the modules relating to the seven segment display to cycle the anodes. All other modules run off the 1MHz clock divider to facilitate a synchronous communication between all modules. The clock dividers count up to a specified value at each rising clock edge and toggle their outputs every time this value is reached. See Clock Divider 1 MHz and Clock Divider 1KHz in Appendix B.

4.3 Debounce Modules

Two debounce modules were used in this project, one for the reset button and one for the Keypad Flag, which is discussed in section 4.6.

The reset debounce uses four ANDed flip flops (Reset Debounce Module), but the keypad flag debounce Key Flag Debounce Module) was slightly more complex and was the source of many problems on this project. Normally a button can be debounced for several milliseconds, however the developer was unable to get a solid output with a debounce timer larger than roughly 900 microseconds, just under 1ms. It is hypothesized that this may be due to the nature of the keypad code developed wherein the columns shift roughly every 1001 microseconds, resulting in a somewhat flickering output.

The basic operation of the keypad debounce is that the keypad flag is synchronized to the clock with two flip flops and compared to the current output state of the keypad flag debouncer. If the output does not match the current input, a counter starts. If the counter reaches a maximum value, the output is toggled.

This current design works ok, and is able to output key presses to the LCD, but if a key is held down then it will continue to print the key until 4 digits have been received at a rate of roughly .5 seconds. This interval is likely due to the buffer in the LCD control module, which is discussed further below.

4.4 Two Bit Counter

The next module constructed was the 2-bit counter, which receives its input from the 1KHz clock divider. The output of the counter is used to drive the select bits of the multiplexers, effectively allowing them to cycle through the 7-segments at 1KHz without any flicker. The basic idea behind the counter is very simple; a count variable is incremented at each rising clock edge, and since the output is only 2 bits, it is truncated back to 0b00 each time the count exceeds 0b11.The code for the counter can be viewed in Counter 2-bit Module.

4.4 Seven Segment Decoder

The seven segment decoder module (Seven Segment Module) uses a switch case for its main operations. Taking as input a 4-bit value, it outputs a 7-bit value to drive the seven segments of the LED. A truth table constructed to determine the output values based on inputs- the max value of the input being 0b1111, the highest output would be the hex value of ‘f’. See the table in Appendix C.

Additional verification of the Seven Segment Decoder was done in simulation using tasks to verify that output would match the input. The test bench file can be viewed in Seven Segment Decoder Task Test bench and the console output can be viewed in Appendix E.

4.5 Anode Counter

The anode counter is a simple multiplexer module using a switch case. It takes a 2-bit select input and outputs a 4-bit number depending on the value of the select bits. The output of the anode module is used to determine which of the 7-segments is active at any given rising clock edge of the 1KHz divider. The code for the Anode multiplexer can be viewed in Anode Counter Module in Appendix B.

4.6 Keypad Decoder

The basic architecture of the Keypad Decoder module consists of several procedural blocks, with the second containing a switch case to set the column values. The design is modified from the version developed in lab to use the concatenate function to shift the columns. A delay flag is used to change the state at specified times, with the main three states being a set column state, a read row state, and a shift column state.

Row and column in the Keypad Decoder submodule are declared as inout, allowing them to be both driven and read. A secondary variable was declared for the Columns, allowing value and impedance to be assigned to them.

Additional procedural blocks manage the decoding portion of the module in a similar fashion to what can be done in software. This requires some additional variables – a 2 bit column tracker that indicates which column is active, and a subtract variable. The subtract variable is dependent on which row is active.

```
/*** Variable used to determine key index ****/
assign sub = (row == 4'b1110) ? 2 :
             (row == 4'b1101) ? 5 :
             (row == 4'b1011) ? 7 :
             (row == 4'b0111) ? 7 :
                                0 ;
```

Figure 2 Subtract Variable

The Key index will determine what key value the module outputs, and the key index is determined by taking the adding the active row value to the 2 bit column tracker variable, and subtracting the sub variable. The result will be a number from 0 -15, the key index.

```
/**** BUTTON READER BLOCKS ****/
//Determine index
always @ (posedge clk) begin
    if  ((state == stateRead) && (row != 4'b1111))
        key_index <= (row + col_tracker) - sub;
    else
        key_index <= key_index;
end
//Output Key value
always @ (posedge clk) begin
    if  ((state == stateRead) && (row != 4'b1111))
        decode <= keypad[key_index];
    else
```

Figure 3 Logic for decoding the key press

In addition to outputting the decoded key value, a key flag is also set as an output port whenever a key is pressed. This key flag is debounced and sent to the upper level LCD Control Module. An LED on the BASYS also indicates that the key is pressed and has been debounced.

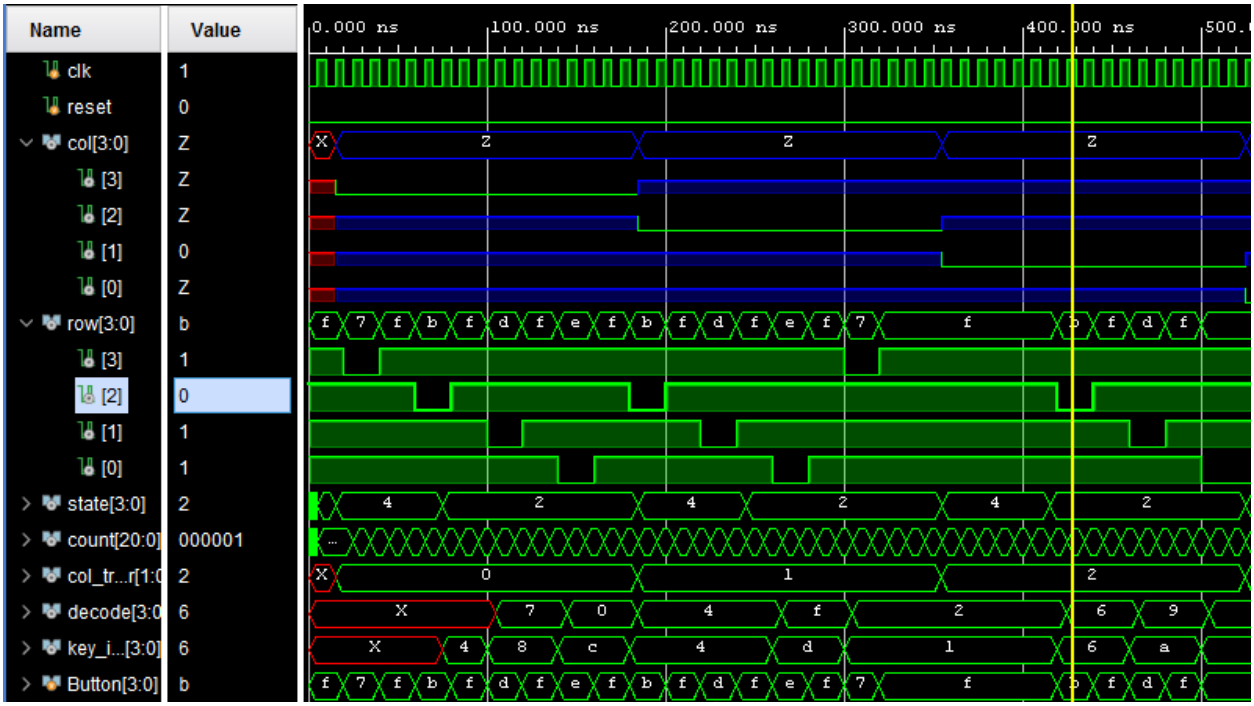


Figure 4 New keypad decoder simulation

To simulate the Keypad Decoder module, a more basic set of delay values was used to cycle the columns every few simulated clock cycles. After some effort in navigating the ins and outs of the inout port type, a successful simulation was performed, which showed the simulated keys decoding properly as well as the value of the last button pressed being held while no keys were pressed. The count variable was included in the simulation to verify the columns were changing approximately as desired. See Keypad Decoder Module.

