

# EECS 151/251A FPGA Lab

## Lab 4: Debouncers, Finite State Machines, Synchronous Resets, Synchronous RAM, Testbench Techniques, Hex Keypads

Prof. John Wawrzynek, Nicholas Weaver  
TAs: Arya Reais-Parsi, Taehwan Kim  
Department of Electrical Engineering and Computer Sciences  
College of Engineering, University of California, Berkeley

### Contents

<b>1</b>	<b>Before You Start This Lab</b>	<b>2</b>
<b>2</b>	<b>Lab Overview</b>	<b>2</b>
<b>3</b>	<b>Synchronizer, Debouncer</b>	<b>3</b>
3.1	Synchronizer . . . . .	3
3.1.1	Testing in Simulation . . . . .	4
3.2	Debouncer and Edge Detector . . . . .	4
3.2.1	Edge Detector . . . . .	6
3.2.2	Testing in Simulation . . . . .	6
3.2.3	Testing on the FPGA . . . . .	7
3.3	Shift Register (Optional) . . . . .	7
3.3.1	Testing in Simulation . . . . .	7
<b>4</b>	<b>Testbench Techniques</b>	<b>8</b>
<b>5</b>	<b>Synchronous Resets In Design and Simulation</b>	<b>9</b>
<b>6</b>	<b>Hex Keypad (Optional)</b>	<b>10</b>
6.1	PMOD Expansion Headers . . . . .	11
6.2	Installing our keypad . . . . .	11
6.3	Developing a Decoder . . . . .	11
<b>7</b>	<b>Music Streamer Tempo Control</b>	<b>12</b>
<b>8</b>	<b>Music Streamer FSM</b>	<b>12</b>
<b>9</b>	<b>Building a Music Sequencer FSM (Optional)</b>	<b>13</b>

## 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. [fpga\\_labs\\_sp18/resources/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. [fpga\\_labs\\_sp18/resources/pmodkypd\\_rm.pdf](#)

Skim the reference manual for the hex keypad we will be using. See if you can roughly figure out how it works given the manufacturer's description.

## 2 Lab Overview

In this lab, we will

- learn about input conditioning circuits that make control signals from physical input devices, like the buttons and switches we've been using up to now, more reliable;
- use board LEDs to confirm that our keypad input and conditioning circuits are working correctly;
- discuss how to use synchronous resets to reset our circuits to a known initial state;
- create a basic FSM in the `music_streamer` that uses the buttons and hex keypad to change states and alter the music playback;
- (optionally) learn how to interface with a hey-keypad, as made famous by security systems in movies; and
- (optionally) extend our `music_streamer` into a sequencer.

Run `git pull` in your `git cloned fpga_labs_sp18` directory to fetch the latest skeleton files for this lab.

## 3 Synchronizer, Debouncer

### 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 slide switches on the PYNQ-Z1 board don't have an associated clock signal. For the push-buttons in particular, 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.

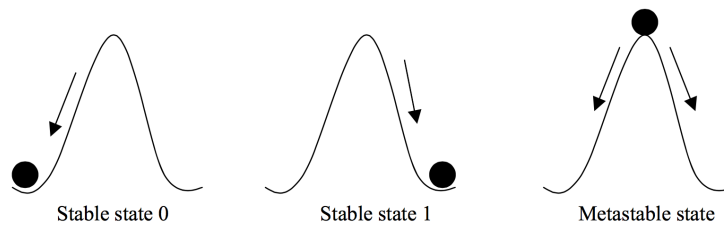


Figure 1: The 'ball on a hill' metaphor for metastability. If a register's timing constraints are violated, its output voltage oscillates and after some time unpredictably settles to a stable state.

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 of 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.

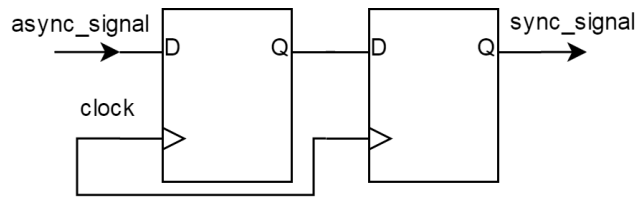


Figure 2: 1-bit 2 Flip-Flop Synchronizer

Edit the `synchronizer.v` (in your project sources) 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.

