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 review the 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 circuits you did in lab 3. Some need to be changed such as the debouncer but others such as the synchronizer do not need changes. **When we ask you to copy your Verilog from a previous lab, please don't copy over the entire file, but just copy and paste the code you wrote inside your module.** Some of the port declarations for various modules will have changed from previous labs.

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 `lab4/` 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.

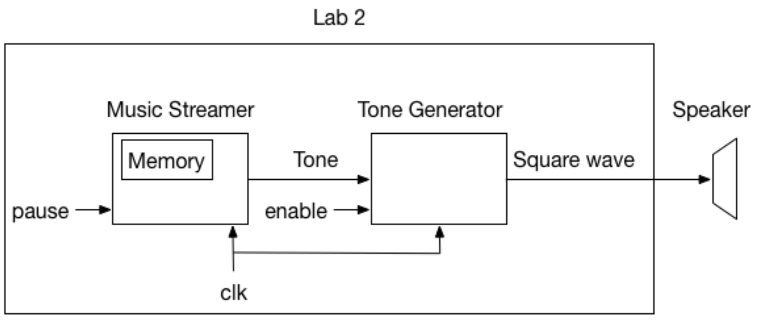
2 Lab Overview

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 `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 lab4/src/music_streamer.v and lab4/src/tone_generator.v respectively.

4.1 Generating a ROM

Run the following scripts in the lab4/ 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 lab4/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 upper 8 bits of your ROM address through the GPIO_leds 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). i.e., your ROM address should increment every $\frac{1}{25}$th of a second.
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.

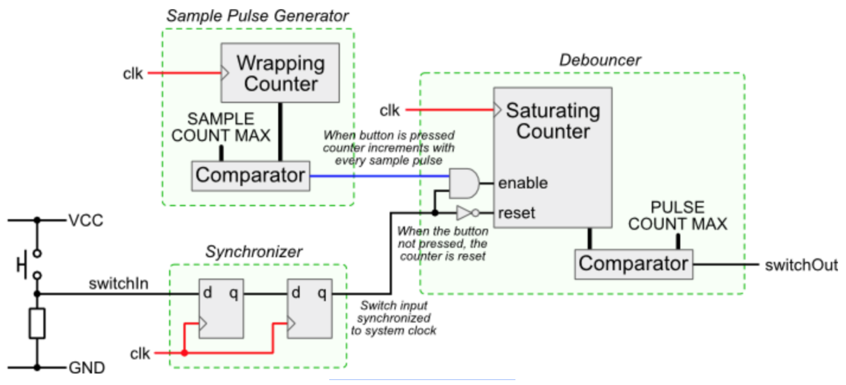
You shouldn't need to make any changes to your tone_generator from lab 2.

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.

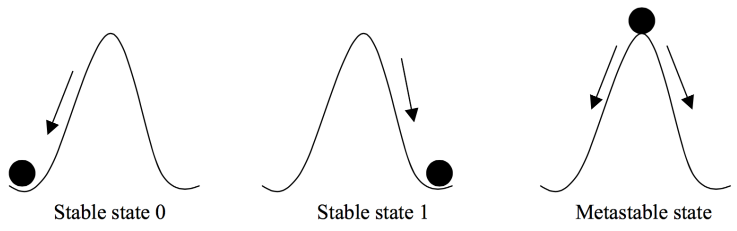
6 Synchronizer, Debouncer, and Rotary Encoder



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 in Simulation

The testbenches to be run are stored in `lab4/sim/tests`. Each `.do` file in this directory is run when you run `make` in the `lab4/sim` directory. If you only want to run one testbench, you can rename all the other `.do` files in this directory to have a different file extension.

Run `make` in the `lab4/sim` directory to run the testbenches. We have provided a testbench for your synchronizer called `sync_testbench` in `lab4/src/sync_testbench.v`. Take a look at the code for this testbench and run it; **the testbench should pass and you should inspect the waveform before you move on.** For details on the techniques/syntax used in this testbench, refer to Section 7 of this lab.

