EECS 151/251A FPGA Lab
Lab 4: Rotary Encoder and Debouncer, Finite State Machines, Synchronous Resets, Synchronous RAM, Testbench Techniques

Prof. Borivoje Nikolic
TA: Vighnesh Iyer
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Contents

1 Before You Start This Lab

Before you proceed with the contents of this lab, we suggest that you look through three documents that will help you better understand some concepts we will be covering.

1. **labs_fa16/docs/Verilog/verilog_fsm.pdf**

   Goes over concepts of FSM in Verilog. Provides an example of implementing FSM's in Verilog and pitfalls to watch out for.

2. `http://www.labbookpages.co.uk/electronics/debounce.html`

   Read the "What is Switch Bounce" section to get idea of why we need a debouncer circuit. Read the "Digital Switch Debouncing" section to get a general overview of the circuit, its parts, and their purposes. You may want to pay attention to the purpose of the synchronizer as meta-stability is something you will go over in class.

3. `http://www.xilinx.com/products/boards/s3estarter/files/s3esk_rotary_encoder_interface.pdf`

   Read slide 5 (Rotary Encoder and Signals) to get an idea of how the encoder works and the signal it generates. You can read the next few pages to get a better idea of how to use the signals. You will be implementing the circuit described in these slides in this lab.

In the first couple sections of this lab, we will be revisiting the circuit you did in lab 2 and confirming their functionality with music generated by the updated Python scripts. Then we will be going over some new circuits that you will be using in future assignments.

1.1 Helpful Hint: Synthesis Warnings and Errors

At various times in this lab, things will just not work on the FPGA or in simulation. To help with debugging, you can run `make synth` in the `lab3/` folder. This will just run `xst` which will only take a few seconds. Then you should run `make report`. In the window that opened, click on `Synthesis Messages` on the left under `Errors and Warnings`. Any synthesis warnings you see here are a possible alert to some issue in your circuit. If you don't understand a warning, ask a TA; it almost always reveals some issue in your RTL.

You should keep the depth and width of the ROM consistent throughout the lab. We recommend generating a ROM that's 24 bits wide and 4096 addresses deep (12-bit addresses).

2 Lab Overview

The first part of this lab is a repetition of Lab 3, however, there are a couple differences in this new version of the debouncer and a new edge detector circuit. Go through this lab from top to bottom even if you have gone through Lab 3 already.

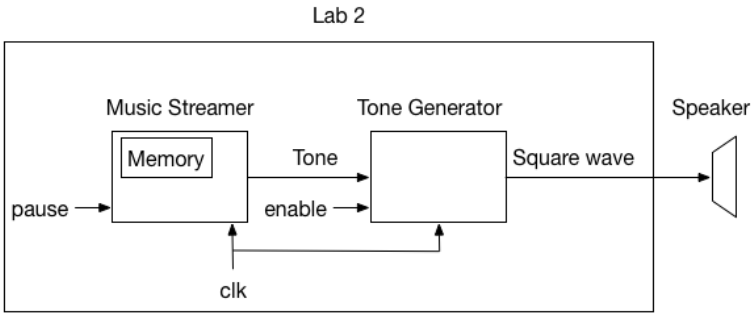
In this lab, we will begin by taking your rotary encoder circuit and adding a debouncer in front it. With the implementation from last lab, some rotary encoders would interpret more rotations than expected. We will add a debouncer to get reduce glitched or buggy signals

For the first part of the lab, we will fix an issue most rotary encoders had using a debouncer circuit and elaborate on the changes you have to make to the debouncer circuit between the buttons and rotary encoder. We will give you some testbenches for these circuits and introduce you to some tools Verilog provides for making debugging easier. Finally, you will use what you've made so far to create a sequencer where you will use synchronous RAM as memory to store your tune.

In this lab, we will begin by taking your `tone_generator` and `music_streamer` design from Lab 2 and feeding it some music. We will learn about circuits to take the signals generated by the buttons and rotary encoder on the ML505 board and convert them into a digital signal we can use in our FPGA design. You will be using the LED's to confirm they are working correctly. We will discuss how to use synchronous resets to set an initial state of your circuits. We will be creating a basic FSM in the `music_streamer` and will then create an extension on that FSM to implement a music sequencer.