4.7 LCD Lower level Module

A state machine was developed to handle sending the 8bits of data for write or command functions to the LCD. The finite state machine implemented consists of four states: stateBegin, stateSetData, stateEnLow, and stateDelay.

Two synchronous always blocks are used in the FSM module; the first manages the delay variables such that a counter is only engaged when in stateDelay and the delay flag variable is LOW. The flag variable is driven high when the counter reaches the value specified by the delay register.

The second always block contains two conditions- if the reset button (BtnC on BASYS3) is pressed the key item here is that the finish variable is driven HIGH, communicating to the top level that the module is read to receive data and restarts the initialization procedure. The state variable is also assigned the value of stateBegin. If reset is not pressed, the always block enters the state machine. The first state, stateBegin, contains an if-else condition dependent on the values of start and finish; if start and finish

are not both HIGH, the else condition is met and finish is driven HIGH, signaling to the top level that start should also be driven HIGH and data sent from the top level to the state machine.

Once start and finish are driven high, the rs pin is assigned for either data write or command write and the delay value from the top level is assigned to the delay_set register. Here, finish is also set LOW. StateSetData is then entered, where the data value from the top level is assigned to the data pins. The next state, stateEnLow is then entered, which simply drives the enable pin LOW, setting the data that has been sent to the LCD. The final state, stateDelay, is then entered, where the state machine will remain until the data flag variable is set high. From stateDelay, the state machine will reengage the first state, stateBegin. Because the finish variable is still LOW, the state machine will enter the else condition of this state, where finish is driven HIGH, signaling to the top level that it is ready for the next set of data. See Lower Level LCD Module (writes data to LCD) for the full Verilog code.
The state machine was simulated to verify that all states were entered and that the data and enable pins were driven in the proper sequence so that data would successfully be sent to the LCD.

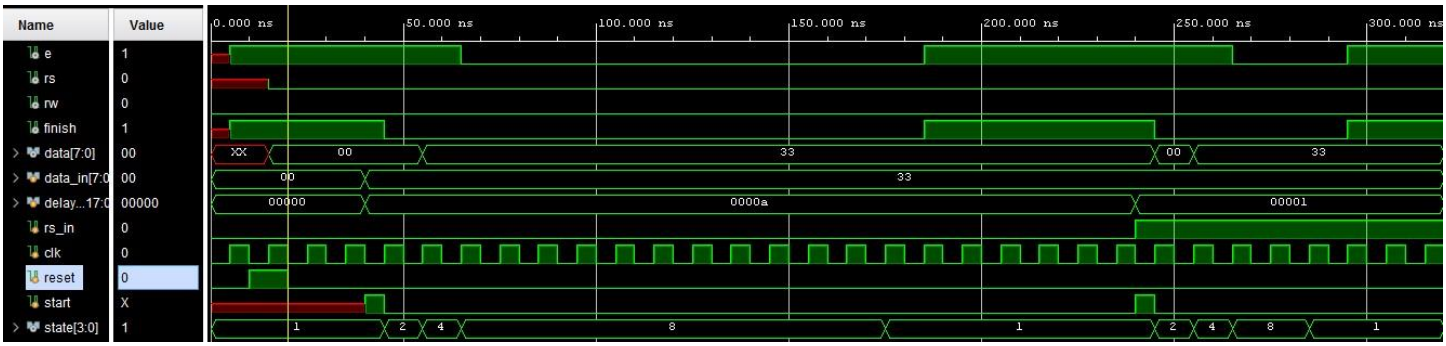


Figure 5 Lower Level LCD module simulation

4.8 Upper Level LCD Control Module

The upper level state machine controls what is being sent to the LCD and takes in input from the keypad. Using three switch cases to manage input and output, the basic flow is

- Initialize LCD -> send all initialization commands
- Write “Display 4 dig PIN” to the LCD

Figure 6 Initial LCD display

- Set address to line 2, position 6
- Load -> in the load state, the FSM waits for keypad input and a “ready” flag is high.
    - When the keypad flag goes high, a “read” flag is toggled for one clock cycle.

        - The key value is loaded into the current PIN index (the current PIN index value is displayed in binary by two LEDs on the BASYS).
        - A buffer state is entered to keep the "ready" flag low for .5 seconds, ignoring the keypad until it is low.
- Write Key -> While ignoring the keypad, the loaded key press is printed to the LCD
- Idle -> The FSM is then idle until the "ready" flag goes high.
    - The idle state checks the value of the PIN index
        - If the last PIN digit has been entered, set the LCD address home
            - If the PIN is valid, print "Door is unlocked" to the LCD
            - If the PIN is invalid, print "PIN invalid"
    - If the ready flag goes high and the last PIN digit has not been loaded, return to Load state and wait for input.
- Done -> after valid or invalid message has been printed, enter done state. The program will remain here until the reset button is pressed.

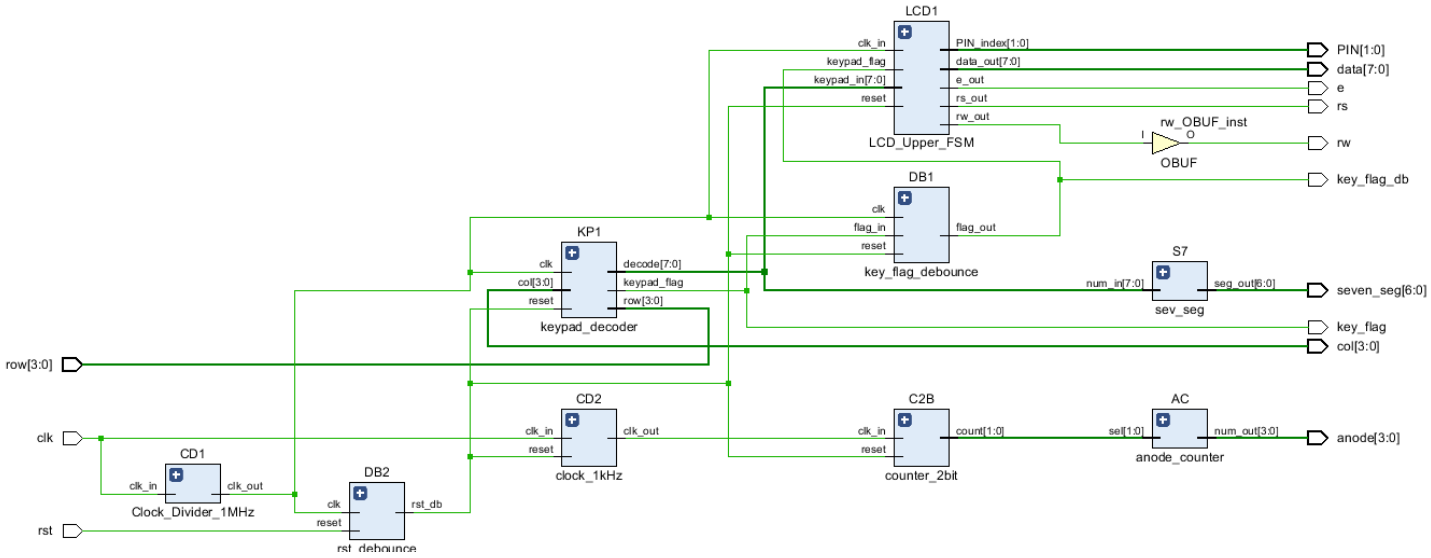
A large number of flags are used in this module to manage the LCD and ensure that only the desired data are written to the LCD in the desired order. See Control Module.

