E E P 598

# DIGITAL SYSTEMS DESIGN WITH FPGAS

Lab 2

Apply Sequential Logic and LFSR in FPGA

# Lab 2 - Advanced Digital Logic

## Lab 2 - Learning Objectives

This lab will be separated into three parts.

- Part 1:
  - Familiarity with the FSM and Sequential Logic function
- Part 2:
  - Familiarity with the Communicating Sequential Logic
- Part 3:
  - Understanding the LFSR (linear feedback shift register)

If you have any questions at any point in the lab, make sure you have read the entire lab thoroughly. If you still cannot find the answer you should ask your TA. And as always, there are no dumb questions; please ask them, you don't want to damage any of the expensive hardware that is provided for you.

Note: With all the labs, keep your files when you are done with the lab – they will often get reused in subsequent labs!

# Lab 2 Instructions

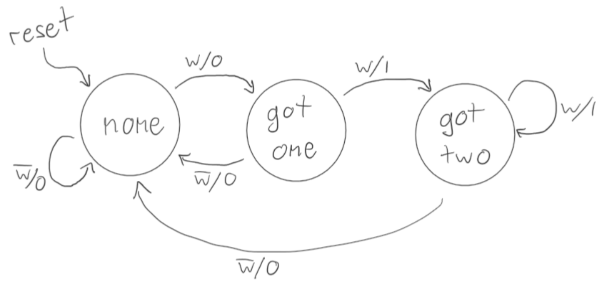
## 1. Sequential Logic

Copy the previous project folder and rename it lab2-1.

### 1.1 Tutorial Task – Mapping sequential logic with Mealy Machine

You do not have to turn in anything for the tutorial task, but running through it will help a LOT in getting the design problem working.

- To understand the sequential logic design, simulation, and execution process we'll use a simple state machine given by the diagram below that has one input w and one output out. The output is true whenever w has been true for the previous two clock cycles.



```
module simple (clk, reset, w, out);
  input logic clk, reset, w;
  output logic out;
  // State variables
  enum { none, got_one, got_two } ps, ns;
  // Next State logic
  always_comb begin
    case (ps)
      none: if (w) ns = got_one;
            else ns = none;
      got_one: if (w) ns = got_two;
               else ns = none;
      got_two: if (w) ns = got_two;
               else ns = none;
    endcase
  end
  // Output logic - could also be another always_comb block.
  assign out = (ps == got_two);
  // DFFs
  always_ff @(posedge clk) begin
    if (reset)
      ps <= none;
    else
      ps <= ns;
  end
endmodule
```

Copy this module and its testbench (next page) inside a SystemVerilog file named simple.sv.

- To simulate this logic, we not only have to provide the inputs, but must also create a simulated clock. For that, we can embed some logic in the testbench. Here is the testbench for this FSM:

```
module simple_testbench();
  logic clk, reset, w;
  logic out;
  simple dut (clk, reset, w, out);
  // Set up a simulated clock.
  parameter CLOCK_PERIOD=100;
  initial begin
    clk <= 0;
    //Forever toggle the  clock
    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); // Always reset FSMs at start
    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
```

  **Note:** In this testbench, instead of waiting for a specific amount of time with “#10;”, we wait for a clock edge with “@(posedge clk)”. In this way we wait for a clock edge to occur (and thus the FSM moves to the next state) before applying new inputs. Simulate the design with ModelSim, and make sure it works. Recall that the spec of the machine is ‘The output is true whenever w has been true for the previous two clock cycles’.

- A note on state encoding
  - Recall that to build a circuit out of a state diagram, you must first assign an arbitrary but unique encoding to each state. In the given code, the state variables

are defined with the enumeration keyword enum. This auto-assigns encodings to the states, by default as integers starting from 0:

```
// none = 0, got_one = 1, got_two = 2
enum { none, got_one, got_two } ps, ns;
```

- But if you want to use a custom encoding, you can explicitly assign their values:

```
enum { none = 2, got_one = 1, got_two = 0 } ps, ns;
```

- And if you want a bit-like encoding instead of integer encoding, you can also do that:

```
enum logic [1:0] { none = 2'b10, got_one = 2'b01, got_two = 2'b00 }
ps, ns;
```

- Next, set up the design to run on the FPGA. For this we need to provide a clock to the circuit, but the clocks on the FPGA are VERY fast (50 MHz, so a clock tick every 20 ns!). This poses an issue that may have been invisible in ModelSim - the circuit may run too fast to be practical. For example, if we are trying to animate the LEDRs, having them animate at 50 MHz would be way too fast for our eyes to discern.
    - This motivates why we'd like a slower clock, so we provide a clock divider – a module that generates slower clocks from a master clock by indexing a bit of a counter.

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

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

Copy this module inside a new SystemVerilog file named clock_divider.sv.

- Lastly, here is the top-level module that loads the simple FSM to the board. **Read this code and its comments carefully**.

```
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 reset;
 logic [31:0] div_clk;

 assign reset = SW[9];
 parameter whichClock = 25; // 0.75 Hz clock
 clock_divider cdiv (.clock(CLOCK_50),
 .reset(reset),
 .divided_clocks(div_clk));

 // Clock selection; allows for easy switching between simulation
and board clocks
 logic clkSelect;
 // Uncomment ONE of the following two lines depending on intention

 //assign clkSelect = CLOCK_50; // for simulation
 assign clkSelect = div_clk[whichClock]; // for board
 // Set up FSM inputs and outputs.
 logic w, out;
 assign w = SW[0]; // input is SW[0]

 simple s (.clk(clkSelect), .reset, .w, .out);

 // Show signals on LEDRs so we can see what is happening
 assign LEDR[9] = clkSelect;
 assign LEDR[8] = reset;
 assign LEDR[0] = out;

endmodule
```

Copy this module inside a SystemVerilog file named DE1_SoC.sv. (continues on next page)

- In the code above we select the speed of the clock via “whichClock”. whichClock = 25 yields a clock speed of about 0.75 Hz, 24 is twice as fast, 26 is twice as slow, etc. Load this design to your board. This design uses SW[9] for reset (toggle up, then down to reset the circuit), SW0 as the “w” input, and LEDR[0] as the output. You should see the effective system clock tick on LEDR[9]. Holding SW[9] up for 2 or more cycles should cause the output to be true.

