

# Digital Design: A Systems Approach

## Prelab + Lab 3: *Bike Light FSM*

### Introduction

In lab 3, you are going to get your first taste of sequential logic by building a system of finite state machines, timers, and shifters to create a programmable bike light. To make this lab easy to understand, implement, and test, we've divided up the project into small chunks, but it's up to you to implement and hook up the chunks.

The bike light project consists of a flashing LED that goes through 6 “master” states when the right button is pressed on the FPGA board. These are:

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

After the 6<sup>th</sup> state it returns to the 1<sup>st</sup> state. When the bike light is in the 4<sup>th</sup> and 6<sup>th</sup> states the LED flashes at a programmable rate, and the user can press the up and down buttons to change this rate. The rates are stored independently. For example, you should be able to set Flash 1 to blink at twice a second and Flash 2 to blink at four times a second, and cycling through the states will not erase those blink rates.

When the module is in the Flash 1 or Flash 2 states, pushing the up or down buttons on the FPGA board will change the blinking rate by a factor of two. There are a total of four blinking periods for each flash state. Flash 1 should be able to blink at 1s, 2s, 4s, and 8s intervals. Flash 2 should blink at 1/8s, 1/4s, 1/2s and 1s intervals. Both states should start blinking after power up or reset at once per second (i.e. 1s on, 1s off, 1s on...).

Each time we press the up button in a flash state we should flash twice as fast, and each time we press the down button in a flash state we should flash twice as slowly. As stated above, 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. One final note: it may seem these blink rates are a lot slower than what you would want on an actual bike light, but it makes it easy for your TA to time and see if the rate is correct =)

### **A Warning**

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

## Architectural Overview

### Top-level module

The top-level module has been provided for you for this lab. It takes in the `up_button`, `down_button`, `right_button`, and `left_button` (used for reset), and the clock, and outputs the `rear_light` signal (for your LED).

### Processing Inputs: The `button_press_unit`

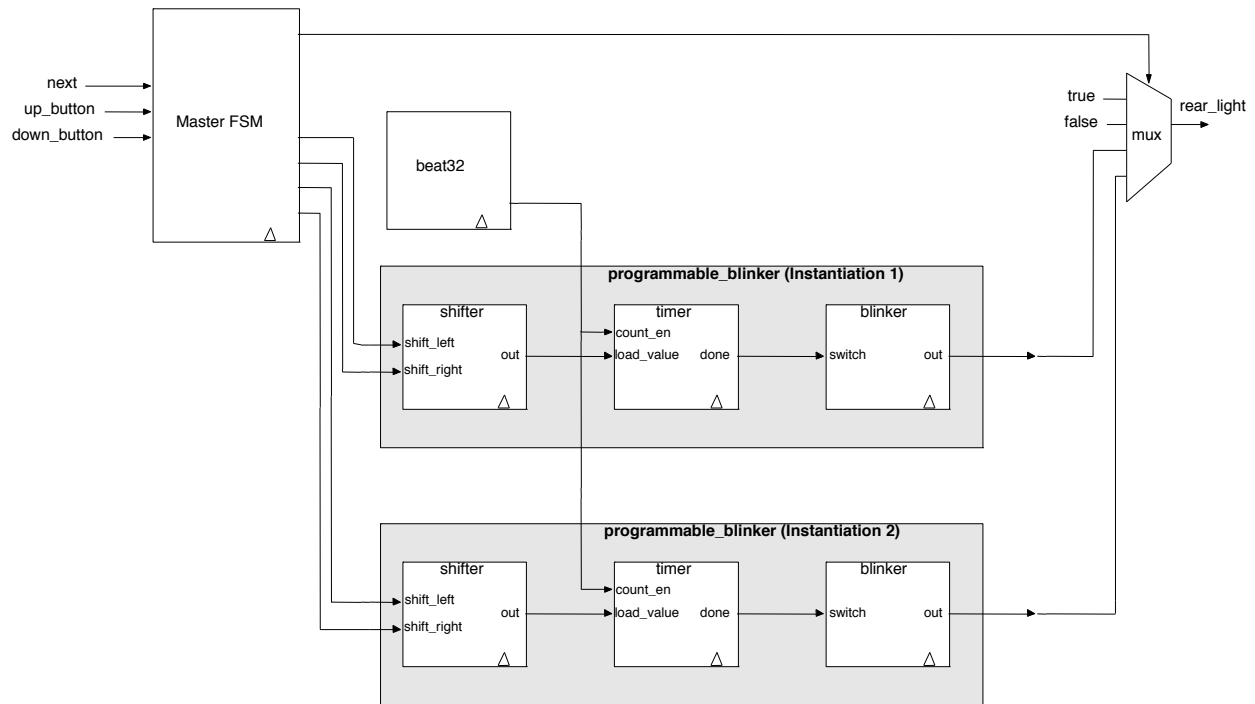
The button inputs in our FPGA are pure mechanical, asynchronous, bouncy switches. This leads to several problems if we use the button inputs directly in our logic:

- 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. While we have not yet covered flip-flop timing, we can briefly explain why this causes timing issues. A DFF can only sample an input during a specified window of time. If the button is pressed outside of the valid sampling window, we can hang a DFF in an unknown state where it is storing neither a 0 nor a 1. To combat this problem, we need to 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 the mechanical springs in the button reach their equilibrium state. This leads to many unwanted 0→1 and 1→0 transitions while the button is oscillating, 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 an awful lot of clock cycles, so you probably 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, 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  $1/100\text{MHz} = 10$  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 the `button_press_unit` that combines the synchronizer, debouncer, and one-pulse units and have instantiated them for you in the top-level module. You should inspect these modules to understand how they work before you turn in your prelab.

## The Bicycle FSM

The `bicycle_fsm` module is responsible for implementing all of the functionality outlined above. However, to simplify things we're going to break it apart into several smaller, easy-to-debug modules. Your job is to understand each of them, how they come together to make the whole thing work, and then to build and debug them. Figure 2 includes all the key signals you'll need in your design, but it omits resets and clocks. Don't forget them or you'll have a hard time simulating your design.



**Figure 1:** *Bicycle FSM module*

### Master FSM Module

The Master FSM is the control unit that actually goes through the states listed in the Overview section.

As you can see in the above diagram, the output from the `bicycle_fsm` comes from a mux which selects between 4 inputs depending on the state of the Master FSM. It is either off, on, or whatever is coming out of the first or second programmable\_blinker. The Master FSM is also responsible for sending signals to the blinker modules to adjust their speed as appropriate. (Remember that you can only change them when you are in the right state.)

### Beat32 Module

Our FPGA runs on a 100MHz clock. This means that if you want to blink an LED once a second you need to wait for 100 million clock cycles between each blink. Since it's a pain to deal with counting so high all the time you are going to implement a `beat_32` module which generates a one cycle enable pulse every  $1/32$  of a second. This way we only need to have one big counter. To build this module you simply need a counter that counts up to  $100,000,000/32 = 1/32^{\text{nd}}$  of a second, and then resets to zero and starts over again. Each time it resets to zero it should output true for one cycle.

You should think carefully about what this means for your testbenches. The `beat32` module will take 100million/32 cycles before it does anything. You will want to change this when you are simulating so it counts up to a much lower number so your simulations don't take forever! (But remember to put it back for your final version.)

### Programmable Blinker Module

The Programmable Blinker module is itself made up of three modules. Its job is to take in a signal that tells it to go faster or slower and then generate the appropriate blinking output. To do this it uses the beat signal from the `beat32` so it doesn't have to count up so high. Since the flashing rates in the Flash 1 and Flash 2 states are independent, we need two instances of the Programmable Blinker module, one for each flashing state.

The programmable blinker module consists of a shifter which outputs a value to a timer which then uses that value to count down to zero. When the timer expires it tells the blinker module to "blink", that is, if it is on, switch to off, and vice versa. The timer then re-loads the value from the shifter and starts over. This way if the shifter's value changes the timer will count by a different amount on the next blink. Make sure you understand this flow before you dig into the details below.

**1.Shifter module:** The shifter is responsible for keeping track of how long the blinker module should blink. It does this by storing a 4-bit one-hot value, and shifting it right or left each time the `shift_right` or `shift_left` input goes high. Once the value gets down to `4'b0001` or up to `4'b1000` it should remain there even if `shift_right` or `shift_left` is asserted again, respectively.

This works very nicely for our design because when the shifter gets a "shift\_left" signal it will shift its output one to the left (multiply by two) so the timer will now get an input that is twice as big. Similarly when the shifter shifts to the right it will divide its output by two and the timer will

have half as big a value.

**2. Timer module.** The timer module is very similar to the ones you've seen in class. It simply counts down by one and when it reaches zero it emits a one-cycle pulse on the output, and re-loads itself from the `load_value` input. However, to make sure our counter doesn't count at 100MHz you will add a `count_en` (count enable) input which prevents the counter from counting except when the enable is high. By doing so we can hook up the `beat_32` output to the counter and end up with a counter that counts down one count every 32<sup>nd</sup> of a second instead of 100 million times a second. (Remember that this is an enable and not the clock. You should never put anything other than the one system clock into the clock input.)