4 Conclusion

The design was successfully implanted, albeit the keypad debounce did leave something to be desired. Nearly 100 iterations of this overall design were sent to the board and many debounce values were attempted, the current design has been the most reliable.

Figure 7 (Left) Invalid PIN (Right) Valid PIN

Appendix A.



Appendix B.

TOP LEVEL MODULE

```
/********************
*File :          LCD_Keypad_Top
*Assignment:     Final Project
*Course:         EGR 224
*Instructor:     Professor Zuidema
*Author:         Dustin Matthews
*Date:           12/11/20
*Description:    Manages all input and output of LCD and Keypad
*********************/
`timescale 1ns / 1ps

module
LCD_Keypad_Top  (clk, rst, data, rs, rw, e,
                 row, col, anode, seven_seg, PIN,
                 key_flag_db, key_flag);
input wire clk,
           rst;
input [3:0] row;
output [3:0] col;
output wire [7:0] data;
output wire [3:0] anode;
output wire [6:0] seven_seg;
output wire [1:0] PIN;      //display current pin index on LEDs
output wire rs,
            rw,
            e;
output wire key_flag,       //indicate key pressed on LED
            key_flag_db;    //indicate debounced key press on LED
```

```
wire rst_deb,
     clk_div,
     clk_1K,
     key_Press;
wire [7:0] key_data;
wire [1:0] select;

//////////////////////
//INSTANTIATE MODULES//
//////////////////////
Clock_Divider_1MHz CD1(.clk_in(clk),  .clk_out(clk_div));

rst_debounce DB2 (.clk(clk_div), .reset(rst), .rst_db(rst_deb));

LCD_Upper_FSM LCD1 (.clk_in(clk_div), .reset(rst_deb), .keypad_flag(key_flag_db),
                    .keypad_in(key_data), .data_out(data), .rs_out(rs),
                     .rw_out(rw), .e_out(e), .PIN_index(PIN));


keypad_decoder KP1 (.clk(clk_div), .reset(rst_deb), .col(col), .row(row),
                    .decode(key_data), .keypad_flag(key_Press));

key_flag_debounce DB1  (.clk(clk_div), .reset(rst_deb),
                        .flag_in(key_Press), .flag_out(key_flag_db));

clock_1kHz CD2 (.clk_in(clk), .reset(rst_deb), .clk_out(clk_1K));

counter_2bit C2B(.reset(rst_deb), .clk_in(clk_1K), .count(select));

anode_counter AC (.sel(select), .num_out(anode));

sev_seg S7 (.num_in(key_data), .seg_out(seven_seg));

assign key_flag = key_Press;

endmodule
```

