Design Project

12pm-2pm

Calvin Liu

804182525

Darin Minamoto

704140102

**Introduction:**

The purpose of the lab was to design something using digital logic design. For our design project, we decided to create a game in which the user would input a combination of the 8 switches on the FPGA and if correct, will win the game. If wrong the user will be prompted to choose another combination of switches. The number of tries is limited for the user and there is a timer that counts down and determines the number of points the user gets for winning the game. The faster the user wins the game, the more points they get. The number of points starts at 200 and the points count down the longer the user doesn’t get the right answer. To make it easier for the user, the FPGA seven segment display show how many tries and the number of switches that is correct, but not which ones.

**Design:**

For the point system we decided to base it off the time of completion and give them a limited amount of time to guess the right combination so we decided to make the point system based off a clock and when the number of points becomes 0 then the user loses. For this we used a clock module to demultiplex the master clock and create a timer.

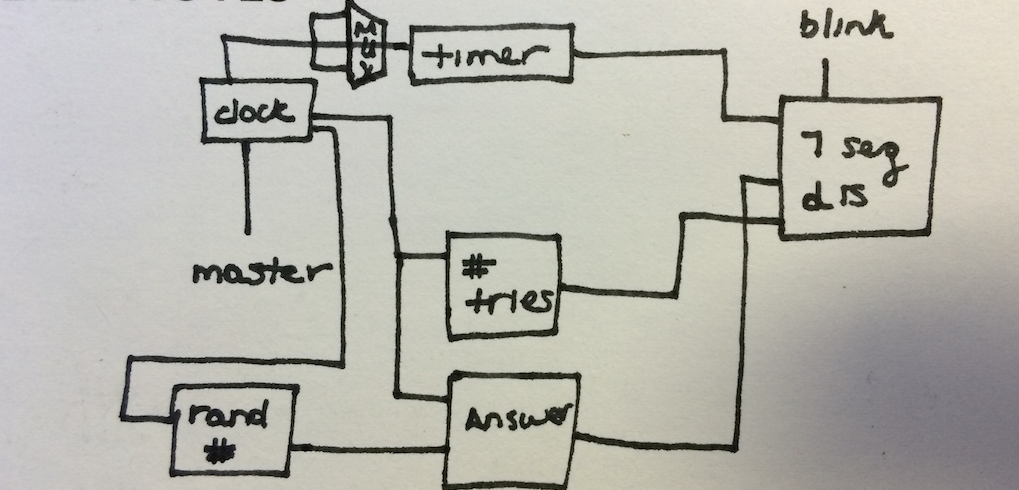
            To check for the right combination we did a brute force technique to check each switch with the random number we generated. We cycled through the 8 bits and checked each one to see if it is correct.

            To tell if the user has won or lost, we implemented blinking part of our experiment where if the user won the game then it would blink the number of points left and if the user lost the computer would blink the words ‘L’, ‘O’, ‘S’, ‘E’, on the seven segment display.

            For the buttons we used the middle button to be used for the submit button to check whether the right combination of switches was given and the right button was used as a reset button for resetting the game.

            The answer to the combination was also random. To make the random number generator we used a clock that would count really fast and stop when the user first presses the submit button. The submit button would then stop the counter and that number would be the solution.

A simple schematic of our design:



**Implementation:**

We had to demultiplex the clock from the master clock. We made simple counters to do this that counted at a certain frequency in relation to the frequency of the masterclock. Sample:

always @(posedge masterclock)

begin

///////////////////500Hz//////////////////////

if(Hz3 == 100000)

begin

Hz3 <= 0;

clock3 <= ~clock3;

end

else

begin

Hz3 <= Hz3 + add1;

end

//////////////////Blinking 5Hz//////////////////

if(Hz4 == 10000000)

begin

Hz4 <= 0;

clock4 <= ~clock4;

end

else

begin

Hz4 <= Hz4 + add1;

end

end

  To make the timer we made a counter that would count at a certain frequency. The timer would count down based on this and essentially the counter became a clock using a demultiplexed clock signal. A sample timer module:

always @(posedge masterclock)

begin

////////////////////1Hz - Seconds////////////////

if(!freeze)

begin

if(Hz1 == 50000000)

begin

Hz1 <= 0;

points <= points + takeAwayOne;

end

else

begin

Hz1 <= Hz1 + add1;

end

end

if(rst)

begin

points <= 200;

