EECS 151/251A FPGA Lab
Lab 3: Rotary Encoder and Debouncer, Finite State Machines, Synchronous Resets, Synchronous RAM, Testbench Techniques

Prof. Elad Alon
TAs: Vighnesh Iyer, Bob Zhou
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 these 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 Lab Overview

In this lab, 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 that your input conditioning circuits are working correctly. We will discuss how to use synchronous resets to reset our circuits to a known initial state. We will be creating a basic FSM in the `music_streamer` that uses the buttons and rotary encoder to change states and alter the music playback.

Run `git pull` in your **git cloned** `labs_sp17` directory to fetch the latest skeleton files for this lab.

3 Synchronizer, Debouncer, and Rotary Encoder

3.1 Synchronizer

In Verilog (RTL), digital signals are either 0's or 1's. In a digital circuit, a 0 or 1 corresponds to a low or high voltage. If the circuit is well designed and timed (fully synchronous), we only have to worry about the low and high voltage states, but in this lab we will be dealing with asynchronous signals.

The signals coming from the push buttons and rotary encoder on the ML505 board don't have an associated clock signal. Thus, when those signals are put through a register, the hold or setup time constraints of that register may be violated. This may put that register into a **metastable** state.

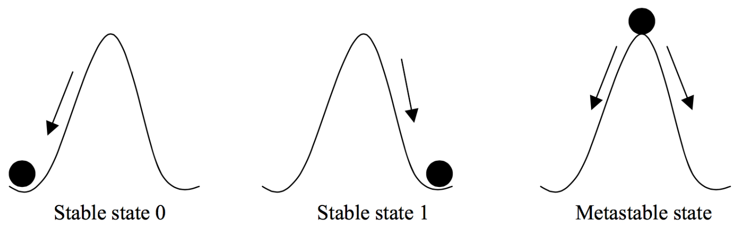


Figure 1: The 'ball on a hill' metaphor for metastability. If a register's timing constraints are violated, its output voltage oscillates and after some time unpredictably settles to a stable state.

In a fully synchronous circuit, the timing tools will determine the fastest clock frequency under which the setup time constraints are all respected and the routing tools will ensure that any hold time constraints are handled. Introducing an asynchronous signal that isn't changing with respect to a clock signal can cause a register to go into a metastable state. This is undesirable since this will cause a 'mid-rail' voltage to propagate to other logic elements and can cause catastrophic timing violations that the tools never saw coming.

We will implement a synchronizer circuit that will safely bring an asynchronous signal into a synchronous circuit. The synchronizer needs to have a very small probability of allowing metastability to propagate into our synchronous circuit.

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 asynchronous off-FPGA signals into the clock domain of our FPGA design.

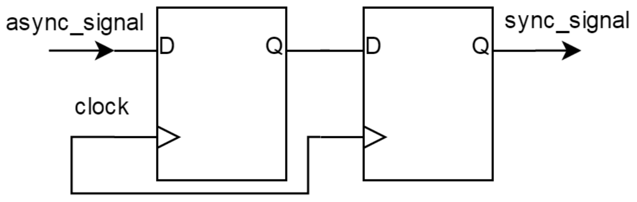


Figure 2: 1-bit 2 Flip-Flop Synchronizer

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.

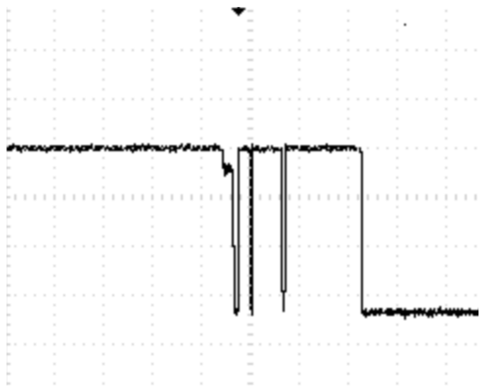
3.1.1 Testing in Simulation

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.