Control Module

```
/*******************
*File :         lcd_top
*Assignment:    Final Project
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          12/11/20
*Description:   Writes messages and key presses from keypad to the LCD
*******************/
`timescale 1ns / 1ps

module
LCD_Upper_FSM  (clk_in, reset, keypad_flag, keypad_in, data_out,
                rs_out, rw_out, e_out, PIN_index);
`define initIndex 5
`define startIndex 14
```

```
`define matchIndex 15
`define invalidIndex 15
`define charDelay 26000
`define PT_MAX 4
//////////////////////////////
/******STATE PARAMETERS*******/
parameter   LCD_init = 4'b0000,
            write_Enter_PIN = 4'b0001,
            shift_Add_Line2 = 4'b0010,
            load = 4'b0011,
            write_Key = 4'b0100,
            idle = 4'b0101,
            return_Home = 4'b0110,
            write_Match = 4'b0111,
            write_Invalid = 4'b1000,
            done = 4'b1001;
//////////////////////////
/*****KEYPAD LOADER******/
parameter   load_3 = 2'b00,
            load_2 = 2'b01,
            load_1 = 2'b10,
            load_0 = 2'b11;
//////////////////////////
/***** BUFFER *****/
parameter wait_for_press = 2'b00,
          halt = 2'b01,
          check_key_status = 2'b11;
//////////////////////////////////
input wire  clk_in,
            reset,
            keypad_flag;

input wire [7:0] keypad_in;
output wire rs_out,
            e_out;
output wire [7:0] data_out;
output reg [1:0] PIN_index = 2'b11;
output reg rw_out = 0;

reg [20:0] delay_wire;
reg [20:0] buffer_count;
reg [7:0] load_key,
          data_wire;
reg [4:0] count = 0;
reg [3:0]  state = LCD_init;
reg [1:0]  buffer = wait_for_press;
reg [1:0]  load_PIN = load_1;

reg ready,
    rs_wire,
    start_out;

wire read,
     toggle,
     PIN_flag,
```

```
     idle_flag,
     count_flag,
     complete_flag,
     wait_flag,
     check_PIN,
     finish_in,
     rs_assign;

wire [7:0] data_assign;
wire [20:0] delay_assign;

reg [7:0] PIN_entry [3:0];
wire [7:0] data_cmd [6:0];
wire [20:0] delay_cmd [6:0];

wire [7:0] startMsg [`startIndex:0];
wire [7:0] matchMsg [`matchIndex:0];
wire [7:0] invalidMsg [`invalidIndex:0];
wire [31:0] system_PIN;

////////////////////////////////////
//----------COMMAND ARRAYS-----------//
////////////////////////////////////
assign data_cmd[0] = 8'b00000000; assign delay_cmd[0] = 20000;   //pwr on delay
assign data_cmd[1] = 8'b00111000; assign delay_cmd[1] = 40;    //function set
assign data_cmd[2] = 8'b00001101; assign delay_cmd[2] = 40;   //disp set
assign data_cmd[3] = 8'b00000001; assign delay_cmd[3] = 1600;   //disp clr
assign data_cmd[4] = 8'b00000010; assign delay_cmd[4] = 1600;   //return home
assign data_cmd[5] = 8'b00000110; assign delay_cmd[5] = 40;
assign data_cmd[6] = 8'b11000110; assign delay_cmd[6] = 40;   //set cursor to second
line address 46H
////////////////////////////////////
//----------CHARACTER ARRAYS----------//
////////////////////////////////////
assign startMsg[0] ="E";   assign matchMsg[0] =  "D";   assign invalidMsg[0] =   "I";
assign startMsg[1] ="n";   assign matchMsg[1] =  "o";   assign invalidMsg[1] =   "n";
assign startMsg[2] ="t";   assign matchMsg[2] =  "o";   assign invalidMsg[2] =   "v";
assign startMsg[3] ="e";   assign matchMsg[3] =  "r";   assign invalidMsg[3] =   "a";
assign startMsg[4] ="r";   assign matchMsg[4] =  " ";   assign invalidMsg[4] =   "l";
assign startMsg[5] =" ";   assign matchMsg[5] =  "i";   assign invalidMsg[5] =   "i";
assign startMsg[6] ="4";   assign matchMsg[6] =  "s";   assign invalidMsg[6] =   "d";
assign startMsg[7] =" ";   assign matchMsg[7] =  " ";   assign invalidMsg[7] =   " ";
assign startMsg[8] ="d";   assign matchMsg[8] =  "U";   assign invalidMsg[8] =   "P";
assign startMsg[9] ="i";   assign matchMsg[9] =  "n";   assign invalidMsg[9] =   "I";
assign startMsg[10] ="g";   assign matchMsg[10] = "l";  assign invalidMsg[10] =  "N";
assign startMsg[11] = " ";   assign matchMsg[11] = "o"; assign invalidMsg[11] =  " ";
assign startMsg[12] = "P";   assign matchMsg[12] = "c"; assign invalidMsg[12] =  " ";
assign startMsg[13] = "I";   assign matchMsg[13] = "k"; assign invalidMsg[13] =  " ";
assign startMsg[14] = "N";   assign matchMsg[14] = "e"; assign invalidMsg[14] =  " ";
                             assign matchMsg[15] = "d"; assign invalidMsg[15] =  " ";
////////////////////////
//--SYSTEM PIN VALUE---//
////////////////////////
assign system_PIN = "A49F";
```

```
//////////////////////////////////////////
//Handshake with lower level LCD FSM MODULE//
//////////////////////////////////////////
always @ (posedge clk_in) begin
    if ((finish_in == 1'b1) && (!idle_flag == 1'b1) && (!complete_flag ==
1'b1))//(((finish_in == 1'b1) && (!idle_flag)) && (!complete_flag))// || (reset))
////((finish_in == 1) && (!idle_flag) && (state != done)) || (reset == 1))
        start_out <= 1;
    else
        start_out <= 0;
end
//////////////////////////////////////////////////
//Ready goes high when ready to accept keypad input//
//////////////////////////////////////////////////
always @ (posedge clk_in) begin
    if ((keypad_flag == 1'b1) || (wait_flag == 1'b1))
        ready <= 1'b0;
    else
        ready <= 1'b1;
end
////////////////////////////////
//Data Feed to lower level FSM//
////////////////////////////////
always @ (posedge clk_in) begin
    data_wire <= data_assign;
    delay_wire <= delay_assign;
    rs_wire <= rs_assign;
end
//////////////////////////
//Top Level Index Counter//
//////////////////////////
always @ (posedge clk_in) begin
    if (reset == 1'b1)
        count <= 3;         //RETURN HOME -> CLR DISP
    else if ((count_flag == 1'b1) || (idle_flag == 1'b1))
        count <= 0;         //Stop/reset counter
    else if (toggle == 1'b1)
        count <= count + 1;
    else
        count <= count;
end
////////////
// BUFFER //
////////////
always @ (posedge clk_in) begin
    if (reset == 1'b1)
        buffer <= wait_for_press;
    else begin
        case (buffer)
            wait_for_press : begin
                if (read == 1'b1) begin
                    buffer_count <= 0;
                    buffer <= halt;
                end
                else
```

```
                buffer <= wait_for_press;
            end
            ////////Ignore keypad while in halt
            halt : begin
                buffer_count <= buffer_count + 1;
                if (buffer_count == 500000)
                    buffer <= check_key_status;
                 else
                    buffer <= halt;
            end
            ////////Check if key is still pressed
            check_key_status : begin
                if (keypad_flag == 1'b1) begin
                    buffer_count <= 0;
                    buffer <= halt;
                end
                else
                    buffer <= wait_for_press;
            end
            default : begin
                buffer <= wait_for_press;
            end
        endcase
    end
end
//////////////
//PIN LOADER//
//////////////
always @ (posedge clk_in) begin
    if (reset) begin            //reset the PIN index
        load_PIN <= load_3;
        PIN_index <= 3;
    end
    else if (read == 1'b1) begin
    //Keypad toggled, load value into PIN array
        case (load_PIN)
            load_3 : begin
                PIN_entry[3] <= keypad_in;
                PIN_index <= 3;
                load_PIN <= load_2;
            end
            load_2 : begin
                PIN_entry[2] <= keypad_in;
                PIN_index <= 2;
                load_PIN <= load_1;
            end
            load_1 : begin
                PIN_entry[1] <= keypad_in;
                PIN_index <= 1;
                load_PIN <= load_0;
            end
            load_0 : begin
                PIN_entry[0] <= keypad_in;
                PIN_index <= 0;
                load_PIN <= load_PIN;
```

```
            end
        endcase
    end
    else
        load_PIN <= load_PIN;
end

////////////////////////////
//FSM TO CONTROL LCD OUTPUT//
////////////////////////////
always @ (posedge clk_in) begin
    if (reset == 1'b1)
        state <= LCD_init;
    else begin
        case (state)
            /////////initialize LCD
            LCD_init : begin
                if (count_flag == 1'b1)
                    state <= write_Enter_PIN;
                else
                    state <= LCD_init;
            end
            /////////Write first display message
            write_Enter_PIN : begin
                if (count_flag == 1'b1) begin
                    state <= shift_Add_Line2;
                end
                else
                    state <= write_Enter_PIN;
            end
            ///////Set address to line 2
            shift_Add_Line2 : begin
                if (count_flag == 1'b1)
                    state <= load;
                else
                    state <= shift_Add_Line2;
            end
            ///////wait for Keypress
            load : begin
                if (read == 1'b1)
                    state <= write_Key;
                else
                    state <= load;
            end
            ///////Write keypress to the LCD
            write_Key : begin
                if (count_flag == 1'b1)
                    state <= idle;
                else
                    state <= write_Key;
            end
            ///////Idle until ready for next input
            ///////or if 4 digits entered, return LCD home
            idle : begin
                if (PIN_flag == 1'b1)
```

```
                state <= return_Home;
            else if (ready == 1'b1)
                state <= load;
            else
                state <= idle;
        end
        ////////set LCD address home, check if PIN matches
        return_Home : begin
            if (count_flag == 1'b1) begin
                if (check_PIN == 1'b1)
                    state <= write_Match;
                else
                    state <= write_Invalid;
            end
            else
                state <= return_Home;
        end
        ////////PIN valid, unlock door
        write_Match : begin
            if (count_flag == 1'b1)
                state <= done;
            else
                state <= write_Match;
        end
        ////////PIN invalid, display error msg
        write_Invalid : begin
            if (count_flag == 1'b1)
                state <= done;
            else
                state <= write_Invalid;
        end
        ////////show message until reset
        done : begin
            state <= done;
        end
        ////////
        default : begin
            state <= LCD_init;
        end
    endcase
end
end
////////////////////
//Lower LCD Module//
////////////////////
LCD_FSM sm1 (.e(e_out), .rs(rs_out), .data(data_out), .clk(clk_in), .reset(reset),
.start(start_out), .finish(finish_in), .data_in(data_wire), .rs_in(rs_wire),
.delay_in(delay_wire));
////////////////////
//FLAG ASSIGNMENTS//
////////////////////
//FLAG for top level index counter//
assign count_flag =     (
                    ((state == LCD_init)        &&  (count == `initIndex + 1))   ||
                    ((state == write_Enter_PIN) &&  (count == `startIndex +1))   ||
```

```
                    ((state == shift_Add_Line2) &&  (count == 1))               ||
                    ((state == return_Home)     &&  (count == 2))               ||
                    ((state == write_Match)     &&  (count == `matchIndex + 1))  ||
                    ((state == write_Invalid)   &&  (count == `invalidIndex + 1))||
                    ((state == write_Key)       &&  (count == 1))
                      ) ? 1'b1 : 1'b0;

//FLAG to idle the LCD//
assign idle_flag =  (
                    ((state == idle) && (!PIN_flag == 1'b1))  ||
                    (state == load)
                    )    ? 1'b1 : 1'b0;
//FLAG goes high if 4 PIN digits received
assign PIN_flag = (PIN_index == 1'b0) ? 1'b1 : 1'b0;
//FLAG goes high while ignoring keypad
assign wait_flag = (
                   (buffer == halt) ||
                    (buffer == check_key_status)
                    ) ? 1'b1 : 1'b0;
//FLAG goes high if all tasks completed
assign complete_flag = ((state == done) && (!reset)) ? 1'b1 : 1'b0;
//FLAG goes high if entered PIN matches stored PIN//
assign check_PIN = (
                  ((PIN_index == 1'b0) &&
                  ({PIN_entry[3], PIN_entry[2],
                    PIN_entry[1], PIN_entry[0]} == system_PIN))
                   ) ? 1'b1 : 1'b0;
////////////////////////////////////
//Data assignments to lower level FSM//
assign data_assign =    (state == LCD_init)         ? data_cmd[count] :
                        (state == write_Enter_PIN)  ? startMsg[count] :
                        (state == shift_Add_Line2)  ? data_cmd[6] :
                        (state == write_Key)        ? keypad_in :
                        (state == return_Home)      ? data_cmd[4] :
                        (state == write_Match)      ? matchMsg[count] :
                        (state == write_Invalid)    ? invalidMsg[count]:
                        0;
assign delay_assign =   (state == LCD_init)         ? delay_cmd[count] :
                        (state == write_Enter_PIN)  ? `charDelay       :
                        (state == shift_Add_Line2)  ? delay_cmd[6]     :
                        (state == write_Key)        ? `charDelay :
                        (state == return_Home)      ? delay_cmd[4] :
                        (state == write_Match)      ? `charDelay :
                        (state == write_Invalid)    ? `charDelay :
                        0;
assign rs_assign =      (state == LCD_init)         ? 1'b0 :
                        (state == write_Enter_PIN)  ? 1'b1 :
                        (state == shift_Add_Line2)  ? 1'b0 :
                        (state == write_Key)        ? 1'b1 :
                        (state == return_Home)      ? 1'b0 :
                        (state == write_Match)      ? 1'b1 :
                        (state == write_Invalid)    ? 1'b1 :
                        (state == load )            ? 1'b1 :
                        0;
////////////////////////////////////
```

```
//READ FLAG toggles high for one clock cycle to load keypad value
assign read = ((ready == 1'b1) && (keypad_flag == 1'b1)) ? 1'b1 : 1'b0;
//TOGGLE flag toggles high for one clock cycle to increment top counter
assign toggle = (start_out & finish_in) ? 1'b1 : 1'b0;

endmodule
```

