Lab 2: Game Access Control on FPGA

Jose Navarro

ECE5440

4207

**Introduction:**

The Game Access Control is an extension of the Mental Binary Game. The Mental Binary Game is a simple two player game testing the players ability to convert hexadecimal numbers into binary numbers. In this game player one will think of a binary number and input the number to the FPGA using the FPGA switches. Without looking at player one’s switches, player two is to look at the FPGA’s seven segment decoder displaying player one’s number in hexadecimal. Player two is then to input the number, using the switches on the FPGA board, they believe will sum with player one’s number to equal 15. Player two’s goal is to mentally come up with the number that when added to player one’s number the total sum will equal 15 (or F when displayed on the FPGA). Player one’s goal is to come up with a number difficult enough so that player two is not able to come up with the right number to match the original output. If player two comes up with the correct number to match player one’s number player two gets a point. If player two does not come up with the correct number player one gets a point. Players change roles after each point is scored and play as long as they desire. The Game Access Control adds a level of security to the game. In order to play the game a user must input a four-digit password prior to being able to play the game. If the user inputs the correct password the game will be available to play, otherwise the FPGA will continue to accept password attempts. Once allowed to play the game player 1 and two will be able to input their numbers and play the game. The Game Access also adds a confirmation button. Player inputs will not be displayed until the confirmation button is pressed.

**System Architecture Design:**

Figure 1 below shows the top-level system architecture of the Game Access Control.

A close up of text on a white background

Description automatically generated

*Figure 1. Top-Level System Architecture for Access Control on FPGA*

The figure above shows the system as a whole. A player is first to input the password, to be approved by the Access Controller. Afterwards both players may enter their numbers and confirm their input with the Load Register. Finally, the two binary inputs are summed together and decoded using a seven segment display as shown.

Figure 2 below shows the Finite State Machine (FSM) design of the Button Shaper module.

A picture containing text

Description automatically generated

*Figure 2. FSM of Button Shaper Module*

The Button Shaper is a two procedure FSM. Meaning that there are two “always” blocks running simultaneously. One is tasked with outputting values at the correct times and the other is tasked with changing the state the FSM is in. As shown in Figure 2 the Button Shaper has three states. The first state, also known as the initial state, outputs a value of zero and continues to do so, so long as the button is not pressed (an input of 1). The FSM will move onto the next state when button is pressed (an input of 0). The state that it will move on to is known as the pulse state. This will output a value of one for a short period of time and move onto the next state. The next state is known as the wait state. The output at this state will be a zero and the FSM will stay at this state so long the button is pressed (an input of 0) and will move on back to the initial state when the button is let go (an input of 1). The expected wave forms from this FSM according the previous description are shown in Figure 3 below.

A screenshot of a cell phone

Description automatically generated

*Figure 3. Expected Wave Forms for Button Shaper FSM*

Figure 4 on the next page shows the FSM design for the Access Control Module.

A picture containing text

Description automatically generated

*Figure 4. FSM of Access Controller*

The Access Control Module has 6 states and is a single procedure FSM. Therefore, there is only a single “always” block in charge of outputs and directing the FSM to the next state. The first state, also the “First Number” state outputs a zero to both player input pass variables and the green LED (indicating access). Regardless of whether the user inputs the correct or incorrect password, so long as the user presses the access button the FSM will eventually reach the Flag Check state. If the password is correct access will be granted, indicated by a one output to green, and player pass variables will pass the input of the player buttons. If the password is incorrect the FSM will move to the First Number state and allow the user another try to input the password.

**Simulation Results:**

Figure 5 and 6 on the next page show the simulation results for the Access Control Module. As shown the Access Control properly works. It does not allow the user to pass their input when attempting an incorrect password (1234) but does allow the user to pass their input when attempting a correct password (4207).

A screenshot of a computer

Description automatically generated

*Figure 5. Password attempt: 1234 not allowing user to pass input*

*A screenshot of a video game

Description automatically generated*

*Figure 6. Password attempt: 4207 allowing user to pass input*

Figure 7 shows the simulation results for the Load Register Module. This module is created so that player inputs don’t pass to the seven-segment display until they are confirmed by the player. As shown by the Figure on the next page the module works correctly, not passing inputs through when not confirmed while passing inputs when they are confirmed.

A screenshot of a computer

Description automatically generated

*Figure 7. Simulation Result for Load Register Module Test*

Figure 8 below shows the simulation results for the Button Shaper Module. As shown in the Figure the module acts as expected. Turning the negative logic button push into a positive logic pulse which simplifies its use when implementing with other modules.