3.2 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
```

3.2.1 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.

3.2.2 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`.

3.2.3 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.

3.2.4 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
```

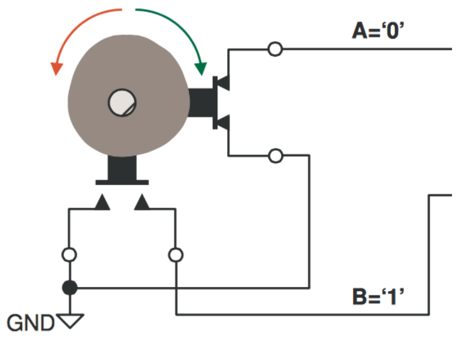
Make sure that your report gives you **zero warnings for synthesis.** You must fix any and all warnings before your debouncer will work well on the FPGA.

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

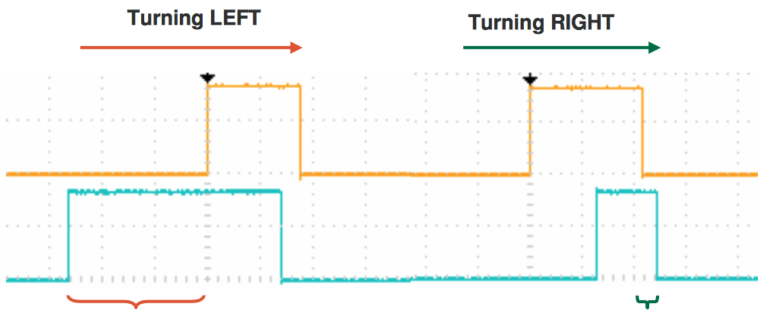
You will discover when playing with your debouncer that the buttons have a way that they like being pressed to minimize bounce; get a good feel for them.

3.3 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.

3.3.1 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.

3.3.2 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.

4 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>) - These are a different type of delay statement from what you have seen before. #10 would advance the simulation by 10 timesteps. These commands will advance the simulation until the <signal> rises or falls.

For example:

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

Simulation time will advance until we have seen two rising edges of `signal`.

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

  For example:

  ```
  repeat (2) @(negedge clk);
  repeat (10) begin
          @(posedge clk);
  end
  ```

  The simulation will advance until we have seen 2 falling clock edges and will then advance further until we have seen 10 rising clock edges.

- `$display` - acts as a print statement. Similar to languages like C, if you want to print out a wire, reg, integer, etc... value in your testbench, 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 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` - Allows you to execute testbench code in parallel. You create a fork block with the keyword `fork` and end the block with the keyword `join`.

  For example:

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

  Multiple threads of execution are created by putting multiple begin/end blocks in the fork block. In this example, thread 1 runs `task1()`, while thread 2 first `$display`s some text then runs `task2()`. The threads operate in parallel.

- Hierarchical Paths - you can access signals inside an instantiated module for debugging purposes. This can be helpful in some cases where you want to look at an internal signal but don't want to create another output port just to look at it.

  For example:

  ```
  tone_generator tone_gen ();
  $display("Signal inside my tone_generator instance, clock_counter: %b",
  ↪  tone_gen.clock_counter);
  ```

5 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 as below 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. You might also want to modify your debouncer, synchronizer, edge detector, and rotary decoder to use the provided reset signal.

After doing this, run the `tone_generator_testbench` again using `make` in the `lab4/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.

6 Music Streamer Tempo Control

Let's use the new user inputs we now have access to. 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.

