Lab 2: FPGA-based Mental Binary Math Game with Game Access Control

Mubashar Khan

ECE5440

# Introduction

In this lab, we add access control capabilities to the design of the 2-player binary mental math game using the DE2-115 FPGA board. Before players can play the game, they must first enter a 16-bit password which has been predefined. User inputs will only be accepted once the user presses the corresponding button. The game functions in a turn-based system with Player 1 choosing a 4-bit binary number and inputting it using a set of toggle switches. That number is then displayed on a 7-segment display which can display the terms 0-F in hexadecimal. Player 2 can look at the number being displayed on the 7-segment display and mentally should convert it to binary. Then Player 2 should enter a number using their 4-bit toggle switches such that the sum of Player 1’s and Player 2’s 4-bit binary numbers add to 1111 in binary. Player 2’s 4-bit binary input will also be displayed using a second 7-segment display. The FPGA will compute the sum of both players numbers and display the sum on a third 7-segment display. If the third 7-segment display shows ‘F’, signifying 1111 in hexadecimal, Player 2 wins the round and gains 1 point. Players should switch roles each round and record their scores. After reaching a predetermined number of rounds, the player with the most points is the winner.

# System Architecture Design

The diagram in figure 1 shows the top-level system architecture for the binary mental math game with access control features. To begin, players will have to enter a password to gain access to the FPGA. The password for boards using the code in this document is 2949. Each decimal digit must be entered as a binary number and then the player must press the password button shown in figure 1. When pushed, the button sends a signal to the button shaper which outputs a pulse that is one clock cycle long. This process is repeated for each digit of the password. If players enter an incorrect password, the program will reset after every 4 inputs until the correct password is entered. Another option is to press the reset button. The board will determine if the entered password is valid.

Each player will have their own set of 4 toggle switches to input their 4-bit binary numbers. When a player inputs a value, they must press their respective load buttons. The input is sent to the load register. If the correct password has been entered, the access controller will allow the load button signal through to the load register. This will take the input and send it to a 7-segment display and to the adder.

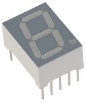
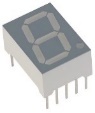
The game uses a 7-segment display which displays a number based on a 4-bit binary input. This is accomplished using a 4-16 decoder. The decoder takes the 4-bit input and converts it to a hexadecimal value. That value is then displayed on a 7-segment display. When both players have entered a value into the FPGA, the sum is found using a binary adder. The adder takes two 4-bit inputs and outputs a 4-bit value. We do not care about the overflow bit and any sums found that are over binary 1111 will wrap around and start counting from binary 0000. The output from the adder is passed onto the 4-16 decoder which converts the sum to a hexadecimal value that can be displayed on a 7-segment display.

FPGA: DE2-115

7-Segment Display 2

7-Segment Display 1

7-Segment Display 3



4-16 Decoder

4-16 Decoder

4-16 Decoder

4-bit Adder

Load Register Player 2

Clock

Reset

Load Register Player 1

Player 1

B2

B3

Player 2

B2

Green LED

Red LED

B3

Access Controller

Clock

Reset

B3

B2

B1

Button Shaper

Clock

Reset

Button Shaper

Button Shaper

B2

B3

B1

Password

Reset

Load P2

Load P1

Player 1

Password

Password

Player 1

Player 2

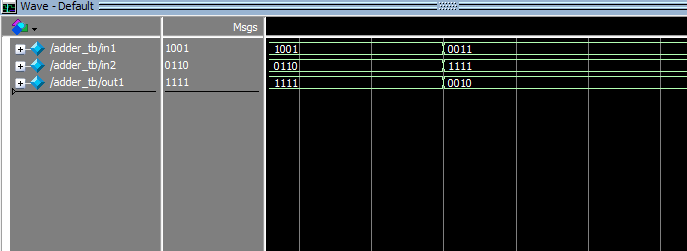
**Figure 1.** Top-level system architecture for the binary mental math game. Connections are shown using labels rather than wires

for some connections. Each button has its own button shaper. The reset button does not use a button shaper.

