Bert Zhao
EE 271

Abstract: The purpose of this lab is to continue developing our sequential logic skills. The new concept we learn is this lab is a linear feedback shift register (LFSR). This can be used to create a random pattern of bits that we will use to complete the primary design problem of this lab.

Introduction: The objective of this lab is to add additional features to the tug-of-war game from the previous lab. We will be adding a round system, in which players continue playing until a total of seven round wins are achieved. Then we also design a computer player, where we use a 10-bit LFSR to create a series of random inputs that will simulate another player's inputs.

Materials:

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

Procedure: CyberWar

There are three major modules required for the implementation of this design. The first is a round counter system that keeps track of the number of rounds won by each player. It displays these numbers in the HEX5 and HEX0 for the left and right player (or computer player) respectively. The counter also resets the playfield at the end of each round. Once a player wins the game by winning seven rounds, the game is over and the playfield is frozen. A new game can then be started with a full reset.

The second part of this lab is the random input generator for the computer opponent. This is done through the use of a LFSR. We start analysing smaller three and four bit LFSRs to observe the randomness of the pattern generated by the LFSR. We then design a 10-bit LFSR for this lab as the foundation for our computer player's inputs.

State Diagrams:

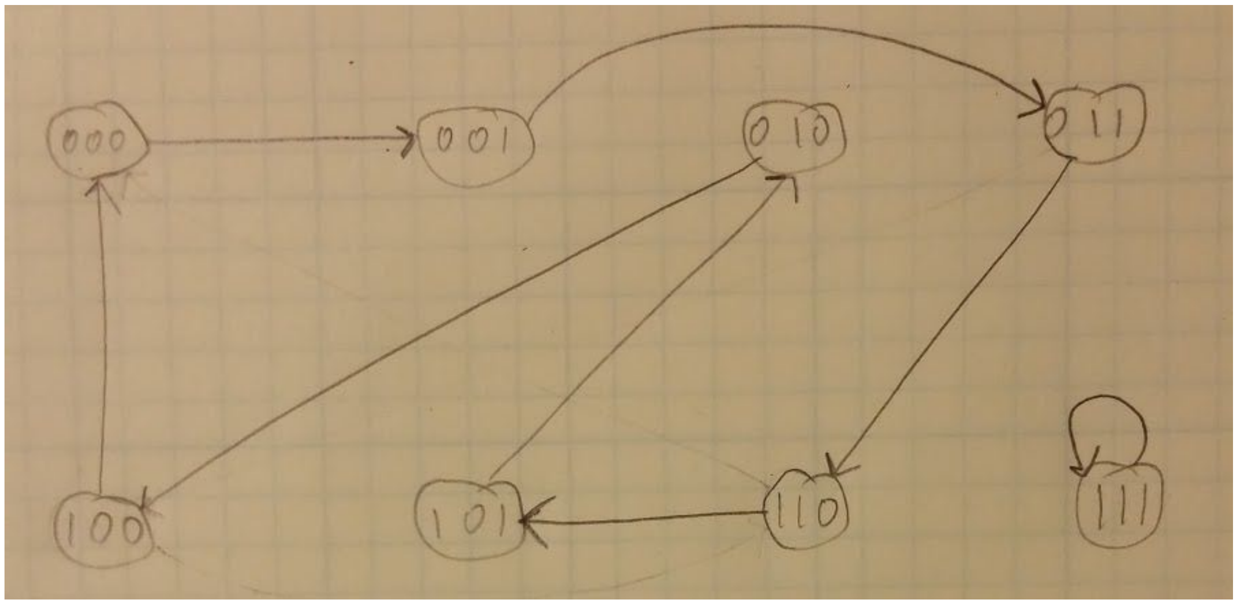


Figure 1: 3-bit LFSR

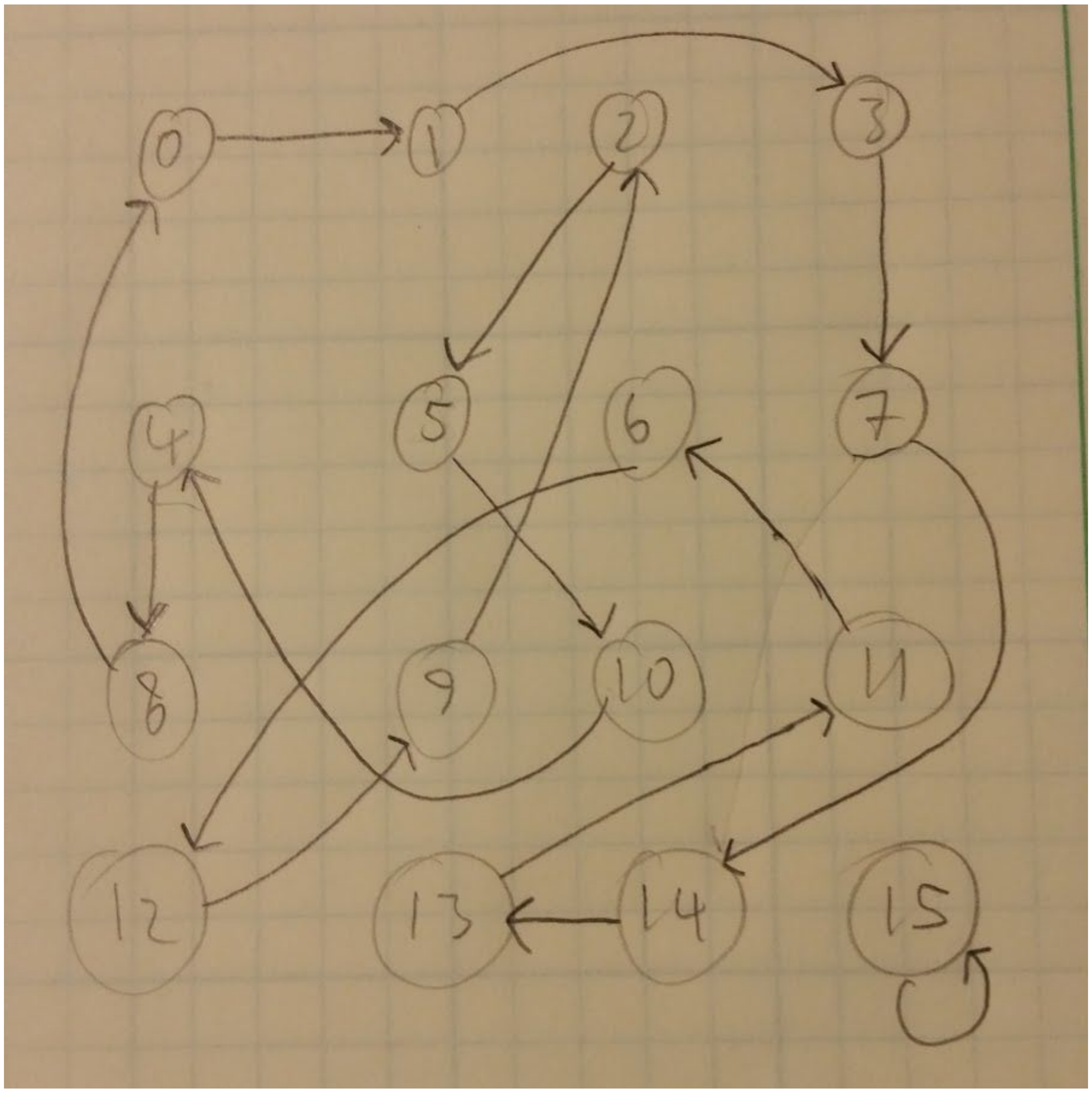


Figure 2: 4-bit LFSR

The final part of the lab is a comparator that takes the random pattern from the LFSR and an input of our choosing to determine the highs and lows of the computer button presses. We will enter a nine-bit number on switches 8-0 to be passed to the comparator. If the LFSR generated input is less than our input on SW[8:0], then the computer is granted a button press. This will allow us to adjust the difficulty of the game through the number we set on the switches.

Results:
The updated modules of this lab were each tested using ModelSim. This includes the counter, LFSR, and comparator, but also an updated victory module. The other modules remained the same from the previous lab, with the only change to the overall circuit being HEX5 and HEX0 are now used to display a rounds won counter for each player. The playfield is frozen at the end of a complete game. The final light pattern of the playfield can be unpredictable when the computer wins due to the speed of the computer inputs. A player win will not cause a frozen playfield of three lights on. A new total cost was calculated in the analysis and summary. The total resource utilization is (66 - 16) + (49 - 16) = 83.

ModelSim Screenshots:

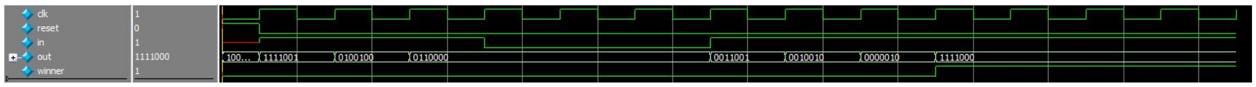


Figure 3: counter ModelSim screenshot

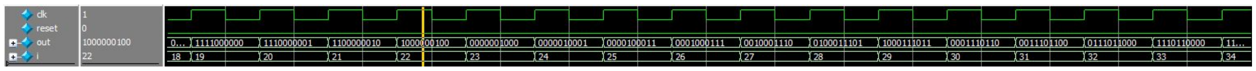


Figure 4: LFSR ModelSim screenshot

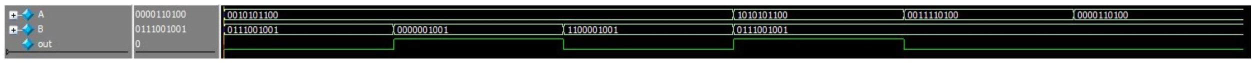
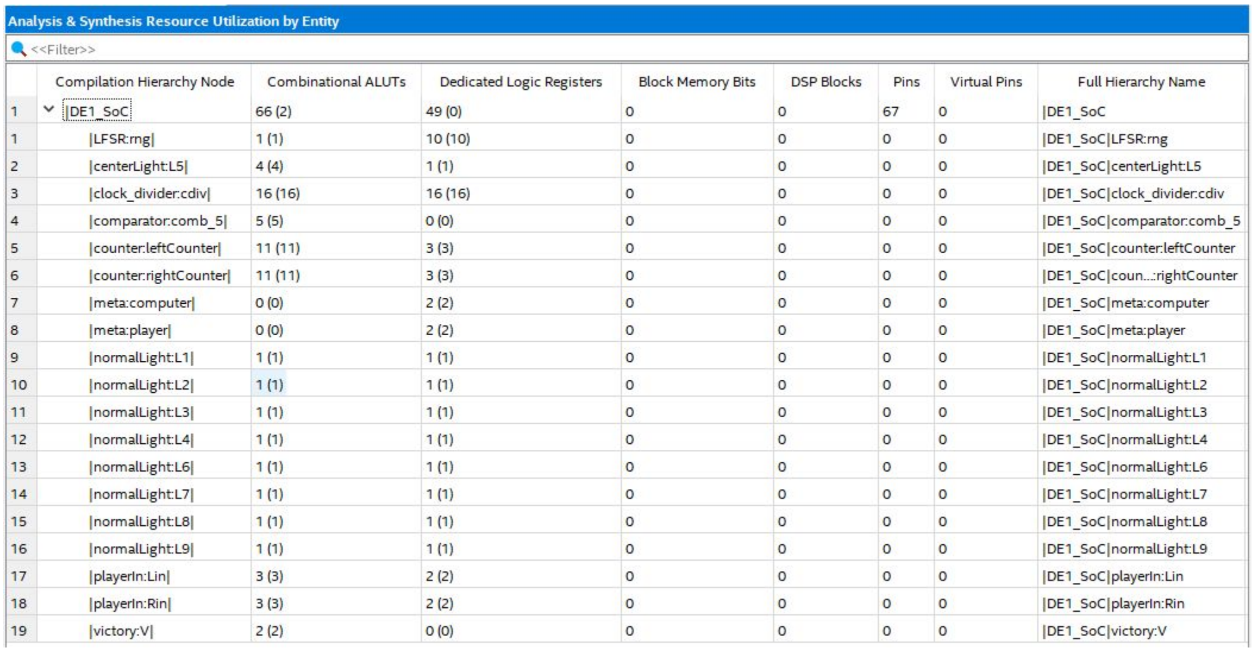


Figure 5: comparator ModelSim screenshot

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	66 (2)	49 (0)	0	0	67	0	|DE1_SoC
1	|LFSR:rng|	1 (1)	10 (10)	0	0	0	0	|DE1_SoC|LFSR:rng
2	|centerLight:L5|	4 (4)	1 (1)	0	0	0	0	|DE1_SoC|centerLight:L5
3	|clock_divider:cdiv|	16 (16)	16 (16)	0	0	0	0	|DE1_SoC|clock_divider:cdiv
4	|comparator:comb_5|	5 (5)	0 (0)	0	0	0	0	|DE1_SoC|comparator:comb_5
5	|counter:leftCounter|	11 (11)	3 (3)	0	0	0	0	|DE1_SoC|counter:leftCounter
6	|counter:rightCounter|	11 (11)	3 (3)	0	0	0	0	|DE1_SoC|coun...:rightCounter
7	|meta:computer|	0 (0)	2 (2)	0	0	0	0	|DE1_SoC|meta:computer
8	|meta:player|	0 (0)	2 (2)	0	0	0	0	|DE1_SoC|meta:player
9	|normalLight:L1|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L1
10	|normalLight:L2|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L2
11	|normalLight:L3|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L3
12	|normalLight:L4|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L4
13	|normalLight:L6|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L6
14	|normalLight:L7|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L7
15	|normalLight:L8|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L8
16	|normalLight:L9|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalLight:L9
17	|playerIn:Lin|	3 (3)	2 (2)	0	0	0	0	|DE1_SoC|playerIn:Lin
18	|playerIn:Rin|	3 (3)	2 (2)	0	0	0	0	|DE1_SoC|playerIn:Rin
19	|victory:V|	2 (2)	0 (0)	0	0	0	0	|DE1_SoC|victory:V