A screenshot of a video game

Description automatically generated

*Figure 8. Simulation Results for Button Shaper Module Test*

**FPGA Board Testing Results:** Figure 9 below shows the FPGA when initially turned on with the Game Access Control modules. As shown the board does not display any numbers and players can’t input their numbers into the board, this is indicated by the two red LED’s lit up.

*A circuit board

Description automatically generated*

*Figure 9. Initial set up of FPGA board when first turned on*

Figure 10 on the next page shows the FPGA board after the access has been granted by inputting the correct pass code (4207), indicated by the green LED lighting up and before player one and payer two have confirmed their numbers. As shown despite having flipped switches the displays still show a value of zero.

A circuit board

Description automatically generated

*Figure 10. FPGA board after access has been granted and players have not confirmed numbers*

Figures 11 and 12 show a correct confirmed match, indicated by the second green light lighting up and an incorrect confirmed match by player 2, indicated by the red LED still lit up.

A circuit board

Description automatically generated

*Figure 11. Player 2 inputs and confirms a correct match to player 1’s number*

*A circuit board

Description automatically generated*

*Figure 12. Player 2 inputs and confirms an incorrect match to player 1’s number*

The log out feature is displayer in Figure 13 below. When the access button is pressed again after being allowed access the current inputs are saved and locked into the display. However, access is no longer granted, therefore players cannot input numbers to the board, this is indicated by the two LED’s lit up.

A circuit board

Description automatically generated

*Figure 13. Players have logged out of FPGA and no longer have access to change input, while last input is still saved*

**Video Demo:**

<https://drive.google.com/file/d/15eRojbqSy8LEagXuaOl5oPHS3gGZge3q/view?usp=sharing>

**Conclusion:**

The Game Access Control allows the user of the FPGA to have a simple form of security to ensure not just anyone can play the Mental Binary Math Game. Using the FPGA board and modules written, shown visually in Figure 1, and code in the Appendix below, the lab is fully completed. The bonus feature included in this lab is the “log-out” feature. If a player wishes to log out mid game without resetting the board the player can push the “Access Button” (see Figure 1) again, to log out of the board.

**Appendix:**

loadRegister.v:

module loadRegister(clk, rst, loadPlayer, playerInput, playerOutput);

input clk, rst, loadPlayer;

input [3:0] playerInput;

output [3:0] playerOutput;

reg [3:0] playerOutput;

always @ (posedge clk)

begin

if(rst == 0)

begin

playerOutput <= 0;

end

else

begin

if(loadPlayer == 1)

begin

playerOutput <= playerInput;

end

end

end

endmodule

loadRegister\_tb.v:

`timescale 10ns/100ps

module loadRegister\_tb();

reg clk;

reg rst;

reg loadPlayer1;

reg [3:0] playerInput1;

wire [3:0] playerOutput1;

loadRegister DUT\_loadRegister\_1(clk, rst, loadPlayer1, playerInput1, playerOutput1);

always

begin

#10 clk = 1;

#10 clk = 0;

end

initial

begin

#70 rst = 0;

loadPlayer1 = 0;

playerInput1 = 4'b1011;

#70 rst = 1;

loadPlayer1 = 0;

playerInput1 = 4'b1110;

#70 rst = 0;

loadPlayer1 = 1;

playerInput1 = 4'b1010;

#70 rst = 1;

loadPlayer1 = 1;

playerInput1 = 4'b1111;

end

endmodule

buttonShaper.v:

module buttonShaper(clk, rst, buttonInput, buttonOutput);

input clk, rst, buttonInput;

output buttonOutput;

reg buttonOutput;

reg [1:0] state, nextState;

parameter initialState = 2'b00;

parameter pulseState = 2'b01;

parameter waitState = 2'b10;

parameter state3 = 2'b11;

always @ (posedge clk)

begin

if(rst == 0) //might be a one but not sure

begin

state <= initialState;

end

else

begin

state <= nextState;

end

end

always @ (state, buttonInput)

begin

case(state)

initialState:

begin

buttonOutput = 1'b0;

if(buttonInput == 1)

begin

nextState <= initialState;

end

else

begin

nextState <= pulseState;

end

end

pulseState:

begin

buttonOutput = 1'b1;

nextState <= waitState;

end

waitState:

begin

buttonOutput = 1'b0;

if(buttonInput == 1)

begin

nextState <= initialState;

end

else

begin

nextState <= waitState;

end

end

default:

begin

buttonOutput = 1'b0;

nextState = initialState;

end

endcase

end

endmodule

buttonShaper\_tb.:

`timescale 10ns/100ps

module buttonShaper\_tb();

reg clk;

reg rst;

reg buttonIn;

wire buttonOut;

buttonShaper DUT\_buttonShaper\_1(clk, rst, buttonIn, buttonOut);

always

begin

#10 clk = 1;

#10 clk = 0;

end

initial

begin

#3 rst = 0;

buttonIn = 0;