# Simulation Results

4-bit Adder Test:

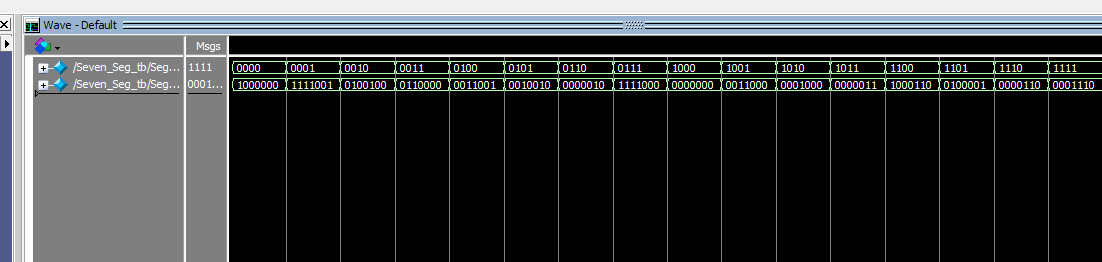
Figure 2 shows the simulation done to test the adder Verilog file. Results show that the adder is working as intended.



**Figure 2.** 4-Bit Adder Simulation. Using two sets of inputs, this simulation shows the respective outputs. By doing binary addition, we can see that the output shows the correct sum for both tests.

4-16 decoder: 7-segment display:

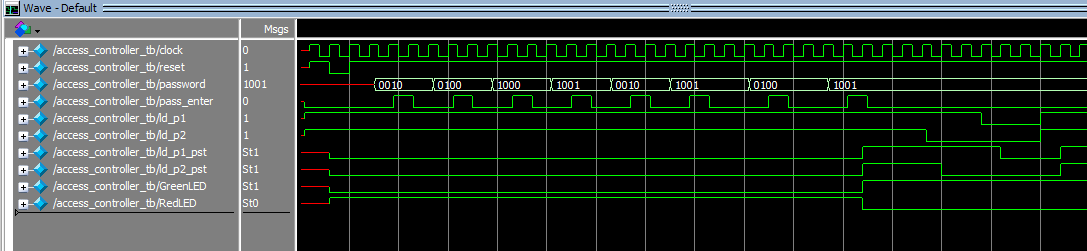
Figure 3 shows the simulation done to test the 7-segment display. The LEDs on the 7-segment display are active-low meaning an output of 0 for a bit enables that bit.



**Figure 3.** 4-16 Decoder Simulation. This simulation shows all possible outputs using the full range of a 4-bit input.

Access Controller:

Figure 4 shows the access controller simulation. Three tests were conducted to check the viability of the code. The first test was an incorrect password input. The second test was a correct password input. The final test was to see if outputs would change when inputs were changed.



**Figure 4.** Access Controller Testbench Simulation.

Button shaper:

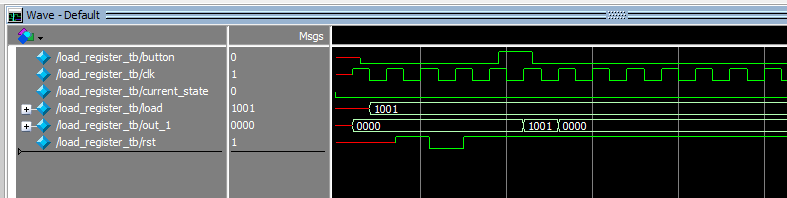
Figure 5 shows the simulation done for the button shaper module.

A screen shot of a clock

Description automatically generated

**Figure 5.** Button Shaper Testbench Simulation. Note that b\_out only remains in the active state for one clock cycle regardless of the length of b\_in.

Figure 6 shows the simulation results for the load register. The output remains 0 until the button is pressed, at which point the output changes to the value that the player entered.



**Figure 6.** Load Register Testbench Simulation.

# FPGA Board Testing Results

The following images in figures 7-10 show the DE2-115 FPGA board with the Quartus file uploaded to it.

A circuit board

Description automatically generated

**Figure 7.** Reset State. In this figure, the reset button has been pressed. All values have been zeroed out and players have been logged out of the board.

A circuit board

Description automatically generated

**Figure 8.** Access Granted. This image shows the green LED enabled after the user has entered the correct password using switches 10-7.

A circuit board

Description automatically generated

**Figure 9.** Player 1 input. This image shows the 7-segment displays after player 1 has used switches 17-14 to input a 4-bit value and then pressed the load button.

A circuit board

Description automatically generated

**Figure 10.** Player 2 input. Here, player 2 has used switches 3-0 to enter a 4-bit number and pressed their load button. The sum has also been computed and displayed.