- **IMPORTANT**: Note that you will need to use different system clocks depending on if you are loading to the board or simulating in ModelSim. This is explained in the ‘Clock selection’ portion of the code. The reason for this is because we want a slower clock to discern transitions in real life, but in simulation, the clock speed doesn't matter. We don't want simulations to go through the clock divider because we will need to wait for many 'simulated' clock ticks in order for the system clock to tick once. Thus, in the DE1_SoC module, uncomment the appropriate line for clkSelect:

```
// Uncomment ONE of the following two lines depending on intention

 assign clkSelect = CLOCK_50; // for simulation
 //assign clkSelect = div_clk[whichClock]; // for board
```

**Note:** You will need to switch between these two any time you wish to go between running a ModelSim simulation or loading onto your board.

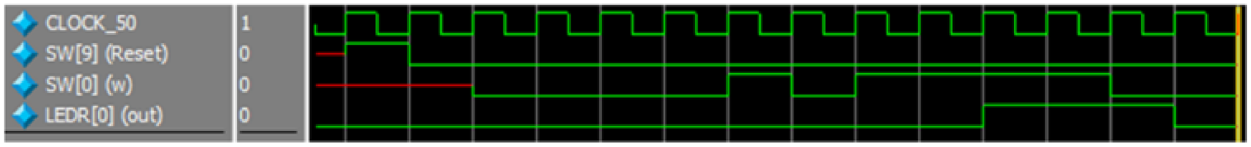
Lastly, copy the code below and simulate the DE1_SoC testbench. Read the code and comments carefully. Note the use of the repeat(N) keyword, which does the same thing as copy-pasting @posedge(CLOCK_50)N times.

```
module DE1_SoC_testbench();
 logic CLOCK_50;
 logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
 logic [9:0] LEDR;
 logic [3:0] KEY;
 logic [9:0] SW;

 DE1_SoC dut (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR,
SW);

 // Set up a simulated clock.
 parameter CLOCK_PERIOD=100;
 initial begin
   CLOCK_50 <= 0;
   // Forever toggle the clock
   forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;
 end

 // Test the design.
 initial begin
  repeat(1) @(posedge CLOCK_50);
  // Always reset FSMs at start
  SW[9] <= 1; repeat(1) @(posedge CLOCK_50);
  // Test case 1: input is 0
  SW[9] <= 0; repeat(1) @(posedge CLOCK_50);
  SW[0] <= 0; repeat(4) @(posedge CLOCK_50);
  // Test case 2: input 1 for 1 cycle
  SW[0] <= 1; repeat(1) @(posedge CLOCK_50);
  SW[0] <= 0; repeat(1) @(posedge CLOCK_50);
  // Test case 3: input 1 for >2 cycles
  SW[0] <= 1; repeat(4) @(posedge CLOCK_50);
```

```
    SW[0] <= 0; repeat(2) @(posedge CLOCK_50);
    $stop; // End the simulation.
  end
endmodule
```

Copy this testbench code into DE1_SoC.sv

Upon setting up your runlab.do and DE1_SoC_wave.do properly, you should get something that looks like this:



**Note:** When you are turning in ModelSim screenshots for the Paper Supplement, it is highly preferred that you crop just the important information - no need for full-window screenshots (See helpful tips for ModelSim at @Appendix A)

## 1.2 Assigned Task – Hazard lights

*Review the Verilog Tutorial on the website through to the end.*
**Note**: ALL FSMs in this class need to be structured similar to the "simple" code above – next state and output logic in either "always_comb" or "assign" statements, each using the "=" to assign values to signals, and stateholding elements created in an "always_ff @(posedge clk)" block with assignments using "<=".

- The landing lights at Sea-Tac are busted, so we have to come up with a new set. In order to show pilots the wind direction across the runways we will build special wind indicators to put at the ends of all runways at Sea-Tac.

- Your circuit will be given two inputs (SW[0] and SW[1]), which indicates wind direction, and three lights to display the corresponding sequence of lights: Your circuit must also have a reset signal, that resets the machine to whatever state you choose, when key[0] is pressed (i.e. if you hold the button down, the machine stays in that state).

| SW[1] | SW[0] | Meaning | Pattern (LEDR[2:0]) |
|---|---|---|---|
| 0 | 0 | Calm | 1 0 1<br>0 1 0 |
| 0 | 1 | Right to Left | 0 0 1<br>0 1 0<br>1 0 0 |
| 1 | 0 | Left to Right | 1 0 0<br>0 1 0<br>0 0 1 |

- For each situation, the lights should cycle through the given pattern. Thus, if the wind is calm, the lights will cycle between the outside lights lit, and the center light lit, over and over. The right to left and left to right crosswind indicators repeatedly cycle through three patterns each, which have the light "move" from right to left or left to right respectively.

