

Intro to Digital Logic, Lab 4

Sequential Logic

Lab Objectives

Now that we have mastered combinational logic, it is time to figure out sequential circuits. In this lab, you will first play with a pre-made finite state machine (FSM) design before designing your own circuit.

Task #1 – Mapping sequential logic to the FPGA

To understand the sequential logic design, simulation, and execution process we'll use a state machine that has one input `w` and one output `out`. The output is true whenever `w` has been true for the previous two clock cycles (*i.e.* it recognizes the input sequence 1, 1). The Verilog code for the machine is given in Figure 1:

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

    logic [1:0] ps; // Present State
    logic [1:0] ns; // Next State

    // State encoding
    parameter [1:0] A = 2'b00, B = 2'b01, C = 2'b10;

    // Next State logic
    always_comb
        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;
            default:    ns = 2'bxx;
        endcase

    // Output logic - could have been in new or previous always block
    assign out = (ps == C);

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

endmodule
```

Figure 1: Verilog code for simpleFSM. Note the four main elements – the state (DFFs), state encoding, NS logic, and Out logic.

To simulate this logic, we not only have to provide the inputs, but must also specify the clock. For that, we can embed some simple logic in the testbench. The testbench for this FSM is found in *Figure 2*:


```
module simpleFSM_testbench();
    logic clk, reset, w;
    logic out;

    simpleFSM 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
```

Figure 2: Verilog code for `simpleFSM_testbench`. The input signal changes occur just after the rising edge of the clock.

 Instead of waiting for a specific amount of time (e.g. `#10;`), we wait for a clock edge with “`@ (posedge clk)`”. This way we let the FSM update its state before applying new inputs. Note that embedding `@ (posedge clk)` inside `initial` blocks is fine for testbenches and simulation, but should not be part of the code that is downloaded onto the FPGA.

Simulate the design with ModelSim, and make sure it works.



ModelSim Tip for sequential logic waveform viewing:

Clocks are really important to FSMs, so it is useful to set the grey gridlines in the Wave pane to line up with the positive edge of the clocks. Click on the “Edit Grid & Timeline Properties...” icon in the lower-left corner of the Wave pane (*Figure 3*).

<< continued on next page >>

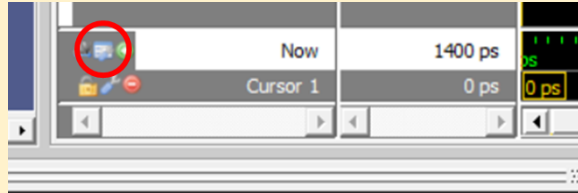


Figure 3: The location of the “Edit Grid & Timeline Properties...” button in the Wave pane.

Set the “Grid Period” to 100 ps, which is the simulation clock period (set by parameter `CLK_PERIOD = 100`), and the grid lines will line up with the clock. Make sure to save the formatting to the `simpleFSM_wave.do` file!

Next, set up the design to run on the FPGA:

We need to connect a (real) clock to the circuit, but the clocks on the chip are very fast (50 MHz → a clock cycle every 20 ns). We’d like to use a slower clock, so we provide a *clock divider* – a circuit that generates slower clocks from a master clock. Below is a version of the `DE1_SoC` main file (Figure 4) that includes the clock divider and sets up `simpleFSM` to use the LEDs and pushbuttons.

```
module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
    input logic      CLOCK_50;    // 50 MHz "master" 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(CLOCK_50), .divided_clocks(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];    // FSM input is KEY[1]

    simpleFSM s (.clk(clk[whichClock]), .reset, .w, .out);

    // Show signals on LEDR[5]-LEDR[0] 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
// HARDWARE ONLY - not to be used in simulation
module clock_divider (clock, divided_clocks);
    input logic      clock;
    output logic [31:0] divided_clocks;

    initial
        divided_clocks = 0;

    always_ff @(posedge clock)
        divided_clocks <= divided_clocks + 1;