In other words, the main state DFF for your timer should be the `dffre` module found in `ff_lib.v`. This is a DFF with synchronous reset and enable inputs: the DFF resets to zero if reset goes high, and it will not copy D to Q unless the enable signal is high. So we should pass the output of the `beat32` module into the enable signal of the DFF, so that we only enable the DFF to count once every 1/32 of a second.

You are responsible for sizing the timer to be able to capture speeds from 1/8<sup>th</sup> of a second to 8 seconds using the 1/32 second enable signal. Work out how high the counter needs to be able to count before you start building it. Make sure your counter is counting down by 1 and not by the `load_value`. If you count down by the `load_value` you'll have problems with wrapping around.

**3. Blinker module.** The blinker is almost too simple to bother describing. It simply switches from "on" to "off" each time it gets an input. However, this module does have state, so you'd better make sure you're instantiating a flip flop, and you are required to draw out the state diagram for it.

## Notes About Inferred state

We've warned you several times about not inferring latches but this is the first time it will actually be a problem, so here's an overview of what to do and what to not do. Remember, *whenever you need state storage (counters, FSMs, etc.) you must explicitly instantiate a flip flop from the provided `ff_lib.v` file.* This file explicitly infers state to generate a flip flop. To understand what you need to do to avoid inferring state, let's take a look inside the flip flop library:

## The Flip Flop Module:

```

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

```

You should note two things here that you probably haven't seen before in Verilog. The first is the `@(posedge clk)` and the second is the `<=` operator. You are not allowed to use either of these in this lab. The reason we don't teach them is that it is important that you learn to distinguish the combinational and sequential portions of your design so you understand what the tools are building from your code. If you write Verilog using these constructs it is a lot harder to understand what you're building.

So what is that always block doing? Well, because it is not using `always @*` it will only evaluate when `@(posedge clk)` is true. This means that on every rising edge of the clock (and **only** on the rising edges of `clk`) this block will be evaluated. 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.)

Now the real question is what is the value of `q` when we're not on the rising edge of the clock? What happens is that Verilog infers a latch for `q`, and remembers its value whenever the clock is not a rising edge. While this is exactly the behavior we want for a flip flop, we don't want that to happen anywhere else in our designs.

### DFFs with Reset and Enable

The `ff_lib.v` module also has two other flipflops you can use: `dffr` and `dffre`. The "r" here stands for reset, with each of these flipflops having a reset input which will set them to zero when asserted. If you want your logic to reset to zero then you don't have to do anything other than use a `dffr` and put the reset signal into all your flipflops. If you need it to reset to something else, say `0d1000`, you will need to explicitly build logic to set the flipflop's input when reset is asserted. The `dffre` also has an enable signal. When enable is 1, the flipflop will latch in the input value on the rising edge of the clock. When enable is 0, it won't. You should use the `dffre` in your timer module to enable counting only when the `beat32` output goes high.

### Mistakes that infer state

Now that we've seen how to build a flip flop on purpose, let's see how we can build them by accident.

Verilog will only infer state in your design if you don't build combinational logic in always

blocks. That is, **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 in that case. You are not allowed to do this and we will take off points if you do.

Let's look at an example:

```
reg result, status;
always @* begin
    if (input_button) begin
        status = 1'b1;
        result = 1'b0;
    end
    else if (done_signal) begin
        status = 1'b0; // ERROR! Latch inferred for "result"
    end
    // ERROR! No else statement to cover case where
input_button
    // is low or done_signal is low. Latch inferred.
end
```

This code does not define every output for every combination of inputs. There are two different problems here. The first is that we don't define the value of the result output for the case where done\_signal is true. The second is that we don't define the value of either status or result in the case where both done\_signal and input\_button are false. As a result this will infer latches for both result and status.

The correct code is as follows, with changes underlined:

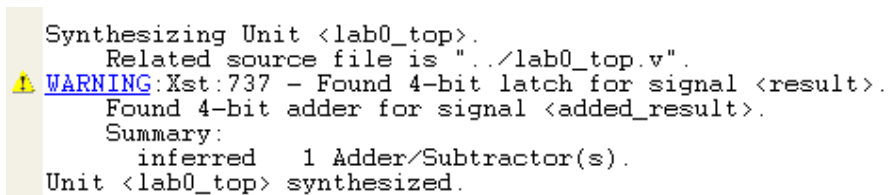
```
reg result, status;
always @* begin
    if (input_button) begin
        status = 1'b1;
        result = 1'b0;
    end
    else if (done_signal) begin
        status = 1'b0;
        result = 1'b1;
    end
    else begin
        status = 1'b0;
        result = 1'b0;
    end
end
```

This version is purely combinational because it defines all outputs for all combinations of inputs. The same thing can happen with a case statement where you don't include a default.

