EE108 Lab 3

Bike Light FSM

Lab due: Wednesday 2/6/19 at noon

Objective: The purpose of lab 3 is to introduce you to the construction of finite state machines and sequential logic design in Verilog.

1. Introduction

A Warning

This lab is significantly more involved than any of the previous labs. Do NOT put off starting on this until the weekend!!

Also, make sure you fully read over the Appendix, or at least the flip flop and "back to lab 3…" sections. They contain important information that you'll need to keep in mind for this and all future labs! Taking 5 minutes to read them now will save you hours later on.

So, what are we doing here?

We're going to build a programmable 'bike light' out of finite state machines, timers, and shifters. The bike light consists of a flashing LED that goes through 6 "master" states when the right push-button is pressed on the FPGA board:

1. Off (resets to here)
2. On
3. Off
4. Flash 1
5. Off
6. Flash 2

After the 6^{th} state, the bike light returns to the 1^{st} state.

When the bike light is in the 4^{th} and 6^{th} states, the LED blinks and the user can press the up and down buttons to make the rate twice as fast or twice as slow, respectively. Flash 1 should be able to blink with a half-period of 1s, 2s, 4s, or 8s (in the last case, for example, that would be 8 seconds on and 8 seconds off). Flash 2 should be able to blink with a half-period of 1s, 1/2s, 1/4s or 1/8s. Both states should start at 1s.

These rates are stored independently. For example, you should be able to set Flash 1 to blink at a period of 4s and Flash 2 to blink at a period of 1/2s, and cycling through the states will not reset

those blink rates. The rate control should be independent: pushing the up/down buttons in Flash 1 should only change the flash rate in state Flash 1, and pushing the up/down buttons in Flash 2 should only change the flash rate in state Flash 2.

Project architecture

The top-level module has been provided for you for this lab. Its inputs are up_button, down_button, right_button, left_button (used for reset), and the clock, and its lone output is the rear_light bit which lights an LED when set to 1.

The project is broken down into the following modules:

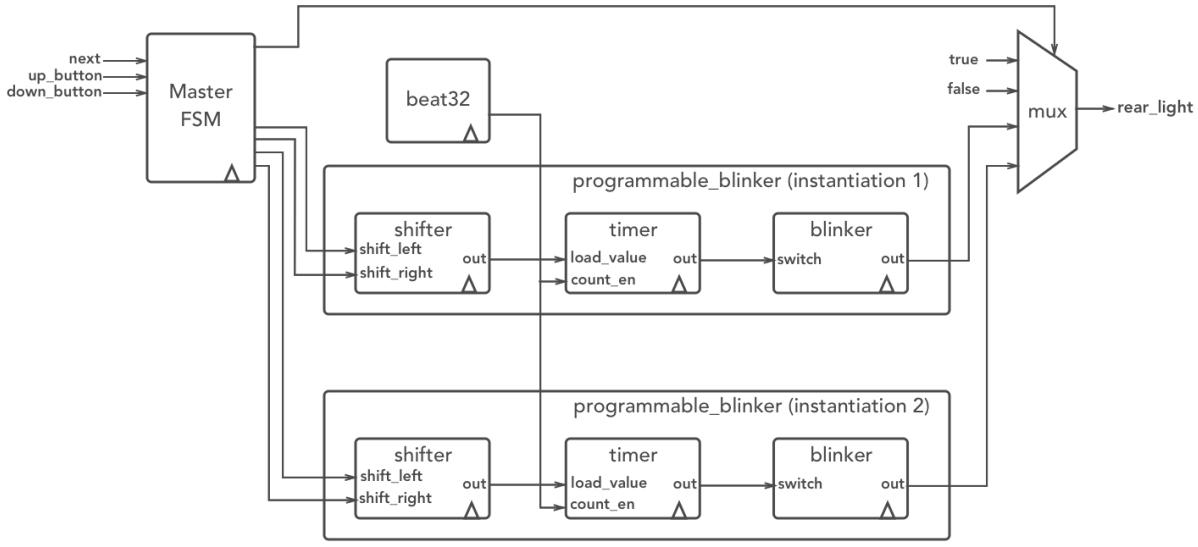
button_press_unit

The button_press_unit module sanitizes our button inputs and is described in section 5 of this document. It has been completely implemented and is provided with the starter code.