Run `git pull` in your **git cloned** `labs_fa16` directory to **fetch the latest skeleton files for this lab.**

3 Overview of Your tone_generator and music_streamer



We included this diagram so you can review the circuit you made in lab 2. There are two main modules that you created: `tone_generator` and `music_streamer`.

Your `tone_generator` should take an input signal representing the tone or frequency you want to play and output a square wave that goes to the piezo speaker. It has an enable signal that allows you to disable the output, and this enable signal should be hooked up to one of the DIP switches.

Your `music_streamer` is responsible for providing the input signal to your `tone_generator` so it knows what frequency to play. Inside this generator is the memory (ROM), that holds the frequencies you want to play. Your music generator will output one tone for a certain amount of time (1/5 of a second for lab 2) before incrementing the ROM's address input. Your `music_streamer` will keep incrementing its address until it reaches the end of the ROM after which it should reset back to the first address, and loop through the ROM again.

4 ROM and music_streamer Modifications

Copy your `music_streamer` and `tone_generator` implementations from lab 2 into `lab3/src/music_streamer.v` and `lab3/src/tone_generator.v` respectively.

4.1 Generating a ROM

Run the following scripts in the `lab3/` directory to generate a Verilog ROM from sheet music.

First, use the `musicxml_parser.py` script to convert sheet music (in the form of a MusicXML file) into the contents of the ROM.

```
python scripts/musicxml_parser.py musicxml/Row_Row_Row_Your_Boat.xml music.txt
```

You will now have a `music.txt` file in the `lab3/` directory with the ROM's contents. Now we use `rom_generator.py` to create a ROM using this file.

```
python scripts/rom_generator.py src/rom.v music.txt 4096 24
```

You will now have a `rom.v` file in `lab3/src`. Take a look at this file. The ROM is 4096 entries deep and 24 bits wide with a 12-bit address input.

You can use any sheet music you want that's available in MusicXML. This website `https://musescore.com/` has a lot of good quality sheet music that you can download and play on the FPGA!

4.2 music_streamer Modifications

Make the following changes to the `music_streamer`:

1. Modify your code to use a 12-bit ROM address
2. Output the ROM address through the `rom_address` output. The most significant 8 bits of your ROM address will show up on the 8 GPIO LEDs.
3. Modify the amount of time each tone in the ROM will play to be $\frac{1}{25}$th of a second from $\frac{1}{5}$th of a second in lab 2.
4. There is a new output from the ROM called `last_address`. It contains the last address of the ROM that should be played. After you traverse the ROM and get to the last address, you should loop back to address 0.

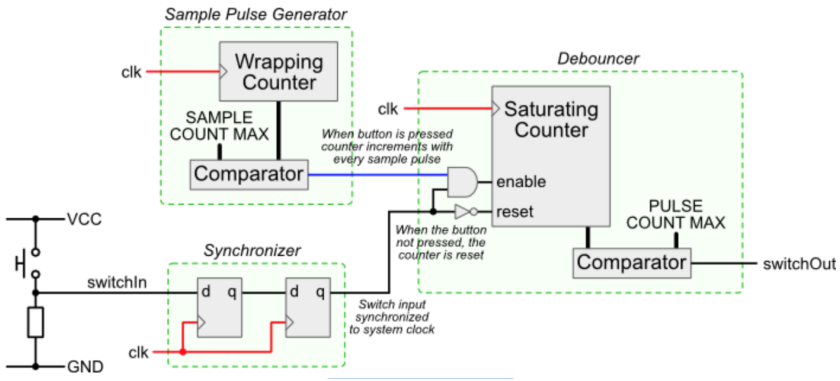
You shouldn't need to make any changes to your `tone_generator` from lab 2.

5 Playing Music

Go through the FPGA tool flow by running `make`. Check `make report` to see if there are any unexpected synthesis warnings; ask the TA if you need clarification. Run `make impact` and you should be able to hear 'Row Row Row Your Boat' playing through the piezo speaker.