6.1.2 Testing on the FPGA

As a rudimentary test of your synchronizer, we have provided a file called `lab4/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
```

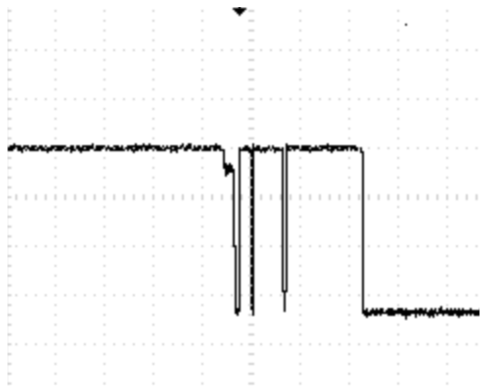
We are changing the default value of `TOP` that the `Makefile` uses, to point to `synchronizer_fpga_test` as the top level module rather than `ml505top`. This allows us to build and impact different top level modules in the same folder. Keep in mind that each top level module needs a UCF file with the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

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

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.

6.2 Debouncer and Edge Detector

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 `lab4/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`. The saturating counter should increment by one every time the input signal is 1 and the sample pulse generator tells us to sample the input signal. At any clock edge, if the input signal is 0, the saturating counter should be reset to 0. Once the saturating counter reaches `saturating_counter_max`, it should hold that value indefinitely until the input signal falls to 0, upon which the saturating counter should be reset to 0. The output of your debouncer should be an equality check between the saturating counter and `saturating_counter_max`.

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.

Here is an example of creating a 2D array and using a generate-for loop:

```
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) begin:LOOP_NAME
                always @ (posedge clk) begin
                        // Insert synchronous Verilog here
                end
            end
endgenerate
```

6.2.1 Debouncer Clarifications For Lab 4

To clarify, you should use the same sample pulse generator for all input signals into your debouncer, but you should have a separate saturating counter per input signal.

*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.

6.2.2 Edge Detector

The debouncer we created in the last lab (Lab 3) 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 intentionally decouple the edge detector from the debouncer. An edge detector will take an input signal and will output a one clock period wide pulse on a rising edge of the input signal.

You will feed the output of your button debouncer through an edge detector before passing the signal to the `music_streamer` or reset net.

Create a variable-width edge detector in `lab4/src/edge_detector.v`.

6.2.3 Testing in Simulation

We've provided a testbench to test your debouncer and edge detector circuits in `lab4/src/debouncer_testbench.v` and `lab4/src/edge_detector_testbench.v`. Run the testbench, make sure it passes, and inspect the waveforms before FPGA testing.

6.2.4 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. This module will use both your `debouncer.v` and `edge_detector.v`.

Pressing any button will cause the register to increment and the LEDs will show the current value of the register. Pressing the South compass button however, will cause the register to decrement.

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

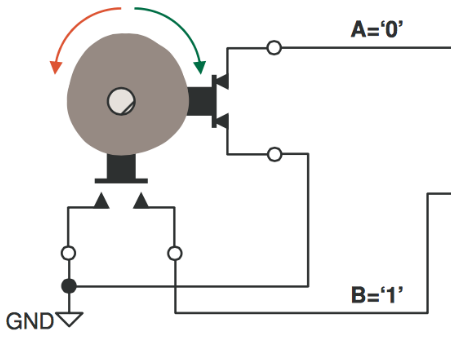
Make sure that your report gives you **zero warnings for synthesis.** You must fix any and all warnings before your debouncer will work well on the FPGA.

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

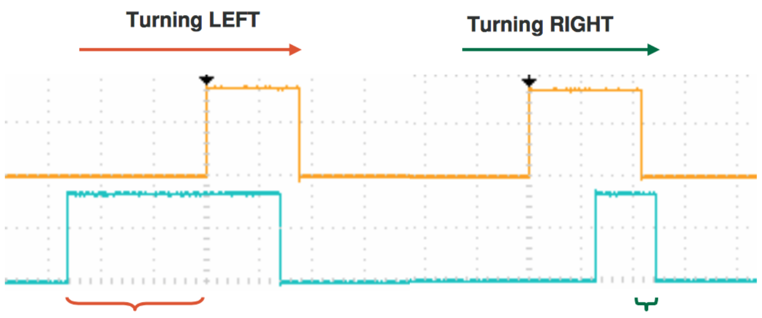
You will discover when playing with your debouncer that the buttons have a way that they like being pressed to minimize bounce; get a good feel for them.

6.3 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.

6.3.1 Testing in Simulation

We have provided a rotary decoder testbench for you in `lab4/src/rotary_decoder_testbench.v`. Run this testbench and make sure it passes; inspect the waveforms. Proceed to the FPGA test once you have confirmed expected behavior in simulation.

6.3.2 Testing on the FPGA

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 four highly useful and practical digital circuits. Now let's integrate them into our larger music streamer design.

7 Testbench Techniques

There are several testbenches included in this lab for your synchronizer, edge detector, rotary encoder, debouncer, and music streamer that introduce you to some useful Verilog testbench constructs.

- `@(posedge signal)` and `@(negedge signal)` - instead of having an always block before, you can just use these to trigger just once at the rising or falling edge of a signal, note you can also use this to advance the current timestep to where the event occurs

  For example:

  ```
  @(posedge signal);
  @(posedge signal);
  ```

  You will be taken to the time step where the signal goes high for the second time.

- `repeat` - it acts like a `for` loop but without an increment variable

  For example:

  ```
  repeat (10) @(posedge clk);
  ```

  You will be taken to the time step after the tenth rising edge of the clock.

- `$display` - acts essentially as a print statement. Similar to languages like C, if you want to print out a wire, reg, integer, etc... value, you will need to format the string.

  For example:

  ```
  \$display("Wire x in decimal is %d", x);
  \$display("Wire x in binary is %b", x);
  ```

- `tasks` - tasks are basically subroutines where you can essentially group and organize some commands rather than haphazardly putting them everywhere. They can take inputs and outputs. An few examples are shown in the provided testbenches.

  - `fork/join` - a powerful tool that lets you execute other calls in parallel. Similar to a begin/end block, you create a fork block with the keyword fork and end the block with the keyword join.

  For example:

  ```
  fork
      task1();
      begin
                  \$display("Another thread");
                  task2();
              end
  join
  ```

- hierarchical paths - you can access signals within instantiated modules for debugging purposes. This can be helpful in some cases where you want to look at a signal but don't want to create another output port just to look at or if you don't want to use the waveform viewer for a simple check.

  For example:

  ```
  my_device device(); //device instantiation
  \$display("Signal inside my device %b", device.x);
  ```

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.

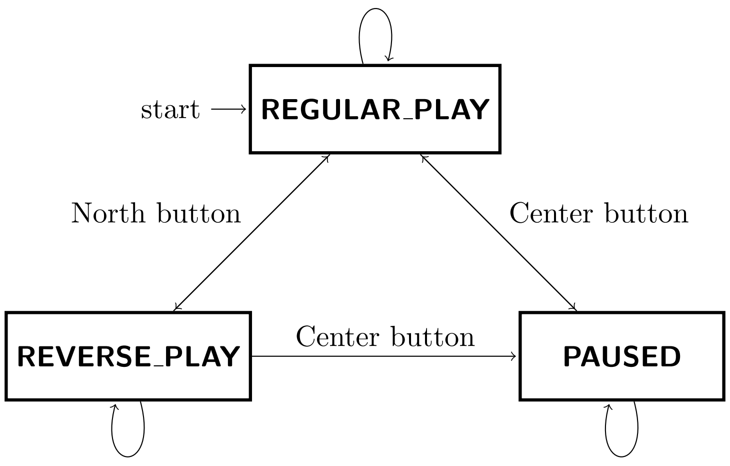
9 Music Streamer Tempo Control

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 Synchronous RAM

10.2 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 pressing the wheel button. You can move to the next memory location by pressing the east or west compass buttons. HOWEVER, note that if you want to save the tone, you must remember to press the wheel button before moving to a different memory address. 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 looping through the 8 frequencies. A reset should return the tones in all memory location to some initial value.

11 Putting Everything Together

One final task will be to combine the sequencer into your existing FSM. This means aside from the regular, pause, and reverse states, you will have one more state which is the `SEQUENCER` state. Within the `SEQUENCER` state, you should have the behavior described earlier. You should design the transitions such that if you press the north compass button, the `REGULAR_PLAY` state should move into the `SEQUENCER` state and pressing the north button again in the `SEQUENCER` state should take you back to the `REGULAR_PLAY` state.

12 Conclusion + Checkoff

You are done with lab 4! 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. Demonstrate that you can transition into the `SEQUENCER` state and that you can edit your tones and play them back.
5. 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.