The password control and load registers function in tandem with the button pusher modules to control player input. We can see through these pictures that the 4-bit binary adder is working and that inputs are properly being converted from their 4-bit form to their hexadecimal equivalent.

# Video Demo

The following links are to videos showing the operation of binary mental math game. For each lab.

Lab 1:

<https://drive.google.com/file/d/1CQv7KU_kPwXUhqRWP6rJahYrf8mgxPv_/view?usp=sharing>

Lab 2:

<https://drive.google.com/file/d/1FHTdyxH9T4_aaNNfueHTf-uf1d4xCvPb/view?usp=sharing>

# Conclusion

Lab 2 had us design and build an access controller for the binary mental math game built in lab 1. The game is a 2-player game that can be used to practice and test a person’s ability to convert numbers between hexadecimal and binary. Additionally, the game tests a person’s ability to add in binary. The goal of the game is for Player 2 to determine which binary number must be added to Player 1’s input in order to reach binary 1111. The additions made in lab 2 allow us to restrict access to only those players who have been given the correct password. Using the load register and button shaper modules, the game now displays player input only when the player presses the input button.

# Appendix

The following code includes all 7 files used for Lab 2.

**Access Controller:**

// ECE 5440

// Author: Mubashar Khan, 2949

// access\_controller

// This module is used to prevent access to the game until users have entered the correct password

// The program waits for players to enter a 4-digit password that has been predefined.

// The password that players should enter is 2949.

//

module access\_controller(reset, clock, input\_button\_press\_p1, input\_button\_press\_p2, password, pass\_enter, ouput\_to\_load\_reg\_p1, ouput\_to\_load\_reg\_p2, RedLED, GreenLED);

input reset, clock, input\_button\_press\_p1, input\_button\_press\_p2, pass\_enter;

input [3:0] password;

output ouput\_to\_load\_reg\_p1, ouput\_to\_load\_reg\_p2, RedLED, GreenLED;

reg password\_correct\_flag;

reg ouput\_to\_load\_reg\_p1, ouput\_to\_load\_reg\_p2, RedLED, GreenLED;

reg [2:0] state;

parameter [2:0] s1\_first\_digit\_check = 3'b000;

parameter [2:0] s2\_second\_digit\_check = 3'b001;

parameter [2:0] s3\_third\_digit\_check = 3'b010;

parameter [2:0] s4\_fourth\_digit\_check = 3'b011;

parameter [2:0] s5\_success = 3'b100; // state 4 will also check for successful password entry

always @(posedge clock)

begin

//reg [1:0] password\_correct\_flag;

if (reset == 1'b0)

begin //reset all output variables to default values, goto state 1

state <= s1\_first\_digit\_check;

ouput\_to\_load\_reg\_p1 <= 1'b0;

ouput\_to\_load\_reg\_p2 <= 1'b0;

GreenLED <= 1'b1;

RedLED <= 1'b1;

password\_correct\_flag <= 1'b1; //reset flag

end

else begin

case (state)

//State 1

s1\_first\_digit\_check:begin

ouput\_to\_load\_reg\_p1 <= 1'b0;

ouput\_to\_load\_reg\_p2 <= 1'b0;

GreenLED <= 1'b0;

RedLED <= 1'b1;

password\_correct\_flag <= 1'b1;

if (pass\_enter == 1'b1)

begin

if (password == 4'b0010)

begin

//PW is correct so far, no action needed

end

else begin

password\_correct\_flag <= 1'b0;

end

state <= s2\_second\_digit\_check; // goto state 2 for second input

end

else begin

state <= s1\_first\_digit\_check; //repeat s1 until first digit has been entered

end

end

//State 2

s2\_second\_digit\_check:begin

//no change to outputs

if (pass\_enter == 1'b1)

begin

if (password == 4'b1001)

begin

//PW is correct so far, no action needed

end

else begin

password\_correct\_flag <= 1'b0;

end

state <= s3\_third\_digit\_check; // goto state 3 for third input

end

else begin

state <= s2\_second\_digit\_check; //repeat s2 until second digit has been entered

end

end

//State 3

s3\_third\_digit\_check:begin

if (pass\_enter == 1'b1)

begin

if (password == 4'b0100)

begin

//PW is correct so far, no action needed

end

else begin

password\_correct\_flag <= 1'b0;

end

state <= s4\_fourth\_digit\_check; // goto state 4 for fourth input

end

else begin

state <= s3\_third\_digit\_check; //repeat s3 until first digit has been entered

end

end

//State 4

s4\_fourth\_digit\_check: begin

if (pass\_enter == 1'b1)

begin

if (password == 4'b1001)

begin

if(password\_correct\_flag == 1'b1) //password correct

begin

ouput\_to\_load\_reg\_p1 <= input\_button\_press\_p1;

ouput\_to\_load\_reg\_p2 <= input\_button\_press\_p2;