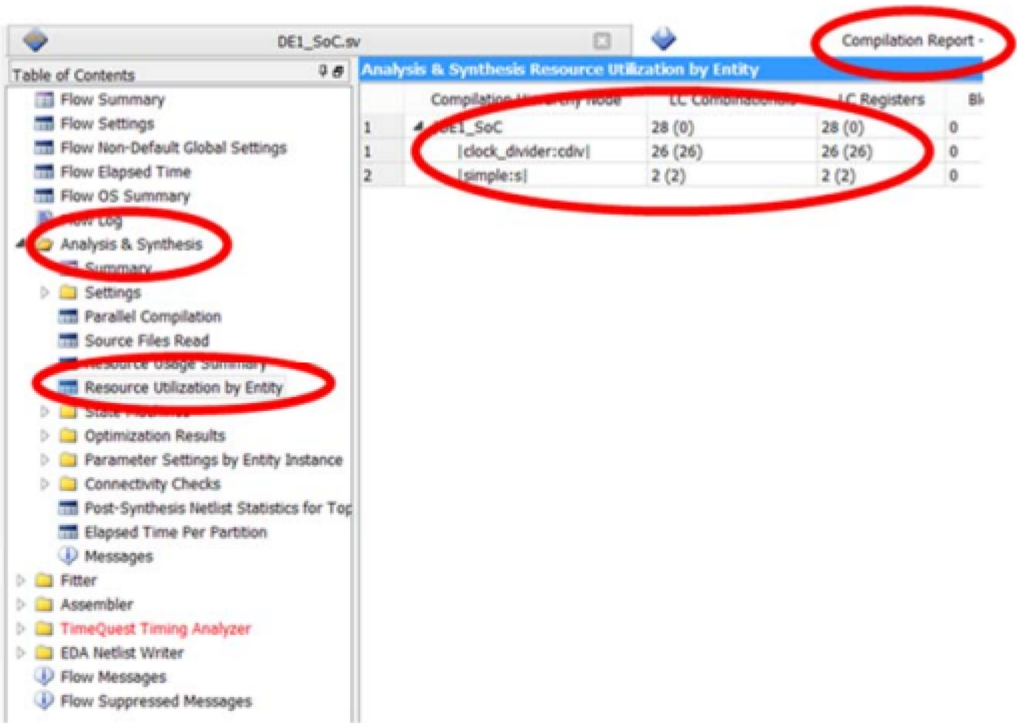
- The switches will never both be true. The switches may be changed at any point during the current cycling of the lights, and the lights must switch over to the new pattern as soon as possible (however, it can enter into any point in the other pattern's behaviors).

- Your design should be in the style of the "simple" module given above. That is, you can use "if" and "case" statements to implement the next-state logic and the outputs.

  Before you do any coding, start this problem by reasoning out what the states should be, and draw a state diagram. You will need to turn in this state diagram as part of your paper supplement.

  With some practices you should find that translating a state diagram into code is very straightforward. Like simple.sv, we recommend building and testing your FSM in its own module, then instantiating it in the top level entity DE1_SoC.sv and testing it again.

- You will be graded 100 points on correctness, style, testing, testbenches, etc. Your bonus goal is developing the smallest circuit possible, in terms of # of FPGA logic and DFF resources. We need to compute the size without the cost of the clock_divider. To do this, perform a compilation of your design, and look at the Compilation Report. In the left hand column select Analysis & Synthesis > Resource Utilization by Entity. The "LC Combinationals" column lists the amount of FPGA logic elements being used, which are logic elements that can compute any Boolean combinational function of at most 6 inputs. The "LC Registers" is the number of DFFs used. For each entry there is the listing of the amount of resources used by that specific module (the number in parentheses), and the amount of resources used by that specific module plus its submodules (the number outside the parentheses).



Analysis & Synthesis Resource Utilization by Entity

| | Compilation Hierarchy Node | LC Combinationals | LC Registers | Bl |
|---|---|---|---|---|
| 1 | DE1_SoC | 28 (0) | 28 (0) | 0 |
| 1 | \|clock_divider:cdiv\| | 26 (26) | 26 (26) | 0 |
| 2 | \|simple:s\| | 2 (2) | 2 (2) | 0 |

To compute the size of your FSM, add the numbers outside the parentheses for the entire design under “LC Combinationals” and “LC Registers”. Subtract from that the same numbers from the “clock_divider” line. The original “simple” FSM from the first part of this lab has a score of (28+28)-(26+26) = 4, though obviously it doesn’t perform the right functions for the runway lights…

## 2. Communicating Sequential Logic

Copy previous project folder and rename it lab2-2.

### 2.1 Assigned Task– Tug of War

Sweat pouring from their brow, body straining, muscles pulsing back and forth, we have the epic conflict which is: Tug Of War! It’s time to update this rope-based team sport into an electronic analog of finger-pounding power!

- We’re going to build a 2-player game using the KEY[0] and KEY[3] buttons, and the leds from LEDR9 to LEDR1, skipping LEDR0, as the playfield. When the game starts, only the centermost LED is lit (LEDR5). Each time the first player presses the KEY[0] button, the light moves one LED right. Each time the second player presses the KEY[3] button, the light moves one LED to the left. If the light ever goes off the end of the playfield, the player that moved it off the end wins, and the HEX0 7-segment display shows 1 for first player, 2 for second player. You can use SW9 as the reset signal.

- If you try to design this as one big state machine, you will never get anything working. Instead, think about breaking it down into smaller pieces. We will help you with some ideas, but we STRONGLY advise putting together a block diagram of the system early in the design process.

