Bert Zhao
EE 271

Abstract: The purpose of the lab is to expand on the concept of sequential logic as well continuing to develop larger multi-module logic circuits in Verilog. The fundamental elements of this lab require detailed planning and strategic decomposition of modules so that we can easily manage multiple modules with very few states rather than just one module with many states.

Introduction: The objective of this lab is to build a game of tug-of-war. The row of leds will be used to identify the location of the center of the rope and two of the buttons will be used as the inputs for each player to tug on the rope. Once one of the players win the game, the hex display will indicate the winner of the game. After a round has finished, a switch can be used to start a new game.

Materials:

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

Procedure: Tug of War

As one of the main goals of this lab break down the overall logic circuit to smaller and more easily manageable modules, the core functionality of this lab can be programmed in three key modules.

The first major component of the circuit is constructing a system that appropriately accepts the player inputs. In this implementation of tug of war, each button press must only prompt one “tug” of the rope. This means that a single key press must only provide an input for one clock period, regardless of the duration the player holds down the button. In other words, the user input module must detect only the instance when the player clicks their button.

The second major component of the game involves controlling the lights on the playfield. Each new game must start with the center light (LED5) on and all other lights off. Each time a player clicks their button, the light will move over one led towards that player. The function for each light is the same during gameplay, with the only exception being the reset condition. This means that the entire playfield can be implemented with a single central light and eight normal light modules.

The final component of the game is victory check condition. A game of tug-of-war is won when a player pulls the center of the rope to their side, and in our logic implementation, a player wins when they move the LED indicator over the LED closest to them (LED9 for left player and LED1 for right player). The victory check module must analyze the states of LED9 and LED1 and the user inputs to determine if a player has won the game. Once a winner is decided, the winner player will be declared through the HEX0 display.

State Diagrams:

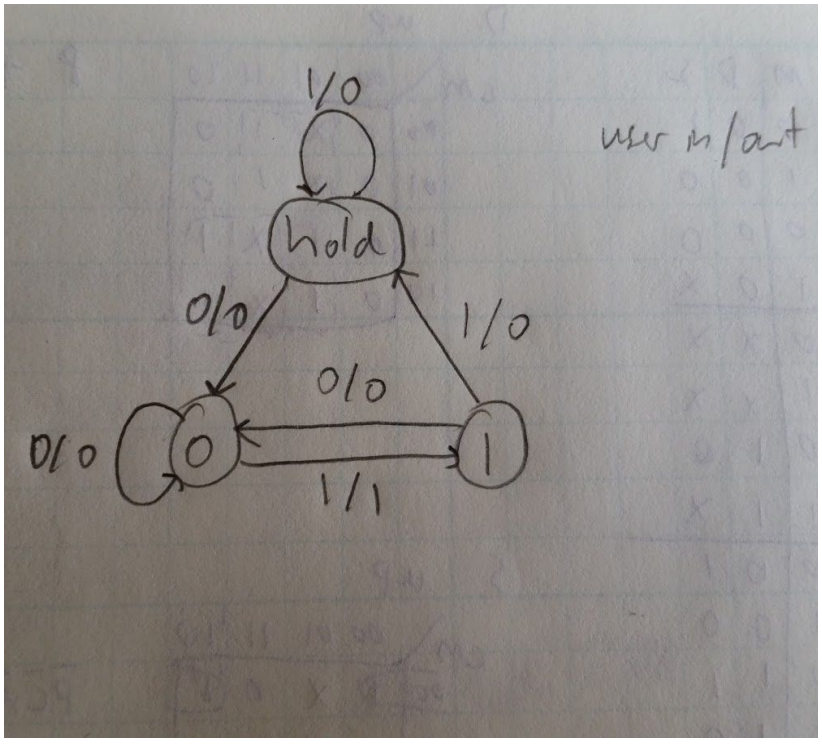

Figure 1: User input state diagram

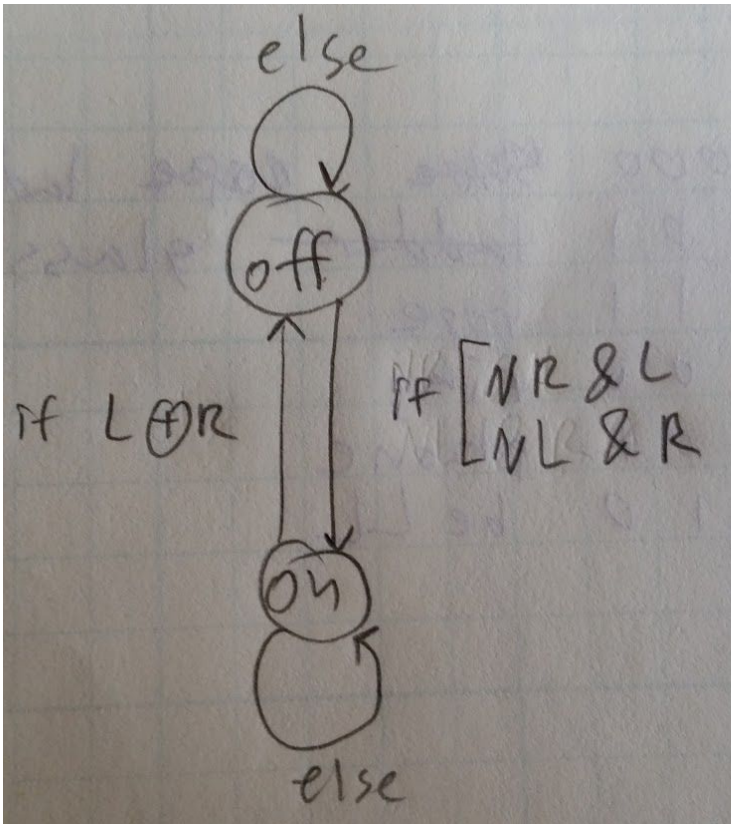

Figure 2: LED state diagram

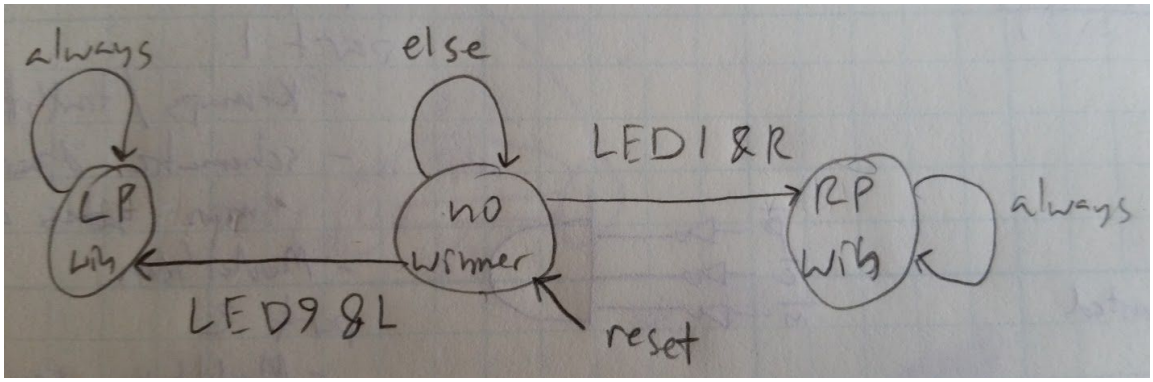

Figure 3: Victory state diagram

The final detail of the lab involved resolving the issue with metastability. A metastability occurs when an input to a DFF occurs at a clock edge. In this case, the output becomes metastable and therefore is unreliable. To solve this issue, inputs should pass through two DFFs in series. In this lab, we must design a simple module of two DFFs to eliminate the potential of a metastable output before using the user input for the rest of the circuit.