Lower Level LCD Module (writes data to LCD)

```
/*******************
*File :         LCD_FSM
*Assignment:    Lab 8
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          11/13/20
*Description:   work_state Machine to send data to LCD
********************/
`timescale 1ns / 1ps

module
LCD_FSM(e, rs, data, clk, reset, start, finish, data_in, rs_in, delay_in);
//INPUTS
input clk, reset;
input start;            // flag to start work_state machine and process data
input rs_in;            //determines command or data write
input [7:0] data_in;    // data sent from top level for FSM to process
input [20:0] delay_in;  // delay length sent from top level
//OUTPUTS
output reg e, rs;       //signals sent out from module
output reg [7:0] data;  //data sent out from module
output reg finish;      // flag to top level to signal data has been sent

//REGISTERS
reg [20:0] delay_current;
reg [20:0] delay_set;
wire delay_flag;
//work_state VARIABLES
parameter work_stateBegin = 4'b0001,
          work_stateSetData = 4'b0010,
          work_stateEnLow = 4'b0100,
          work_stateDelay = 4'b1000;

reg [3:0] work_state = work_stateBegin;

//delay flag goes high if work_state is work_stateDelay and the delay counter matches
the delay_set variable
assign delay_flag = ((work_state == work_stateDelay) && (delay_set == delay_current))
? 1'b1 : 1'b0;

//current delay count will set to zero if work_state is not work_stateDelay or delay
flag goes high
always @(posedge clk)
    begin
        if ((delay_flag == 1) || (work_state != work_stateDelay)) begin
```

```
            delay_current <= 0;
        end
        else begin
            delay_current <= (delay_current + 1);
        end
    end

always @(posedge clk)
    begin
        if (reset) begin
            data <= 0;
            finish <= 1;
            work_state <= work_stateBegin;
            rs <= 0;
            e <= 1;
            delay_set <= 0;
            end
        else begin
        case(work_state)
            work_stateBegin : begin
                if (start && finish) begin
                    data <= data_in;        //load data input
                    rs <= rs_in;            //assign the rs value
                    e <= 1;
                    finish <= 0;
                    delay_set <= delay_in;  //assign the delay value
                    work_state <= work_stateSetData;
                end
                /* IDLE UNTIL START IN GOES HIGH */
                else begin
                    data <= data;
                    rs <= rs;
                    e <= 1;
                    finish <= 1;            //set finish high, which will set start
high in top level
                    delay_set <= delay_set;
                    work_state <= work_stateBegin;
                end
            end
            work_stateSetData : begin
                    data <= data;
                    rs <= rs;
                    e <= 1;
                    finish <= 0;                //set finish LOW
                    delay_set <= delay_set;
                    work_state <= work_stateEnLow;
            end
            work_stateEnLow : begin
                    data <= data;
                    rs <= rs;
                    e <= 0;                     //drive E LOW  to lock in data
                    finish <= 0;
                    delay_set <= delay_set;
                    work_state <= work_stateDelay;
            end
```

```
            work_stateDelay : begin              //remain in this work_state until
the delay is complete
                if (delay_flag == 1'b1) begin
                    data <= data;
                    rs <= rs;
                    e <= e;
                    finish <= 0;
                    delay_set <= delay_set;
                    work_state <= work_stateBegin;
                    end
                else begin
                    data <= data;
                    rs <= rs;
                    e <= e;
                    finish <= 0;
                    delay_set <= delay_set;
                    work_state <= work_stateDelay;
                    end
            end
            default : work_state <= work_stateBegin;
        endcase
    end
end
endmodule
```