- You should use the 50MHz clock directly (pin CLOCK_50) to control the whole design – we'll assume no player can press the button faster than 25 million times a second…

- **User Input**

  Since we are using a fast clock, each time the user presses a button the button will be ON for many cycles, and OFF for many cycles. However, you want to design a simple FSM that detects the moment the button is pressed – its output is TRUE for only 1 cycle for every button press. This will handle all user input.

- **Playfield**

  There are 9 lights, which is too big to do as a single huge FSM. However, what about an FSM for each location? A given playfield light needs to know the following:
  - Does it start as TRUE (the center LED) or FALSE? This could be an input to the module.

  - During play, it needs to know which button(s) were just pressed, whether its light is currently lit, and whether its right and left neighbors are currently lit.

  - Given all that data, plus the reset signal, it's now easy to figure out whether you should be lit during the next clock cycle.

- **Victory**

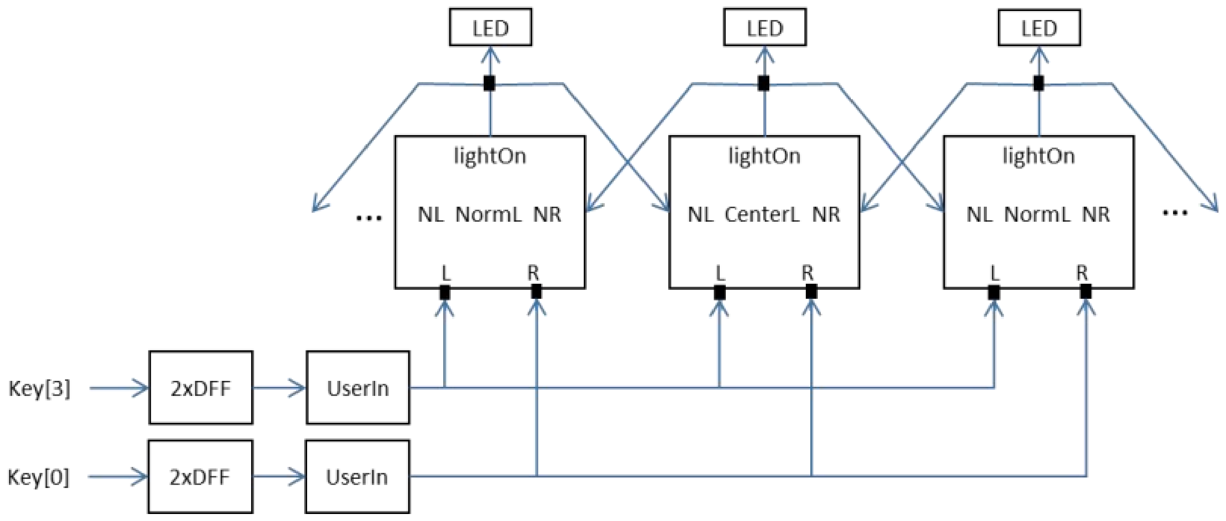
  You can tell when someone wins by watching the ends of the playfield – when the leftmost LED is lit and only the left button is pressed, the left player wins. Similar logic can be found for the right player. So, build a unit that controls the HEX0 display, based on these victory conditions.

- **Suggested FSMs**

  Your playfield and victory lights could be controlled by the following FSMs. Note that these are only suggestions. You are free to create the design using any number of different FSMs.

```
module centerLight (Clock, Reset, L, R, NL, NR, lightOn);
  input logic Clock, Reset;
  // L is true when left key is pressed, R is true when the right key
  // is pressed, NL is true when the light on the left is on, and NR
  // is true when the light on the right is on.
  input logic L, R, NL, NR;
  // when lightOn is true, the center light should be on.
  output logic lightOn;
  // Your code goes here!!
  endmodule

module normalLight (Clock, Reset, L, R, NL, NR, lightOn);
  input logic Clock, Reset;
  // L is true when left key is pressed, R is true when the right key
  // is pressed, NL is true when the light on the left is on, and NR
  // is true when the light on the right is on.
  input logic L, R, NL, NR;
  // when lightOn is true, the normal light should be on.
  output logic lightOn;
  // Your code goes here!!
endmodule
```

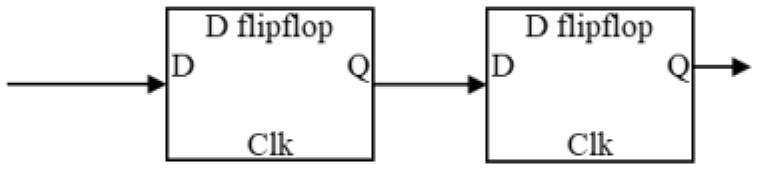
  If you were to use the above FSMs in your design, you'd need 1 center light and 8 normal light FSMs. In addition, you'd need two instantiations of a user input FSM, and logic to determine when someone has won the competition. None of the FSMs should require more than four states.



- **Metastability**

  This lab has user input going into a somewhat high-speed circuit. That means there's a pretty good chance you can get metastability – the input to a DFF changing at about the same time as the clock edge occurs. **If you do not deal with this problem, your circuit may randomly screw up.**

  To deal with metastability, make sure you send each of the user inputs (KEY[3] and KEY[0]) to a pair of D-flipflops in series BEFORE you use it in your logic (i.e. the rest of your circuit won't use KEY[3] nor KEY[0] directly, but instead the input will go in as the D of the first DFF, and the circuit will listen to the Q output of the second DFF).



