# Laboratory Exercise #1
# Using the Vivado

ECEN 449-511

Seth Pregler

ELECTRICAL & COMPUTER ENGINEERING
TEXAS A&M UNIVERSITY

**Introduction**

The purpose of this lab was to review synthesizing Verilog HDL using Vivado to program Xilinx FPGAs. The lab introduced three tasks; the first task was to output LEDs given user input, the second task was to count up and down using one-hot-encoding and the final task was to build a jackpot game. These tasks will be explained in greater detail in the subsequent section.

**Procedure**

switch.v:

- Create a program that assigns LEDs as output corresponding to DIP switches as input.
- Create a design constraint file that connects hardware ports to module ports. This is done using the switch.xdc file.
- Generate bitstream and program FPGA, verify results.

four_bit_counter.v:

- Implement a 4-bit counter using the LEDs.
- Create a clock divider to reduce 125MHz clock frequency to 1 Hz.
- Include both up/down counting capabilities using BTN inputs.

jackpot.v:

- Design a jackpot game that works as follows: The LEDs glow in a one-hot fashion, which means that the LEDs are turned on one at a time in a sequential manner. Assign a DIP switch to each of the LEDs. At any point in time, if you turn on the switch corresponding to the glowing LED, you win a Jackpot and all the LEDs start glowing.
- Reuse the clock divider to slow down clock frequency.

**Results**

Both the switch and 4-bit counter implementations were quite straightforward. In order to achieve satisfactory results for the counter, a clock divider needed to be implemented. I implemented this by using a 28-bit count register and incremented the counter every clock cycle. When the counter reached $125,000,00/2$, I would invert the output signal. The result is a module that takes in a 125 MHz clock and outputs a 1 Hz clock cycle.

The final jackpot implementation was less straightforward. After reviewing some concepts on state machine coding, I was able to implement the logic. I used a combinational “next state” block and a sequential “current state” block. The reason for this is that next state logic doesn’t change, and is therefore easily implemented in combinational logic. The current state logic uses registered outputs and this is why I output in my sequential block. The main challenge with this part of the lab was implementing an edge detector to counter cheating the game. Without edge detection, a player can just flip a switch corresponding to an LED that is not flashing and wait for the signals to propagate. I implemented an edge detector using the schematic found in Appendix A, however, I was unable to get the final design working.

Reflecting on this lab, I should have created a testbench file to simulate outputs to see if my edge detector was being used correctly, if at all. This was a major flaw in my design process and I paid the price.

**Conclusion**

As previously stated, the purpose of this lab was to review concepts previously learned including the Vivado development environment and Verilog programming. This lab helped me recall many concepts taught in ECEN 248 such as clock division, Verilog semantics and syntax, and state machine coding. The lab for ECEN 449 will continue to build on these fundamentals and Lab 1 set the foundation for subsequent labs.

**Questions**

**(a) How are the user push-buttons wired on the ZYBO Z7-10 board (i.e. what pins on the FPGA do each of them correspond to and are the signals pulled up or down)? You will have to consult the Master XDC file for this information.**

There are four user push buttons on the ZYBO board: BTN3(Y16), BTN2(K19), BTN1(P16), BTN0(K18). After consulting the Master XDC file and Digilent.com, it appears that the Basic I/O buttons are pull-down.

**(b) What is the purpose of an edge detection circuit and how should it have been used in this lab?**

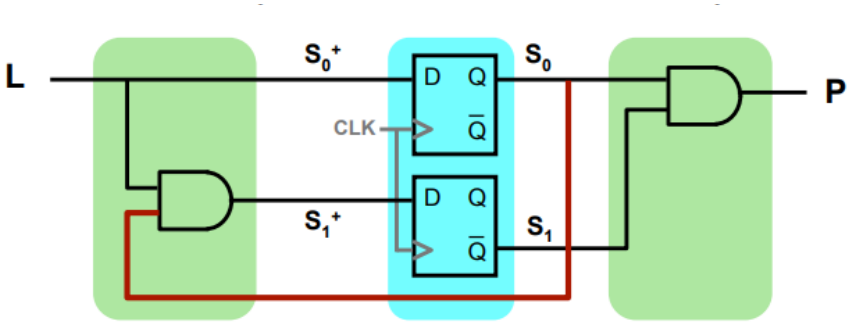
The purpose of the edge detection is to act as a trigger condition for the winning state. My design failed due to the fact that a user could activate an arbitrary switch and wait for the LEDs to propagate. The edge detection circuit used in this lab resolves this by converting a level to pulse. Whenever switch input goes from low to high, the output of the edge detection circuit produces an output pulse, one clock period wide. This is the signal that needs to trigger a winning state.

