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. `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 functions.

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 signals 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.

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.

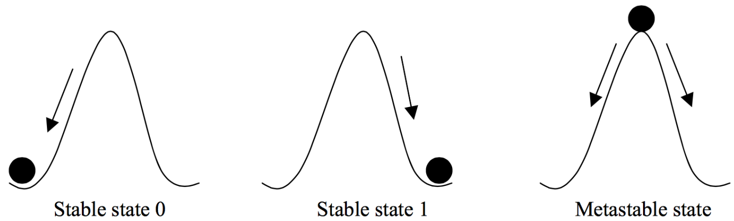


Figure 1: The 'ball on a hill' metaphor for metastability. If a register's timing constraints are violated, its output voltage oscillates and after some time unpredictably settles to a stable state.

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.

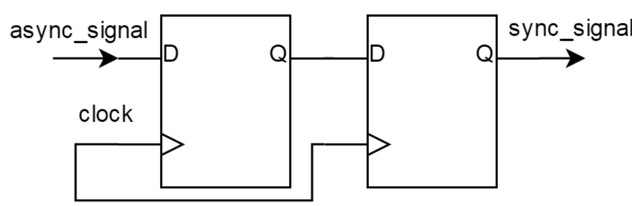


Figure 2: 1-bit 2 Flip-Flop Synchronizer

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.

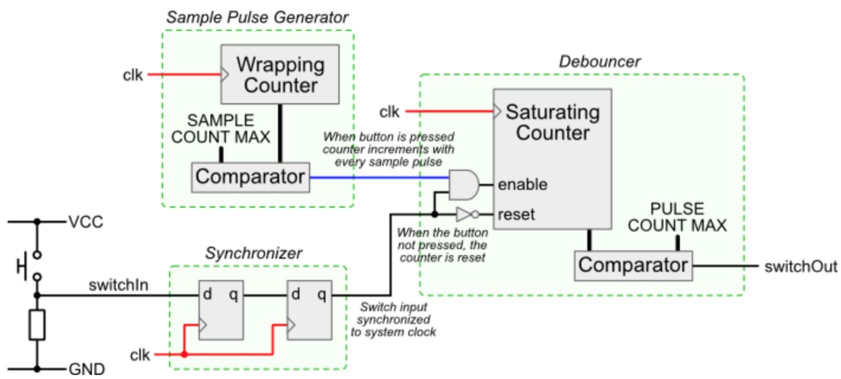
3.1.1 Testing in Simulation

The testbenches to be run are stored in `lab3/sim/tests`. Each `.do` file in this directory is run when you run `make` in the `lab3/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. Alternatively, you can leave the file extensions alone, and specify the test you want to run like this:

```
cd lab3/sim
make CASES="tests/sync_testbench.do"
```

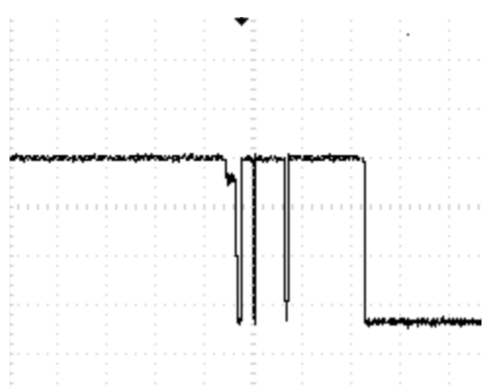
We have provided a testbench for your synchronizer called `sync_testbench` in `lab3/src/sync_testbench.v`. Take a look at the code for this testbench and run it; **the testbench shouldn't print any failure messages and you should inspect the waveform before you move on.** For details on the constructs/techniques/syntax used in this testbench, refer to the 'Testbench Techniques' section of this lab.

3.2 Debouncer and Edge Detector



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

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



When we press the button, the signal doesn't behave like a perfect step function. Instead the button signal is glitchy due to mechanical 'bounce'. A debouncer turns this waveform, which shows a single button press, into a clean signal with a single voltage transition.

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 saturating counter when to sample the input signal. It should output a 1, every `sample_count_max` clock cycles. By default `sample_count_max` is set to 25000.

2. **Saturating Counter** - This is a counter that counts up to `pulse_count_max`. The saturating counter should increment by one if both the sample pulse and the input signal are high at a clock edge. At any clock edge, if the input signal is 0, the saturating counter should be reset to 0. Once the saturating counter reaches `pulse_count_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 `debounced_signal` of your debouncer should be an equality check between the saturating counter and `pulse_count_max`.

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. You will likely need to use a 2D reg in Verilog to create the saturating counters. You will also likely need to use generate-for.

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 Edge Detector

The debouncer will act to 'smooth-out' the button press signal. It is then followed up with an edge detector that can take the high-to-low transition of the debouncer output and use it to generate a 1 clock period wide pulse that the rest of our digital design can use.