Keypad Decoder Module

**Revised from lab version to a smaller FSM and decoder block*

```
/********************
*File :          keypad_decoder
*Assignment:     Final Project
*Course:         EGR 224
*Instructor:     Professor Zuidema
*Author:         Dustin Matthews
*Date:           12/11/20
*Description:    Reads the 4x4 matrix keypad
*********************/
`timescale 1ns / 1ps

module
keypad_decoder (clk, reset, row, col, decode, keypad_flag);

//REGISTERS AND WIRES
inout [3:0] row, col;
output reg [7:0] decode;
output reg keypad_flag;

input wire  clk,
            reset;
wire [3:0]  state,
            sub;
wire delay_flag;
reg [20:0] count;
```

```
reg [3:0] nextState,
          ColType,
          key_index;
reg [1:0] col_tracker;              //2bit number to track which column is active
wire [7:0] keypad [15:0];

parameter   stateShift = 4'b1000,
            stateWait = 4'b0100,
            stateRead = 4'b0010,
            stateInit = 4'b0001;

// Next state logic
assign state = nextState;

//Manage high impedance columns
assign col[3] = ColType[3]?1'bZ:1'b0;
assign col[2] = ColType[2]?1'bZ:1'b0;
assign col[1] = ColType[1]?1'bZ:1'b0;
assign col[0] = ColType[0]?1'bZ:1'b0;

/*** Variable used to determine key index ****/
assign sub = (row == 4'b1110) ? 2 :
             (row == 4'b1101) ? 5 :
             (row == 4'b1011) ? 7 :
             (row == 4'b0111) ? 7 :
                                0 ;
/**** KEYPAD ARRAY ****/
assign keypad[0]    = "1";
assign keypad[1]    = "2";
assign keypad[2]    = "3";
assign keypad[3]    = "A";
assign keypad[4]    = "4";
assign keypad[5]    = "5";
assign keypad[6]    = "6";
assign keypad[7]    = "B";
assign keypad[8]    = "7";
assign keypad[9]    = "8";
assign keypad[10]   = "9";
assign keypad[11]   = "C";
assign keypad[12]   = "0";
assign keypad[13]   = "F";
assign keypad[14]   = "E";
assign keypad[15]   = "D";

/**** DELAY FLAG ****/
//1MHz clock input, 1000 cycles = 1ms
assign delay_flag = ((state == stateWait) && (count == 100)) ||
                    ((state == stateRead) && (count == 900)) ? 1'b1: 1'b0;
/************ DELAY COUNTER *************/
//Increment counter when setting columns//
//********and when reading rows*********//
always @ (posedge clk) begin
    if ((delay_flag == 1) || (reset))
        count <= 0;
    //increment counter only when in stateWait or stateRead
```

```
    else if  ((state == stateWait) || (state == stateRead))
        count <= (count + 1);
    else count <= 0;
end

/**** FINITE STATE MACHINE ****/
//Cycle the columns to be read//
always @ (posedge clk) begin
    if (reset)
        nextState <= stateInit;
    else begin
        case (state)
            stateInit : begin
                ColType <= 4'b0111;                         //initialize column value
                col_tracker <= 2'b00;                       //initialize tracker value
                nextState <= stateWait;
            end
            stateWait : begin
                ColType <= ColType;           //maintain column value while in state
                col_tracker <= col_tracker;
                if (delay_flag == 1'b1)       //allow new column value to 'settle'
                    nextState <= stateRead;
                else nextState <= stateWait;
            end
            stateRead : begin
                ColType <= ColType;
                col_tracker <= col_tracker;
                if (delay_flag == 1'b1)
                    nextState <= stateShift;              //move to delay state
                else nextState <= stateRead;
            end
            stateShift : begin
                ColType <= {ColType[0], ColType[3:1]};    //shift column
                col_tracker <= col_tracker + 1;           //increment tracker
                nextState <= stateWait;                   //move to delay state
            end
            default : begin
                ColType <= ColType;
                col_tracker <= col_tracker;
                nextState <= stateInit;
            end
        endcase
    end
end

/**** BUTTON READER BLOCKS ****/
//Determine index
always @ (posedge clk) begin
    if  ((state == stateRead) && (row != 4'b1111))
        key_index <= (row + col_tracker) - sub;
    else
        key_index <= key_index;
end
//Output Key value
always @ (posedge clk) begin
```

```
    if  ((state == stateRead) && (row != 4'b1111))
        decode <= keypad[key_index];
    else
        decode <= decode;
end
/****KEYPAD FLAG OUTPUT TO LCD CONTROLLER MODULE****/
always @ (posedge clk) begin
    if ((row == 4'b0111) || (row == 4'b1011) ||
        (row == 4'b1101) || (row == 4'b1110))
        keypad_flag <= 1;
    else
        keypad_flag <= 0;
end

endmodule
```

Clock Divider 1 MHz

```
/********************
*File :         Clock_Divider_1MHz
*Assignment:    Lab 8
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          11/13/20
*Description:   takes clock signal of BASYS3
*               and outputs a 1MHz clock signal
*********************/
`timescale 1ns / 1ps

module Clock_Divider_1MHz//(clk_in, reset, clk_out);
    (clk_in, clk_out);
input clk_in;
//input reset;
output reg clk_out;
reg [6:0] count;
//For 1MHz, period = 100 000 000 / 1 000 000 = 100
//DC = .5; therefore the counter must toggle every 50 clock cycles
always @(posedge clk_in)
begin
    if (count == 50)
    begin
        clk_out <= ~clk_out;
        count <= 0;
    end
    else
    begin
        count <= (count + 1);
    end
end

endmodule
```

Clock Divider 1KHz

```
/******************
*File :         clock_1kHz
*Assignment:    Lab 6
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          11/4/20
*Description:   Modified file from part 1 of lab, takes clock signal of BASYS3
*               and outputs a 1kHz clock signal
*******************/
`timescale 1ns / 1ps

module clock_1kHz(clk_in, reset, clk_out);

input clk_in;
input reset;
output reg clk_out;
reg [16:0] count;
//For 1kHz, period = 100 000 000 / 1000 = 100 000
//DC = .5; therefore the clock must switch every 50 000 clock cycles

always @(posedge clk_in or posedge reset) //always @ similar to while(1)
begin
    if(reset)
    begin //like {} from c
        clk_out <= 1'b0;
        count <=0;
    end
    else if (count == 50_000)
    begin
        clk_out <= ~clk_out;
        count <= 0;
    end
    else
    begin
        count <= (count + 1);
    end
end
endmodule
```

Counter 2-bit Module

```
/******************
*File :         2Bit Counter
*Assignment:    Lab 6
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          11/4/20
*Description:   Module counts 00 01 10 11 00... if reset (BTNC on BASYS3) is pressed,
*               resets count to 0
*******************/
`timescale 1ns / 1ps
```

```
module counter_2bit(reset, clk_in, count);
input reset, clk_in;
output reg [1:0] count;

always @(posedge clk_in)
    begin
    if (reset) count <= 2'b00;
    else count <= (count + 1);
    end
endmodule
```

Anode Counter Module

```
/*******************
*File :         anode_counter
*Assignment:    Lab 6
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          11/4/20
*Description:   Multiplexer that outputs a 4bit number based on the 2 bit select
input
*               to drive the individual seven-segment LEDs on the BASYS3
*********************/
`timescale 1ns / 1ps
module anode_counter(sel, num_out);
input wire [1:0] sel;
output reg [3:0] num_out;

always @(sel)
begin
    case(sel)
        2'b00 : num_out <= 4'b0111;
        2'b01 : num_out <= 4'b1011;
        2'b10 : num_out <= 4'b1101;
        2'b11 : num_out <= 4'b1110;
    endcase
end
endmodule
```

