Bert Zhao
EE 271

Abstract: The purpose of this lab is to introduce the concept of sequential logic circuits. This involves the use of new logic components; flip flops and latches. These components are capable of storing a previous input, which makes them suitable as a 1-bit memory module. Sequential logic operate on the basis of a clock, which is discrete signal alternating between high and low at a constant rate. The different flip flops and latches update their states corresponding to different points of the clock. Once we have designed our sequential logic systems, we will also be programming them onto the FPGA development board.

Introduction: The main purpose of this lab is to design a sequential logic system. The first part simply requires running some provided test code. This is to help us with understanding the basics of sequential logic and its implementation in verilog. The second part of the lab is a design task which involves creating a system of hazard lights which cycle through a distinct lighting pattern based on the wind conditions. There are three different light patterns that need to be implemented for this part and we must create a system that is capable of displaying all three given the appropriate inputs.

Materials:

- Altera Terasic DE1-SoC Board with power and USB cables
- PC with Altera Quartus Prime Lite installed

Procedure:

Part 1: Mapping sequential logic to the FPGA

The first part of this lab serves primarily as an introduction to sequential logic programming in verilog. We run some provided test code with the following function: The output is true whenever the input has been true for the previous two clock cycles. The testing process is also slightly different in that the iteration through inputs is not done using a loop and wait commands, but rather the natural delay of the clock. In this specific part, we are using a positive edge flip flop in our system. In this lab we are also provided the clock, in which we can change certain properties such as the speed. This design displays the clock on LEDR5, the output on LEDR0, and LEDR3 for reset. KEY1 is the push button used for the input and KEY0 is used for reset.

Part 2: Design Problem – Hazard lights

The second part of this lab is the design problem where we must create a system of light patterns based on the wind conditions. There are three different wind conditions; calm, right to left, and left to right. These can be all accounted for with a 2-bit input. The outputs are three leds controlled by a 3-bit output. The patterns of lights can be created using four static sets of 3-bit outputs, giving us four states. The logic behind our system will determine how the light patterns transition from one to the next, therefor giving us our three patterns.

State Diagram:

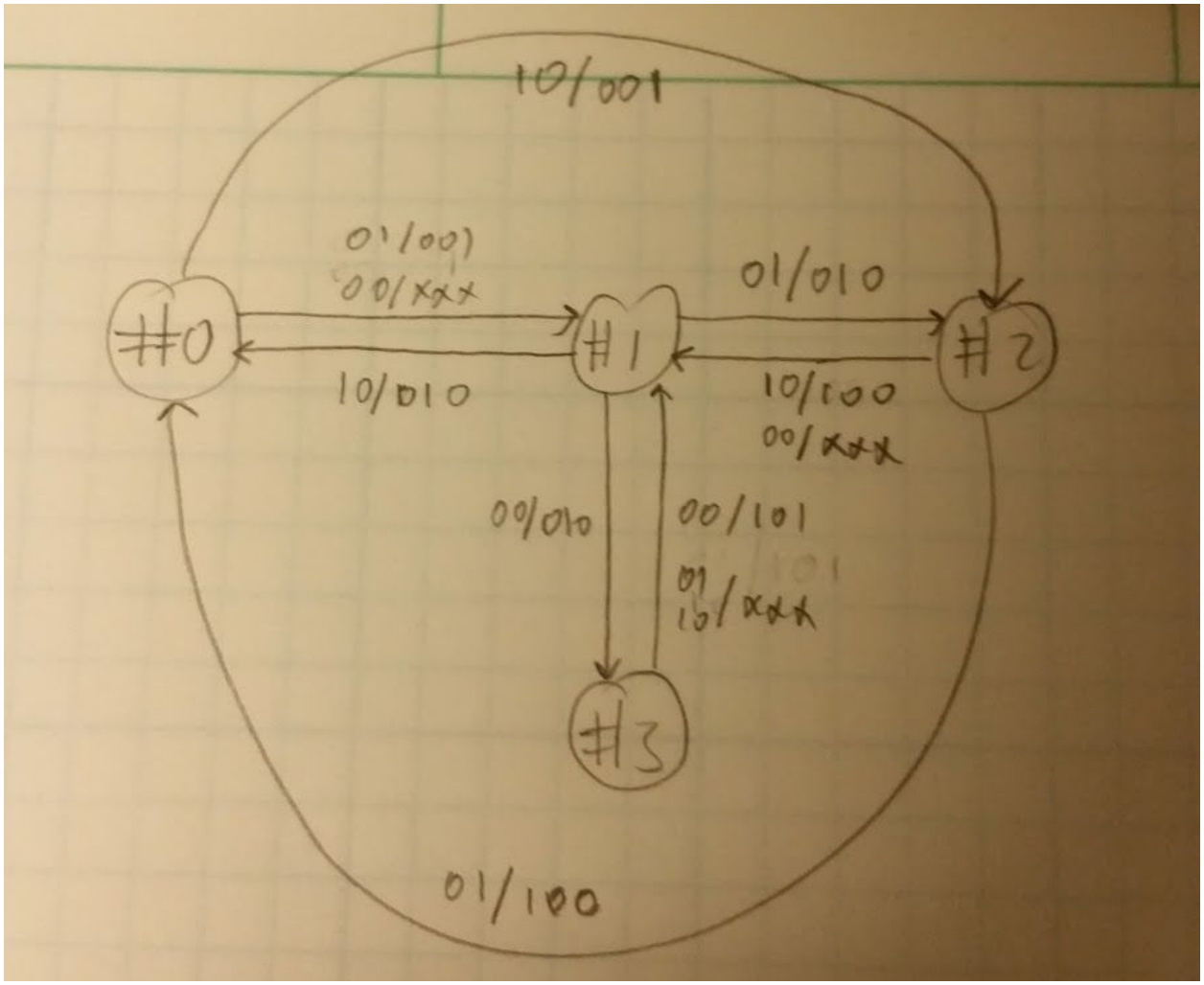

Figure 1: Lighting state diagram

Truth Table:

SW1	SW0	PS1	PS0	LEDR2	LEDR1	LEDR0	NS1	NS0
0	0	0	0	x	x	x	0	1
0	0	0	1	0	1	0	1	1
0	0	1	0	x	x	x	0	1
0	0	1	1	1	0	1	0	1
0	1	0	0	0	0	1	0	1
0	1	0	1	0	1	0	1	0
0	1	1	0	1	0	0	0	0

0	1	1	1	x	x	x	0	1
1	0	0	0	0	0	1	1	0
1	0	0	1	0	1	0	0	0
1	0	1	0	1	0	0	0	1
1	0	1	1	x	x	x	0	1
1	1	0	0	x	x	x	x	x
1	1	0	1	x	x	x	x	x
1	1	1	0	x	x	x	x	x
1	1	1	1	x	x	x	x	x

Results:

All parts of the lab were first tested using ModelSim. The waveform plots could be adjusted so that the grid would line up with the clock periods for easier analysis. The design problem could be implemented with a minimum of four states, one for each of the static light patterns. The lights were displayed in LEDR 2-0. The programming could be further optimised by setting all the don't cares to the output 010, since all three patterns of light cycled through that lighting configuration. This also made transitions between each pattern more efficient. The reset was again represented by KEY0 and holding that down caused a static 010 output to be displayed.

ModelSim Screenshots:

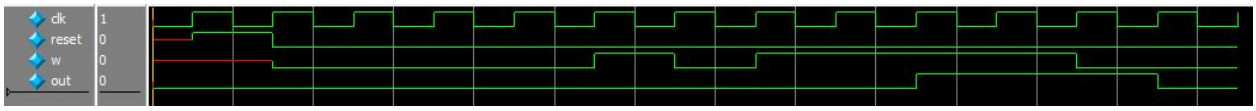

Figure 2: simple ModelSim screenshot (Task 1)

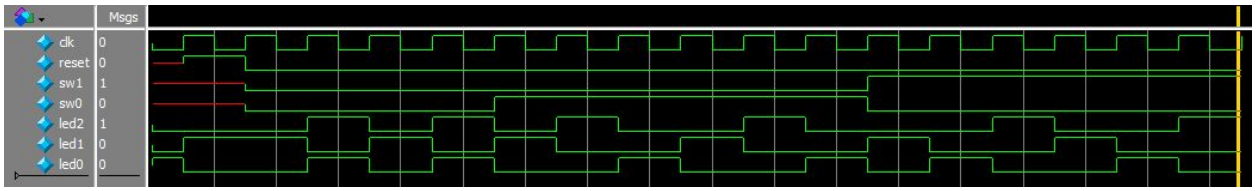

Figure 3: runway ModelSim screenshot (Task 2)

Analysis and Conclusion:

Overall, this lab provided an introduction to sequential logic design. The lab introduced the style of programming associated with sequential logic, the fundamentals of which will be important continuing into the remaining labs as we focus more heavily on the details of sequential logic.

Appendix:

```
module simple (clk, reset, w, out);
   input logic clk, reset, w;
   output logic out;

   // State variables.
   enum {A, B, C} ps, ns;

   // Next State logic
   always_comb begin
      case (ps)
         A: if (w) ns = B;
            else ns = A;
         B: if (w) ns = C;
            else ns = A;
         C: if (w) ns = C;
            else ns = A;
      endcase
   end

   // Output logic - could also be another always, or part of above block.
   assign out = (ps == C);

   // DFFs
   always_ff @(posedge clk) begin
      if (reset)
         ps <= A;
      else
         ps <= ns;
   end
endmodule

module simple_testbench();
   logic clk, reset, w;
   logic out;

   simple dut (clk, reset, w, out);

   // Set up the clock.
   parameter CLOCK_PERIOD = 100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Set up the inputs to the design. Each line is a clock cycle.
   initial begin
                              @(posedge clk);
      reset <= 1;             @(posedge clk);
      reset <= 0; w <= 0;     @(posedge clk);
                              @(posedge clk);
                              @(posedge clk);
                              @(posedge clk);
                  w <= 1;     @(posedge clk);
                  w <= 0;     @(posedge clk);
                  w <= 1;     @(posedge clk);
                              @(posedge clk);
                              @(posedge clk);
                              @(posedge clk);
                  w <= 0;     @(posedge clk);
                              @(posedge clk);
      $stop; // End the simulation.
   end
endmodule
```