Results:

All modules of this lab were tested separately using ModelSim. The largest advantage of splitting the system into smaller modules is that so any given module could be managed in under four states. This also makes the system easier to debug for errors. The overall circuit was then analyzed using both ModelSim and a block diagram. There are many different inputs and outputs for each module, but the block diagram provides better visualization of the connections between each module. In the end, the block diagram lets us clearly map how the overall inputs; KEY3 (left player), KEY0 (right player), and SW9 (reset), controls LED1-9 (playfield), and HEX0 (winner indicator). Quartus can also calculate a total cost for our full circuit shown in the analysis and synthesis summary.

ModelSim Screenshots:

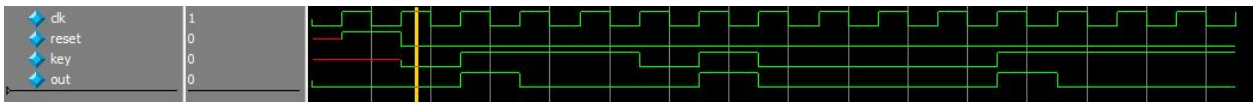

Figure 4: playerIn ModelSim screenshot

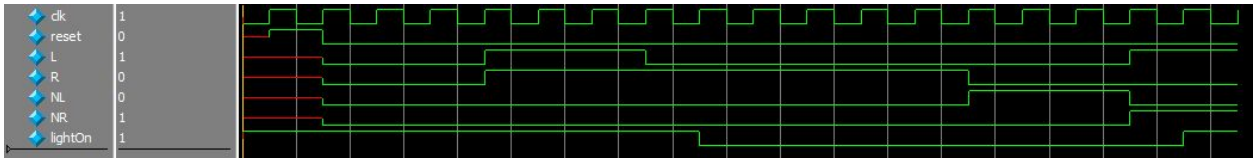

Figure 5: centerLight ModelSim screenshot

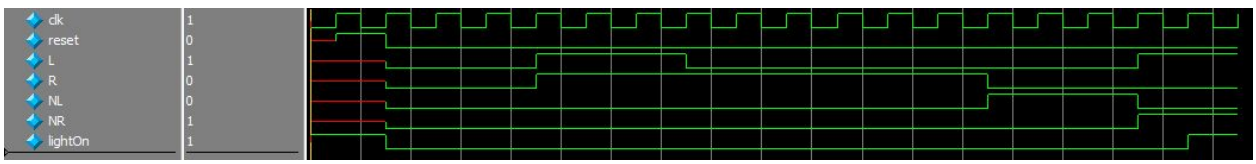

Figure 6: normalLight ModelSim screenshot

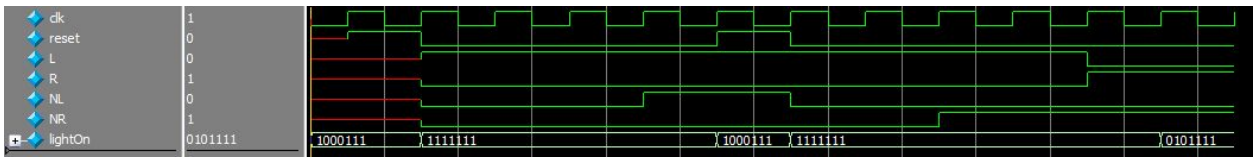

Figure 7: victory ModelSim screenshot

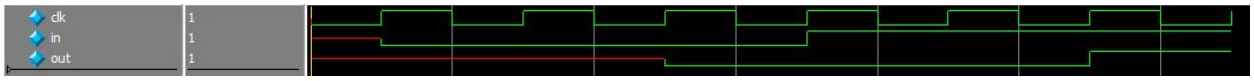

Figure 8: meta ModelSim screenshot

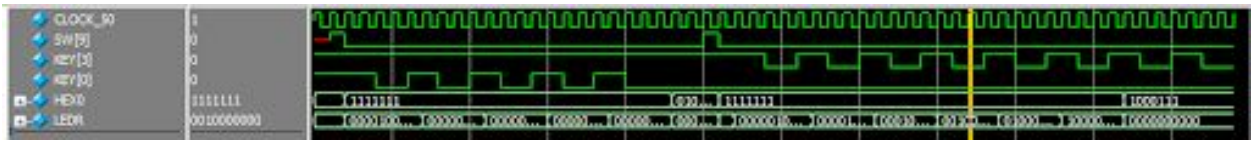

Figure 9: DE1_SoC ModelSim screenshot

System Cost:

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins	Full Hierarchy Name
1	DE1_SoC	23 (0)	19 (0)	0	0	67	0	|DE1_SoC
1	|centerLight:L5|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|centerLight:L5
2	|meta:leftPlayer|	1 (1)	2 (2)	0	0	0	0	|DE1_SoC|meta:leftPlayer
3	|meta:rightPlayer|	1 (1)	2 (2)	0	0	0	0	|DE1_SoC|meta:rightPlayer
4	|normalLight:L1|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L1
5	|normalLight:L2|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L2
6	|normalLight:L3|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L3
7	|normalLight:L4|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L4
8	|normalLight:L6|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L6
9	|normalLight:L7|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L7
10	|normalLight:L8|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L8
11	|normalLight:L9|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L9
12	|playerIn:Lin|	3 (3)	2 (2)	0	0	0	0	|DE1_SoC|playerIn:Lin
13	|playerIn:Rin|	3 (3)	2 (2)	0	0	0	0	|DE1_SoC|playerIn:Rin
14	|victory:V|	6 (6)	2 (2)	0	0	0	0	|DE1_SoC|victory:V

Figure 10: Resource utilization by entity screenshot

Block Diagram:

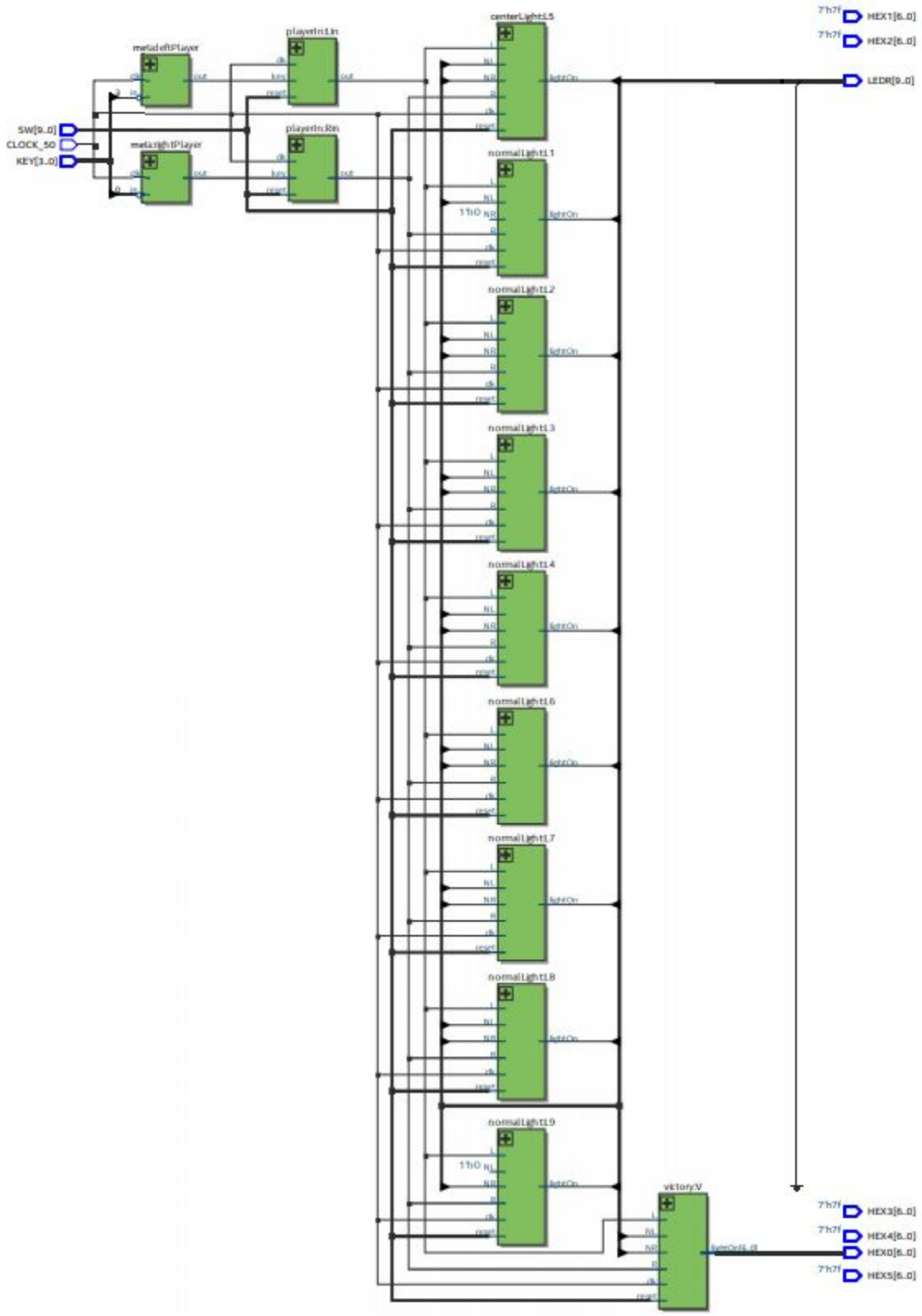

Figure 11: DE1_SoC block diagram

Analysis and Conclusions:
Overall, this lab continued the application of sequential logic components, mainly the DFF. It is also important to note that despite the size of the design task, careful organization of multiple modules can help streamline the process and reduce points of error along the way.

Appendix:

```
module playerIn(clk, reset, key, out);
    input logic clk, reset, key;
    output logic out;

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

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

    // Output logic
    assign out = (ns == B);

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

endmodule

module playerIn_testbench();
    logic clk, reset, key;
    logic out;

    playerIn dut(clk, reset, key, 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; key <= 0;    @(posedge clk);
                   key <= 1;    @(posedge clk);
                                @(posedge clk);
                                @(posedge clk);
                   key <= 0;    @(posedge clk);
                   key <= 1;    @(posedge clk);
                   key <= 0;    @(posedge clk);
                                @(posedge clk);
                                @(posedge clk);
                                @(posedge clk);