#20 rst = 1;

buttonIn = 0;

#40 rst = 0;

buttonIn = 1;

#70 rst = 1;

buttonIn = 1;

end

endmodule

accessController.v:

module accessController(clk, rst, loadPlayer1, loadPlayer2, passWord, passWordEnter, loadPlayer1Pass, loadPlayer2Pass, green, red);

input clk, rst, loadPlayer1, loadPlayer2, passWordEnter;

input [3:0] passWord;

output loadPlayer1Pass, loadPlayer2Pass, green, red;

reg loadPlayer1Pass, loadPlayer2Pass, green, red;

reg passWordFlag;

reg [2:0] state;

parameter state0 = 3'b000, state1 = 3'b001;

parameter state2 = 3'b010, state3 = 3'b011;

parameter state4 = 3'b100, state5 = 3'b101;

always @ (posedge clk)

begin

passWordFlag = 1'b1;

if(rst == 0)

begin

state <= state0;

passWordFlag = 1'b1;

end

else

begin

case(state)

state0:

begin

loadPlayer1Pass <= 1'b0; //might need to change to arrows

loadPlayer2Pass <= 1'b0;

green <= 1'b0;

red <= 1'b1;

if(passWordEnter == 1'b1)

begin

if(passWord == 4'b0100)

begin

//good

end

else

begin

//not good forever, update flag

passWordFlag = 1'b0;

end

//go to next state

state <= state1;

end

else

begin

//stay in this state

state <= state0;

end

end

state1:

begin

loadPlayer1Pass <= 1'b0; //might need to change to arrows

loadPlayer2Pass <= 1'b0;

green <= 1'b0;

red <= 1'b1;

if(passWordEnter == 1'b1)

begin

if(passWord == 4'b0010)

begin

//good

end

else

begin

//not good forever

passWordFlag = 1'b0;

end

state <= state2;

end

else

begin

//stay in this state

state <= state1;

end

end

state2:

begin

loadPlayer1Pass <= 1'b0; //might need to change to arrows

loadPlayer2Pass <= 1'b0;

green <= 1'b0;

red <= 1'b1;

if(passWordEnter == 1'b1)

begin

if(passWord == 4'b0000)

begin

//good

end

else

begin

//not good forever

passWordFlag = 1'b0;

end

state <= state3;

end

else

begin

//stay in this state

state <= state2;

end

end

state3:

begin

loadPlayer1Pass <= 1'b0; //might need to change to arrows

loadPlayer2Pass <= 1'b0;

green <= 1'b0;

red <= 1'b1;

if(passWordEnter == 1'b1)

begin

if(passWord == 4'b0111)

begin

//good

end

else

begin

//not good forever

passWordFlag = 1'b0;

end

state <= state4;

end

else

begin

//stay in this state

state <= state3;

end

end

state4:

begin

if(passWordFlag == 1'b0)

begin

state <= state0;

end

else

begin

state <= state5;

end

end

state5:

begin

red <= 1'b0;

green <= 1'b1;

loadPlayer1Pass <= loadPlayer1;

loadPlayer2Pass <= loadPlayer2;

end

default:

begin

state <= state0;

passWordFlag = 1'b1;

end

endcase

end

end

endmodule

accessController\_tb.v:

module accessController\_tb();

reg[3:0] passIn;

reg Clk, Rst, loadPass, p1\_Load, p2\_Load;

wire p1\_LoadOut, p2\_LoadOut, green, red;

accessController DUT\_accessController\_1(Clk, Rst, p1\_Load, p2\_Load, passIn, loadPass, p1\_LoadOut, p2\_LoadOut, green, red);

always

begin

#10 Clk = 1;

#10 Clk = 0;

end

initial

begin

// Following three signals come from the output of

// the button shaper.

// Need to generate a single cycle pulse when

// the button is pushed.

p1\_Load = 0;

p2\_Load = 0;

loadPass = 0;

Rst = 1;

@(posedge Clk);

@(posedge Clk);

// Reset the system

Rst = 0;

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

Rst = 1;

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

// test inputing incorrect pass

// align the the next rising clock edge

@(posedge Clk);

// Change data 5 ns after the rising edge of the clock

#5 passIn = 4'b0000;

@(posedge Clk);

#5 loadPass = 1;

@(posedge Clk);

#5 loadPass = 0;

@(posedge Clk);

#5 passIn = 4'b0011;

@(posedge Clk);

#5 loadPass = 1;

@(posedge Clk);

#5 loadPass = 0;

@(posedge Clk);

#5 passIn = 4'b0010;

@(posedge Clk);

#5 loadPass = 1;

@(posedge Clk);

#5 loadPass = 0;

@(posedge Clk);

#5 passIn = 4'b0011;

@(posedge Clk);

#5 loadPass = 1;

@(posedge Clk);

#5 loadPass = 0;

// Test if the Access Controll blocks the player's load signals

// before the board is authenticated

// P1\_LoadOut should remain 0.