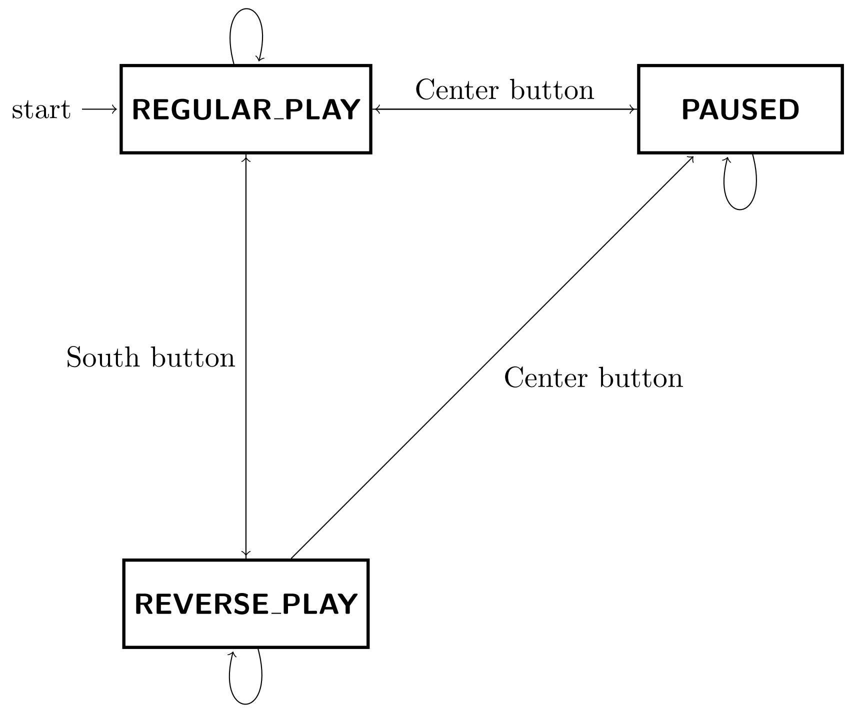
You should implement this by using a register to hold the number of clock cycles per note. Instead of this number being hardcoded in Verilog to represent $\frac{1}{25}$th of a second, you can change it at runtime. Spinning the rotary encoder once should add or subtract a fixed number from this register which should alter the time each tone is played (i.e. tempo).

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.

7 Music Streamer FSM

Now, you will implement a simple FSM in the `music_streamer`. **If you've done this part in Lab 3, make sure your FSM matches this slightly modified spec**.

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 south 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.
    - Small caveat: as you play your ROM in reverse, make sure that if the current ROM address is 0, that you loop back to the `last_address` of the ROM rather than to address 4095.
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 user button inputs that it can use to transition states. You should drive the compass LEDs in this fashion corresponding to the three states:

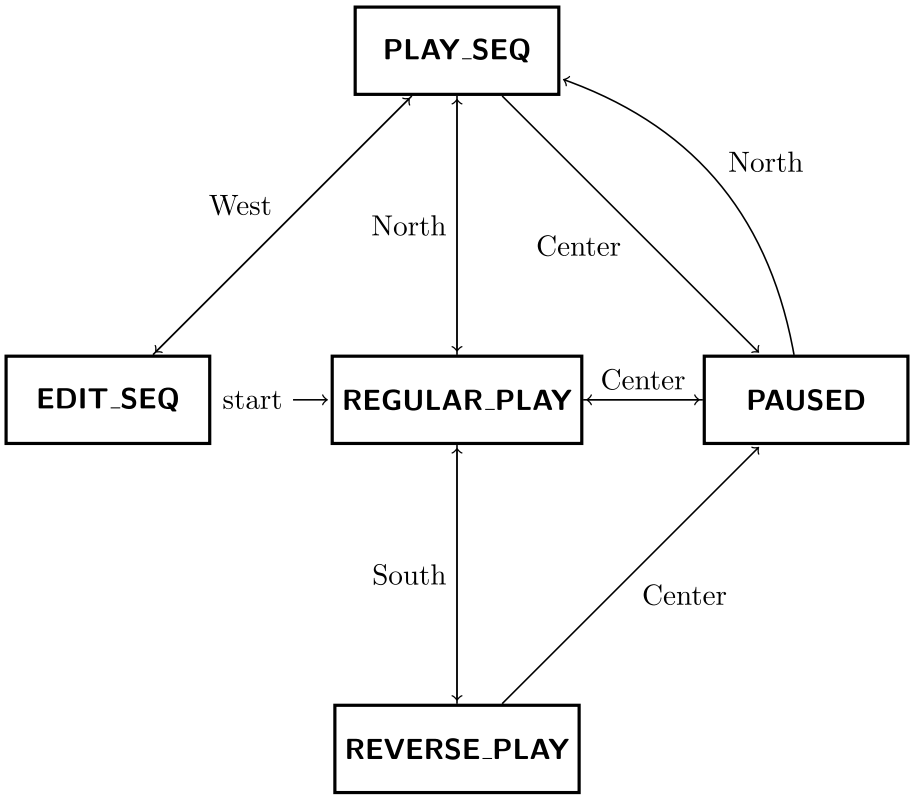
LED	**Value**
Center	`current_state` == `REGULAR_PLAY`
East	`current_state` == `PAUSED`
South	`current_state` == `REVERSE_PLAY`
North	0
West	0