                   key <= 1;    @(posedge clk);
                                @(posedge clk);
                                @(posedge clk);
                                @(posedge clk);
       $stop; // End the simulation.
    end
endmodule
```

Code 1: playerIn.sv

```
module centerLight(clk, reset, L, R, NL, NR, lightOn);
    input logic clk, reset, L, R, NL, NR;
    output logic lightOn;

    // State variables
    enum {on, off} ps, ns;

    // Next state logic
    always_comb begin
        case (ps)
            off: if (NR & L)
                    ns = on;
                 else if (NL & R)
                    ns = on;
                 else
                    ns = off;
            on: if (L ^ R)
                    ns = off;
                 else
                    ns = on;
        endcase
    end

    // Output logic
    always_comb begin
        case (ps)
            off: lightOn = 1'b0;
            on: lightOn = 1'b1;
        endcase
    end

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

endmodule
```

```
module centerLight_testbench();
   logic clk, reset, L, R, NL, NR;
   logic lightOn;

   centerLight dut(clk, reset, L, R, NL, NR, lightOn);

   // 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; NL <= 0; L <= 0;  NR <= 0; R <= 0;      @(posedge clk);
                                                          @(posedge clk);
                                                          @(posedge clk);
                  NL <= 0; L <= 1;  NR <= 0; R <= 1;      @(posedge clk);
                                                          @(posedge clk);
                                                          @(posedge clk);
                  NL <= 0; L <= 0;  NR <= 0; R <= 1;      @(posedge clk);
                                                          @(posedge clk);
                                                          @(posedge clk);
                  NL <= 0; L <= 0;  NR <= 0; R <= 1;      @(posedge clk);
                                                          @(posedge clk);
                                                          @(posedge clk);
                  NL <= 1; L <= 0;  NR <= 0; R <= 0;      @(posedge clk);
                                                          @(posedge clk);
                                                          @(posedge clk);
                  NL <= 0; L <= 1;  NR <= 1; R <= 0;      @(posedge clk);
                                                          @(posedge clk);
      $stop; // End the simulation.
   end
endmodule
```