Seven Segment Module

```
/*******************
*File :         sev_seg
*Assignment:    Final Project
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          12/11/20
*Description:   drives the segments of the seven segment LEDs based on the input
*               Shows the last key pressed on the keypad
*********************/
`timescale 1ns / 1ps

module sev_seg(num_in, seg_out);
```

```
input wire [7:0] num_in;
output reg [6:0] seg_out;

always @(num_in)
begin
    case(num_in)
        "0" : seg_out <= 7'b0000001;//0
        "1" : seg_out <= 7'b1001111;//1
        "2" : seg_out <= 7'b0010010;//2
        "3" : seg_out <= 7'b0000110;//3
        "4" : seg_out <= 7'b1001100;//4
        "5" : seg_out <= 7'b0100100;//5
        "6" : seg_out <= 7'b0100000;//6
        "7" : seg_out <= 7'b0001111;//7
        "8" : seg_out <= 7'b0000000;//8
        "9" : seg_out <= 7'b0001100;//9
        "A" : seg_out <= 7'b0001000;//a
        "B" : seg_out <= 7'b1100000;//b
        "C" : seg_out <= 7'b0110001;//c
        "D" : seg_out <= 7'b1000010;//d
        "E" : seg_out <= 7'b0110000;//e
        "F" : seg_out <= 7'b0111000;//f
        default : seg_out <= 7'b1111111;

    endcase
end
endmodule
```

Reset Debounce Module

```
/*******************
*File :         rst_debounce
*Assignment:    Final Project
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          12/11/20
*Description:   Debounce the reset button
********************/
`timescale 1ns / 1ps

module
rst_debounce(clk, reset, rst_db);

input clk, reset;
output rst_db;
reg wait1, wait2, wait3, wait4;

always @ (posedge clk) begin
        wait1 <= reset;
        wait2 <= wait1;
        wait3 <= wait2;
        wait4 <= wait3;
end
```

```
assign rst_db = (wait1 & wait2 & wait3 & wait4);

endmodule
```

Key Flag Debounce Module

```
/*******************
*File :         key_flag_debounce
*Assignment:    Final Project
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          12/11/20
*Description:   Short debounce before key flag is sent to LCD Control Module
*********************/
`timescale 1ns / 1ps

module
key_flag_debounce(clk, reset, flag_in, flag_out);

input clk, reset, flag_in;
output reg flag_out;
reg [10:0] db_count;
reg  key_sync1, key_sync2;
reg [10:0] max_count = 900;                //max count of debounce


/****** SYNCHRONIZE KEYPAD_FLAG TO CLK ********/
always @ (posedge clk) begin
    key_sync1 <= flag_in;
    key_sync2 <= key_sync1;
end

wire flag_idle = (flag_out == key_sync2);   //high when state of key unchanged

/*** DEBOUNCE ***/
always @ (posedge clk) begin
    if ((flag_idle) || (reset))             //If key stable, maintain flag_out
        db_count <= 0;                      //or reset counter
    else begin
        db_count <= db_count + 1;           //Increment the counter if state of the
key changes
        if (db_count == max_count)
            flag_out <= ~flag_out;          //If max count reached, toggle the state
of the output
    end
end

endmodule
```

Appendix C.

Table 1 Decoder Truth Table for Seven Segment Decoder

Input	'a' seg	'b' seg	'c' seg	'd' seg	'e' seg	'f' seg	'g' seg	Hex Out
0000	0	0	0	0	0	0	1	0
0001	1	0	0	1	1	1	1	1
0010	0	0	1	0	0	1	0	2
0011	0	0	0	0	1	1	0	3
0100	1	0	0	1	1	0	0	4
0101	0	1	0	0	1	0	0	5
0110	0	1	0	0	0	0	0	6
0111	0	0	0	1	1	1	1	7
1000	0	0	0	0	0	0	0	8
1001	0	0	0	1	1	0	0	9
1010	0	0	0	1	0	0	0	A
1011	1	1	0	0	0	0	0	B
1100	0	1	1	0	0	0	1	C
1101	1	0	0	0	0	1	0	D
1110	0	1	1	0	0	0	0	E
1111	0	1	1	1	0	0	0	F

Appendix D.

Test Bench Files

Keypad Decoder Test Bench

```
/*******************
*File :          key_flag_debounce tb
*Assignment:     Final Project
*Course:         EGR 224
*Instructor:     Professor Zuidema
*Author:         Dustin Matthews
*Date:           12/11/20
*Description:    Short debounce before key flag is sent to LCD Control Module
*******************/


`timescale 1ns / 1ps

module Keypad_Reader_tb();
reg clk, reset;
wire[3:0] col, row, state;
wire [20:0] count;
wire [1:0] col_tracker;
wire [3:0] decode, key_index;
reg [3:0] Button;
```

```
Keypad_Reader kr1 (clk, reset, col, row, state, count, col_tracker, decode,
key_index);
assign row = Button;

initial begin
    clk = 0;
    Button = 4'b1111;
    reset = 0;
end

always #5 clk = ~clk;

initial begin
    #20 Button = 4'b0111;
    #20  Button = 4'b1111;
    #20 Button = 4'b1011;
    #20  Button = 4'b1111;
    #20  Button = 4'b1101;
    #20 Button = 4'b1111;
    #20  Button = 4'b1110;
    #20 Button = 4'b1111;
    #20 Button = 4'b1011;
    #20  Button = 4'b1111;
    #20  Button = 4'b1101;
    #20 Button = 4'b1111;
    #20  Button = 4'b1110;
    #20  Button = 4'b1111;
    #20 Button = 4'b0111;
    #20  Button = 4'b1111;
    #100 Button = 4'b1011;
    #20  Button = 4'b1111;
    #20  Button = 4'b1101;
    #20 Button = 4'b1111;
    #20  Button = 4'b1110;
end
endmodule
```

Lower Level LCD FSM test bench

```
/*******************
*File :         LCD test bench
*Assignment:    Final Project
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          12/11/20
*Description:   Verify the data and delay input, see that e pin toggles to set data
********************/
`timescale 1ns / 1ps

module LCD_FSM_tb();
wire e, rs, rw, finish;
wire [7:0] data;
reg [7:0] data_in;
```

```
reg [17:0] delay_in;
reg rs_in, clk, reset, start;

wire [3:0] state;

LCD_FSM M1(e, rs, rw, data, clk, reset, start, finish, data_in, rs_in, delay_in,
state);

initial begin
data_in <= 0;
delay_in <= 0;
rs_in <= 0;
clk <= 0;
reset <= 0;

end

always #5 clk = ~clk;

always@ (posedge clk)begin
start <= finish;
end

initial begin
#20
data_in <= 8'b00110011;
delay_in <= 10;
rs_in <= 0;

#155
data_in <= 8'b00010011;
delay_in <= 5;
rs_in <= 1;

end

initial #320 $finish;

endmodule
```

Top Level Test Bench – Keypad and LCD together

```
/*******************
*File :         Top Level Test Bench
*Assignment:    Final Project
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          12/11/20
*Description:   Verify that the LCD and Keypad communicate and appropriate data is
displayed
*******************/


`timescale 1ns / 1ps
```

```
module LCD_Keypad_Top_tb();
wire [3:0]row;
reg [3:0]Button;
wire [3:0]col;
reg clk;
reg rst;
wire [7:0] data;
wire rs, rw, e;
wire [3:0] anode;
wire [6:0] seven_seg;
wire toggle;
wire [7:0] key_data;

LCD_Keypad_Top LK (clk, rst, data, rs, rw, e, row, col, anode, seven_seg, toggle,
key_data);
assign row = Button;

initial begin
clk = 0;
rst = 0;

Button = 4'b1111;
end
always #1 clk = ~clk;

initial begin
    #480 Button <= 4'b0111;
    #120  Button <= 4'b1111;
    #80 Button <= 4'b1101;
    #94  Button <= 4'b1111;
end

endmodule
```