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

#5 p1\_Load = 1;

@(posedge Clk);

#5 p1\_Load = 0;

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

// P2\_LoadOut should remain 0.

#5 p2\_Load = 1;

@(posedge Clk);

#5 p2\_Load = 0;

// test inputing correct pass

// Password "1234"

// Replace with your LAST FOUR DIGITS OF ID

@(posedge Clk);

#5 passIn = 4'b0100;

@(posedge Clk);

#5 loadPass = 1;

@(posedge Clk);

#5 loadPass = 0;

@(posedge Clk);

#5 passIn = 4'b0010;

@(posedge Clk);

#5 loadPass = 1;

@(posedge Clk);

#5 loadPass = 0;

@(posedge Clk);

#5 passIn = 4'b0000;

@(posedge Clk);

#5 loadPass = 1;

@(posedge Clk);

#5 loadPass = 0;

@(posedge Clk);

#5 passIn = 4'b0111;

@(posedge Clk);

#5 loadPass = 1;

@(posedge Clk);

#5 loadPass = 0;

// Test if the Access Controll blocks the player's load signals

// after the board is authenticated

// P1\_LoadOut should be 1 for one cycle.

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

#5 p1\_Load = 1;

@(posedge Clk);

#5 p1\_Load = 0;

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

// P2\_LoadOut should be 1 for one cycle.

#5 p2\_Load = 1;

@(posedge Clk);

#5 p2\_Load = 0;

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

@(posedge Clk);

end

endmodule

adder.v :

module adder(num1, num2, sum);

input [3:0] num1, num2;

output [3:0] sum;

reg [3:0] sum;

always @ (num1, num2)

begin

sum = num1 + num2;

end

endmodule

adder\_tb.v :

`timescale 10ns/100ps

module adder\_tb();

reg [3:0] in1, in2;

wire [3:0] out1;

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

initial

begin

in1 = 0;

in2 = 0;

#10;

in1 = 1;

in2 = 2;

#10;

in1 = 4'b0011;

in2 = 4'b1111;

end

endmodule

sevenSegDecoder.v :

module sevenSegDecoder(numIn, numOut);

input [3:0] numIn;

output [6:0] numOut;

reg [6:0] numOut;

always @ (numIn)

begin

case(numIn)

4'b0000: begin numOut = 7'b1000000; end

4'b0001: begin numOut = 7'b1111001; end

4'b0010: begin numOut = 7'b0100100; end

4'b0011: begin numOut = 7'b0110000; end

4'b0100: begin numOut = 7'b0011001; end

4'b0101: begin numOut = 7'b0010010; end

4'b0110: begin numOut = 7'b0000010; end

4'b0111: begin numOut = 7'b1111000; end

4'b1000: begin numOut = 7'b0000000; end

4'b1001: begin numOut = 7'b0011000; end

4'b1010: begin numOut = 7'b0001000; end

4'b1011: begin numOut = 7'b0000011; end

4'b1100: begin numOut = 7'b1000110; end

4'b1101: begin numOut = 7'b0100001; end

4'b1110: begin numOut = 7'b0000110; end

4'b1111: begin numOut = 7'b0001110; end

default: begin numOut = 7'b1111111; end

endcase

end

endmodule

sevenSegDecoder\_tb.v :

`timescale 10ns/100ps

module sevenSegDecoder\_tb();

reg [3:0] inputPin;

wire [6:0] outputSeg;

sevenSegDecoder DUT\_sevenSegDecoder(inputPin, outputSeg);

initial

begin

inputPin = 4'b0000;

#10;

inputPin = 4'b0001;

#10;

inputPin = 4'b0010;

#10;

inputPin = 4'b0011;

#10;

inputPin = 4'b0100;

#10;

inputPin = 4'b0101;

#10;

inputPin = 4'b0110;

#10;

inputPin = 4'b0111;

#10;

inputPin = 4'b1000;

#10;

inputPin = 4'b1001;

#10;

inputPin = 4'b1010;

#10;

inputPin = 4'b1011;

#10;

inputPin = 4'b1100;

#10;

inputPin = 4'b1101;

#10;

inputPin = 4'b1110;

#10;

inputPin = 4'b1111;

#10;

end

endmodule

ledToggle.v :

module ledToggle(sum, red, green);

input [3:0] sum;

output red, green;

reg red, green;

always @ (sum)

begin

if (sum == 15)

begin

red = 0;

green = 1;

end

else

begin

red = 1;

green = 0;

end

end

endmodule