Code 2: centerLight.sv

```
module normalLight(clk, reset, L, R, NL, NR, lightOn);
   input logic clk, reset, L, R, NL, NR;
   output logic lightOn;

   // State variables
   enum {on, off} ps, ns;

   // Next state logic
   always_comb begin
      case (ps)
         off: if (NR & L)
                ns = on;
              else if (NL & R)
                ns = on;
              else
                ns = off;
         on: if (L ^ R)
                ns = off;
              else
                ns = on;
      endcase
   end
```

```
    // Output logic
    always_comb begin
        case (ps)
            off: lightOn = 1'b0;
            on: lightOn = 1'b1;
        endcase
    end

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

endmodule
module normalLight_testbench();
    logic clk, reset, L, R, NL, NR;
    logic lightOn;

    normalLight dut(clk, reset, L, R, NL, NR, lightOn);

    // 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; NL <= 0; L <= 0;  NR <= 0; R <= 0;  @(posedge clk);
                                                        @(posedge clk);
                                                        @(posedge clk);
                    NL <= 0; L <= 1;  NR <= 0; R <= 1;  @(posedge clk);
                                                        @(posedge clk);
                                                        @(posedge clk);
                    NL <= 0; L <= 0;  NR <= 0; R <= 1;  @(posedge clk);
                                                        @(posedge clk);
                                                        @(posedge clk);
                    NL <= 0; L <= 0;  NR <= 0; R <= 1;  @(posedge clk);
                                                        @(posedge clk);
                                                        @(posedge clk);
                    NL <= 1; L <= 0;  NR <= 0; R <= 0;  @(posedge clk);
                                                        @(posedge clk);
                                                        @(posedge clk);
                    NL <= 0; L <= 1;  NR <= 1; R <= 0;  @(posedge clk);
                                                        @(posedge clk);
        $stop; // End the simulation.
    end
endmodule
```

Code 3: normalLight.sv

```
module victory(clk, reset, L, R, NL, NR, lightOn);
    input logic clk, reset, L, R, NL, NR;
    output logic [6:0] lightOn;

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

```
    // Next state logic
    always_comb begin
       case (ps)
          A: ns = A; // left player wins
          B: if (L & NL)
                ns = A;
             else if (R & NR)
                ns = C;
             else
                ns = B;
          C: ns = C; // right player wins
       endcase
    end

    // Output logic
    always_comb begin
       case (ps)
          A: lighton = 7'b1000111;
          B: lighton = 7'b1111111;
          C: lighton = 7'b0101111;
       endcase
    end

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

module victory_tesbench();
    logic clk, reset, L, R, NL, NR;
    logic [6:0] lighton;

    victory dut(clk, reset, L, R, NL, NR, lighton);

    // 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; NL <= 0; NR <= 0; L <= 1; R <= 0;@(posedge clk);
                                                          @(posedge clk);
                                                          @(posedge clk);
                   NL <= 1; NR <= 0;                      @(posedge clk);
       reset <= 1;                                        @(posedge clk);
       reset <= 0; NL <= 0; NR <= 0;                      @(posedge clk);
                                                          @(posedge clk);
                   NL <= 0; NR <= 1; L <= 1; R <= 0;@(posedge clk);
                                                          @(posedge clk);
                                     L <= 0; R <= 1;@(posedge clk);
                                                          @(posedge clk);
       $stop; // End the simulation.
    end
endmodule
```

Code 4: victory.sv

```
module meta(clk, in, out);
    input logic clk, in;
    output logic out;
    logic q;

    always_ff @ (posedge clk) begin
        q <= in;
        out <= q;
    end
endmodule

module meta_testbench();
    logic clk, in, out;

    meta dut(clk, in, 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);
        in <= 0;    @(posedge clk);
                    @(posedge clk);
                    @(posedge clk);
        in <= 1;    @(posedge clk);
                    @(posedge clk);
                    @(posedge clk);
        $stop; // End the simulation.
    end
endmodule
```

Code 5: meta.sv