* One issue you probably won't run into but should be aware of is that the tools will infer distributed memory on SLICEMs as long as our ROM module is small enough. If the ROM we specify is too big, then the tools will infer block RAMs instead. Block RAMs are synchronous memories which our `music_streamer` isn't designed to use; we will learn more about synchronous RAM in the next lab.

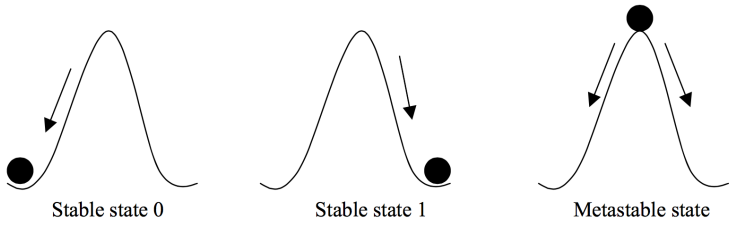
6 Synchronizer, Debouncer, and Rotary Encoder



Recall this graphic from the prelab debouncer reading. It is an overview of the debouncer circuit which includes the synchronizer circuit.

6.1 Synchronizer

In RTL, digital signals can be interpreted as 0's and 1's. However, in reality these correspond to low and high voltages and there are other states that can occur. In the lab we have to be concerned about the metastable state.



Normally, when there are no timing issues, we only have to worry about the high and low states. However, if there are timing issues, we can run into metastability where a net is basically stuck between the two states. This is an oversimplification but what you ultimately need to know is that metastability is generally undesirable and we want a circuit to get rid of it.

This synchronizer circuit we want you to implement for this lab is relatively simple. For synchronizing one bit, it is a pair a flip-flops connected serially. This circuit synchronizes an asynchronous signal (not related to any clock) coming into the FPGA. We will be using our synchronizer circuit to bring any off-FPGA signals into the clock domain of our FPGA design.

Edit the `lab3/src/synchronizer.v` file to implement the two flip-flop synchronizer. This module is parameterized by a `width` parameter which indicates the number of one-bit signals to synchronize.

6.1.1 Testing

As a rudimentary test of your synchronizer, we have provided a file called `lab3/src/synchronizer_fpga_test.v` which will synchronize various button press signals and send them to the GPIO LEDs. Execute the following commands:

```
make TOP=synchronizer_fpga_test
make TOP=synchronizer_fpga_test report
make TOP=synchronizer_fpga_test impact
```

We are changing the default value of `TOP` that the `Makefile` uses, to point to `synchronizer_fpga_test` as the top level module rather than `ml505top`. This allows us to build and impact different top level modules in the same folder. Keep in mind that each top level module needs a UCF file with the same name defining its I/O.

Take a look at this test module and try it out on the board. Pressing the various buttons should light up the respective LED.

You will notice that when pressing certain buttons, the LED flickers on and off before setting at on. This can create issues in our digital logic if we just assume that a low to high logic level transition constitutes a button press, since we will get erroneous presses. We use a debouncer to fix this issue.

7 Debouncer Review

Last lab, there were some questions about using an array for the debouncer. We will elaborate more and review the debouncer circuit so there is less confusion this time.

The debouncer for one button uses two counters. Let us start with the saturating counter. The saturating counter receives two essential signals: the synchronized input and an enable signal. Every time the enable signal goes high, one of the two things can happen:

- if the input synchronized signal was high, we increment the counter. Once this counter equals `PULSE_COUNT_MAX`, we indicate the button has been pressed long enough. The larger `PULSE_COUNT_MAX` is, the longer we have to hold the button for the circuit to realized we actually pushed the button. - if the input synchronized signal was low, we reset the counter. This essentially means we stopped pressing the button before the circuit realized we pushed the button.

The wrapping counter essentially tells the saturating counter when to count. Everything we reach `SAMPLE_COUNT_MAX`, we tell the saturating counter to count by one For example, if we made `SAMPLE_COUNT_MAX` equal to 1, we'd be telling the saturating counter to count or reset every cycle.

The reason we had an array was because, we have multiple buttons. Each button will at the very least need its own saturating counter. We don't want the counter of one button affecting the others. The sampling counter doesn't necessarily need multiple counters because it can tell all saturating counters to sample at the same time and this won't cause them to interfere with one another. We could have had a separate wire for each counter but we wanted you to make the module parameterizable. Thus we use an array whose size can be adjusted with a parameter and one row of the array represents one of the saturating counters for one of your buttons.