Figure 6: Resource utilization by entity screenshot

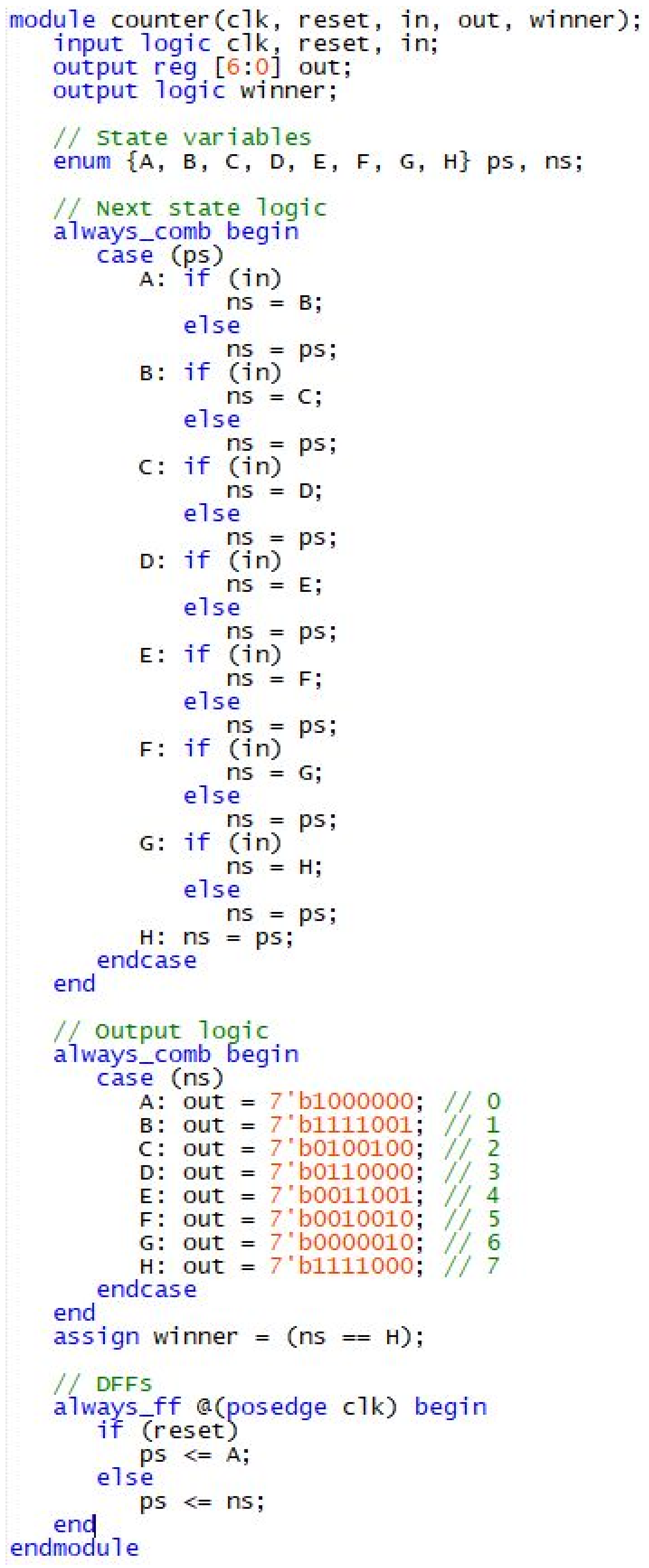
Analysis and Conclusion:

Overall, this lab continued the application of sequential logic components. We learned how DFFs could be used to create a LFSR, which is useful as a pseudo random sequence generator. These addition of these new features to the previous lab help us better solidify our understanding of sequential logic.