```
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;

    // Hook up FSM inputs and outputs.
    assign reset = SW[9];
    logic Lmeta, Rmeta;
    logic Lplayer, Rplayer;
    logic led9, led8, led7, led6, led5, led4, led3, led2, led1;
    assign key3 = ~KEY[3];
    assign key0 = ~KEY[0];

    // Metastability tolerance
    meta leftPlayer(.clk(CLOCK_50), .in(key3), .out(Lmeta));
    meta rightPlayer(.clk(CLOCK_50), .in(key0), .out(Rmeta));

    // player input single pulse
    playerIn Lin(.clk(CLOCK_50), .reset, .key(Lmeta), .out(Lplayer));
    playerIn Rin(.clk(CLOCK_50), .reset, .key(Rmeta), .out(Rplayer));

    // lights of playfield
    normalLight L9(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(1'b0), .NR(led8), .lightOn(led9));
    normalLight L8(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(led9), .NR(led7), .lightOn(led8));
    normalLight L7(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(led8), .NR(led6), .lightOn(led7));
    normalLight L6(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(led7), .NR(led5), .lightOn(led6));
    centerLight L5(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(led6), .NR(led4), .lightOn(led5));
    normalLight L4(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(led5), .NR(led3), .lightOn(led4));
    normalLight L3(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(led4), .NR(led2), .lightOn(led3));
    normalLight L2(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(led3), .NR(led1), .lightOn(led2));
    normalLight L1(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(led2), .NR(1'b0), .lightOn(led1));

    // check victory condition
    victory V(.clk(CLOCK_50), .reset, .L(Lplayer), .R(Rplayer), .NL(led9), .NR(led1), .lightOn(HEX0));
```

```
	// Show signals on LEDRs so we can see what is happening.
	assign LEDR = {led9, led8, led7, led6, led5, led4, led3, led2, led1, 1'b0};
	
	// Sets all unused hex displays
	assign HEX1 = 7'b1111111;
	assign HEX2 = 7'b1111111;
	assign HEX3 = 7'b1111111;
	assign HEX4 = 7'b1111111;
	assign HEX5 = 7'b1111111;
endmodule
```

```
module DE1_SoC_testbench();
	logic CLOCK_50; // 50MHz clock.
	logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
	logic [9:0] LEDR;
	logic [3:0] KEY; // True when not pressed, False when pressed
	logic [9:0] SW;
	
	DE1_SoC dut(CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
	
	initial CLOCK_50 = 1;
	initial KEY[3:0] = 4'b1111;
	
	// Set up clock
	parameter CLOCK_PERIOD = 100;
	always begin
		forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;
	end
	
	// Set up the inputs to the design. Each line is a clock cycle.
	initial begin
		                                 @(posedge CLOCK_50);
		SW[9] <= 1;                      @(posedge CLOCK_50);
		SW[9] <= 0; KEY[0] <= 1;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[0] <= 0;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[0] <= 1;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[0] <= 0;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[0] <= 1;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[0] <= 0;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[0] <= 1;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[0] <= 0;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[0] <= 1;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[0] <= 0;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		SW[9] <= 1;                      @(posedge CLOCK_50);
		SW[9] <= 0;                      @(posedge CLOCK_50);
		            KEY[3] <= 1;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[3] <= 0;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[3] <= 1;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
		            KEY[3] <= 0;         @(posedge CLOCK_50);
		                                 @(posedge CLOCK_50);
```

```
                      @(posedge CLOCK_50);
        KEY[3] <= 1;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 0;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 1;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 0;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 1;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 0;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 1;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 0;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 1;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 0;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 1;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);
        KEY[3] <= 0;  @(posedge CLOCK_50);
                      @(posedge CLOCK_50);

    $stop; //End the simulation
  end
endmodule
```

Code 6: DE1_SoC.sv