The basic rules to avoid this problem are:

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

You will only see warnings about this in Xilinx because the simulator assumes that you just wanted to build a latch and so it just gives you one. When you run Xilinx you need to look at the output from the synthesis step and make sure you don't see any warnings except where you have instantiated a flip flop from the ff\_lib.v module. For example, the following warning is not allowed in this class (Figure ):

A screenshot of the Xilinx ISE synthesis output window. The text shows the synthesis of a unit named 'lab0\_top'. It indicates the source file is '../lab0\_top.v'. A yellow warning icon is shown next to the text 'WARNING: Xst:737 - Found 4-bit latch for signal <result>.'. Below this, it says 'Found 4-bit adder for signal <added\_result>.'. A summary section follows, stating 'inferred 1 Adder/Subtractor(s)'. The final line says 'Unit <lab0\_top> synthesized.'.

```
Synthesizing Unit <lab0_top>.  
  Related source file is "../lab0_top.v".  
  WARNING: Xst:737 - Found 4-bit latch for signal <result>.  
  Found 4-bit adder for signal <added_result>.  
  Summary:  
    inferred 1 Adder/Subtractor(s).  
  Unit <lab0_top> synthesized.
```

**Figure 2:** The yellow exclamation mark means “points off of your prelab”

You must make sure you have run your prelab through Xilinx ISE before you submit it and that you did not receive any warnings about inferred latches for anything other than flip flops. If we see problems like this in your prelab code (and we're pretty good at recognizing them) you will lose points.



## What 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 lab3/included 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. Remember that anything that has a clock input should also have a reset input.
5. Implement the master FSM, shifter, blinker, timer, 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. (The modules are small, so this will be easy.)
7. Implement the Programmable Blinker module and write a test bench for it.
8. Hook everything up inside the bicycle\_fsm module (bicycle\_fsm is already instantiated in the top module) and simulate it.
9. Edit the top module to enable the instantiations of the button\_press\_unit in the top module for synthesis (or try the design without them to discover first-hand why they are necessary).
10. Generate a .bit file for the design so that you can be out of lab that much sooner.

## Submission Information

### Prelab

Please turn in the following files, with the submission requirements found on Blackboard, to Blackboard:

1. **lab3\_readme.txt.** This file includes the answers to the following prelab questions:
  - a. If you have a de-bounced button (that is one that only goes on when you push it and immediately turns off 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 rather impractical. Describe the states of a “one-pulse” FSM that outputs a 1-cycle pulse every time the button is pressed, regardless of how long it is held down.
  - b. Assume there is an error in a one-hot encoded FSM and a bit is flipped such that either two bits are high or zero bits are high. What will happen to the design if it was written as a Verilog case statement with no default entry? (Hint: the other states are now handled like don't cares in Kmaps.) What if there was a default entry?
2. **lab3\_schematic.pdf.** This file 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 + digital photo (or scan) is acceptable also, but make sure the image is readable.
  - b. Block diagrams of the timer and beat32 modules.
3. **lab3\_wave.pdf.** This should include the following:
  - a. Simulation results showing that your design works. These can be annotated waveforms,

\$display output, whatever; but you must clearly describe what the simulation results show and how that indicates it works. You need a simulation (and therefore 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.

#### 4.prelab3\_synth.pdf

a.Extract timing information for the *whole* design from the timing analysis report. This tool is hidden under Implement Design → Place & Route → Generate Post-Place & Route Static Timing → Analyze Post-Place & Route Static Timing (Timing Analyzer). When you run it click the button for “Analyze Against Auto-Generated Constraints” and accept the defaults. The report consists of paths sorted from slowest (largest delay) to fastest (shortest delay). Each path starts with “Delay:” and then describes the source and destination, and lists the delay of each net (or wire) in-between. At the end it reports the total delay broken up into % logic and % routing. Report the fastest speed the design will run and the critical path. (The critical path is the one that keeps the design from going faster, which is the path at the top of the report.) Does this path make any sense to you?

b.Find the resource utilization for the *whole* design. (Click on the “Design Summary” tab in the right-hand pane in ISE.) How many Slice Flip Flops are used? How many 4 input LUTs and how many occupied Slices are there?

c.Repeat (a) and (b) above for *just the bicycle\_fsm portion of the design*. You can do this by selecting the “bicycle\_fsm” module in Xilinx ISE and re-running the Timing Analyzer using that as the top-level. Comment on the differences (why do you see any?).

d.Look at your *whole* design in the Xilinx FPGA Editor and include a screenshot with the master FSM’s state flip-flops highlighted. One way is to select the flip flops on the left side, and then zoom in so they and their connections are visible using the ‘zoom to selection’ tool.

5.Verilog files (.v files) of any module/testbench that you wrote.

### Lab Day

Just come to lab and show us it works. Make sure to bring suggestions on how to make this lab better in the future!