- **Overall**

  Build each of the pieces and test them independently in ModelSim before combining them together. TEST EACH ELEMENT IN MODELSIM BEFORE TRYING TO HOOK IT ALL UP. TEST THE WHOLE THING IN MODELSIM BEFORE DOWNLOADING TO THE FPGA. If you try to do everything by just downloading it to the FPGA you will have LOTS of trouble getting this lab working, and subsequent labs will be MUCH harder – simulation and good complete testbenches are your friend, will SIGNIFICANTLY speed up your debugging. Only once you have all the pieces, and then the entire system, working in Modelsim should you download the design to the FPGA and test the working game (the fun part…). Note that during testing you may want a slower clock – you can always use the clock divider from section 1 to help you in this process.

  You can watch the example demo video to get a sense of what the final product should look like. You will be graded 100 points on correctness, style, testing, etc. Your bonus goal is developing the smallest circuit possible – measure this the same way you did in section 1.
  **Note:** that the "Resource Utilization by Entity" report will give you the sizes of each of the modules in your design, so you can focus your sizing improvement efforts accordingly.

# 3. LFSRs (Linear-feedback shift register)

Copy previous project folder and rename it lab2-3.

## 3.1 Assigned Task– CyberWar

In the last lab we built a simple Tug of War game, and by now you've already crushed your room-mate into submission. Now it's the hardware's turn. Your goal is to develop a computer opponent to play against, as well as a scorekeeper that can show exactly how badly it beats you…

- **Counters**

  First off, take your lab2-3 and replace the "winner" system with counters. Specifically, develop a 3-bit counter (holds values 0..7). It starts at 0, and whenever a "win" comes in, it increments its current value by 1. This is a simple FSM. Note that we assume once one player gets to 7 the game is over, so it doesn't matter what happens when a player with 7 points gets one more.

  Now, alter your lab2-3 so that there is a counter for each player, which drives a per-player 7- segment display with the current score for that player. Whenever someone wins, you increment the appropriate player's score, then restart the game (i.e. automatically reset the playfield). Resetting the entire game will reset the playfield and score, while winning only resets the playfield.

- **LFSRs**

  To build a cyber-player, we need to create a random number generator to simulate the button presses. In hardware, the simplest way to do this is generally an LFSR (linear feedback shift register). It consists of a set of N D-flip-flops (DFF1…DFFN), where the output of DFFi is the input of DFFi+1. The magic comes in on the input of DFF1. It is the XNOR of 2 or more outputs of the DFF. By picking the bits to XNOR carefully, you get an FSM that goes through a fairly random pattern, but with very simple hardware (note that these LFSRs can never go into the state with all 1's, but reach all others). Two examples are given below.

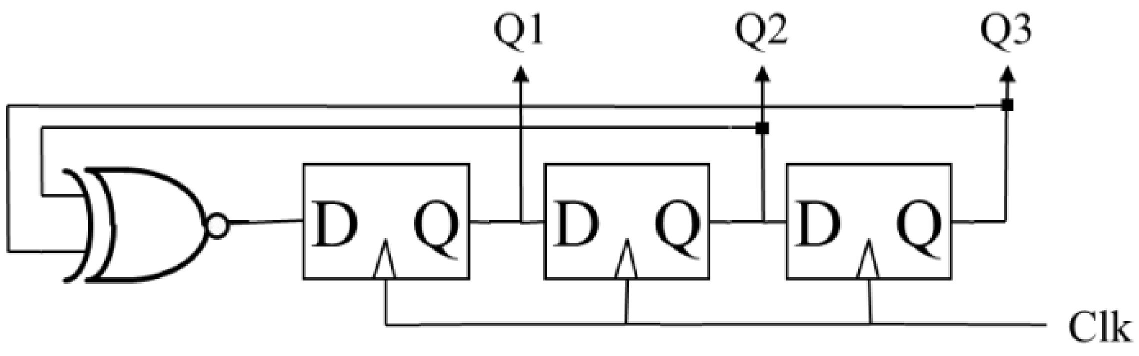


Figure 1. 3-bit LFSR

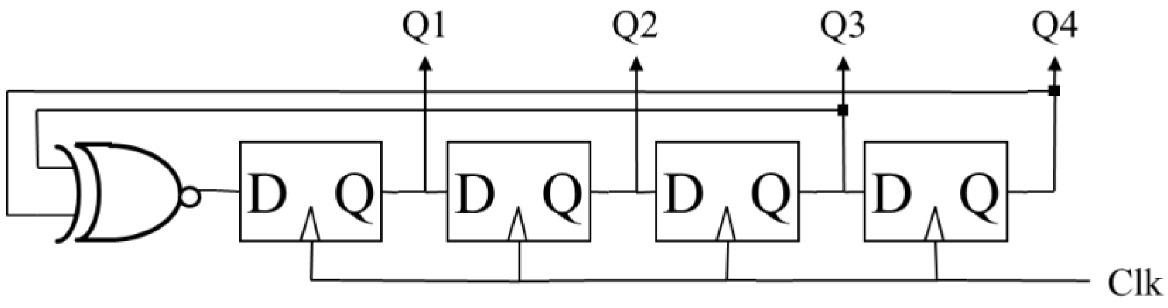


Figure 2. 4-bit LFSR

First, draw the state diagram for these two circuits. It will show every possible state for the machine (8 for the 3-bit, 16 for the 4-bit), with arrows showing the next state they enter after that state.

Next, create a 10-bit LFSR in Quartus and simulate it. You can find the list of bits to XNOR together in the table at **Appendix B** (do NOT make up your own – most choices don't work well, so the table shows the "best" connections to make).