RedLED <= 1'b0;

GreenLED <= 1'b1;

state <= s5\_success; // goto state 5 for infinite success loop

end

end

else begin

password\_correct\_flag <= 1'b0;

end

if(password\_correct\_flag == 1'b0)

begin

state <= s1\_first\_digit\_check; // goto state 1 for first input if password is incorrect

end

end

else begin

state <= s4\_fourth\_digit\_check; //repeat s4 until first digit has been entered

end

end

//State 5

s5\_success: begin

ouput\_to\_load\_reg\_p1 <= input\_button\_press\_p1;

ouput\_to\_load\_reg\_p2 <= input\_button\_press\_p2;

RedLED <= 1'b0;

GreenLED <= 1'b1;

state <= s5\_success;

end

//default

default: begin

state <= s1\_first\_digit\_check;

end

endcase

end

end

endmodule

**Access Controller Testbench:**

// ECE 5440

// Author: Mubashar Khan, 2949

// access\_controller\_tb

// This module is used to test access\_controller.v

// The test cycles through the 4 states using a incorrect and correct password sequence to test if the module works properly

//

`timescale 1 ns/100 ps

module access\_controller\_tb ();

reg [0:0] reset, clock, ld\_p1, ld\_p2, pass\_enter;

reg [3:0] password;

wire [0:0] ld\_p1\_pst, ld\_p2\_pst; // 1 on success only

wire [0:0] RedLED, GreenLED; //on/off

access\_controller DUT\_access\_controller (reset, clock, ld\_p1, ld\_p2, password, pass\_enter, ld\_p1\_pst, ld\_p2\_pst, RedLED, GreenLED);

always begin //enable clock

#10 clock = 1'b1;

#10 clock = 1'b0;

end

initial begin //testing

#5

ld\_p1 = 1'b1;

ld\_p2 = 1'b1;

pass\_enter = 1'b0;

@(posedge clock);

reset = 1'b1;

@(posedge clock);

reset = 1'b0;

@(posedge clock);

reset = 1'b1;

//password is 2949

//incorrect input test

@(posedge clock);

#5 password = 4'b0010; // 2

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

#5 password = 4'b0100; // 4 <-should trigger password\_correct\_flag

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

#5 password = 4'b1000; // 8

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

#5 password = 4'b1001; // 9

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

//correct input test

//@(posedge clock);

//#5 reset = 0;

@(posedge clock);

#5 password = 4'b0010; // 2

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

#5 password = 4'b1001; // 9

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

@(posedge clock);

#5 password = 4'b0100; // 4

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

@(posedge clock);

@(posedge clock);

#5 password = 4'b1001; // 9

@(posedge clock);

#5 pass\_enter = 1'b1;

@(posedge clock);

#5 pass\_enter = 1'b0;

// now testing output if inputs are changed while in state 5

@(posedge clock);

@(posedge clock);

@(posedge clock);

#20

ld\_p1 = 1'b1;

ld\_p2 = 1'b0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

#20

ld\_p1 = 1'b0;

ld\_p2 = 1'b0;

@(posedge clock);

@(posedge clock);

@(posedge clock);

#20

ld\_p1 = 1'b1;

ld\_p2 = 1'b1;

//reset everything

@(posedge clock);

reset = 0;

@(posedge clock);

@(posedge clock);

reset = 1;

end

endmodule

**Load Register:**

// ECE 5440

// Author: Mubashar Khan, 2949

// load\_register

// This module is used to prevent inputs from showing up on the displays until the load button has been pressed

// The program waits for players to enter a 4-bit input and press the load button.

// The signal then allows the input to be displayed.

//

module load\_register(rst, clk, four\_bit\_input, button, four\_bit\_output);

input [3:0] four\_bit\_input;

input rst, clk, button;

output reg [3:0] four\_bit\_output;

always @(posedge clk) begin

if (rst == 1'b0) begin

four\_bit\_output <= 4'b0000;

end

else begin

if (button == 1'b1) begin

four\_bit\_output <= four\_bit\_input;

end

end

end

endmodule

**Load Register Testbench:**

// ECE 5440

// Author: Mubashar Khan, 2949

// load\_register\_tb

// This module is used to test the load register

// The program waits for players to press the load button.

