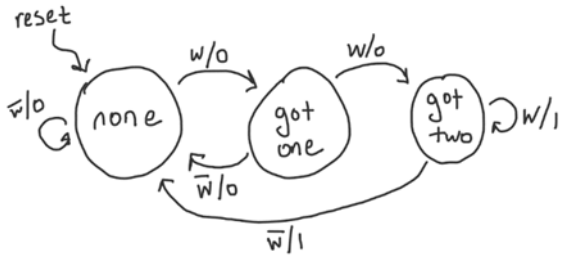
Intro to Digital Logic, Lab 5
Sequential Logic

Lab Objectives

Now that we have mastered combinational logic, it is time to figure out sequential circuits. In this lab you will download a premade design to your board. Then, you get to design your own circuit and get it working.

Tutorial Task – Mapping sequential logic to the FPGA

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.



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

```
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
```

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 #(CLOCK_PERIOD/2) clk <= ~clk;     //   Forever   toggle   the   clock
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 that 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:

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

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 (50MHz, so a clock tick every 20ns!). 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.

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

```
// 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
```

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

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

```
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
```

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.

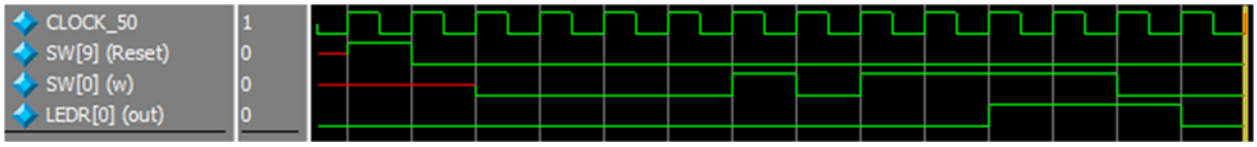
Lastly, copy the code below and simulate the DE1_SoC testbench. Read the code and comments carefully. Note the use of the repeat(N) keyword, which does the same thing as copy-pasting `@posedge(CLOCK_50)`N times.

Copy this testbench code into **DE1_SoC.sv**.

```
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 #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;    // Forever toggle the clock
    end

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

