

Intro to Digital Logic, Lab 3

Sequential Logic

Lab Objectives

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

Tutorial Task – Introduction to sequential logic

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 this machine is given here:

Note: There are two ways to assign variables in SystemVerilog, and it is important you understand which to use and when. When you are assigning a variable in an "always_comb" block, or after an "assign" statement, you should ALWAYS use "=" to assign that variable a value.

If you are assigning a variable in an "always_ff@(posedge clk)" block, use "<=" to assign the variable a value. These rules will hold true for all your labs.

```

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
    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, which creates a constantly-changing clock signal, clk. Here is the testbench for the above 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

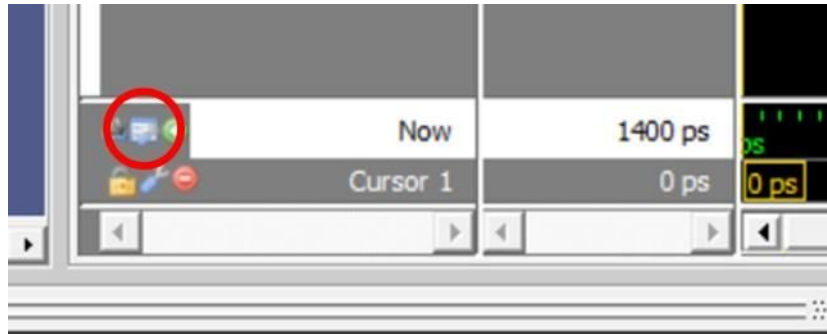
```

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.

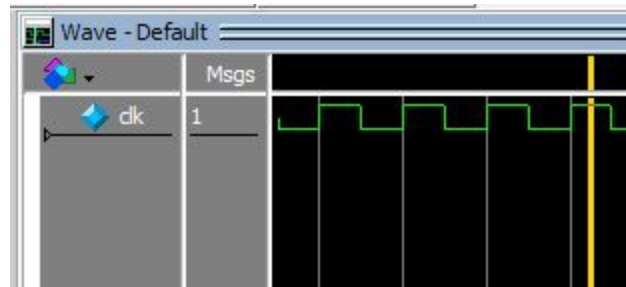
Note: embedding @(posedge clk) inside initial blocks is fine for testbenches, but should not be part of code intended to be actually mapped into the FPGA.

Simulate the design with ModelSim, and make sure that 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, and set “Grid Offset” to 50ps, the grid lines will now line up with the positive edge of the clock:



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 creates an instance of the simple module to run on the board.

Note: In the code below we select which clock we’re using via the “whichClock” parameter. whichClock = 25 yields a clock with a period of a bit over 1 second, while whichClock=24 is twice as fast, whichClock=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_50, 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, 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

```

```
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) and KEY1 as the “w” input, while 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 understanding sequential logic and getting the design problem to work.

Graded Task – Hazard lights

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 indicate 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).

Create a module that will control the runway lights. Your design should be in the style of the “simple” module given above. That is, you should use “if” and “case” statements to implement the next-state logic and the outputs, and an “always_ff @(posedge clk)” block to

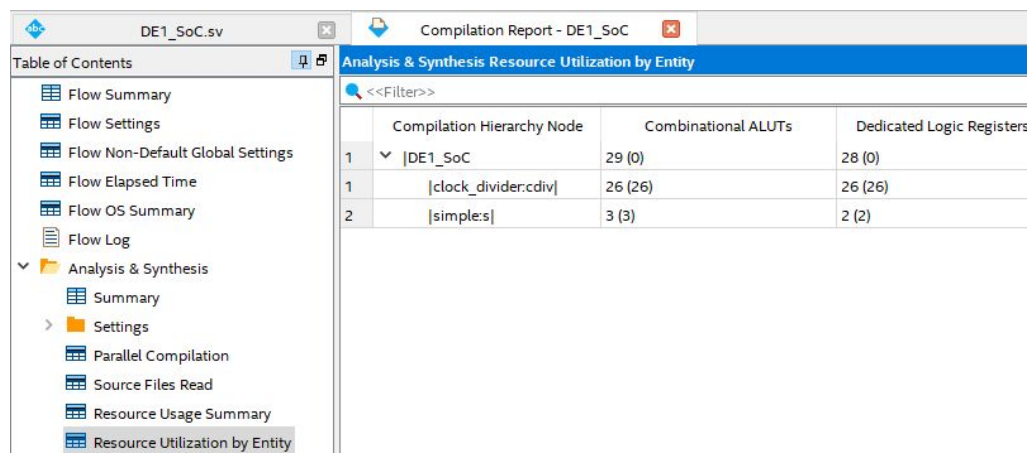
handle state transitions. Simulate your design to verify correctness. Then, connect it to the switches and LEDs on your DE1 board and make sure that it works.

Note: you won't be able to perceive LEDs blinking 50,000,000 times per second, so you will need a slower clock. The clock divider from the first task will work well for the runway lights. However, simulations won't work well with the clock divider in use, because the first positive edge of the divided clock won't happen until after ~34 million clock cycles! As such, you should simulate your design separately from the top-level module.

If you would like to simulate your top-level module, you need to temporarily remove the clock divider, and instead connect CLOCK_50 as the clock input to your submodules.

Hardware Allocation

Compute the size of your design in terms of #FPGA logic and DFF resources 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 “Combinational ALUTs” 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 “Dedicated Logic 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).



The screenshot shows the Quartus II software interface. The top window is titled 'DE1_SoC.sv' and the bottom window is titled 'Compilation Report - DE1_SoC'. The left sidebar shows a 'Table of Contents' with 'Analysis & Synthesis' expanded, and 'Resource Utilization by Entity' selected. The main window displays a table titled 'Analysis & Synthesis Resource Utilization by Entity'.

| | Compilation Hierarchy Node | Combinational ALUTs | Dedicated Logic Registers |
|---|----------------------------|---------------------|---------------------------|
| 1 | DE1_SoC | 29 (0) | 28 (0) |
| 1 | clock_divider:cdiv1 | 26 (26) | 26 (26) |
| 2 | simple:s1 | 3 (3) | 2 (2) |

To compute the size of your FSM, 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 original “simple” FSM from the first part of this lab has a score of $(29+28)-(26+26) = 5$, though obviously it doesn't perform the right functions for the runway lights...

Lab Demonstration/Turn-In Requirements

- Demonstrate your runway lights (both in simulation and on the DE1 board) to the TA.
- Submit a short lab report that includes the following three sections:

Procedure

- Describe how you approached the problem and any relevant truth tables, circuit schematics, or state machine diagrams.

Results

- Include screenshots of your ModelSim simulations.
- Describe what you tested in your simulations, and what the results show.
- Include a screenshot of your “Resource Utilization by Entity” page.
- Include the computed size of your design.
- Give a brief overview of the finished project, referencing what was asked of you.

Problems Faced and Feedback

- Describe any issues you had while completing the lab, and how/if you were able to overcome them.
 - Include tips and tricks that you found out about that may be useful to you in future labs.
- Submit the SystemVerilog files (files with extension .sv) for your runway lights. Make sure to follow the commenting guide provided.
 - Submit your report and files to Canvas. No hard copies.