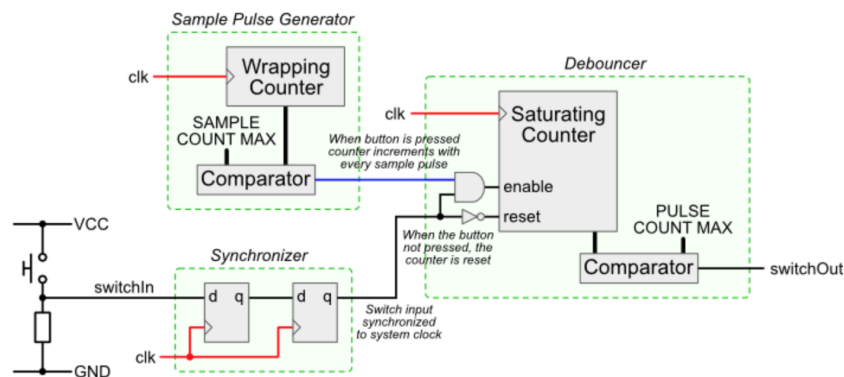
### 3.1.1 Testing in Simulation

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. Alternatively, you can leave the file extensions alone, and specify the test you want to run like this:

```
cd lab4/sim
make CASES="tests/sync_testbench.do"
```

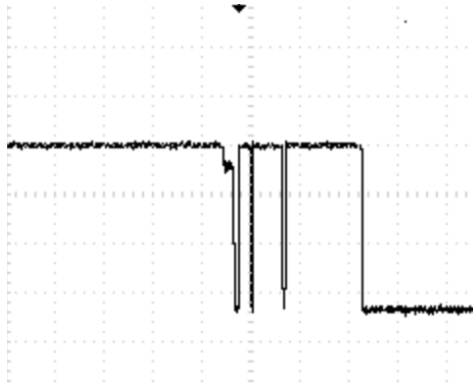
We have provided a testbench for your synchronizer called `sync_testbench` in `sync_testbench.v` (in the project source files). Take a look at the code for this testbench and run it; **the testbench shouldn't print any failure messages and you should inspect the waveform before you move on**. For details on the constructs/techniques/syntax used in this testbench, refer to Section 4: Testbench Techniques in 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 can be seen 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 `debouncer.v` in your project sources. 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:

```
reg [7:0] arr [3:0]; // 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)
```

And here is an **example** of using a generate-for loop:

```
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 low-to-high 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 `edge_detector.v`.

### 3.2.2 Testing in Simulation

We've provided a testbench to test your debouncer and edge detector circuits in `debouncer_testbench.v` and `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 6-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 BTN1 or BTN2 will cause the register to increment by 1. Pressing BTN0 will cause the register to decrement by 1. BTN3 resets the register to 0. The LEDs will show the current value of the register.

In your Vivado project, set `debouncer_fpga_test` as the top module and run the build flow, then program the board. Make sure that you **investigate any warnings for synthesis**. You can ignore some: the top-level module has unused input/output ports to avoid having to change the XDC file. Make sure you fix any other warnings you find so that the your debouncer will work as expected on the FPGA.

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 Shift Register (Optional)

A shift register “bubbles” new inputs from one end of a flip-flop chain to the other. It looks like a FIFO queue: with every input trigger, a new input is loaded into the first register, and all other values are moved along one. The last value is discarded.

Shift registers are useful for loading larger values than you have inputs available. Consider, for example, inputting a 32-bit binary number with only two inputs. Or inputting an 8-digit alarm code with a hey keypad: you should be able to enter press one key at a time and to store the complete value in the circuit.

The input trigger is usually *not* the clock: you want the registers only to take new values as some input event occurs. Your task is now to build a variable-depth and variable-width shift register module using the techniques you’ve seen so far (in particular in the Debouncer code). You’ll need:

- a WIDTH and a DEPTH parameter;
- a single output signal that is WIDTH×DEPTH-many bits wide;
- a WIDTH-wide input signal; and
- an input trigger signal.

Complete the code in `shift_register.v` to implement the shift register.

### 3.3.1 Testing in Simulation

As for the other circuits, a testbench for the shift register has been provided for you. Run it and confirm that there are no reported failures. Inspect the waveform to make sure you understand why, in any case.

Congratulations! You just built three or four 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, shift register, 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
```



```
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 3. **Do not change the port declaration of the lab 4 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;
```