end

end

To make the combination we used a random counter with a really fast clock where the user would press the middle button on the FPGA and froze the counter and saved it to be the answer. The answer would then only reset if we pressed the reset button.  The code this was like any other counter:

module RandomNumber(input clock,

output [7:0] randomSol

);

reg [7:0] randomSol = 0;

reg add1 = 1;

always @(posedge clock)

begin

if(randomSol == 8'b11111111)

randomSol <= 0;

else

randomSol <= randomSol + add1;

end

endmodule

To check whether or not the combination was correct, we used a brute force method in checking each switch each time the user pushed the submit button. A sample on how we checked each bit; 1 bit per cycle:

if(sequentialCounter == 0)

begin

if(sw[7] == answer[7])

correctness <= correctness + 1;

sequentialCounter <= sequentialCounter + 1;

end

if(sequentialCounter == 1)

begin

if(sw[6] == answer[6])

correctness <= correctness + 1;

sequentialCounter <= sequentialCounter + 1;

end

if(sequentialCounter == 2)

begin

if(sw[5] == answer[5])

correctness <= correctness + 1;

sequentialCounter <= sequentialCounter + 1;

end

if(sequentialCounter == 3)

begin

if(sw[4] == answer[4])

correctness <= correctness + 1;

sequentialCounter <= sequentialCounter + 1;

end

if(sequentialCounter == 4)

begin

if(sw[3] == answer[3])

correctness <= correctness + 1;

sequentialCounter <= sequentialCounter + 1;

end

if(sequentialCounter == 5)

begin

if(sw[2] == answer[2])

correctness <= correctness + 1;

sequentialCounter <= sequentialCounter + 1;

end

if(sequentialCounter == 6)

begin

if(sw[1] == answer[1])

correctness <= correctness + 1;

sequentialCounter <= sequentialCounter + 1;

end

if(sequentialCounter == 7)

begin

if(sw[0] == answer[0])

correctness <= correctness + 1;

sequentialCounter <= -1;

end

For the display we use the .utf file to create the seven segment display. We parsed the number into a digit between 0 and 9 and to display it on the seven segment display, we used a case statement to enable certain parts of the segments based on what the parsed number is. We then cycled through the 4 seven segment displays using a really fast clock so it looks like the clock is always displaying all 4 lights. A sample can be seen:

case(caseStatements)

0:

begin

sseg\_num = correctness0;

an = 4'b1110;

end

1:

begin

sseg\_num = correctness1;

an = 4'b1101;

end

2:

begin

sseg\_num = numOfTries0;

an = 4'b1011;

end

3:

begin

sseg\_num = numOfTries1;

an = 4'b0111;

end

endcase

always @(\*)

begin

case (sseg\_num)

0: seg = 8'b11000000;

1: seg = 8'b11111001;

2: seg = 8'b10100100;

3: seg = 8'b10110000;

4: seg = 8'b10011001;

5: seg = 8'b10010010;

6: seg = 8'b10000010;

7: seg = 8'b11111000;

8: seg = 8'b10000000;

9: seg = 8'b10010000;

100: seg = 8'b11000111; //L

101: seg = 8'b11000000; //O

102: seg = 8'b10010010; //S

103: seg = 8'b10000110; //E

default: seg = 8'b11111111;

endcase

end

The buttons for submitting the guess and the reset were also debounced correctly. If the user were to hold the button down, it would not see the input twice. To do this we make a large counter along with certain flags that saw that if the button was pushed, the flag would be enabled based off the least significant bit. This flag would then tell when the button was being pushed down, and when the button was released.

counter = counter +1;

if (&counter)

begin

btnM\_buf [24:0] = {btnM, btnM\_buf[24:1]};

end

btnM\_VLD = &btnM\_buf;

.

.

.

if (btnM\_VLD && btnM\_mode == 0)

else if (btnM\_mode == 1 && btnM\_VLD == 0)

begin btnM\_mode = 0;

end

else if (btnM\_mode ==0 && btnM\_VLD == 0)

begin

end

else if (btnM\_mode == 1 && btnM\_VLD == 1)

begin end

**Conclusion:**The design project was very similar to the previous lab with the stopwatch so we were able to implement the game successfully. The game proposal that was given was followed exactly and correctly and works the way we wanted it to. We were able to play our game successfully in which the number of tries and the number of correct switches was displayed as well as the win/lose display. The submit and reset buttons also worked fine for the game.