bicycle_fsm

The bicycle_fsm module is the brains of our bike light and is implemented using several smaller, easy-to-debug modules outlined in the block diagram below. These are the modules you're going to have to implement:

Note that clock and reset inputs aren't drawn in this diagram, but will be present in your Verilog wherever flipflop modules are instantiated.



Master FSM Module
The Master FSM is the control unit that will go through the master states listed earlier. It drives the select signal of a multiplexer which, in turn, drives the actual LED. It is also responsible for sending signals to the blinker modules which adjust their speed.

Beat32 Module
The Beat32 module should contain a counter that counts up to 100,000,000/32 = 1/32nd of a second, and then resets to zero and starts over again. Each time it resets to zero its single output bit should go high for one cycle.

We need to count this high because our FPGA runs on a 100MHz clock, so blinking an LED once a second requires us to wait 100,000,000 clock cycles between each blink. Instead of implementing many huge counters throughout our design, which is expensive in chip area and power, we use a single module which provides a 'heartbeat' every 1/32 of a second.

Think carefully about what this means for your testbenches, though. The beat32 module will wait 100,000,000/32 cycles before it does anything. When simulating you need to reduce the period of beat32 significantly so your testbenches don't take forever! Remember to set it back when synthesizing.

Programmable Blinker Module
The programmable_blinker module should oscillate its output bit at a rate which can be changed with its **shift_left** and **shift_right** inputs. Since the rates in the Flash 1 and Flash 2 states are independent we need two instances of this module, one for each flashing state.

We control the blink rate by using the **timer** module, similar to the one in beat32 but significantly smaller. The timer counts down and when it reaches zero pulses its output bit high for one cycle. The timer then starts counting down again from the value fed to it from the **shifter**. The shifter stores the count-down time for the next blink, and can shift it left or right to double or halve it. The output bit of the timer feeds into the **blinker**, whose output bit inverts whenever its input goes high. Each of these modules stores some sort of state and will need a flipflop inside of them.

Timer module
Store a counter value loaded in from the **load_value** input and decrease it by 1 every time the **count_en** input (fed in from beat32) goes high. When the counter reaches 0, pulse the output for a cycle and reload the counter value from **load_value**.

You should do this by instantiating a flipflop with an enable port. Connect its clock input to the clock signal and its enable port to **count_en**. See section 4 of this handout for how to instantiate flipflops in this class. Remember, **we never put anything other than the top-level clock signal into the clock input.**

You are responsible for sizing this counter to support durations from 1/8th of a second to 8

seconds, based on the fact it decrements once every 1/32nd of a second. Figure the number of bits necessary ahead of time.

Shifter Module
Store the blink rate as a 4-bit one-hot value, and shift it right or left each time the shift_right or shift_left input goes high. Don't let this value shift beyond 4'b1000 or 4'b0001.

Output this value with enough 0s appended to it to represent a time between either 1/8th of a second and 1 second for the fast timer or 1 second and 8 seconds for the slow timer. The number of 0's appended should be parameterizable to allow for the same module to be used in both the fast and slow timers.

2. Lab

What you need to do

1. Look through the code, particularly the comments—we've provided much of the infrastructure for you already. The files are located in the class's labstarterfiles/lab3 directory.
2. Draw state diagrams for the master FSM, the shifter, and the blinker.
3. Draw block diagrams for the timer and the beat32.
4. Determine the inputs and outputs for each module. Anything with a clock input should also have a reset input.
5. Implement the master FSM, shifter, blinker, timer, programmable_blinker, and beat_32 modules. Use case statements for the FSMs and remember to include default cases.
6. Write test benches for each module and simulate them.
7. Hook everything up inside the bicycle_fsm module, already instantiated in the top module, and simulate it.
8. Edit the top module to enable the instantiations of the button_press_unit in the top module for synthesis (or try the design in hardware without them to discover first-hand why they are necessary).
9. Generate a .bit file for the design and confirm it works so that you can be in-and-out of lab as fast as possible!