//

`timescale 1 ns/100 ps

module load\_register\_tb();

reg clk, rst, button;

reg [3:0]load;

reg current\_state = 2'b00;

wire [3:0] out\_1;

load\_register DUT\_load\_register(rst, clk, load, button, out\_1);

always begin

#10 clk = 1;

#10 clk = 0;

end

initial begin

@(posedge clk);

#5 button = 0;

#5 load = 4'b1001;

@(posedge clk);

#5 rst = 1;

@(posedge clk);

#5 rst = 0;

@(posedge clk);

#5 rst = 1;

@(posedge clk);

#5 button = 1;

@(posedge clk);

#5 button = 0;

end

endmodule

**Button Shaper:**

// ECE 5440

// Author: Mubashar Khan, 2949

// button\_shaper

// This module is a button shaper that emits a pulse for 1 clock cycle when the button is pressed

// This module was created using a two procedure fsm

//

module button\_shaper(reset, clk, button\_shaper\_input, button\_shaper\_output);

input clk, reset, button\_shaper\_input;

output reg button\_shaper\_output;

reg [1:0] current\_state;

reg [1:0] next\_state;

parameter [1:0] init\_state = 2'b00;

parameter [1:0] pulse\_state = 2'b01;

parameter [1:0] wait\_state = 2'b10;

always @ (current\_state, button\_shaper\_input)

begin

case(current\_state)

init\_state:

begin

button\_shaper\_output <= 1'b0;

if (button\_shaper\_input == 1'b0)

next\_state <= pulse\_state;

else

next\_state <= init\_state;

end

pulse\_state:

begin

button\_shaper\_output <= 1'b1;

next\_state <= wait\_state;

end

wait\_state:

begin

button\_shaper\_output <= 1'b0;

if (button\_shaper\_input == 1'b1)

next\_state <= init\_state;

else

next\_state <= wait\_state;

end

default:

begin

button\_shaper\_output <= 1'b0;

next\_state <= init\_state;

end

endcase

end

always @(posedge clk)

begin

if (reset == 1'b0)

current\_state <= init\_state;

else

current\_state <= next\_state;

end

endmodule

**Button Shaper Testbench:**

// ECE 5440

// Author: Mubashar Khan, 2949

// button\_shaper\_tb

// This module is a button shaper that emits a pulse for 1 clock cycle when the button is pressed

// This module was created using a two procedure fsm

//

`timescale 10ns/100ps

module button\_shaper\_tb ();

reg clk;

reg reset;

reg b\_in;

reg current\_state = 2'b00;

wire b\_out;

button\_shaper DUT\_button\_shaper\_1 (clk, reset, b\_in, b\_out);

always begin

#10 clk = 1;

#10 clk = 0;

end

initial

begin

//test 1

@(posedge clk);

#5 reset = 1;

@(posedge clk);

#5 reset = 0;

@(posedge clk);

#5 reset = 1;

@(posedge clk);

#5 b\_in = 1;

@(posedge clk);

#5 b\_in = 0;

@(posedge clk);

#5 b\_in = 0;

@(posedge clk);

#5 b\_in = 1;

//test 2

@(posedge clk);

#5 reset = 0;

@(posedge clk);

#5 reset = 1;

@(posedge clk);

#5 reset = 0;

@(posedge clk);

#5 b\_in = 1;

@(posedge clk);

#5 b\_in = 1;

@(posedge clk);

#5 b\_in = 0;

@(posedge clk);

#5 b\_in = 1;

end

endmodule

**Top Level File for Lab 2:**

// ECE 5440

// Author: Mubashar Khan, 2949

// Lab2\_KHAN\_M

// Top level file for lab 2.

