EECS 151/251A FPGA Lab 4: ROMs and IO Circuits

Prof. Borivoje Nikolic and Prof. Sophia Shao
TAs: Cem Yalcin, Rebekah Zhao, Ryan Kaveh, Vighnesh Iyer
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

1 Before You Start This Lab

Run `git pull` in `fpga_labs_fa19`.

Review these documents that will help you better understand some concepts we will be covering.

1. verilog_fsm.pdf - constructing FSMs in Verilog.
2. Debouncer Circuit

   Read the "What is Switch Bounce" section about why we need a debouncer circuit. Read the "Digital Switch Debouncing" section about the implementation of a debouncer circuit.

2 Lab Overview

In this lab, we will

- use a ROM to store a melody and play it on the FPGA using the `tone_generator`
- build 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
- verify the conditioning circuits are working correctly using the FPGA LEDs
- use synchronous resets to reset our circuits to a known initial state
- create a basic FSM in the `music_streamer` that uses the buttons to change states and alter the music playback

3 Using an Asynchronous ROM to Build the Music Streamer

An asynchronous memory is a memory block that isn't governed by a clock. In this lab, we will use a Python script to generate a Verilog ROM block.

A ROM is a read-only memory. The ROM's data can be accessed by supplying an address; after some time, the ROM will output the data stored at that address. The number of addresses in a

memory is called its **depth**, while the number of bits stored per address is called the **width** of the memory.

The synthesizer takes the Verilog you write and converts it into a low-level netlist which uses only the primitives available on the FPGA. Our Verilog **describes** the functionality of some digital circuit and the synthesizer **infers** what primitives implement the functional description. In this section, we will examine the Verilog that allows the synthesizer to infer a ROM. This is a minimal example of a ROM in Verilog: (depth of 8 entries/addresses, width of 12 bits)

```
module rom (input [2:0] address, output reg [11:0] data);
  always @(*) begin
    case(address)
      3'd0: data = 12'h000;
      3'd1: data = 12'hFFF;
      3'd2: data = 12'hACD;
      3'd3: data = 12'h122;
      3'd4: data = 12'h347;
      3'd5: data = 12'h93A;
      3'd6: data = 12'h0AF;
      3'd7: data = 12'hC2B;
    endcase
  end
endmodule
```

To power our `tone_generator`, we will be using a ROM that is X entries/addresses deep and 24 bits wide. The ROM will contain notes that the `tone_generator` will play. You can choose the depth of your ROM based on how long the music is.

We've provided you with a few scripts that can generate a ROM from either a file with it's contents or even from sheet music. Run these commands from `lab4/`.

```
python3 scripts/musicxml_parser.py musicxml/Twinkle_Twinkle_Little_Star.mxl music.txt
python3 scripts/rom_generator.py music.txt src/rom.v 1024 24
```

The first script will parse a MusicXML file and turn it into a list of `tone_switch_periods` for each of the notes for a piece of sheet music. The second script will take that list and turn it into a ROM that's 1024 entries deep with a width of 24 bits.

Look at `music.txt` and `src/rom.v`. The `last_address` port of the ROM outputs the last address that contains a valid note.

You can download your own music in MusicXML format from here (`https://musescore.org/`) and run it through the same parser; it should ideally only have one part to work properly. You can also directly edit the `music.txt` file to customize the contents of the ROM as you wish.

4 Designing the music_streamer

Open up the `music_streamer.v` file. You should instantiate the ROM you created earlier and add logic to address the ROM sequentially. The `music_streamer` will play each note in the ROM for

a predefined amount of time by sending it to the `tone_generator` via the `tone` output port. You should play each note for **1/25th of a second**.

In this section, only concern yourself with the `clk` input and the `tone` output; leave the other inputs unused.

Next we'll cover some testbench constructs and test the `music_streamer`.

5 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. See Discussion 3 for more slides and example code.