Lab Deliverables

Put the following files in a zip or tar archive and put it in the Canvas Assignment:

1. **Verilog files (.v files) of every module and testbench that you wrote or modified.**
2. The **lab3_top.syr** and **lab3_top.twr** files in your project directory that are generated after synthesis.
3. **lab3_synthesis.pdf** which contains the following:
    a. The **critical path** of your design – this is the longest chain of logic in the entire design from the output of one flipflop to the input of another.

You can find it by selecting “Design Summary/Reports → Detailed Reports → Post-PAR Static Timing Report”. This report lists the critical paths in your design sorted with the longest at the top. **Report the slack, source, and destination**. Look at the list of locations the signal visits under 'Maximum Data Path'. Do you recognize from where in your logic this path comes from?

   b. Those resource usage statistics we collected in labs 1 and 2. Double click “Design Summary/Reports” and select “Summary” at the top of the reports list.
      - What fraction of the FPGA's slice LUTs did your implementation consume?
      - How many are used for logic and how many are used for route-through?
      - How many slices are occupied?
      - How many slice **registers** did your implementation consume? These registers are configured as flip-flops, which you instantiated using the dff module.

The cool part here is that for the first time in this class, there is no starter code overhead adding to these numbers. Everything you see is a result of your Verilog logic.

   c. Try to open FloorPlan (Tools → PlanAhead → Analyze Timing / Floorplan Design). This time, all of those highlighted slices should be from logic you recognize. Track down the master FSM's state flip-flops in the netlist on the left, right-click one of them, and select 'Mark'. Then zoom into it on the diagram. Take a screenshot and submit it with everything else. If FloorPlan is still refusing to open skip this step and let your TA know.

4. **lab3_discussion.pdf** which contains the answers to these discussion questions:
   a. If you have a de-bounced button (that is, one that only goes high when you push it and immediately goes back low when you let go) as an input to a 100MHz FSM, you would have to hold down the button for about 10ns to have it only advance one state in the FSM. This is obviously impractical. Draw a bubble diagram for a “one-pulse” FSM that outputs a 1-cycle pulse every time the button is pressed, regardless of how long it is held down. Keep in mind there is **no other state or memory in the universe** other than which bubble the FSM is currently on. If you are confused about what is a pure FSM and what is not, ask your TA for help.
   b. Assume some stray radiation hits the state bits of a **one-hot encoded** FSM and a one of those bits is flipped such that either two state bits are high or zero state bits are high. What will happen to the design if it was written as a Verilog case statement with no default entry? What if there was a default entry?
5. **lab3_schematic.pdf** which contains:
   a. FSM state diagrams for the master FSM, the shifter, and the blinker. You can use any program you have available to create the state diagrams; hand drawn and scanned is acceptable also, but make sure the image is readable.
   b. Block diagrams of the timer and beat32 modules.
6. **lab3_sims.pdf** . This should include the following:
   a. Simulation results showing that your design works. These can be annotated waveforms, $display output, whatever, as long as you clearly describe what the simulation results show and how they indicates the module works. **You need a testbench for each of the modules you wrote.** Make sure you hit all interesting edge cases and prove that the design still works under those conditions.

3. Demo session

Just come to lab and show us it works. As always, we're open to suggestions on how to make this lab better in the future!