//

module Lab2\_KHAN\_M (reset, clock, password\_button, player1\_button, player2\_button, password, red\_LED, green\_LED, p1\_four\_bit\_input, p2\_four\_bit\_input,seg7\_num1, seg7\_num2, seg7\_sum);

//lab 2 variables

input clock, reset, password\_button, player1\_button, player2\_button;

input [3:0] password;

output red\_LED, green\_LED;

wire player1\_button\_pressed, player2\_button\_pressed, password\_button\_pressed, ac\_to\_ld\_reg\_p1, ac\_to\_ld\_reg\_p2;

wire [3:0] output\_from\_ld\_reg\_p1, output\_from\_ld\_reg\_p2;

//lab 1 variables

input [3:0] p1\_four\_bit\_input, p2\_four\_bit\_input;

output [6:0] seg7\_num1, seg7\_num2, seg7\_sum;

wire [3:0] sum;

button\_shaper player1\_button\_shaper(reset, clock, player1\_button, player1\_button\_pressed);

button\_shaper player2\_button\_shaper(reset, clock, player2\_button, player2\_button\_pressed);

button\_shaper password\_button\_shaper(reset, clock, password\_button, password\_button\_pressed);

access\_controller game\_access (reset, clock, player1\_button\_pressed, player2\_button\_pressed, password, password\_button\_pressed, ac\_to\_ld\_reg\_p1, ac\_to\_ld\_reg\_p2, red\_LED, green\_LED);

load\_register player1\_load (reset, clock, p1\_four\_bit\_input, ac\_to\_ld\_reg\_p1, output\_from\_ld\_reg\_p1);

load\_register player2\_load (reset, clock, p2\_four\_bit\_input, ac\_to\_ld\_reg\_p2, output\_from\_ld\_reg\_p2);

//lab 1 function calls, all are working correctly

adder adder\_1 (output\_from\_ld\_reg\_p1, output\_from\_ld\_reg\_p2, sum);

Seven\_Seg seg7\_number1 (output\_from\_ld\_reg\_p1, seg7\_num1); //the value sent to the 7 segment display is from

Seven\_Seg seg7\_number2 (output\_from\_ld\_reg\_p2, seg7\_num2);

Seven\_Seg seg7\_summation (sum, seg7\_sum);

endmodule

The following is all the code created for this lab. It consists of 5 modules, Lab1\_KHAN\_M, Seven\_Seg.v, Seven\_Seg\_tb.v, adder.v, and adder\_tb.v.

// ECE 5440/6370

// Author: Mubashar Khan, 2949

// Lab1\_KHAN\_M

// This module in the top code that calls all necessary functions

// Its function is to take thhe 4 bit input from players and pass those values to the adder and decoder functions.

// The returned value from the adder function is also passed to a decoder and all 3 decoder values are used to enable the proper pins on the 7-segment display.

//

module Lab1\_KHAN\_M (

//inputs

switch\_num1, switch\_num2,

//outputs

seg7\_num1, seg7\_num2, seg7\_sum);

input [3:0] switch\_num1, switch\_num2;

output [6:0] seg7\_num1, seg7\_num2, seg7\_sum;

wire [3:0] sum;

adder adder\_1 (switch\_num1, switch\_num2, sum);

Seven\_Seg seg7\_number1 (switch\_num1, seg7\_num1);

Seven\_Seg seg7\_number2 (switch\_num2, seg7\_num2);

Seven\_Seg seg7\_summation (sum, seg7\_sum);

Endmodule

// Seven\_Seg

// This module is the code for the 4-16 decoder.

// Its function is to take a 4-bit input and convert it to a 7-segment number that can be displayed on a 7-segment display

//

module Seven\_Seg (Seg\_in, Seg\_out);

input [3:0] Seg\_in;

output [6:0] Seg\_out;

reg [6:0] Seg\_out;

always @ (Seg\_in)

begin

case(Seg\_in)