endmodule
```

Figure 4: Updated Verilog code for `DE1_SoC.sv` that uses divided clocks for FSMs.



We select our clock via the parameter `whichClock`. The different clocks are generated by using the bits of a 32-bit bus `divided_clocks` (i.e. an integer). We add 1 to the integer every clock cycle, so higher-order bits will flip/change less frequently – exactly half as frequently as their right-neighboring bit.

`whichClock = 25` yields a clock frequency of 0.75 Hz → a clock cycle every 1.33 seconds.

Add the new `DE1_SoC.sv` to your project, compile the design for the FPGA, and test it on the DE1.

This design uses KEY0 for reset (press to reset the circuit) and KEY1 as the w input. LEDR5 shows the clock, LEDR3 shows the reset, and LEDR0 shows the output of the FSM.

You do not have to demonstrate this circuit working, but running this test will help a LOT in getting the design problem working.

Task #2 – Runway Landing Lights



We recommend structuring all FSMs for this course in a similar manner to the “simple” code above – next state and output logic in either “`always_comb`” or “`assign`” statements, each using “`=`” to assign values to signals, and state-holding elements created in an “`always_ff @ (posedge clk)`” block with assignments using “`<=`”. *This holds for labs 4-7.*

The landing lights at the local airport SEA-TAC are broken, 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[1]` and `SW[0]`), which indicate wind direction, and three output LEDs (`LEDR[2:0]`) to display the corresponding sequence of lights:

SW[1]	SW[0]	Meaning	Repeating Pattern
0	0	Calm	P0: 1 0 1 P1: 0 1 0
0	1	Right to Left	P0: 0 0 1 P1: 0 1 0 P2: 1 0 0
1	0	Left to Right	P0: 1 0 0 P1: 0 1 0 P2: 0 0 1

Table 1: Desired sequence of LED patterns for the different airport situations.

For each situation, the lights should cycle through the given pattern. In a “Calm” wind, the lights will cycle between the outside lights lit, and the center light lit (two patterns), over and over again. The “Right to Left” and “Left to Right” crosswind indicators repeatedly cycle through three patterns each, which makes the light “move” from right to left or left to right, respectively.

The switches should 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 situation’s pattern.

Design your own FSM to implement the runway landings lights described above, test it thoroughly, and make sure it runs correctly on the DE1. Make sure you include a reset button. Think carefully about your FSM design! This can be accomplished using 4 states.

Lab Grading

Working Design: 100 points for correctness, style, and testing.

Bonus: 10 points for developing the smallest circuit possible, in terms of number of FPGA logic and DFF resources.

To compute the size of your FSM, compile your design in Quartus and look at the Compilation Report (Figure 5). In the left-hand column, select “Analysis & Synthesis” → “Resource Utilization by Entity.” The “Combinational ALUTs” column lists the # of FPGA logic elements being used, and the “Dedicated Logic Registers” is the # of DFFs used.

Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers
1 DE1_SoC	28 (0)	28 (0)
1 clock_divider:cdv	26 (26)	26 (26)
2 simpleFSMs	2 (2)	2 (2)

Figure 5: How to compute the size of your FSM.

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).

Add the numbers outside the parentheses for the entire design under “Combinational ALUTs” and “Dedicated Logic Registers”. Subtract from that the same numbers from the “clock_divider” line. The `simpleFSM` module 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...

Lab Demonstration/Turn-In Requirements

Lab Report (before Wednesday section, submit as PDF on Canvas)

- A drawing of your Finite State Machine.
- A screenshot of the ModelSim simulations that you will demonstrate in-person.
- A screenshot of the “Resource Utilization by Entity” page, showing your design’s computed size.
- How many hours (estimated) it took to complete this lab in total, including reading, planning, design, coding, debugging, and testing.
- As a *separate* file, upload the Verilog code (including testbench) for your runway lights design.

In-Person Demo (during your demo slot)

- Demonstrate your runway lights circuit working in simulation.
- Demonstrate your runway lights circuit working on the DE1 board.
- Be prepared to answer questions on both the theoretical and practical parts of the lab.

Lab 4 Rubric

Grading Criteria	Points
Q1: Drawing of your FSM	10 pts
▪ Explanation of state diagram	10 pts
Q2: ModelSim screenshot of Runway Landing Lights FSM	10 pts
▪ Explanation of waveforms	15 pts
Q3: Screenshot of Resource Utilization	8 pts
▪ BONUS for small resource utilization	(10 pts)
Time spent	2 pts
Verilog code uploaded	5 pts
LAB DEMO	40 pts
	100 pts