```
endmodule
```

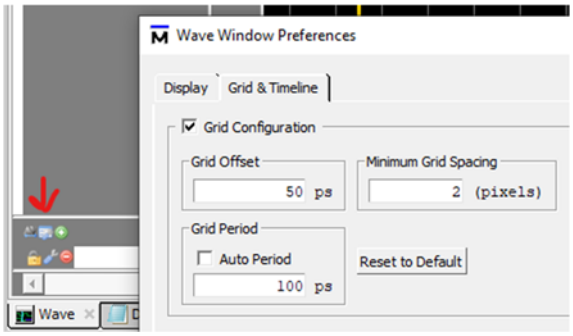
Upon setting up your runlab.do and DE1_SoC_wave.do properly, you should get something that looks like this:



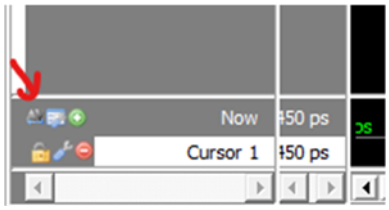
Note that when you are turning in ModelSim screenshots for the Paper Supplement, it is highly preferred that you crop just the important information - no need for full-window screenshots.

Helpful ModelSim Tips

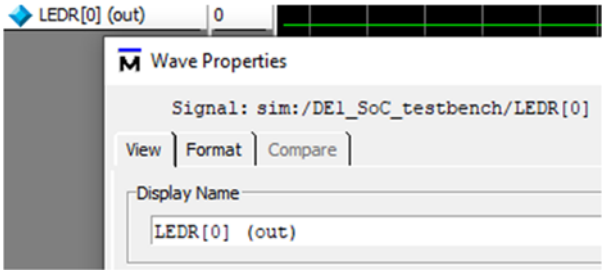
To align the grey tick lines with the rising clock edge, click on the small 'Grid, Timeline, & Cursor Control' icon on the bottom left of the waveform window, and enter in the following parameters:



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:



Design Problem – 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 "<=". This holds for labs 5 – 8.

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 0 1 0
0	1	Right to Left	0 0 1 0 1 0 1 0 0
1	0	Left to Right	1 0 0 0 1 0 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 has the light "move" from right to left or left to right respectively.

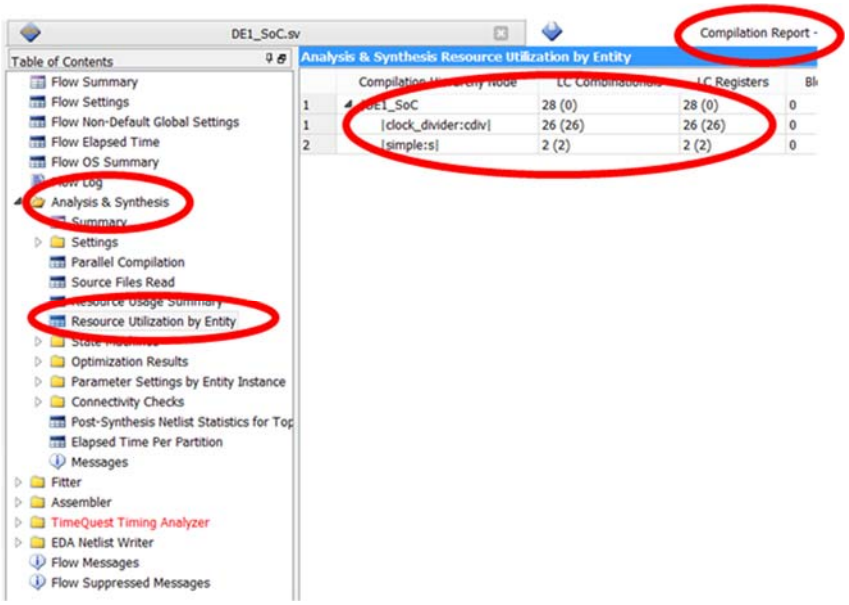
The switches will never both be true. The switches may be changed at any point during the current cycling of the lights, and the lights must switch over to the new pattern as soon as possible (however, it can enter into any point in the other pattern's behaviors).

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 practice 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 lefthand 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).



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…

Lab Demonstration/Turn-In Requirements

Choose and complete a **Demo** option, then prepare your **submission**.

Demo

Choose and complete **one** of these two options to demo your lab.

Option 1: Demo video

Upload a single video to Google Drive clearly showing you performing these demo tasks in order. You must still follow the **Demo Video and Google Drive Access guidelines** from lab #1.

1. Say your name
2. Program your runway lights circuit to the board.
3. 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.

Option 2: Live Demo

During any TA's (not professor's) office hours before the lab deadline, request to perform a live demo of your lab. We will pull you into a breakout room in Zoom for you to demo your lab live. You do not need to make a demo video if you choose this option.

Important: for fairness, if you choose and begin the live demo option, you only get **one attempt regardless of how early you demo**. You may not attempt a live demo more than once, nor switch to the demo video option after completing a live demo.

Submission

File 1/2: Paper supplement

Upload a single PDF document with your responses to the following:

1. Depending on your demo option:
   - Demo video: provide the link to your demo video (hyperlinks preferred)
   - Live demo: write 'Live demoed to (Brian or Nick or Pengyu) on (date)'.
2. A drawing of your state diagram used to implement the runway lights
3. Screenshot of ModelSim simulation of your top-level entity (DE1_SoC) demonstrating runways lights circuit behavior for all 3 possible wind conditions
4. Screenshot of "Resource Utilization by Entity," along with the computed size of your design.
5. Approximately how much time did you spend on this lab (including reading, planning, design, coding, debugging etc.)?

File 2/2: Project & Code Files

Zip your entire project directory and upload it as a .zip file.