or

```
initial clock_counter = 0;
```

Tying one of the push buttons to a reset signal, we can now 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
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 sensible initial state.

After doing this, run the `tone_generator_testbench` again using `make` in the `lab4/sim/` directory (or just use Vivado's simulator). View the waveform using the simulator 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 Hex Keypad (Optional)



The later parts of this lab use multiple input signals to control the Music Streamer you wrote in lab 3. Unfortunately, our Pynq-Z1 buttons only have 6 basic IO inputs: the 4 buttons and the 2 slide switches. In order to upgrade our capabilities, we have procured hex-keypads.

**NOTE for Spring 2018: This is an experiment! We haven't been able to test the keypads yet, so this part of the lab is optional.** If you would like to install and implement the keypad decoder, you will be able to use it as 16 additional inputs to your later design. If not,

you will have to be selective about which of the input signals you implement. We will have the keypad tested and integrated into later labs anyway.

We will be using the Digilent PmodKYPD, as seen [here](#). Peruse the reference manual online or from the lab resources directory [fpga\\_labs\\_sp18/resources/pmodkypd\\_rm.pdf](#).

## 6.1 PMOD Expansion Headers

Digilent has developed an expansion port for their development boards called **PMOD**. There are two PMOD ports on the Pynq-Z1, whose pins are hardwired to GPIO pins on the FPGA itself. Using PMOD ports is just a matter of plugging in compatible circuits and binding the PMOD-wired FPGA pins in our constraints file, as we've done for other basic I/O on the board. The PMOD header has two power rails, two ground rails, and 8 logic pins.

Read about the PMOD headers in the Pynq-Z1 reference manual [here](#) (or in the PDF in the lab resources directory).

## 6.2 Installing our keypad

The PMOD headers are at the top-right of your board, when it's oriented so you can read the text. Plug the hex key into **PMODA** or **PMODB**. In your project's `PYNQ-Z1_C.xdc` constraints file, uncomment the corresponding pins for your PMOD port of choice and bind them to the `KEYPAD.ROWS/KEYPAD.COLS` signals in `z1top.v`.

## 6.3 Developing a Decoder

Now to make the keypad useful. As you will have read in the reference material, you have to sample the keypad in order to figure out which button was pushed. (We already did some sampling in the Debouncer circuit.) Specifically, you have to cycle through each column pin, driving it low while keeping the other column pin signals high. We then read each row pin: if the pin is low, you have detected a keypress at the current (row, column) position, and should emit its value as the output of our decoder.

You should sample columns at about 1 kHz, or about every 1 ms. After another short delay (about 8 clock cycles) you can check the row pins. Keeping track of which column you're in and which row you're sampling, you can directly output the decoded value.

To help get you up and running faster, you can use Digilent's sample code to understand one way of implementing the interface. We have provided it verbatim in the `lab4/PmodKYPD_Demo` directory (complete with offensive newline characters). Read it for ideas, but do not copy it directly. You have enough experience by now to implement it yourself.

## 7 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  $1/25$ th of a second. Extend the functionality of the `music_streamer` so that one input increases the tempo, one input decreases it, and one resets it to a default value. If you have the hex keypad working, you can map any of the keys to these functions, but make a note of it. If not, use the push-buttons left on the Pynq-Z1 (remembering that `BTN3` is now reserved as a global reset signal).