Create a variable-width edge detector in `lab3/src/edge_detector.v`.

3.2.2 Testing in Simulation

We've provided a testbench to test your debouncer and edge detector circuits in lab3/src/debouncer_testbench.v and lab3/src/edge_detector_testbench.v. Run the testbench, make sure it passes, and inspect the waveforms before FPGA testing. Make sure there are no undefined (red line) signals.

If you are seeing issues where certain registers are red lines (X's), make sure you give them an initial state. For a 2D reg initialization, use the following initialization code in debouncer.v:

```
integer k;
initial begin
    for (k = 0; k < width; k = k + 1) begin
        saturating_counter[k] = 0;
    end
end
```

The debouncer testbench has 2 tests:

1. Verifies that if a glitchy signal initially bounces and then stays high for **less** than the saturation time, that the debouncer output never goes high.
2. Verifies that if a glitchy signal initially bounces and then stays high for **more** than the saturation time, that the debouncer goes high and stays high until the glitchy signal goes low.

The edge detector testbench tests 2 scenarios, when the signal_in is a pulse 10 clock cycles wide and a pulse 1 clock cycle wide and verifies that the edge_detect_pulse output goes high twice, both times with a width of 1 clock cycle.

3.2.3 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 compass button will cause the register to increment by 1 and the LEDs will show the current value of the register. Pressing the rotary wheel inward, will cause the register to decrement. Pressing the CPU_RESET button (near the SATA port), will cause the register to reset to 0.

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

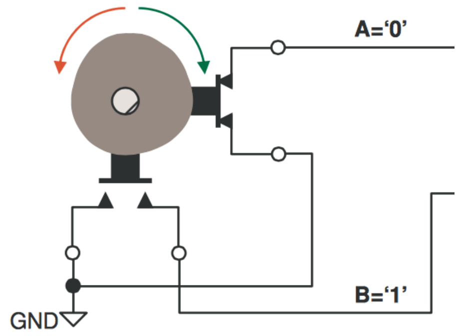
Make sure that your report gives you **zero warnings for synthesis.** You must fix any and all warnings before your debouncer will as expected on the FPGA.

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

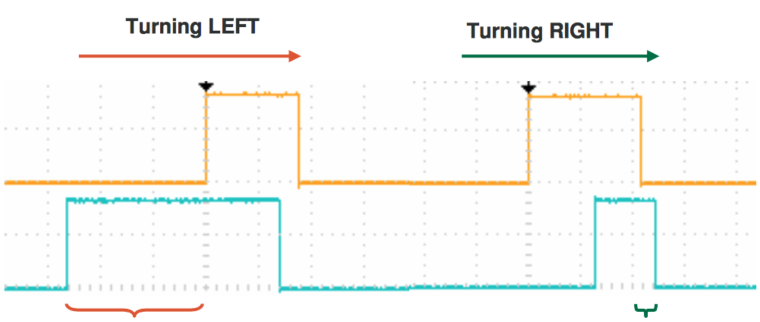
You will discover when playing with your debouncer that the buttons have a way that they like being pressed to minimize bounce; get a good feel for them.

3.3 Rotary Encoder

The rotary encoder is a device that has two switches that close as you rotate the wheel. Recall this diagram 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 B (bottom wave - blue) happens before the pulse from A (top wave - orange), it indicates that the wheel has been turned left. If the wheel is spun in the opposite direction then A's pulse will occur before B. 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 and debounced A and B signals, and a clock input. It outputs a `rotary_event` pulse (one clock cycle wide) when a wheel spin has been detected and `rotary_left` (when `rotary_event` is high) indicates whether that spin was to the right or the left.

You need to implement the circuit from slide 8 of the prelab reading. In that slide `rotary_q1` refers to input A and `rotary_q2` refers to input B. You don't need to use the `rst` input, but it is recommended for future use of this module.

3.3.1 Testing in Simulation

We have provided a rotary decoder testbench for you in `lab3/src/rotary_decoder_testbench.v`. **This testbench isn't self checking. You will have to inspect the waveform manually.** 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
```

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. It works like `printf()` in C.

  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 group and organize some commands rather than haphazardly putting them everywhere. They can take inputs and assign outputs.

```
task wait_for_n_clocks();
input [7:0] num_edges;
begin
    repeat (num_edges) @(posedge clk);
end
endtask
```

- `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-join 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 for debug.

  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

Begin by copying your `tone_generator` and `music_streamer` from lab 2. **Do not change the port declaration of the skeleton files, only copy over your implementation**.

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
    else begin
        clock_counter <= clock_counter + 24'd1;
    end
end
```

Unlike what we did before, this Verilog 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.

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. Use the `rotary_event`, `rotary_left`, and `rotary_push` inputs in the `music_streamer`.

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.