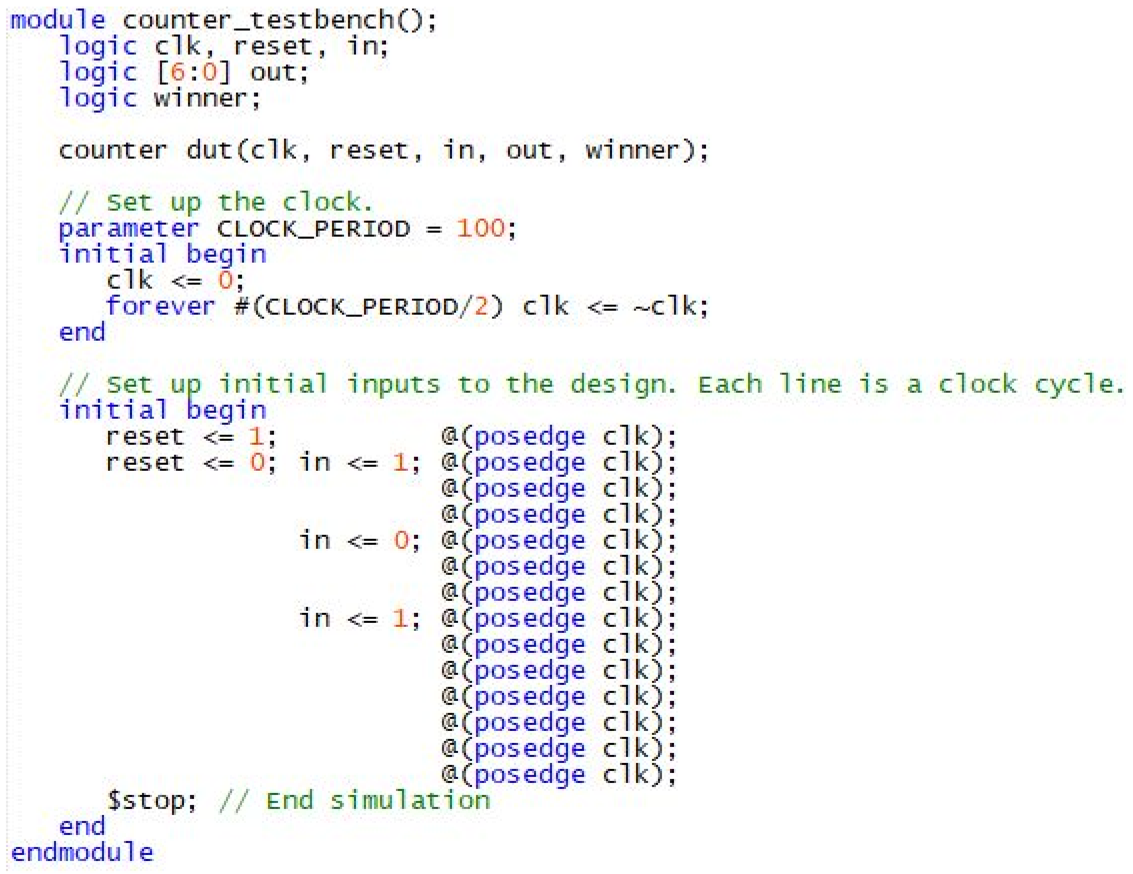
Appendix:

```
module victory(L, R, NL, NR, LW, RW);
   input logic L, R, NL, NR;
   output logic LW, RW;

   assign LW = NL & L & ~R;
   assign RW = NR & R & ~L;
endmodule
```

Code 1: victory.sv

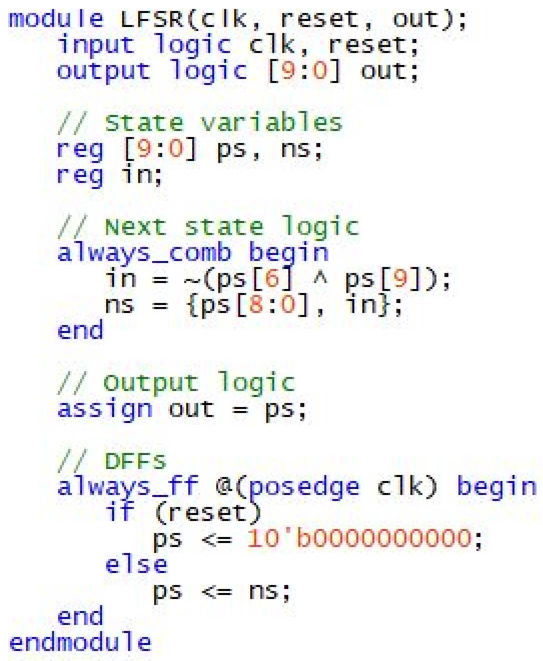


```
module counter(clk, reset, in, out, winner);
   input logic clk, reset, in;
   output reg [6:0] out;
   output logic winner;

   // State variables
   enum {A, B, C, D, E, F, G, H} ps, ns;

   // Next state logic
   always_comb begin
      case (ps)
         A: if (in)
               ns = B;
            else
               ns = ps;
         B: if (in)
               ns = C;
            else
               ns = ps;
         C: if (in)
               ns = D;
            else
               ns = ps;
         D: if (in)
               ns = E;
            else
               ns = ps;
         E: if (in)
               ns = F;
            else
               ns = ps;
         F: if (in)
               ns = G;
            else
               ns = ps;
         G: if (in)
               ns = H;
            else
               ns = ps;
         H: ns = ps;
      endcase
   end

   // Output logic
   always_comb begin
      case (ns)
         A: out = 7'b1000000; // 0
         B: out = 7'b1111001; // 1
         C: out = 7'b0100100; // 2
         D: out = 7'b0110000; // 3
         E: out = 7'b0011001; // 4
         F: out = 7'b0010010; // 5
         G: out = 7'b0000010; // 6
         H: out = 7'b1111000; // 7
      endcase
   end
   assign winner = (ns == H);

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

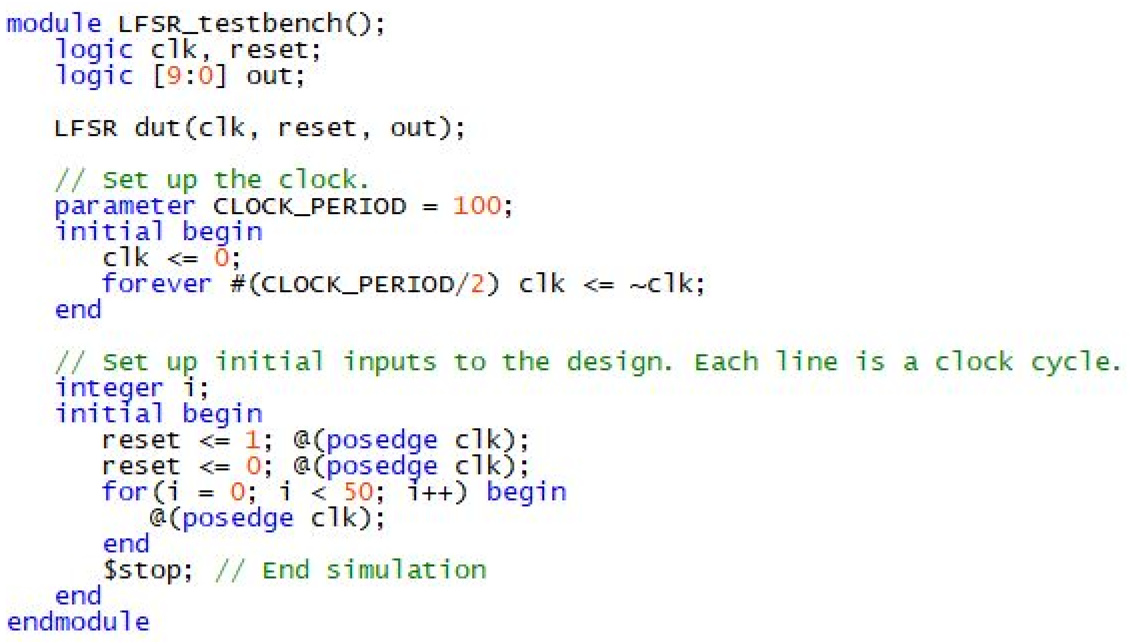