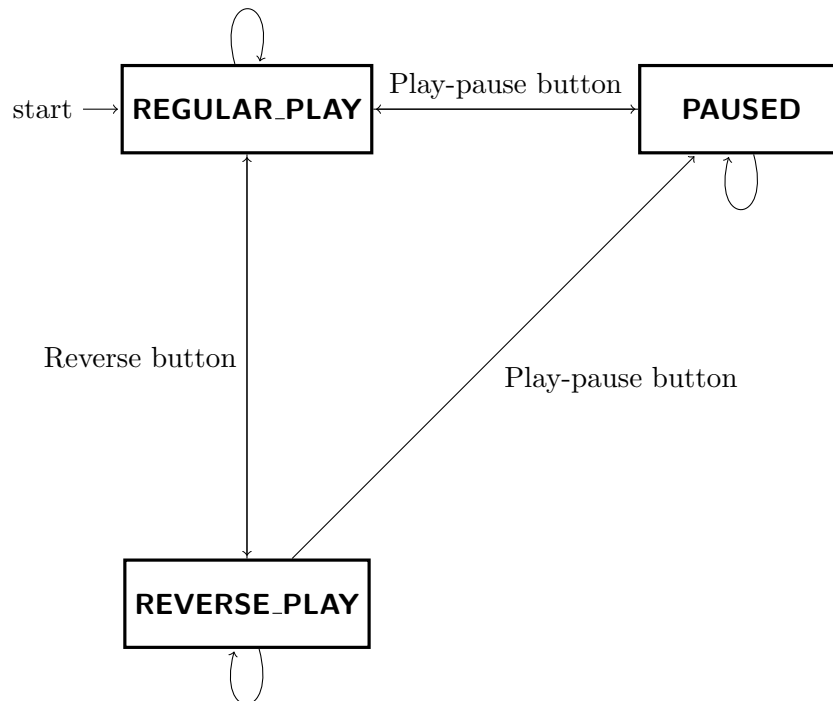
You can implement this by using a register to hold the number of clock cycles per note. Instead of this number being hardcoded in Verilog to represent  $\frac{1}{25}$ th of a second, you can change it at runtime. One input should add or subtract a fixed number from this register, which should alter the 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 inputs. You should be able to speed up and slow down the music you are playing.

## 8 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 one button should transition you into the `PAUSED` state from either the `REGULAR_PLAY` or `REVERSE_PLAY` states. Pressing the same 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 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 a second input 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.

Your `music_streamer` takes in user button inputs that it can use to transition states. If you have a working keypad you will have plenty of inputs to choose from, and you should select new inputs such that the tempo control from the previous section still works (again, remember to note your scheme). If not, you have to be creative: use an edge detector circuit with the slide switches to convert their actuation into a momentary signal (using the switch position as state would be too easy).

Also, drive the LEDs such that they show the current state. An example:

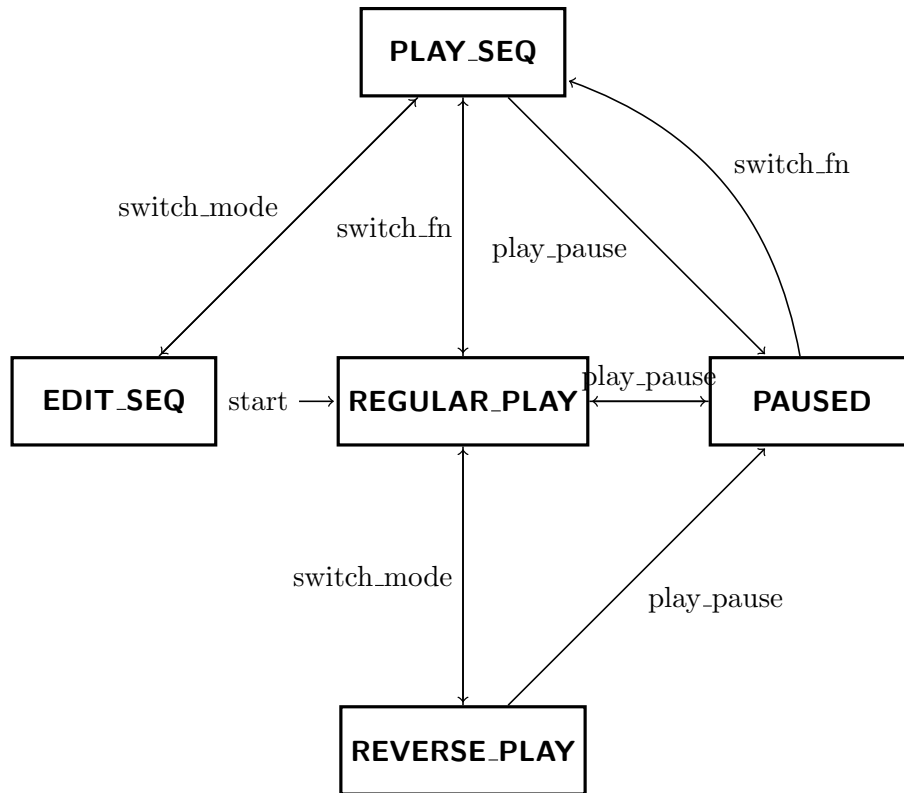
LED	Value
LEDS[5]	<code>current_state == REGULAR_PLAY</code>
LEDS[4]	<code>current_state == PAUSED</code>
LEDS[3]	<code>current_state == REVERSE_PLAY</code>
LEDS[2:0]	0

You can run the testbench in `music_streamer_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.

## 9 Building a Music Sequencer FSM (Optional)

**Note:** This section of the lab relies heavily on additional circuit inputs (e.g. buttons); if you haven't got the keypad input working from Section 6 it will be quite cumbersome. It's recommended that you get the keypad working first, so that you can use it here. Transitions are described in terms of abstract inputs here. It's up to you to map them in a consistent way (remember to record your scheme).

Here is a new state transition diagram for a 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 an LED to `current_state == EDIT_SEQ` and another 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_periods` 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 some input (**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 should show which of the 8 notes we are currently editing. Using the keypad (or a repurposed input), we can select a new pitch for this note. We will also use another input to save the selected pitch in the RAM location for this note. Two more inputs can be used to select between different notes.

You will find some skeleton code in `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 some input.

- (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.
2. The LEDs should be used for the following functions in different states.
    - (a) In the `REGULAR_PLAY`, `PAUSED`, and `REVERSE_PLAY` states, the 6 LEDs should display 6 bits of the rom address (most significant, least significant, or somewhere in the middle, as long as it's useful - you decide).
    - (b) In the `PLAY_SEQ` state, the 6 LEDs should display the note being played. To spare LEDs, you can use binary encoding, so that when note 5 is played the LEDs show 101.
    - (c) In the `EDIT_SEQ` state, the 6 LEDs should display the note being edited. As before, binary encoding is fine.
  3. In the `EDIT_SEQ` state, the speaker should play the current pitch of the note. You should have inputs which increase and decrease the pitch by some amount per click. Another input should save the current pitch of the note into the sequencer's RAM.
  4. In the `EDIT_SEQ` state, you should have inputs which change the note being edited. As you change notes, the 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.

**Hint:** If you have the hex keypad working, you can use it as a series of 16 new input buttons, *or* with the shift register as a mean to directly input the tone values!

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

## 10 Checkoff

1. (Optional) Show the TA that that you can decode hex keypad presses correctly.
2. Show the TA your working design with the FSM. Be able to transition states by clicking on either the hex keypad or using the slide switches, and show that your `music_streamer` matches the spec.
3. Show the tempo control working by using the push buttons to speed up and slow down the music.
4. Demonstrate that hitting the the reset button resets the ROM address back to 0 and puts the FSM into the `REGULAR_PLAY` state.
5. (Optional) Demonstrate that you can transition into the `SEQUENCER` state and that you can edit your tones and play them back.

6. Show the TA your Verilog RTL for all the components you designed for this lab (synchronizer, debouncer, shift register, FSM) and briefly explain the design of each of them.