7.1 Edge Detector

The debouncer we provided last lab had an implicit edge detector: we gave you the specifications of the module and you implemented it as one block. In this lab, we will specifically separate the edge detector part of the debouncer. The edge detector is a signal that essentially marks when an edge is detected on a net.

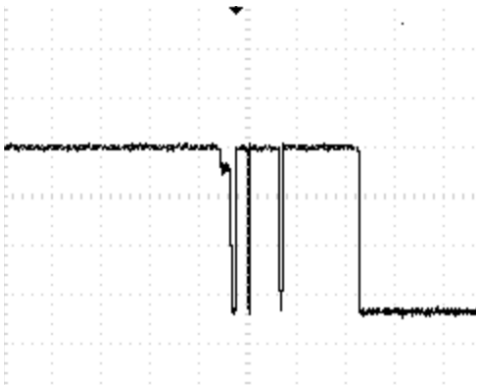
The issue in this lab is that the debouncer of your button outputs a one clock period wide pulse at an edge while your rotary encoder circuit expects the signal to be high much longer. Part of the rotary encoder circuit expects both the incoming signal from the wheel to be high at some point so we cannot just have a one clock period wide pulse when we detect that one of the signals from the wheel have gone high.

We want you to make two versions of the debouncer: one that outputs a short pulse for the debouncing the buttons and other that remains high as long as the edge detected signal remains high.

*Note: Some of you have been asking about the widths of the counters used in the debouncer circuit. We provided a log base 2 macro that will calculate the number of binary bits needed to represent a decimal number.

7.2 Debouncer

For this lab, the debouncer circuit will take a button's digital input and output a single pulse indicating a single button press. The reason we need a somewhat involved circuit for this is shown in the figure below.



When we press the button, we don't get a perfect stable signal. Instead the button signal has a mechanical 'bounce'. A debouncer turns this waveform, which shows a single button press, into a single pulse that our circuit can use.

Take a look at `lab3/src/debouncer.v`. This is a parameterized debouncer which can debounce `width` signals at a time. Your debouncer receives a vector of synchronized 1-bit signals and it outputs a debounced version of those signals. The other parameters reference the constants used in the circuit from the prelab reading.

The debouncer consists of:

1. **Sample Pulse Generator** - Tells our main debouncer counter when to sample the input signal. It should output a 1, every `sampling_pulse_period` clock cycles. By default `sampling_pulse_period` is set to 25000.

2. **Saturating Counter** - This is a counter that counts up to `saturating_counter_max`. Every time the sample pulse generator tells the saturating counter to sample the input signal, it should either count up by 1 if the input signal is high, or it should reset to zero if the input signal is low. When the saturating counter reaches the value of `saturating_counter_max`, by default set to 120, it should output a pulse that's one clock period wide. (i.e. the saturating counter should reset to zero the cycle after the counter reaches `saturating_counter_max`. This isn't shown in the circuit diagram.)

You will likely need to use 2D regs in Verilog to implement a saturating counter for each input signal to debounce. You will also likely need to use generate-for. You can use the same sample pulse generator for all input signals.

As an example:

```
reg [7:0] arr [0:3]; // 4 x 8 bit array
arr[0]; // First byte from arr (8 bits)
arr[1][2]; // Third bit of 2nd byte from arr (1 bit)

genvar i;
generate
        for (i = 0; i < width; i = i + 1):LOOP_NAME begin
                // Insert Verilog here
                always @ (posedge clk) begin
                        // Stuff
                end
        end
endgenerate
```

7.2.1 Testing in Simulation

There is no staff created testbench for this module, but you are welcome to create your own. Instantiate your debouncer and synchronizer module and feed it a simulated button press. First start the button off low, then after x ns make it high and oscillate it a bit before settling at high in your initial block. Run your simulation for a second or so and see how your debouncer module behaves. You will have to write your own testbench file and your own .do file to be placed in `lab3/sim/tests` for this testbench.