4'b0000 : begin Seg\_out = 7'b1000000; end //0 //7'b1000000; end 7'b0000001; end

4'b0001 : begin Seg\_out = 7'b1111001; end //1 //7'b1111001; end 7'b1001111; end

4'b0010 : begin Seg\_out = 7'b0100100; end //2 //7'b0100100; end 7'b0010010; end

4'b0011 : begin Seg\_out = 7'b0110000; end //3 //7'b0110000; end 7'b0000110; end

4'b0100 : begin Seg\_out = 7'b0011001; end //4 //7'b0011001; end 7'b1001100; end

4'b0101 : begin Seg\_out = 7'b0010010; end //5 //7'b0010010; end 7'b0100100; end

4'b0110 : begin Seg\_out = 7'b0000010; end //6 //7'b0000010; end 7'b0100000; end

4'b0111 : begin Seg\_out = 7'b1111000; end //7 //7'b1111000; end 7'b0001111; end

4'b1000 : begin Seg\_out = 7'b0000000; end //8 //7'b0000000; end 7'b0000000; end

4'b1001 : begin Seg\_out = 7'b0011000; end //9 //7'b0011000; end 7'b0001100; end

4'b1010 : begin Seg\_out = 7'b0001000; end //A //7'b0001000; end 7'b0001000; end

4'b1011 : begin Seg\_out = 7'b0000011; end //b //7'b0000011; end 7'b1100000; end

4'b1100 : begin Seg\_out = 7'b1000110; end //C //7'b1000110; end 7'b0110001; end

4'b1101 : begin Seg\_out = 7'b0100001; end //d //7'b0100001; end 7'b1000010; end

4'b1110 : begin Seg\_out = 7'b0000110; end //E //7'b0000110; end 7'b0110000; end

4'b1111 : begin Seg\_out = 7'b0001110; end //F //7'b0001110; end 7'b0111000; end

default : begin Seg\_out = 7'b1000000; end //0 //7'b1000000; end 7'b0000001; end

endcase

end

endmodule

// Seven\_Seg\_tb

// This module is the test bench for the 4-16 decoder

// Its function is to be used to test sample inputs and observe their outputs.

//

`timescale 10ns/100ps

module Seven\_Seg\_tb ();

reg[3:0] Seg\_in;

wire[6:0] Seg\_out;

Seven\_Seg DUTSeven\_Seg(Seg\_in,Seg\_out);

initial

begin

Seg\_in = 4'b0000; // 0

#10;

Seg\_in = 4'b0001; // 1

#10;

Seg\_in = 4'b0010; // 2

#10;

Seg\_in = 4'b0011; // 3

#10;

Seg\_in = 4'b0100; // 4

#10;

Seg\_in = 4'b0101; // 5

#10;

Seg\_in = 4'b0110; // 6

#10;

Seg\_in = 4'b0111; // 7

#10;

Seg\_in = 4'b1000; // 8

#10;

Seg\_in = 4'b1001; // 9

#10;

Seg\_in = 4'b1010; // A

#10;

Seg\_in = 4'b1011; // b

#10;

Seg\_in = 4'b1100; // C

#10;

Seg\_in = 4'b1101; // d

#10;

Seg\_in = 4'b1110; // E

#10;

Seg\_in = 4'b1111; // F

end

endmodule

// adder

// This module adds 2 four-bit inputs. We do not care about overflow and the output is also four-bits.

//

module adder (in1, in2, sum);

input [3:0] in1, in2;

output [3:0] sum;

reg [3:0] sum;

always @ (in1, in2)

begin

sum = in1 + in2;

end

endmodule

// adder\_tb

// This module is the test bench for the adder

// Its function is to be used to test sample inputs and observe their outputs.

//

`timescale 10ns/100ps

module adder\_tb ();

reg [3:0] in1, in2;

wire [3:0] out1;

adder DUT\_adder(in1, in2, out1);

initial

begin

in1 = 4'b1001;

in2 = 4'b0110;

#10;

in1 = 4'b0011;

in2 = 4'b1111;

end

endmodule

The following is all the code created for Lab 1. It consists of 5 modules, Lab1\_KHAN\_M, Seven\_Seg.v, Seven\_Seg\_tb.v, adder.v, and adder\_tb.v.

// ECE 5440/6370

// Author: Mubashar Khan, 2949

// Lab1\_KHAN\_M

// This module in the top code that calls all necessary functions

// Its function is to take thhe 4 bit input from players and pass those values to the adder and decoder functions.

// The returned value from the adder function is also passed to a decoder and all 3 decoder values are used to enable the proper pins on the 7-segment display.

//

module Lab1\_KHAN\_M (

//inputs

switch\_num1, switch\_num2,

//outputs

seg7\_num1, seg7\_num2, seg7\_sum);

input [3:0] switch\_num1, switch\_num2;

output [6:0] seg7\_num1, seg7\_num2, seg7\_sum;

wire [3:0] sum;

adder adder\_1 (switch\_num1, switch\_num2, sum);

Seven\_Seg seg7\_number1 (switch\_num1, seg7\_num1);

Seven\_Seg seg7\_number2 (switch\_num2, seg7\_num2);

Seven\_Seg seg7\_summation (sum, seg7\_sum);

Endmodule

// Seven\_Seg

// This module is the code for the 4-16 decoder.