- `@(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, counter: %b",
  ↪   tone_gen.counter);
  ```

6 Simulating the music_streamer

To simulate your `music_streamer` open up `lab4/src/music_streamer_testbench.v`. Note how the `tone_generator` is wired to the `music_streamer`. This test just runs the `music_streamer` for a second.

Run the test with `make sim/music_streamer_testbench.vpd` for VCS or `make sim/music_streamer_testbench.fst` for Icarus Verilog.

Inspect the waveform and verify that there are no undefined signals (red lines/X). Run the Python

script `python3 scripts/audio_to_sim.py sim/output.txt` to generate a `.wav` file and listen to the `music_streamer`. It should sound like the first second of the song that was loaded on the ROM.

6.1 Verify your Code Works For Rest Notes

In simulation, you can often catch bugs that would be difficult or impossible to catch by running your circuit on the FPGA. You should verify that if your ROM contains an entry that is zero (i.e. generate a 0Hz wave), that the `tone_generator` holds the `square_wave_out` output at either 1 or 0 with no oscillation.

Question 1: Music Streamer Simulation

a) Save a waveform screenshot demonstrating that a `tone_switch_period` of 0 sent by the ROM results in no oscillation of the `square_wave_out` output.

b) Save an audio file from the `music_streamer` simulation for checkoff.

6.2 FPGA Time

Instantiate the `music_streamer` and `tone_generator` in `src/z1top.v` similar to the `music_streamer_testbench`. Connect `SWITCHES[0]` to the `tone_generator`'s `output_enable` and wire the `volume` input to 0. Run `make impl` and `make program` and hear the music played from the FPGA.

7 Input Conditioning Circuits

We want to use the buttons on the Pynq board to control the playback of the music by adjusting the tempo, or pausing the music and playing it in reverse. To safely use the button signals, we have to design input conditioning circuits to handle metastability and button bounce.

7.1 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.

The signals coming from the push buttons and slide switches on the Pynq board don't have an associated clock (asynchronous). When the button signals are put through a register, its hold or setup time may be violated. This may put that register into a *metastable* state (Figure 1).

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. An asynchronous signal could violate those constraints, and cause a 'mid-rail' voltage from a register to propagate to other logic elements. This 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.

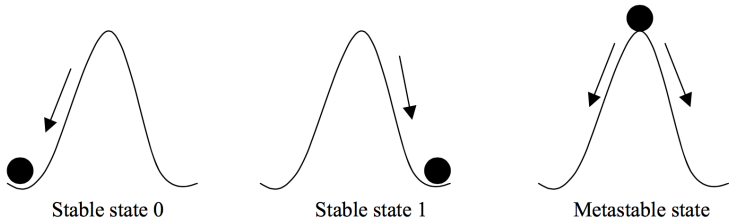


Figure 1: The 'ball on a hill' metaphor for metastability. If a register's timing constraints are violated, its output voltage oscillates and after some time unpredictably settles to a stable state.

This synchronizer circuit for this lab is relatively simple (Figure 2). For synchronizing one bit, it is a pair of flip-flops connected serially.

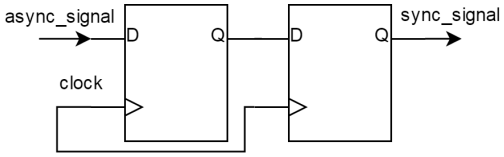


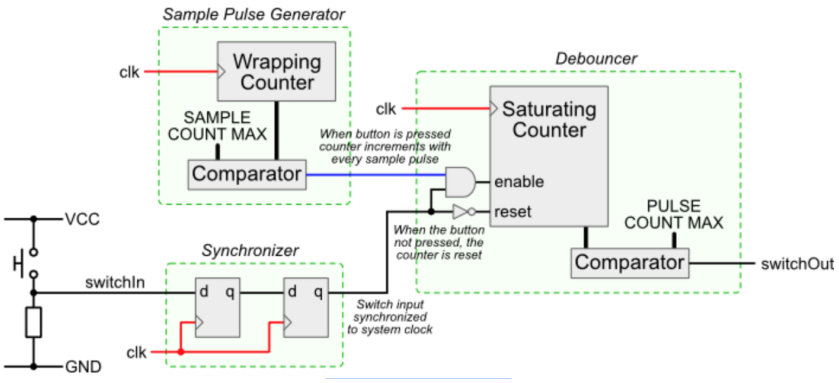
Figure 2: 1-bit 2 Flip-Flop Synchronizer

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

7.1.1 Synchronizer Simulation

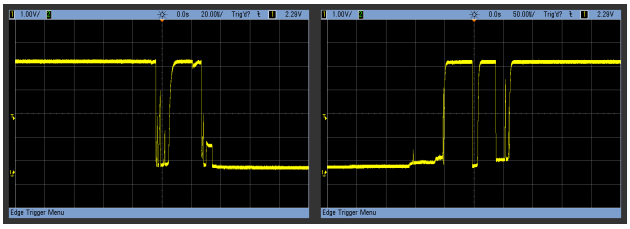
A test is provided in `sim/sync_testbench.v`. Run it as usual `makesim/sync_testbench.vpd`. For details on the constructs/techniques/syntax used in this testbench, refer to Section 5.

7.2 Debouncer



Recall this graphic from the prelab debouncer reading.

The debouncer circuit takes a button's glitchy digital signal and outputs 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 or depress a button, the signal doesn't behave like a perfect step function. Instead the button signal is glitchy due to mechanical "bounce". If we naively used the button signal directly there would be many spurious "button presses".

Look at `src/debouncer.v`. This is a parameterized debouncer which can debounce `width` signals at a time. 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`. If the sample pulse is high at a clock edge, increment the counter if the input signal is also high, else reset the counter to 0. Once the saturating counter reaches `pulse_count_max`, it should hold that value indefinitely until the sampled input signal becomes 0. The `debounced_signal` of your debouncer should be an equality check between the saturating counter and `pulse_count_max`.

You can 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 may 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
```