**ModelSim Tip**: now that you are working with multi-bit signals, it can be helpful in ModelSim to display signals as decimal or hex values. To do this, right-click on a multi-bit signal in ModelSim, and select "Radix". Hex and Unsigned are the most useful choices IMHO.

- **Comparator**

  Develop a 10-bit comparator. The unit takes in two unsigned 10-bit numbers, called A and B, and returns TRUE if A>B, FALSE otherwise. You can think about this as a subtraction problem, or just by considering the individual bits of the number themselves.

- **CyberPlayer**

We now have most of the components to implement a tunable cyber-player. First, let's slow things down so you have a chance – run your entire Tug of War game off of the clock divider's divided_clocks[15] (about 768 Hz). To generate the computer's button presses, compared the LFSR output (a value from 0…1023) to the value on SW[8:0] (a value from 0…511) – you can extend the SW[8:0] with a 0 at the top bit to make it a 10-bit unsigned value. If the SW value is greater than the LFSR value, consider this a computer button-press (i.e. the light should move one space toward the computer player's end, assuming the human doesn't make a move at the same instant). You can speed up or slow down the system by simply playing with the user switches, to see how fast you can go. If the clock is too fast, feel free to adjust to a different divided_clock output (for the ENTIRE design).
**Note**: be sure that EVERYTHING that is clocked in your design (except for the clock_divider circuit) uses the same clock. If you use any other clock then strange things can happen.

> **USE CLOCK_50** during simulation! Use the divided_clock in your compiled design, but EVERYTHING must be clocked using that one clock.

You will be graded 100 points on correctness, style, testing, testbenches, etc. Your bonus goal is developing the smallest circuit possible, measured in the same way as previous sections.

## Deliverables

### What's in the report:

Assigned Task in the section 1:

- A drawing of your state diagram used to implement the runway lights

- Screenshot of ModelSim simulation of your top-level entity (DE1_SoC) demonstrating runways lights circuit behavior for all 3 possible wind conditions.

- Screenshot of "Resource Utilization by Entity," along with the computed size of your design.

- A demo video link:
    - Program your runway lights circuit to the board.
    - Demonstrate your runway lights circuit working on the DE1 board. Show behavior for all 3 wind conditions; clearly state the wind condition (Calm, Right to Left, Left to Right) currently being demonstrated. Show at least 2 complete cycles for each wind condition.

Assigned Task in section 2:

- A drawing of your top-level block diagram for your entire design, showing the major modules and how they are interconnected. It needs to show how the player input (KEYs) reach the lightfield output (LEDRs).

- Drawings of your state diagrams for all FSM that you built.

- A screenshot of ModelSim simulation of your top-level entity (DE1_SoC).

- Screenshots (3+) of ModelSim simulation of each important element you built for this section.

- Screenshot of "Resource Utilization by Entity," along with the computed size of your design.

- A demo video link:
    - Program your Tug of War circuit to the board.
    - Toggle reset, then demonstrate Player 1 (left) winning
    - Toggle reset, then demonstrate Player 2 (right) winning
    - Toggle reset, then simulate a 'real' game by pressing both buttons randomly, causing the light to go back-and-forth, then have any player win.

Assigned Task in the section 3:

- A drawing of your state diagram derived from the 4-bit LFSR.

- Screenshot of ModelSim simulation of your top-level entity showing the human player getting 5 points.

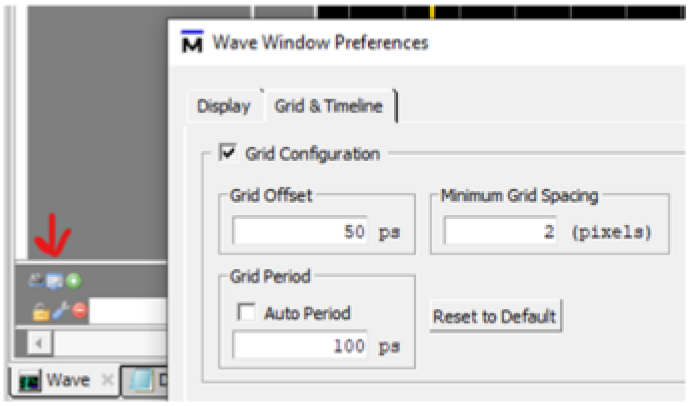
- Screenshot of "Resource Utilization by Entity," along with the computed size of your design.

- A demo video link:
    - Program your game to the board.
    - Set SW[2:0] to 000, then toggle reset. Demonstrate human/Player 2 getting 5 points.
    - Set SW[2:0] to 100, then toggle reset. Demonstrate the bot getting 5 points
    - Set SW[2:0] to 001, then toggle reset. Allow the bot to get 3 point, then demonstrate human/Player 2 getting 7 points
    - **You may use SW[3:1] or SW[4:2] for changing the probability of a button press for the 'computer' opponent if SW[2:0] doesn't noticeably change the probability.**

**Compress the report and all the project folders into a zip file. Then Submit it to the canvas. If the video is too large, you are allowed to use a direct google drive link and add it as a comment.**

# Appendix A - **Helpful ModelSim Tips**

- To align the gray tick lines with the rising clock edge, click on the small 'Grid, Timeline, & Cursor Control' icon on the bottom left of the waveform window, and enter in the following parameters:



Wave Window Preferences

Grid Configuration: ✓

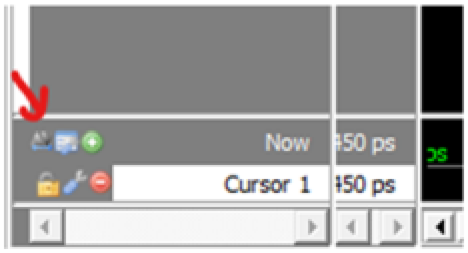
Grid Offset: 50 ps

Minimum Grid Spacing: 2 (pixels)

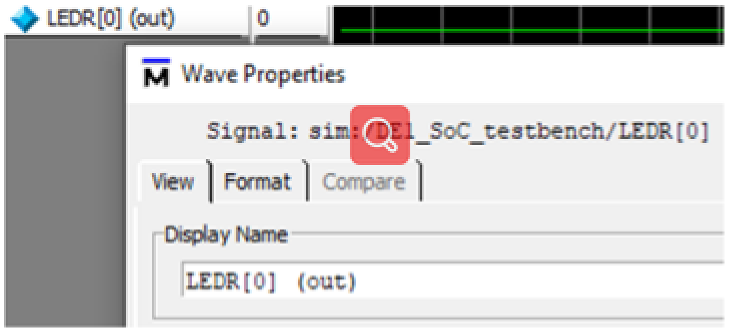
Auto Period: ☐

Grid Period: 100 ps

- To toggle leaf names (i.e., change your signal names from '/DE1_SoC_testbench/SW' to 'SW', click on the 'Toggle leaf names ⇔ full names' icon, also on the bottom left of the waveform window:



- To rename a signal to anything you want, right click on the signal and select 'Properties', then enter a new Display Name:

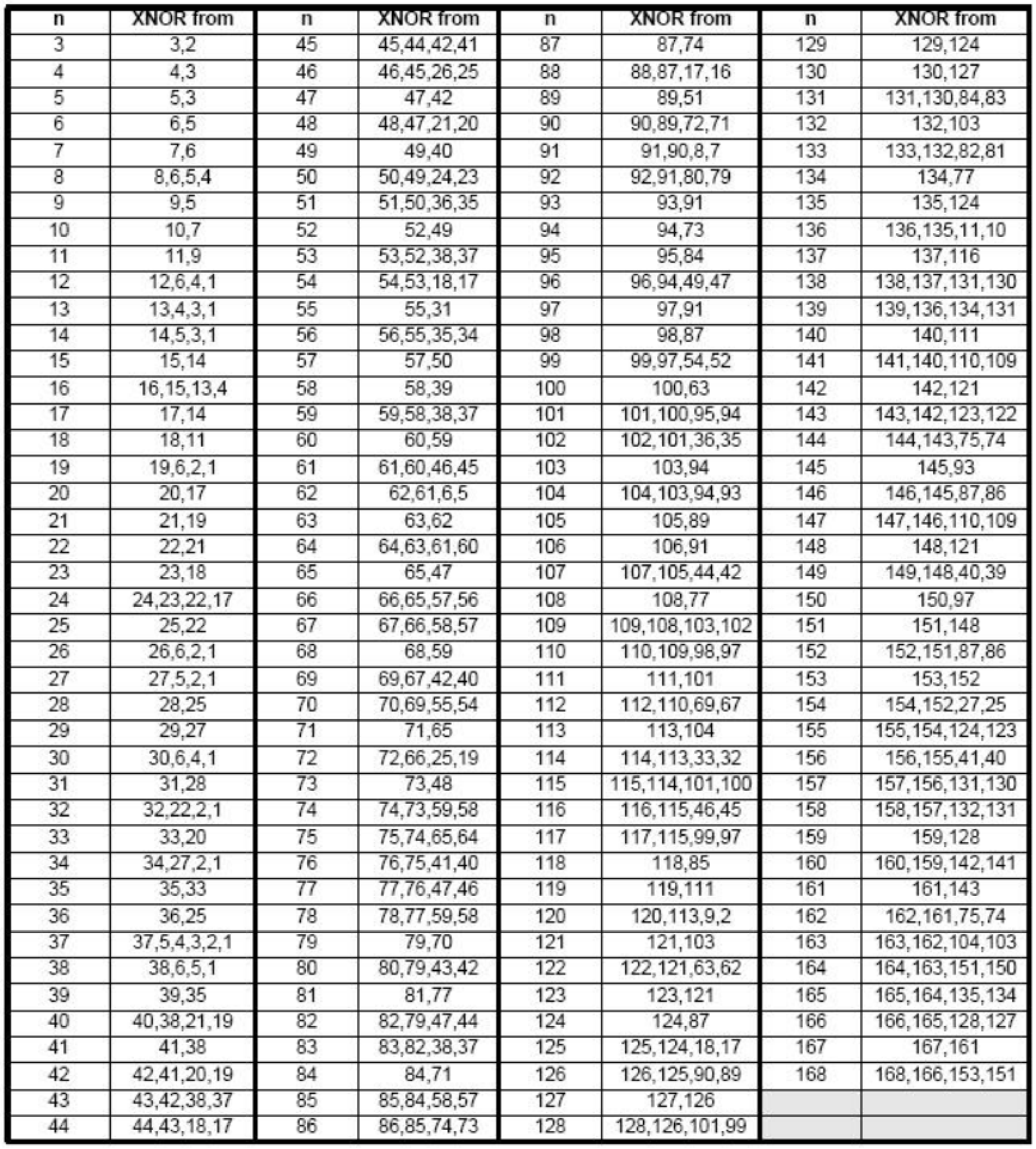


Wave Properties

Signal: sim:/DE1_SoC_testbench/LEDR[0]

Display Name: LEDR[0] (out)