```
module counter_testbench();
   logic clk, reset, in;
   logic [6:0] out;
   logic winner;

   counter dut(clk, reset, in, out, winner);

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

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

Code 2: counter.sv

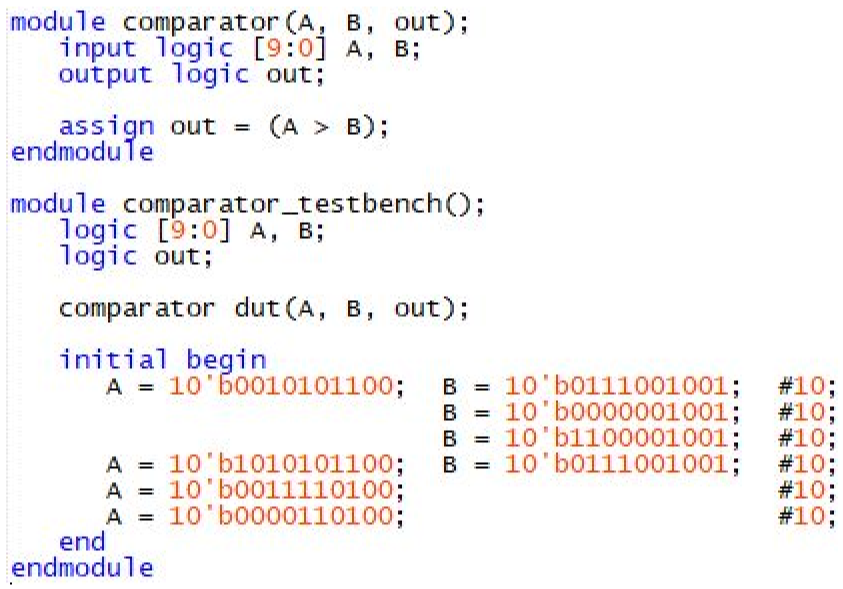


```
module LFSR(clk, reset, out);
   input logic clk, reset;
   output logic [9:0] out;

   // State variables
   reg [9:0] ps, ns;
   reg in;

   // Next state logic
   always_comb begin
      in = ~(ps[6] ^ ps[9]);
      ns = {ps[8:0], in};
   end

   // Output logic
   assign out = ps;

   // DFFs
   always_ff @(posedge clk) begin
      if (reset)
         ps <= 10'b0000000000;
      else
         ps <= ns;
   end
endmodule
```