Code 1: simple (Task 1)

```
module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
   input logic CLOCK_50; // 50MHz clock.
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0] LEDR;
   input logic [3:0] KEY; // True when not pressed, False when pressed
   input logic [9:0] SW;

   // Generate clk off of CLOCK_50, whichClock picks rate.
   logic [31:0] clk;
   parameter whichClock = 25;
   clock_divider cdiv (CLOCK_50, clk);

   // Hook up FSM inputs and outputs.
   logic reset, w, out;
   assign reset = ~KEY[0]; // Reset when KEY[0] is pressed.
   assign w = ~KEY[1];
   simple s (.clk(clk[whichClock]), .reset, .w, .out);

   // Show signals on LEDRs so we can see what is happening.
   assign LEDR = { clk[whichClock], 1'b0, reset, 2'b0, out};
endmodule

   // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
module clock_divider (clock, divided_clocks);
   input logic clock;
   output logic [31:0] divided_clocks;

   initial begin
      divided_clocks <= 0;
   end

   always_ff @(posedge clock) begin
      divided_clocks <= divided_clocks + 1;
   end
endmodule
```

Code 2: DE1_SoC (Task 1)

```
module runway (clk, reset, sw1, sw0, led2, led1, led0);
   input logic clk, reset, sw1, sw0;
   output logic led2, led1, led0;

   // State variables
   enum {A, B, C, D} ps, ns;

   // Next state logic
   always_comb begin
      case (ps)
         A: if ({sw1, sw0} == 2'b10)
               ns = C;
            else
               ns = B;

         B: if ({sw1, sw0} == 2'b00)
               ns = D;
            else if ({sw1, sw0} == 2'b01)
               ns = C;
            else if ({sw1, sw0} == 2'b10)
               ns = A;
            else
               ns = B;

         C: if ({sw1, sw0} == 2'b01)
               ns = A;
            else
               ns = B;
         D: ns = B;
      endcase
   end
```

```
    // Output logic
    always_comb begin
       case (ps)
          A: begin
             led2 = 0; led1 = 0; led0 = 1;
             end

          B: begin
             led2 = 0; led1 = 1; led0 = 0;
             end

          C: begin
             led2 = 1; led1 = 0; led0 = 0;
             end

          D: begin
             led2 = 1; led1 = 0; led0 = 1;
             end
       endcase
    end

    // DFFs
    always_ff @(posedge clk) begin
       if (reset)
          ps <= B;
       else
          ps <= ns;
    end
 endmodule

 module runway_testbench();
    logic clk, reset, sw1, sw0;
    logic led2, led1, led0;

    runway dut(clk, reset, sw1, sw0, led2, led1, led0);

 // Set up clock
 parameter CLOCK_PERIOD = 100;
 initial begin
    clk <= 0;
    forever #(CLOCK_PERIOD/2) clk <= ~clk;
 end

 // Set up the inputs to the design. Each line is a clock cycle.
 initial begin
                                        @(posedge clk);
    reset <= 1;                         @(posedge clk);
    reset <= 0; sw1 <= 0;   sw0 <= 0;   @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
                sw0 <= 1;               @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
                sw1 <= 1;   sw0 <= 0;   @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
                                        @(posedge clk);
    $stop; // End the simulation.
 end
endmodule
```

Code 3: runway (Task 2)

```
module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
   input logic CLOCK_50; // 50MHz clock.
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0] LEDR;
   input logic [3:0] KEY; // True when not pressed, False when pressed
   input logic [9:0] SW;

   // Generate clk off of CLOCK_50, whichClock picks rate.
   logic [31:0] clk;
   parameter whichClock = 25;
   clock_divider cdiv (CLOCK_50, clk);

   // Hook up FSM inputs and outputs.
   logic reset, sw1, sw0, led2, led1, led0;
   assign reset = ~KEY[0]; // Reset when KEY[0] is pressed.
   assign sw1 = SW[1];
   assign sw0 = SW[0];
   runway s (.clk(clk[whichClock]), .reset, .sw1, .sw0, .led2, .led1, .led0);

   // Show signals on LEDRs so we can see what is happening.
   assign LEDR = {led2, led1, led0};
endmodule

   // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
module clock_divider (clock, divided_clocks);
   input logic clock;
   output logic [31:0] divided_clocks;

   initial begin
      divided_clocks <= 0;
   end

   always_ff @(posedge clock) begin
      divided_clocks <= divided_clocks + 1;
   end
endmodule
```

Code 4: DE1_SoC (Task 2)