# Appendix

---

**List of Contents**

# Appendix A
# Edge Detector Schematic




Credit: http://courses.csail.mit.edu/6.111/f2007/handouts/L07.pdf

# Appendix B
# four_bit_counter.v

```
module four_bit_counter(
    input clk,
    input rst,
    input down, up,    // input buttons to control direction of count
    output reg [3:0] out
    );
    wire clk_out;

    // Instantiate clk divider
    clock_divider clk1(clk_out, clk);

    // Always on the posedge of clk
    always@(posedge clk_out) begin
        if (rst) begin
            out <= 4'b0000;
        end

        else
            if (up) begin
                out <= out + 1;
            end

            if (down) begin
                out <= out - 1;
            end
    end
endmodule

module clock_divider(
    output reg clk_out,
    input clk_in
    );
    // Set bit width to 28 bit decimal, 2^28 is just greater than the frequency of the chip (250MHz)
    reg[27:0] counter = 28'd0;
    // Frequency of clk_out is clk_in divided by 125MHz, i.e. 1Hz
    parameter DIVISOR = 28'd125000000;

    always@(posedge clk_in)
    begin
        counter <= counter + 28'd1; // Enumerate counter
        if (counter == (DIVISOR - 1)) begin
            counter <= 28'd0;
        end

        clk_out <= (counter < DIVISOR/2) ? 1'b0 : 1'b1;
    end
endmodule
```

# Appendix C
# Jackpot.v

```
module jackpot(
    output reg [3:0] LEDS,
    input [3:0] SWITCHES,
    input rst,
    input clk
    );

    // Instantiate clock divider
    wire clk_out;
    clock_divider clk1(clk_out, clk);

    reg [4:0] state, next;
    wire flag;
    // Define one-hot encoding states
    parameter
        IDLE = 5'd0,
        S1 = 5'd1,  // 1st LED
        S2 = 5'd2,  // 2nd LED
        S3 = 5'd4,  // 3rd LED
        S4 = 5'd8,  // 4th LED
        YOUWIN = 5'd15;

    // Attempt to implement pulses corresponding to switches
    wire flag1, flag2, flag3, flag4;
    edge_detect edge1(flag1, clk_out, SWITCHES[0]);
    edge_detect edge2(flag2, clk_out, SWITCHES[1]);
    edge_detect edge3(flag3, clk_out, SWITCHES[2]);
    edge_detect edge4(flag4, clk_out, SWITCHES[3]);

    // Define sequential
    always@(posedge clk_out) begin
        //reset condition
        if (rst) begin
            state <= IDLE;
            LEDS <= IDLE;
        end

        else begin
            case(state)
            // If pulse was detected, youwin, else next state
                S1: begin
                    LEDS <= (flag1) ? YOUWIN : next;
                end

                S2: begin
                    LEDS <= (flag2) ? YOUWIN : next;
                end

                S3: begin
                    LEDS <= (flag3) ? YOUWIN : next;
                end

                S4: begin
                    LEDS <= (flag4) ? YOUWIN : next;
                end
                default: state <= IDLE;
            endcase
        end
        state <= next;
     end

     // Define next state logic
     always @(posedge clk_out) begin
        case(state)
            IDLE: next = S1;
            S1: begin
               next = S2;
            end
            S2: begin
               next = S3;
            end
            S3: begin
               next = S4;
            end
            S4: begin
               next = S1;
            end
        endcase
    end
endmodule
```

```
99  module edge_detect(
100     output OUT,
101     input clk,
102     input L     // Switch input
103     );
104
105     reg A, B;
106
107     always@(posedge clk) begin
108         A <= L;
109     end
110
111     always@(posedge clk) begin
112         B <= L && A;
113     end
114
115     assign OUT = A && B;
116 endmodule
117
118 module clock_divider(
119     output reg clk_out,
120     input clk_in
121     );
122     // Set bit width to 28 bit decimal, 2^28 is just greater than the frequency of the chip (250MHz)
123     reg[27:0] counter = 28'd0;
124     // Frequency of clk_out is clk_in divided by 2.083.333 i.e. ~60Hz which should be quite difficult
125     parameter DIVISOR = 28'd200000000;
126
127     always@(posedge clk_in)
128     begin
129         counter <= counter + 28'd1; // Enumerate counter
130         if (counter == (DIVISOR - 1)) begin
131             counter <= 28'd0;
132         end
133
134         clk_out <= (counter < DIVISOR/2) ? 1'b0 : 1'b1;
135     end
136 endmodule
```