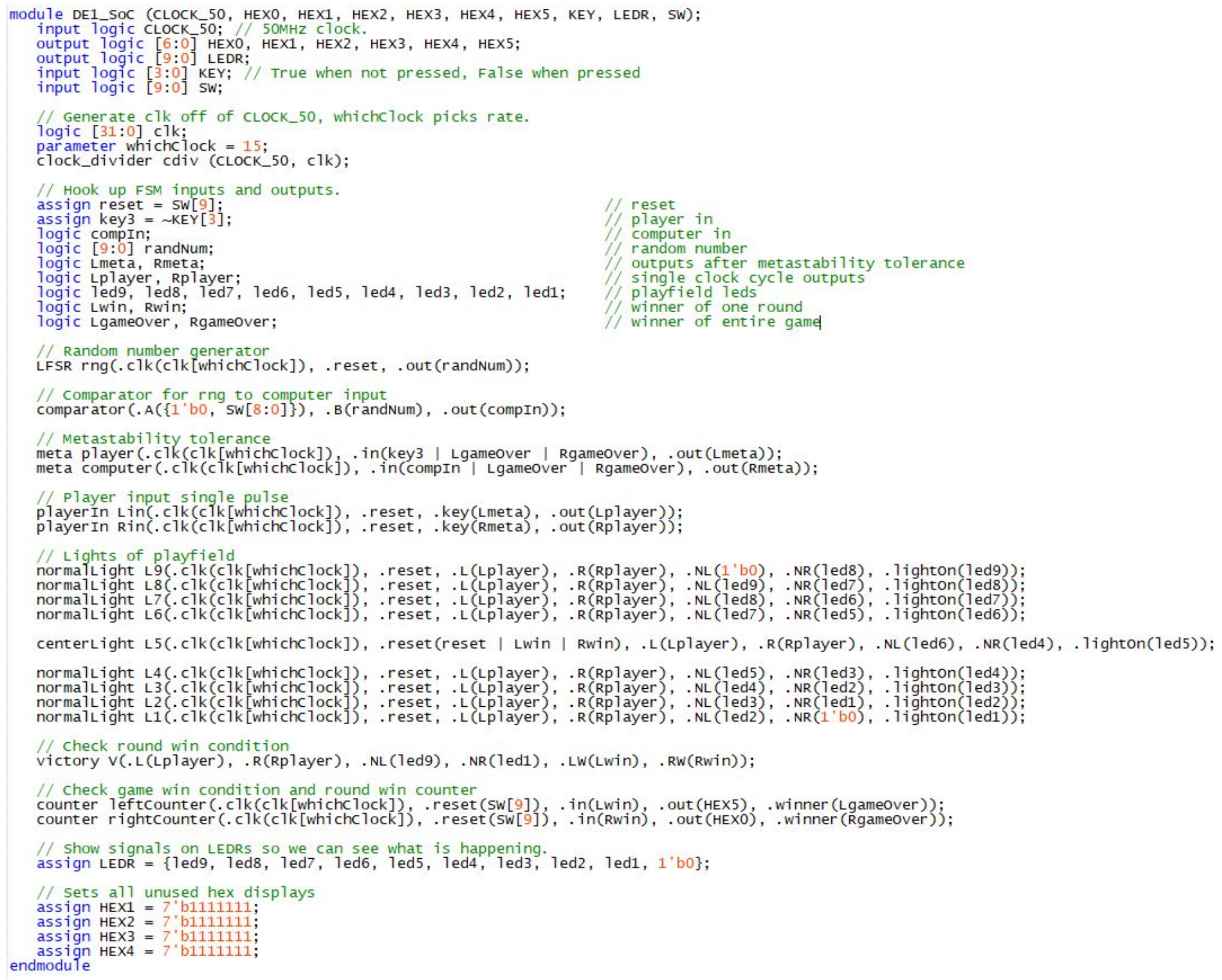
```
module LFSR_testbench();
   logic clk, reset;
   logic [9:0] out;

   LFSR dut(clk, reset, out);

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

   // Set up initial inputs to the design. Each line is a clock cycle.
   integer i;
   initial begin
      reset <= 1; @(posedge clk);
      reset <= 0; @(posedge clk);
      for(i = 0; i < 50; i++) begin
         @(posedge clk);
      end
      $stop; // End simulation
   end
endmodule
```