7.2.2 Testing on the FPGA

We have created a top level module called `debouncer_fpga_test` that will create a 8-bit register and will use button presses to add and subtract from it. Pressing any button will cause the counter to increment and the LEDs will show the current value of the counter. Pressing the South compass button however, will cause the counter to decrement.

```
make TOP=debouncer_fpga_test
make TOP=debouncer_fpga_test report
make TOP=debouncer_fpga_test impact
```

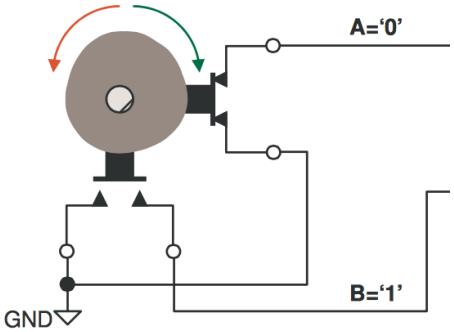
Make sure that your report gives you **zero warnings for synthesis.** You must fix any and all warnings before your debouncer will work well on the FPGA.

Show the TA the debouncer working before moving on. It is critical that your debouncer works properly.

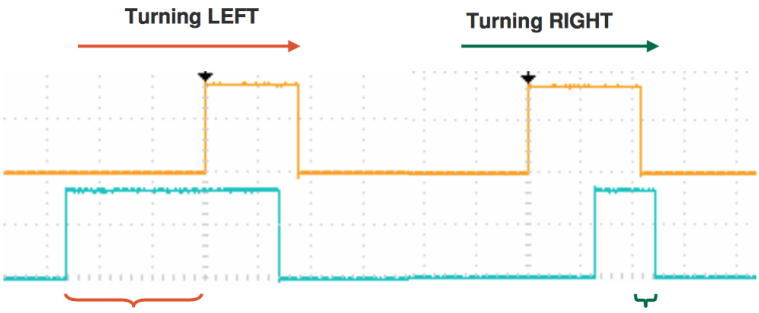
You will discover when playing with your debouncer that the buttons have a way that they like being pressed to minimize bounce; get a good feel for them. You can try holding down a button to see how your debouncer will only output pulses at a regular interval. You can try changing the debouncer parameters to see how they affect the debouncer's performance.

7.3 Rotary Encoder

The rotary encoder consists of a circuit that has two switches that go high as you rotate the wheel. Recall this waveform from the prelab reading.



Our main concern is finding out which direction the wheel turned. The following oscilloscope waveform from the prelab reading illustrates how we will do so.



If the pulse from A (top wave - orange) happens before the pulse from B (bottom wave - blue), it indicates that a the wheel has been turned a certain way. If the wheel is spun in the opposite direction then B's pulse will occur before A. We will take advantage of the fact that we can examine the logic level of wave B at the rising edge of wave A to determine direction of wheel movement.

Open up `lab3/src/rotary_decoder.v`. This module takes the synchronized A and B signals, and a clock input. It outputs `rotary_event` when a wheel click has been detected and `rotary_left` indicates whether that spin was to the right or the left.

Begin by implementing the Rotary Contact Filter as described in slide 7 of the prelab reading. This is a simple filter to remove glitching from the A and B signals. After implementing the filter, refer to slide 8 of the prelab reading for the circuit that will produce the `rotary_event` and `rotary_left` outputs.

7.3.1 Testing

As with the debouncer, there isn't a staff provided testbench, but you may write one yourself. There is a test top level module called `rotary_decoder_fpga_test`. It allows you to spin the rotary encoder to increment and decrement a 8-bit counter whole value is shown on the GPIO LEDs. Pushing the rotary encoder button will cause the counter to reset it to 0. Run it as such.

```
make TOP=rotary_decoder_fpga_test
make TOP=rotary_decoder_fpga_test report
make TOP=rotary_decoder_fpga_test impact
```

Make sure that your report gives you **zero warnings for synthesis.** You must fix any and all warnings before your rotary encoder will work well on the FPGA.

Show the TA the rotary encoder working before moving on. It is critical that your rotary encoder works properly.

Congratulations! You just built three highly useful and practical digital circuits. Now let's integrate them into our larger music streamer design.