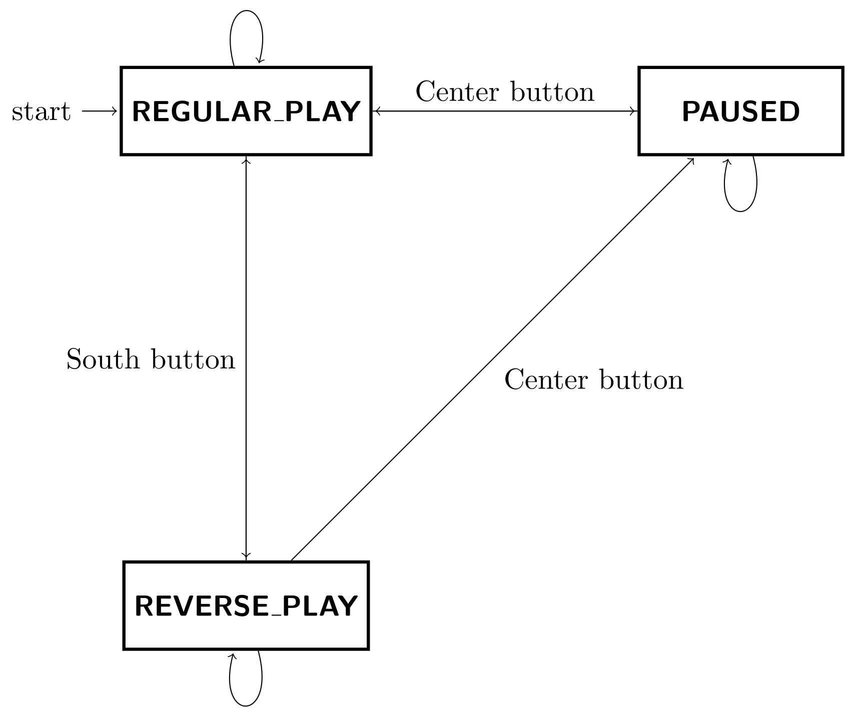
Spinning the rotary encoder once should add or subtract a fixed number from this register which should alter the time each tone is played. You get to choose this number; find something reasonable.

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`.

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.

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:

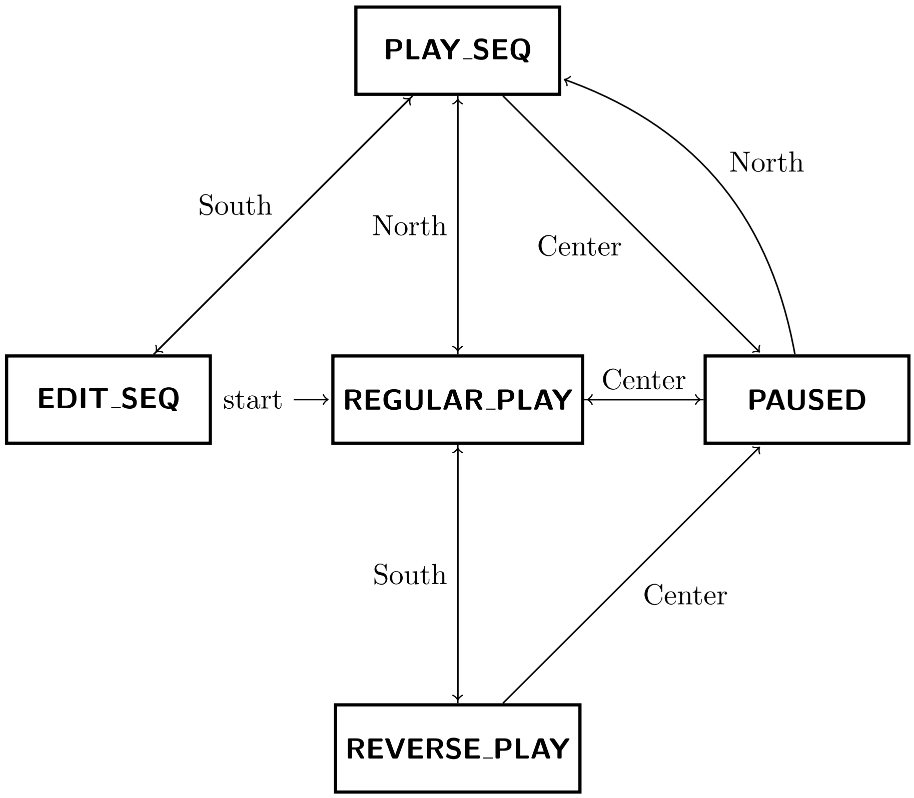
LED	**Value**
Center	`current_state` == `REGULAR_PLAY`
East	`current_state` == `PAUSED`
South	`current_state` == `REVERSE_PLAY`
North	0
West	0

You can run the testbench in `lab3/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 pushing 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 `lab3/src/music_streamer.v` to help you implement the sequencer. This is a complicated circuit, so you should add features slowly and modify the `music_streamer_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 Checkoff

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.