4. Appendix: The ee108 flip-flop library

Your starter code includes the file ff_lib.v, which contains the flipflop modules you will be using for the rest of the quarter. Any state you store will be using one of these modules. They can be instantiated like this:

```
// Replace signal_width with the width of our input_signal, in bits
// If signal_width is 1, the whole #() clause can be omitted

dff #(signal_width) instance_name(
      .clk (clock_signal),
      .d (input_signal), .q (flopped_input_signal)
);

dffr #(signal_width) instance_name(
      .clk (clock_signal),
      .r (reset_signal),
      .d (input_signal), .q (flopped_input_signal)
);

dffre #(signal_width) instance_name(
      .clk (clock_signal),
      .r (reset_signal), .en(enable_signal),
      .d (input_signal), .q (flopped_input_signal)
);
```

dff copies its d input to its q input every time the clock signal rises.
dffr adds a reset port, which if 1 will force the output q to 0 instead of d on the next clock cycle.
dffre adds an enable port, which will only allow q to change when the enable_signal is 1.

If you need to store information in your circuit you will **need** to instantiate one of these modules to do it.

Implementation

It's important for you to understand how we actually represent flipflops in Verilog, though you **should not use any of the following techniques in this class**.

The dff module is implemented as:

```
module dff #(parameter WIDTH = 1) (
    input clk,
    input [WIDTH-1:0] d,
    output reg [WIDTH-1:0] q
);
    always @(posedge clk)
```

```
        q <= d;
endmodule
```

There are two things here we haven't seen before: the “posedge clk” in the always block's sensitivity list and the <= operator.

posedge clk forces the always block to only evaluate only when the signal 'clk' goes from 0 to 1. In fact, we are very carefully inferring a latch here to allow q to be defined even when we're not on the positive edge of clk.

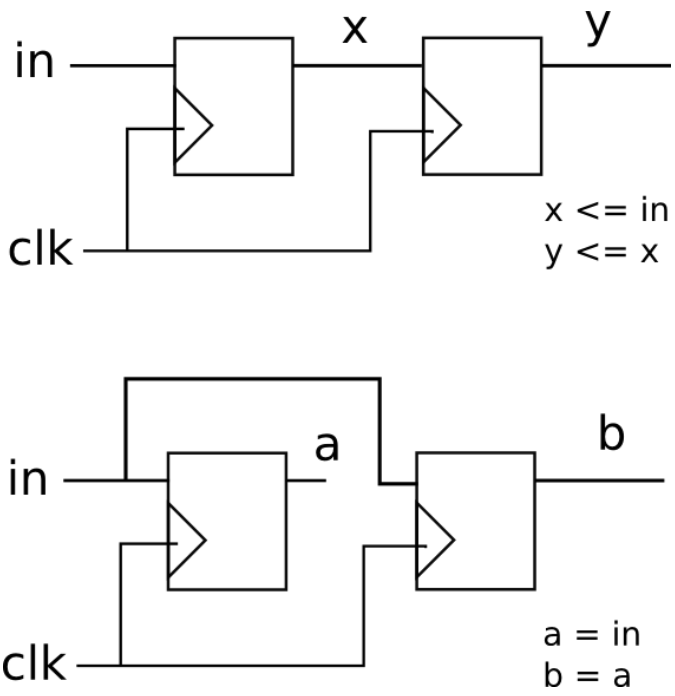
<=, also called **non-blocking assignment**, has a more subtle effect. The importance of non-blocking versus blocking assignment can be seen in the following more complicated example of sequential logic:

```
//DON'T USE CODE LIKE THIS IN ANY OF YOUR LABS

 always @(posedge clk) begin
    x <= in;
    y <= x;
 end

 always @(posedge clk) begin
    a = in;
    b = a;
 end
```

These translate into the following block diagrams:



Example adapted from Sutherland HDL Consulting,
http://www.sutherland-hdl.com/papers/1996-CUG-presentation_nonblocking_assigns.pdf

In words, the `q <= d` statement says that `q` will get the last value of `d`, and will only change to that value when everything else is done being evaluated. That way, the output of the flip flop will only change once even if the input changes multiple times during simulation.

But back to lab 3...