8 Synchronous Resets In Design and Simulation

Now that we have a debouncer that can give us a pulse for a press of a button, we have a way of explicitly resetting our circuits! You will recall that in the previous lab, we set the initial value of registers like follows so that our simulation would have defined signals.

```
reg [23:0] clock_counter = 0;
```

Now that we have a reset signal tied to the CPU_RESET push button, we can do this instead.

```
always @ (posedge clk) begin
        if (rst) begin
                clock_counter <= 24'd0;
        end
end
```

Unlike what we did before, this RTL is synthesizable for all deployment targets, FPGAs, ASICs, and CPLDs alike. Go ahead and modify your `tone_generator` and `music_streamer` to use the provided reset signal to get your registers to a default state.

After doing this, run the `tone_generator_testbench` again using `make` in the `lab3/sim/` directory. View the waveform using ModelSim and see how we used a reset in the testbench to bring all the registers to a defined state without specifying a default value.

Please note that the synchronizer, debouncer, and rotary decoder don't take any reset signals. These circuits should have any registers they use initialized to 0 using the method used in lab 2. This is because these circuits are designed specifically for our FPGA and we are intentionally using our FPGA's initialization capabilities to set their initial states.

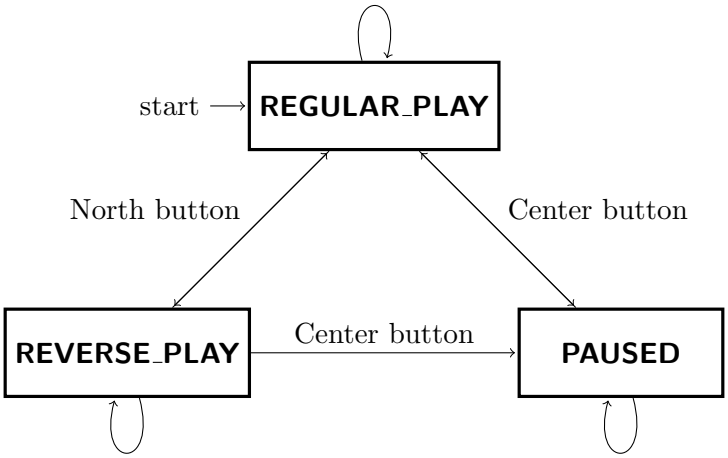
9 Music Streamer Tempo Control

Let's begin using the new user inputs we now have a well defined interface to. Start with the `music_streamer`. You will recall that your `music_streamer` by default chooses to play each tone in the ROM for 1/25th of a second. Extend the functionality of the `music_streamer` so that spinning the rotary encoder changes the tempo of the notes. Pushing in the rotary encoder should reset the tempo back to the default value. You get to choose the amount the tempo changes for each click of the wheel.

Try this out on the FPGA and verify that you have control of your `music_streamer`'s tempo using the rotary encoder. You should be able to speed up and slow down the music you are playing.

10 Music Streamer FSM

To wrap up this lab, you will implement a simple FSM in the `music_streamer`. The FSM will have 3 states: PAUSED, REGULAR_PLAY, REVERSE_PLAY. Here is the state transition diagram:



1. Your initial state should be `REGULAR_PLAY`.
2. Pressing the center compass push button should transition you into the `PAUSED` state from either the `REGULAR_PLAY` or `REVERSE_PLAY` states. Pressing the center compass push button while in the `PAUSED` state should transition the FSM to the `REGULAR_PLAY` state.
3. In the `PAUSED` state, your ROM address should be held steady at its value before the transition into `PAUSED` and no sound should come out of the piezo speaker. After leaving the `PAUSED` state your ROM address should begin incrementing again from where it left off and the speaker should play the tones.
4. You can toggle between the `REGULAR_PLAY` and `REVERSE_PLAY` states by using the north compass button. In the `REVERSE_PLAY` state you should decrement your ROM address by 1 rather than incrementing it by 1 every X clock cycles as defined by your tempo.
5. If you don't press any buttons, the FSM shouldn't transition to another state. Also, the rotary encoder wheel can be used to change tempo regardless of which state you are in.