Code 3: LFSR.sv

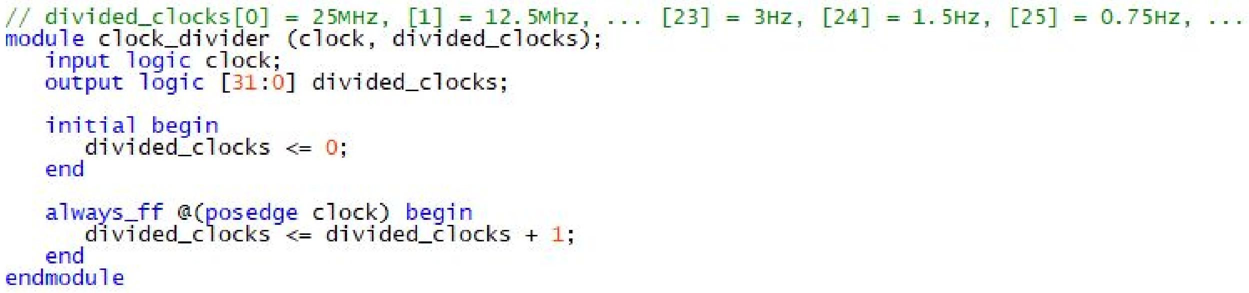


```
module comparator(A, B, out);
   input logic [9:0] A, B;
   output logic out;

   assign out = (A > B);
endmodule

module comparator_testbench();
   logic [9:0] A, B;
   logic out;

   comparator dut(A, B, out);

   initial begin
      A = 10'b0010101100;  B = 10'b0111001001;  #10;
                           B = 10'b0000001001;  #10;
                           B = 10'b1100001001;  #10;
      A = 10'b1010101100;  B = 10'b0111001001;  #10;
      A = 10'b0011110100;                       #10;
      A = 10'b0000110100;                       #10;
   end
endmodule
```

Code 4: comparator.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;

   // Generate clk off of CLOCK_50, whichClock picks rate.
   logic [31:0] clk;
   parameter whichClock = 15;
   clock_divider cdiv (CLOCK_50, clk);

   // Hook up FSM inputs and outputs.
   assign reset = SW[9];                                        // reset
   assign key3 = ~KEY[3];                                       // player in
   logic compIn;                                                // computer in
   logic [9:0] randNum;                                         // random number
   logic Lmeta, Rmeta;                                          // outputs after metastability tolerance
   logic Lplayer, Rplayer;                                      // single clock cycle outputs
   logic led9, led8, led7, led6, led5, led4, led3, led2, led1;  // playfield leds
   logic Lwin, Rwin;                                            // winner of one round
   logic LgameOver, RgameOver;                                  // winner of entire game

   // Random number generator
   LFSR rng(.clk(clk[whichClock]), .reset, .out(randNum));

   // Comparator for rng to computer input
   comparator(.A({1'b0, SW[8:0]}), .B(randNum), .out(compIn));

   // Metastability tolerance
   meta player(.clk(clk[whichClock]), .in(key3 | LgameOver | RgameOver), .out(Lmeta));
   meta computer(.clk(clk[whichClock]), .in(compIn | LgameOver | RgameOver), .out(Rmeta));

   // Player input single pulse
   playerIn Lin(.clk(clk[whichClock]), .reset, .key(Lmeta), .out(Lplayer));
   playerIn Rin(.clk(clk[whichClock]), .reset, .key(Rmeta), .out(Rplayer));

   // Lights of playfield
   normalLight L9(.clk(clk[whichClock]), .reset, .L(Lplayer), .R(Rplayer), .NL(1'b0), .NR(led8), .lightOn(led9));
   normalLight L8(.clk(clk[whichClock]), .reset, .L(Lplayer), .R(Rplayer), .NL(led9), .NR(led7), .lightOn(led8));
   normalLight L7(.clk(clk[whichClock]), .reset, .L(Lplayer), .R(Rplayer), .NL(led8), .NR(led6), .lightOn(led7));
   normalLight L6(.clk(clk[whichClock]), .reset, .L(Lplayer), .R(Rplayer), .NL(led7), .NR(led5), .lightOn(led6));

   centerLight L5(.clk(clk[whichClock]), .reset(reset | Lwin | Rwin), .L(Lplayer), .R(Rplayer), .NL(led6), .NR(led4), .lightOn(led5));

   normalLight L4(.clk(clk[whichClock]), .reset, .L(Lplayer), .R(Rplayer), .NL(led5), .NR(led3), .lightOn(led4));
   normalLight L3(.clk(clk[whichClock]), .reset, .L(Lplayer), .R(Rplayer), .NL(led4), .NR(led2), .lightOn(led3));
   normalLight L2(.clk(clk[whichClock]), .reset, .L(Lplayer), .R(Rplayer), .NL(led3), .NR(led1), .lightOn(led2));
   normalLight L1(.clk(clk[whichClock]), .reset, .L(Lplayer), .R(Rplayer), .NL(led2), .NR(1'b0), .lightOn(led1));

   // Check round win condition
   victory V(.L(Lplayer), .R(Rplayer), .NL(led9), .NR(led1), .LW(Lwin), .RW(Rwin));

   // Check game win condition and round win counter
   counter leftCounter(.clk(clk[whichClock]), .reset(SW[9]), .in(Lwin), .out(HEX5), .winner(LgameOver));
   counter rightCounter(.clk(clk[whichClock]), .reset(SW[9]), .in(Rwin), .out(HEX0), .winner(RgameOver));

   // 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;
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 5: DE1_SoC.sv