// Its function is to take a 4-bit input and convert it to a 7-segment number that can be displayed on a 7-segment display

//

module Seven\_Seg (Seg\_in, Seg\_out);

input [3:0] Seg\_in;

output [6:0] Seg\_out;

reg [6:0] Seg\_out;

always @ (Seg\_in)

begin

case(Seg\_in)

4'b0000 : begin Seg\_out = 7'b1000000; end //0 //7'b1000000; end 7'b0000001; end

4'b0001 : begin Seg\_out = 7'b1111001; end //1 //7'b1111001; end 7'b1001111; end

4'b0010 : begin Seg\_out = 7'b0100100; end //2 //7'b0100100; end 7'b0010010; end

4'b0011 : begin Seg\_out = 7'b0110000; end //3 //7'b0110000; end 7'b0000110; end

4'b0100 : begin Seg\_out = 7'b0011001; end //4 //7'b0011001; end 7'b1001100; end

4'b0101 : begin Seg\_out = 7'b0010010; end //5 //7'b0010010; end 7'b0100100; end

4'b0110 : begin Seg\_out = 7'b0000010; end //6 //7'b0000010; end 7'b0100000; end

4'b0111 : begin Seg\_out = 7'b1111000; end //7 //7'b1111000; end 7'b0001111; end

4'b1000 : begin Seg\_out = 7'b0000000; end //8 //7'b0000000; end 7'b0000000; end

4'b1001 : begin Seg\_out = 7'b0011000; end //9 //7'b0011000; end 7'b0001100; end

4'b1010 : begin Seg\_out = 7'b0001000; end //A //7'b0001000; end 7'b0001000; end

4'b1011 : begin Seg\_out = 7'b0000011; end //b //7'b0000011; end 7'b1100000; end

4'b1100 : begin Seg\_out = 7'b1000110; end //C //7'b1000110; end 7'b0110001; end

4'b1101 : begin Seg\_out = 7'b0100001; end //d //7'b0100001; end 7'b1000010; end

4'b1110 : begin Seg\_out = 7'b0000110; end //E //7'b0000110; end 7'b0110000; end

4'b1111 : begin Seg\_out = 7'b0001110; end //F //7'b0001110; end 7'b0111000; end

default : begin Seg\_out = 7'b1000000; end //0 //7'b1000000; end 7'b0000001; end

endcase

end

endmodule

// Seven\_Seg\_tb

// This module is the test bench for the 4-16 decoder

// Its function is to be used to test sample inputs and observe their outputs.

//

`timescale 10ns/100ps

module Seven\_Seg\_tb ();

reg[3:0] Seg\_in;

wire[6:0] Seg\_out;

Seven\_Seg DUTSeven\_Seg(Seg\_in,Seg\_out);

initial

begin

Seg\_in = 4'b0000; // 0

#10;

Seg\_in = 4'b0001; // 1

#10;

Seg\_in = 4'b0010; // 2

#10;

Seg\_in = 4'b0011; // 3

#10;

Seg\_in = 4'b0100; // 4

#10;

Seg\_in = 4'b0101; // 5

#10;

Seg\_in = 4'b0110; // 6

#10;

Seg\_in = 4'b0111; // 7

#10;

Seg\_in = 4'b1000; // 8

#10;

Seg\_in = 4'b1001; // 9

#10;

Seg\_in = 4'b1010; // A

#10;

Seg\_in = 4'b1011; // b

#10;

Seg\_in = 4'b1100; // C

#10;

Seg\_in = 4'b1101; // d

#10;

Seg\_in = 4'b1110; // E

#10;

Seg\_in = 4'b1111; // F

end

endmodule

// adder

// This module adds 2 four-bit inputs. We do not care about overflow and the output is also four-bits.

//

module adder (in1, in2, sum);

input [3:0] in1, in2;

output [3:0] sum;

reg [3:0] sum;

always @ (in1, in2)

begin

sum = in1 + in2;

end

endmodule

// adder\_tb

// This module is the test bench for the adder

// Its function is to be used to test sample inputs and observe their outputs.

//

`timescale 10ns/100ps

module adder\_tb ();

reg [3:0] in1, in2;

wire [3:0] out1;

adder DUT\_adder(in1, in2, out1);

initial

begin

in1 = 4'b1001;

in2 = 4'b0110;

#10;

in1 = 4'b0011;

in2 = 4'b1111;

end

endmodule