Your `music_streamer` takes in 2 user button inputs that it can use to transition states. You should drive the compass LEDs in this fashion corresponding to the three states:

LED	**Value**
Center	`current_state == PAUSED`
North	`current_state == REGULAR_PLAY`
South	`current_state == REVERSE_PLAY`
East	0
West	0

Put your design on the FPGA with `make` and `make impact` and try transitioning states. For checkoff be able to demonstrate your state machine working and the tempo control with the rotary encoder.

10.1 Testbenches

We will be including some testbenches for your circuits that introduce you to some of the tools you can use in Verilog to help you debug.

- `@(posedge signal)`
- `repeat`
- `tasks`
- `$display`
- `fork/join`
- hierarchical paths - you can access signals within instantiated modules for debugging purposes

11 Sequencer

11.1 Synchronous RAM

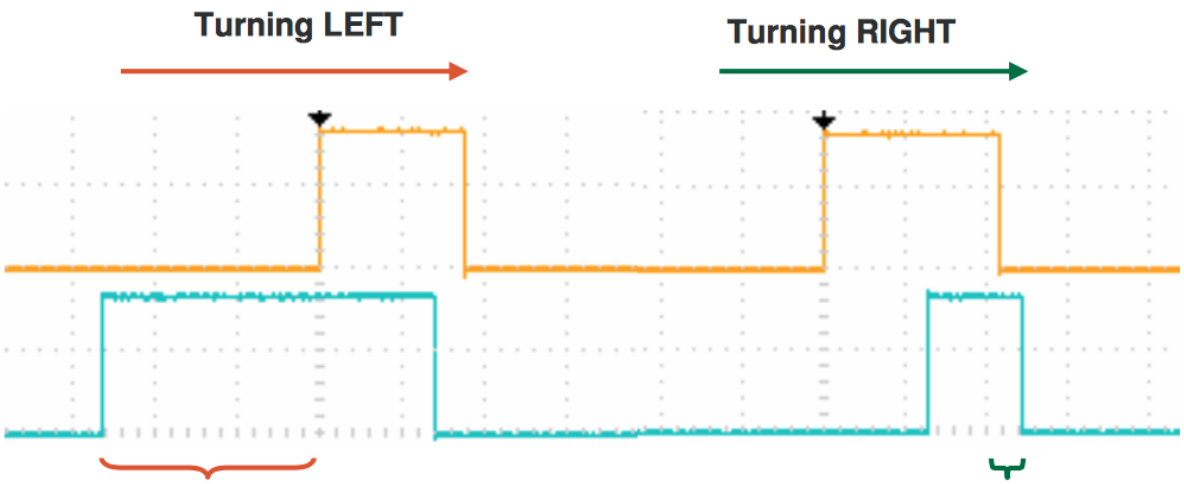
11.2 Sequencer FSM

A simple explanation of what we want you to do for the sequencer is to add one more functionality to your FPGA in which you will actually be able to make your own simple tune. You will have 8 memory locations which you can write to. To make the tune, you should have two states, an edit and play state.

In the edit state, you start off at the first memory location and you should be able to adjust the tone you want to save. You adjust the tone by turning the wheel and you save and move to the next memory address by pressing the wheel button. You should be playing the tones while editing so you can hear the tone that you are saving.

In the play state, you should be playing back

11.3 Putting Everything Together



12 Conclusion + Checkoff

You are done with lab 3! This was the most challenging lab so far, but we hoped you learned a lot. All of this knowledge and experience will be useful for the project. Please write down any and all feedback and criticism of this lab and share it with the TA. This is a brand new lab and we welcome everyone's input so that it can be improved.

12.1 Checkoff Tasks

1. Show the TA your working design with the FSM. Be able to transition states by clicking on the north and center buttons and show that your `music_streamer` matches the spec.
2. Show the tempo control working by spinning the rotary encoder to speed up and slow down the music.
3. Demonstrate that hitting the `CPU_RESET` button resets the ROM address back to 0 and puts the FSM into the `REGULAR_PLAY` state.
4. Show the TA your Verilog RTL for all the components you designed for this lab (synchronizer, debouncer, rotary decoder, FSM) and briefly explain the design of each of them.