# Appendix B-LFSR taps



| n | XNOR from | n | XNOR from | n | XNOR from | n | XNOR from |
|---|---|---|---|---|---|---|---|
| 3 | 3,2 | 45 | 45,44,42,41 | 87 | 87,74 | 129 | 129,124 |
| 4 | 4,3 | 46 | 46,45,26,25 | 88 | 88,87,17,16 | 130 | 130,127 |
| 5 | 5,3 | 47 | 47,42 | 89 | 89,51 | 131 | 131,130,84,83 |
| 6 | 6,5 | 48 | 48,47,21,20 | 90 | 90,89,72,71 | 132 | 132,103 |
| 7 | 7,6 | 49 | 49,40 | 91 | 91,90,8,7 | 133 | 133,132,82,81 |
| 8 | 8,6,5,4 | 50 | 50,49,24,23 | 92 | 92,91,80,79 | 134 | 134,77 |
| 9 | 9,5 | 51 | 51,50,36,35 | 93 | 93,91 | 135 | 135,124 |
| 10 | 10,7 | 52 | 52,49 | 94 | 94,73 | 136 | 136,135,11,10 |
| 11 | 11,9 | 53 | 53,52,38,37 | 95 | 95,84 | 137 | 137,116 |
| 12 | 12,6,4,1 | 54 | 54,53,18,17 | 96 | 96,94,49,47 | 138 | 138,137,131,130 |
| 13 | 13,4,3,1 | 55 | 55,31 | 97 | 97,91 | 139 | 139,136,134,131 |
| 14 | 14,5,3,1 | 56 | 56,55,35,34 | 98 | 98,87 | 140 | 140,111 |
| 15 | 15,14 | 57 | 57,50 | 99 | 99,97,54,52 | 141 | 141,140,110,109 |
| 16 | 16,15,13,4 | 58 | 58,39 | 100 | 100,63 | 142 | 142,121 |
| 17 | 17,14 | 59 | 59,58,38,37 | 101 | 101,100,95,94 | 143 | 143,142,123,122 |
| 18 | 18,11 | 60 | 60,59 | 102 | 102,101,36,35 | 144 | 144,143,75,74 |
| 19 | 19,6,2,1 | 61 | 61,60,46,45 | 103 | 103,94 | 145 | 145,93 |
| 20 | 20,17 | 62 | 62,61,6,5 | 104 | 104,103,94,93 | 146 | 146,145,87,86 |
| 21 | 21,19 | 63 | 63,62 | 105 | 105,89 | 147 | 147,146,110,109 |
| 22 | 22,21 | 64 | 64,63,61,60 | 106 | 106,91 | 148 | 148,121 |
| 23 | 23,18 | 65 | 65,47 | 107 | 107,105,44,42 | 149 | 149,148,40,39 |
| 24 | 24,23,22,17 | 66 | 66,65,57,56 | 108 | 108,77 | 150 | 150,97 |
| 25 | 25,22 | 67 | 67,66,58,57 | 109 | 109,108,103,102 | 151 | 151,148 |
| 26 | 26,6,2,1 | 68 | 68,59 | 110 | 110,109,98,97 | 152 | 152,151,87,86 |
| 27 | 27,5,2,1 | 69 | 69,67,42,40 | 111 | 111,101 | 153 | 153,152 |
| 28 | 28,25 | 70 | 70,69,55,54 | 112 | 112,110,69,67 | 154 | 154,152,27,25 |
| 29 | 29,27 | 71 | 71,65 | 113 | 113,104 | 155 | 155,154,124,123 |
| 30 | 30,6,4,1 | 72 | 72,66,25,19 | 114 | 114,113,33,32 | 156 | 156,155,41,40 |
| 31 | 31,28 | 73 | 73,48 | 115 | 115,114,101,100 | 157 | 157,156,131,130 |
| 32 | 32,22,2,1 | 74 | 74,73,59,58 | 116 | 116,115,46,45 | 158 | 158,157,132,131 |
| 33 | 33,20 | 75 | 75,74,65,64 | 117 | 117,115,99,97 | 159 | 159,128 |
| 34 | 34,27,2,1 | 76 | 76,75,41,40 | 118 | 118,85 | 160 | 160,159,142,141 |
| 35 | 35,33 | 77 | 77,76,47,46 | 119 | 119,111 | 161 | 161,143 |
| 36 | 36,25 | 78 | 78,77,59,58 | 120 | 120,113,9,2 | 162 | 162,161,75,74 |
| 37 | 37,5,4,3,2,1 | 79 | 79,70 | 121 | 121,103 | 163 | 163,162,104,103 |
| 38 | 38,6,5,1 | 80 | 80,79,43,42 | 122 | 122,121,63,62 | 164 | 164,163,151,150 |
| 39 | 39,35 | 81 | 81,77 | 123 | 123,121 | 165 | 165,164,135,134 |
| 40 | 40,38,21,19 | 82 | 82,79,47,44 | 124 | 124,87 | 166 | 166,165,128,127 |
| 41 | 41,38 | 83 | 83,82,38,37 | 125 | 125,124,18,17 | 167 | 167,161 |
| 42 | 42,41,20,19 | 84 | 84,71 | 126 | 126,125,90,89 | 168 | 168,166,153,151 |
| 43 | 43,42,38,37 | 85 | 85,84,58,57 | 127 | 127,126 | | |
| 44 | 44,43,18,17 | 86 | 86,85,74,73 | 128 | 128,126,101,99 | | |

LFSR taps [XAPP 052 July 7, 1996 (Version 1.1), Peter Alfke, Xilinx Inc]