Implement the debouncer.

7.2.1 Debouncer Simulation

A testbench has been provided in `sim/debouncer_testbench.v`. Make sure you understand what the testbench is doing. Run it as usual.

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.

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
```

7.3 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 cycle wide pulse that the rest of our digital design can use.

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

7.3.1 Edge Detector Simulation

A testbench is provided in `edge_detector_testbench.v`. Run as usual.

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.

7.4 Input Conditioning Circuits on the FPGA

Look at `src/button_parser.v` which combines the synchronizer, debouncer, and edge detector in a chain. A test of the `button_parser` is in `src/z1top.v` which contains a 4-bit `count` register which is incremented/decremented by pressing buttons. The LEDs show the current value of `count`.

Run `make lint` and `make program` and try the circuit on the FPGA. Make sure to check synthesis warnings in `build/synth/synth.log`. Fix any that are unexpected. Check that each button performs the right action on the FPGA.

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 as below so that our simulation would have defined signals.

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

or

```
initial 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
    counter <= 24'd0;
  end
  else begin
    counter <= counter + 24'd1;
  end
end
```

Unlike what we did before, this Verilog is synthesizable for all deployment targets, FPGAs, ASICs, and CPLDs. Modify the `tone_generator` and `music_streamer` to use the provided `rst` signal instead of initial assignemnts.

Try rerunning the `music_streamer_testbench`, which pulses the `rst` signal, and verify using the waveform that there are no Xs.

Finally connect `buttons_pressed[0]` to the `music_streamer` and `tone_generator` `rst` input in `z1top.v`. Program the FPGA and verify the reset functionality.

9 Music Streamer Tempo Control

Let's use the new user inputs we now have access to. The `music_streamer` plays each tone in the ROM for 1/25th of a second.

Use the `tempo_up` and `tempo_down` inputs to the `music_streamer` to decrease and increase respectively the time each tone in the ROM is played.

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. You can add or subtract a fixed number from this register, which should alter the time each tone

is played. You get to choose this number.

Next in `z1top.v`, delete the `button_parser` test circuit and connect the `music_streamer` as such:

- `tempo_up = SWITCHES[1] & buttons_pressed[1]`
- `tempo_down = !SWITCHES[1] & buttons_pressed[1]`

Try it out on the FPGA and verify that you have control of your `music_streamer`'s tempo using the buttons.

10 Checkoff

1. Show the RTL for the input conditioning circuits and music streamer
2. Play an audio file generated from a simulation
3. Demonstrate the music streamer on the FPGA (tempo control, reset)

10.1 Lab Report

Submit a short report containing the waveform screenshots to answer the questions in the lab.

Ackowlegement

This lab is the result of the work of many EECS151/251 GSIs over the years including:

- Sp12: James Parker, Daiwei Li, Shaoyi Cheng
- Sp13: Shaoyi Cheng, Vincent Lee
- Fa14: Simon Scott, Ian Juch
- Fa15: James Martin
- Fa16: Vighnesh Iyer
- Fa17: George Alexandrov, Vighnesh Iyer, Nathan Narevsky
- Sp18: Arya Reais-Parsi, Taehwan Kim
- Fa18: Ali Moin, George Alexandrov, Andy Zhou