You can run the testbench in `lab4/src/tone_generator_testbench.v` to test out your state machine. Take a look at the code to see what it does and inspect your waveform to check that your FSM is performing correctly. Verify that you don't have any unexpected synthesis warnings.

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.

8 Building a Music Sequencer FSM

Here is a new state transition diagram for our music sequencer we will build inside the `music_streamer` module.



We have added two new states `PLAY_SEQ` and `EDIT_SEQ`. You should wire up the west compass LED to `current_state == EDIT_SEQ` and the north compass LED to `current_state == PLAY_SEQ`. You should implement the skeleton of this state machine before proceeding with the complete explanation.

Our goal is to create an 8-tap music sequencer that we can use along with our regular music streamer. Our music sequencer will have a place (RAM) where it stores the `tone_switch_period`s of 8 notes. When we are in the `PLAY_SEQ` state, the music streamer will play the 8 notes, one after the other, in a continuous loop. Each note will play for a set amount of time as determined by the **sequencer tempo**. While in the `PLAY_SEQ` state we can change the sequencer tempo using the rotary encoder (**the sequencer tempo is different from the ROM tempo you modified earlier**)

We can edit these 8 notes on the fly by moving into the `EDIT_SEQ` state. In this state, the LEDs will show which of the 8 notes we are currently editing. By spinning the rotary encoder, we can select a new pitch for this note. By clicking in the rotary encoder, we can save the selected pitch in the RAM location for this note. We can use the east and west buttons to edit a different note.

You will find some skeleton code in `lab4/src/music_streamer.v` to help you implement the sequencer. This is a complicated circuit, so you should add features slowly and modify the `tone_generator_testbench` to test out your sequencer. Here is a list of requirements for your sequencer:

1. You should have two tempo controls.
   (a) The **ROM tempo** affects the duration of each note in the ROM and can be modified in either the `REGULAR_PLAY` or `REVERSE_PLAY` states by the rotary encoder.
   (b) The **sequencer/RAM tempo** affect the duration of each note in the sequencer RAM and can be modified only in the `PLAY_SEQ` state by the rotary encoder.
2. The GPIO leds should be used for the following functions in different states.
   (a) In the `REGULAR_PLAY`, `PAUSED`, and `REVERSE_PLAY` states, the 8 GPIO LEDs should display the top 8 bits of the ROM address.
   (b) In the `PLAY_SEQ` state, the 8 GPIO LEDs should display the note being played. For example, if note 5 is currently being played, LED 5 should be on and the rest off.
   (c) In the `EDIT_SEQ` state, the 8 GPIO LEDs should display the note being edited. For example, if we are editing note 6, LED 6 should be on and the rest off.
3. In the `EDIT_SEQ` state, the piezo speaker should play the current pitch of the note. Spinning the rotary encoder should increase and decrease the pitch by some amount per click that you determine. Pressing the rotary encoder button should save the current pitch of the note into the sequencer's RAM.
4. In the `EDIT_SEQ` state, pressing the east or west buttons should change the note being edited. As you change notes, the piezo speaker should play the pitch of the new note being edited.
5. You can use the same `clock_counter` for all parts of this state machine including the sequencer and the regular music streamer.
6. You can use the same `sequencer_address` for the two sequencer states.
7. You can choose what the reset values for the sequencer RAM entries are. A sensible default is provided in the skeleton code.

Please ask the TA if you have any questions about specifications for the sequencer or FSM in general.

9 Conclusion + Checkoff

You are done with lab 4! 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.

9.1 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.