Seven Segment Decoder Task Test bench

```
/*******************
*File :         Sev_seg_task_tb
*Assignment:    Final Project
*Course:        EGR 224
*Instructor:    Professor Zuidema
*Author:        Dustin Matthews
*Date:          12/11/20
*Description:   Use task to verify the output of seven segment decoder
*******************/


`timescale 1ns / 1ps

module seg_tb();

reg [3:0]num_in;
wire [6:0]seg_out;
```

```
sev_seg S7 (num_in, seg_out);

initial begin
    num_in = 0;
end

initial begin
    $display("------------------------------------------------------------------------------");
    $display("Running tests!");
    $display("------------------------------------------------------------------------------");
    #20 num_in = 0;
    #5                                  //Added 5ns buffer between input and task calls,
                                        //otherwise the task did not work properly
    verify(num_in,7'b1101110, seg_out);
    verify(num_in,7'b0000001, seg_out); //0
    #20 num_in = 4'b0001;
    #5
    verify(num_in,7'b1110110, seg_out);
    verify(num_in,7'b1001111, seg_out); //1
    #20 num_in = 4'b0010;
    #5
    verify(num_in,7'b1100110, seg_out);
    verify(num_in,7'b0010010, seg_out);//2
    #20 num_in = 4'b0011;
    #5
    verify(num_in,7'b1100110, seg_out);
    verify(num_in,7'b0000110, seg_out);//3
    #20 num_in = 4'b0100;
    #5
    verify(num_in,7'b0110110, seg_out);
    verify(num_in,7'b1001100, seg_out);//4
    #20 num_in = 4'b0101;
    #5
    verify(num_in,7'b1010110, seg_out);
    verify(num_in,7'b0100100, seg_out);//5
    #20 num_in = 4'b0110;
    #5
    verify(num_in,7'b1011010, seg_out);
    verify(num_in,7'b0100000, seg_out);//6
    #20 num_in = 4'b0111;
    #5
    verify(num_in,7'b1101010, seg_out);
    verify(num_in, 7'b0001111,seg_out);//7
    #20 num_in = 4'b1000;
    #5
    verify(num_in,7'b1111110, seg_out);
    verify(num_in,7'b0000000, seg_out);//8
    #20 num_in = 4'b1001;
    #5
    verify(num_in,7'b1011010, seg_out);
    verify(num_in,7'b0001100, seg_out);//9
    #20 num_in = 4'b1010;
    #5
    verify(num_in,7'b0111010, seg_out);
    verify(num_in,7'b0001000, seg_out);//a
    #20 num_in = 4'b1011;
    #5
    verify(num_in,7'b1001110, seg_out);
    verify(num_in,7'b1100000, seg_out);//b
    #20 num_in = 4'b1100;
    #5
    verify(num_in,7'b1110010, seg_out);
    verify(num_in,7'b0110001, seg_out);;//c
    #20 num_in = 4'b1101;
    #5
    verify(num_in,7'b1110110, seg_out);
    verify(num_in, 7'b1000010, seg_out);//d
    #20 num_in = 4'b1110;
    #5
```

```
    verify(num_in,7'b1010110, seg_out);
    verify(num_in,7'b0110000, seg_out);//e
    #20 num_in = 4'b1111;
    #5
    verify(num_in,7'b1011010, seg_out);
    verify(num_in,7'b0111000, seg_out);//f
    $display("-------------------------------------------------------------------------------");
    $display("Tests are completed!");
    $display("-------------------------------------------------------------------------------");
end

task verify;
    input [3:0] in1;
    input [6:0] out1,out2;
    begin
        if(out1 == out2)
            $display("VALID OUTPUT. Input is %d. Output is confirmed to be 0b%b.",in1,out1);
        else
            $display("OUTPUT FAIL. Input is %d. Output is incorrect. Should be 0b%b. Actually is
0b%b.",in1,out2,out1);
    end
endtask
endmodule
```

Appendix E.

Seven Segment Task Simulation Console Output

```
------------------------------------------------------------------------
Running tests!
------------------------------------------------------------------------
OUTPUT FAIL. Input is  0. Output is incorrect. Should be 0b0000001. Actually is 0b1101110.
VALID OUTPUT. Input is  0. Output is confirmed to be 0b0000001.
OUTPUT FAIL. Input is  1. Output is incorrect. Should be 0b1001111. Actually is 0b1110110.
VALID OUTPUT. Input is  1. Output is confirmed to be 0b1001111.
OUTPUT FAIL. Input is  2. Output is incorrect. Should be 0b0010010. Actually is 0b1100110.
VALID OUTPUT. Input is  2. Output is confirmed to be 0b0010010.
OUTPUT FAIL. Input is  3. Output is incorrect. Should be 0b0000110. Actually is 0b1100110.
VALID OUTPUT. Input is  3. Output is confirmed to be 0b0000110.
OUTPUT FAIL. Input is  4. Output is incorrect. Should be 0b1001100. Actually is 0b0110110.
VALID OUTPUT. Input is  4. Output is confirmed to be 0b1001100.
OUTPUT FAIL. Input is  5. Output is incorrect. Should be 0b0100100. Actually is 0b1010110.
VALID OUTPUT. Input is  5. Output is confirmed to be 0b0100100.
OUTPUT FAIL. Input is  6. Output is incorrect. Should be 0b0100000. Actually is 0b1011010.
VALID OUTPUT. Input is  6. Output is confirmed to be 0b0100000.
OUTPUT FAIL. Input is  7. Output is incorrect. Should be 0b0001111. Actually is 0b1101010.
VALID OUTPUT. Input is  7. Output is confirmed to be 0b0001111.
OUTPUT FAIL. Input is  8. Output is incorrect. Should be 0b0000000. Actually is 0b1111110.
VALID OUTPUT. Input is  8. Output is confirmed to be 0b0000000.
OUTPUT FAIL. Input is  9. Output is incorrect. Should be 0b0001100. Actually is 0b1011010.
VALID OUTPUT. Input is  9. Output is confirmed to be 0b0001100.
OUTPUT FAIL. Input is 10. Output is incorrect. Should be 0b0001000. Actually is 0b0111010.
VALID OUTPUT. Input is 10. Output is confirmed to be 0b0001000.
OUTPUT FAIL. Input is 11. Output is incorrect. Should be 0b1100000. Actually is 0b1001110.
VALID OUTPUT. Input is 11. Output is confirmed to be 0b1100000.
OUTPUT FAIL. Input is 12. Output is incorrect. Should be 0b0110001. Actually is 0b1110010.
VALID OUTPUT. Input is 12. Output is confirmed to be 0b0110001.
OUTPUT FAIL. Input is 13. Output is incorrect. Should be 0b1000010. Actually is 0b1110110.
VALID OUTPUT. Input is 13. Output is confirmed to be 0b1000010.
OUTPUT FAIL. Input is 14. Output is incorrect. Should be 0b0110000. Actually is 0b1010110.

OUTPUT FAIL. Input is 15. Output is incorrect. Should be 0b0111000. Actually is 0b1011010.
VALID OUTPUT. Input is 15. Output is confirmed to be 0b0111000.
------------------------------------------------------------------------
Tests are completed!
```