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.

Assigned Task – Mapping sequential logic to the FPGA

To understand the sequential logic design, simulation, and execution process we'll use a simple 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. The Verilog for the machine is given here:

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

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

   // Next State logic
   always_comb begin
      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;
      endcase
   end

   // Output logic - could also be another always, or part of above block.
   assign out = (ps == C);

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

endmodule
```

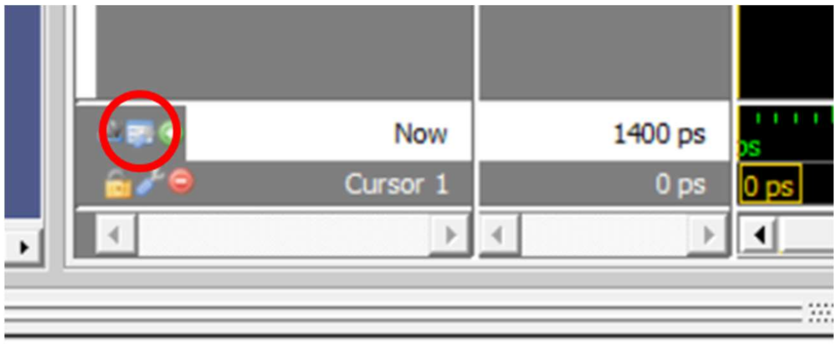
To simulate this logic, we not only have to provide the inputs, but must also specify the clock. For that, we can embed some simple logic in the testbench. Here is the testbench for this FSM:

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

   simple 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
```

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. Note that embedding @(posedge clk) inside initial blocks is fine for testbenches, but should not be part of code intended to be actually mapped into the FPGA.

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

ModelSim Tip: Since clocks are so important to FSMs, it can be useful to set the grey gridlines in the wave display to line up with the positive edge of the clocks. Look in the lower-left corner of the waveform window, and click the "Edit Grid & Timeline Properties…" icon:



If you set the "Grid Period" to 100ps, which is the clock period, the grid lines will now line up with the clock. Make sure to save the formatting to the simple_wave.do file.

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 chip are VERY fast (50MHz, so a clock every 20ns!). For our purposes we'd like 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 that includes the clock divider, and hooks the simple FSM properly to run on the board. In the code below we select which clock via the "whichClock". whichClock = 25 yields a clock with a cycle time of a bit over a second, 24 is twice as fast, 26 is twice as slow, etc.

```
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 = 25;
   clock_divider cdiv (.clock(CLOCK_50),
                       .reset(reset),
                       .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];

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

   // Show signals on LEDRs 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, ...
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
```

Add this to your project, compile the design for the FPGA, and test it on the DE1 board. This design uses KEY0 for reset (press to reset the circuit), 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 to the TA, but running this test will help a LOT in getting the design problem working.

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:

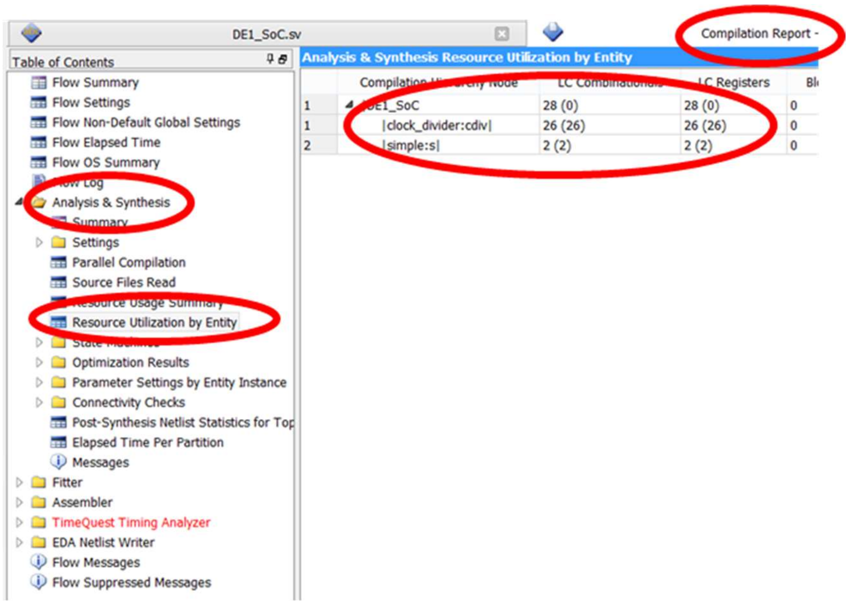
SW[1]	SW[0]	Meaning	Pattern (LEDR[2:0])
0	0	Calm	1 0 1
			0 1 0
0	1	Right to Left	0 0 1
			0 1 0
			1 0 0
1	0	Left to Right	1 0 0
			0 1 0
			0 0 1

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.

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).


DE1_SoC.sv

Compilation Report -

Table of Contents

- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
  - Summary
  - Settings
  - Parallel Compilation
  - Source Files Read
  - Resource Usage Summary
  - Resource Utilization by Entity
  - State Machines
  - Optimization Results
  - Parameter Settings by Entity Instance
  - Connectivity Checks
  - Post-Synthesis Netlist Statistics for Top
  - Elapsed Time Per Partition
  - Messages
- Fitter
- Assembler
- TimeQuest Timing Analyzer
- EDA Netlist Writer
- Flow Messages
- Flow Suppressed Messages

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…

Lab Demonstration/Turn-In Requirements

A TA needs to "Check You Off" for each of the tasks listed below.

- Demonstrate your runway lights circuit working in simulation.
- Demonstrate your runway lights circuit working on the DE1 board.
- Turn in a printout of the Verilog for your runway lights design, including the testbench.
- Turn in a printout of the “Resource Utilization by Entity” page. Write on this the computed size for your design.
- Tell the TA how many hours (estimated) it took to complete this lab, including reading, planning, design, coding, debugging, testing, etc. Everything related to the lab (in total).