We reviewed these topics to give you a complete understanding of what is actually going on underneath the surface of these labs. Indeed, even in ee180 you will be working directly with Verilog code that uses **posedge clk** blocks and non-blocking assignment. That being said, **in ee108 never use @(posedge clk) or non-blocking assignment**.

Now that we're designing sequential logic it will be easier than ever to accidentally infer a latch, and ensuring that your flipflops are explicitly instantiated instead of inferred in the code will make bugs orders of magnitude easier to debug.

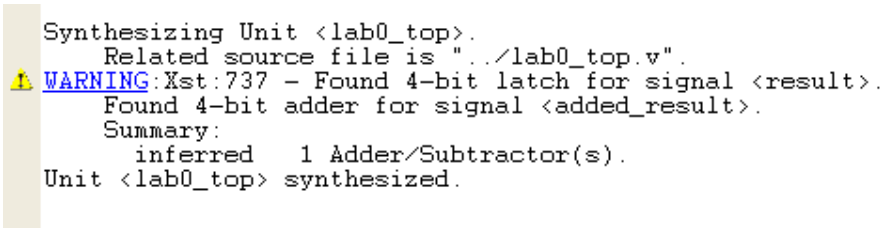
As long as every single output is defined for every single combination of inputs you will never infer state. This should make sense: if all the outputs are defined for all the inputs then the block is purely combinational. The outputs only depend on the current inputs. If an output is not defined for a given set of inputs then Verilog will infer a latch to remember the previous output to use. We will take off points if you do this.

So, your mantra is:

1. Instantiate flipflop modules for any state in the circuit
2. Always have an else for every if

3. Always define the same set of signals in all cases or if clauses
4. Always have a default for every case
5. Always search for warnings in Xilinx ISE about inferred latches.

Going forward it's essential you look at the output from the synthesis step and make sure you don't see any warnings like this:



The yellow exclamation mark means "points off of your lab"

You must make sure you have run your lab through Xilinx ISE before you submit it and that you did not receive any warnings about inferred latches. If we see problems like this in your lab code you will lose points.

5. Appendix: Sanitizing button inputs

The button inputs in our Xilinx XUP board are pure mechanical switches. This leads to several problems:

1. **The buttons are asynchronous.** The signal transitions do not necessarily come on the positive edge of a clock cycle, as the user can press the button at any time. We'll cover exactly why this is bad soon in lecture, but for now it's enough to know that if a signal changes **too close** to the rising edge of the clock, the flipflop can be stuck in a **metastable state** in which it's driving neither a 0 nor a 1 at its output. To combat this problem, we feed the signal through a **synchronizer** module that forces the signal to transition on the positive edge of a clock signal.

2. **The buttons are bouncy.** Even when the button is pushed or released, it can oscillate between 0 and 1 while its mechanical springs reach their equilibrium state. This leads to many unwanted 0→1 and 1→0 transitions when we really want just one 0→1 or 1→0 transition. To combat this problem, we feed the synchronized signal through a **debouncer** to ignore the extraneous transitions. This module works by looking for the first transition, and then ignoring any other transitions until the button has settled for a few milliseconds. **Note that a few milliseconds at 100MHz is a lot of clock cycles, so you don't want to simulate a top-level module that includes the button_press_unit or you'll be waiting for a long time.** See the comments in the top level file for more details.

3. **The buttons need to be one-pulsed**. That is, we need to make sure that whenever the (now synchronized and debounced) input goes high, it generates an output that is high for exactly one cycle. This is important because if your FSM is checking for button presses at 100MHz your user would have to hold the button down for 1/100MHz = 10 nanoseconds or less to make sure the FSM only saw one button press. Pretty unlikely. By one-pulsing the input we guarantee that we get exactly one cycle's worth of input for each button press.

We have provided you with the button_press_unit module which combines the synchronizer, debouncer, and one-pulse units. You should inspect these modules to understand how they work before you turn in your lab.