# EECS 151/251A FPGA Lab 3:
# Tone Generator, Simulation, Connecting Modules, and Memories

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

Before you proceed with the contents of this lab, we suggest that you look through these two documents that will help you better understand some Verilog constructs.

1. wire_vs_reg.pdf - The differences between wire and reg nets and when to use each of them.
2. always_at_blocks.pdf - Understanding the differences between the two types of always @ blocks and what they synthesize to.

## 2 Designing a Tone Generator

Let's create a tone generator/buzzer on the FPGA.

### 2.1 Clock Sources

Look at the Pynq Reference Manual. Read Section 11 about the clock sources available on the Pynq. We are using the 125 MHz clock from the Ethernet PHY IC on the Pynq board connected to pin H12 of the FPGA chip.

Look at the `src/z1top.v` top-level module and its `CLK_125MHZ_FPGA` input.

```
module z1top (
    input CLK_125MHZ_FPGA,
    ...
);
```

Next take a look at the constraints in `src/PYNQ-Z1_C.xdc` and notice how the LOC for the clock net is set to H16, just as specified in the Pynq-Z1 Reference Manual.

```
set_property -dict { PACKAGE_PIN H16 IOSTANDARD LVCMOS33 } \
  [get_ports { CLK_125MHZ_FPGA }];
```

This is how we map top-level signals in Verilog to the physical FPGA pins they are connected to. We can access the clock signal from our Verilog top-level module and can propagate this clock signal to any submodules that may need it.

### 2.2 Audio Out

Look at Section 14 of the Pynq Reference Manual which describes the mono audio out feature on the Pynq board.

The FPGA pin R18 is connected to the `AUD_PWM` net. The FPGA can drive this net with a PWM signal which goes through a low-pass filter and is driven into the audio jack on the Pynq board.

There's also an `AUD_SD` net connected to FPGA pin T17, which turns off the opamps in the low-pass filter. Setting `AUD_SD` to 1 enables the audio output.

Find these signals in the `src/PYNQ-Z1_C.xdc` file, and note how they appear in the `src/z1top.v` port list.

### 2.3 Generating a Square Wave

Let's play a 220 Hz square wave out of the Mono Audio Out port on the Pynq. The square wave should have a 50% duty cycle, so for half of the period of one wave the signal should be high and for the other half, the signal should be low. We have a 125 MHz clock we can use to time our circuit.

Find the following:

**Question 1: Square Wave Calculations**

a) The period of our clock signal (frequency = 125 MHz)?

b) The period of a 220 Hz square wave?

c) How many clock cycles fit in one period of the square wave?

Open `src/tone_generator.v` and design a circuit to output a 220Hz square wave on the `square_wave_out` output. Ignore the `tone_switch_period`, `output_enable`, and `volume` inputs for now.

## 3 Simulating the Tone Generator

Let's run some simulations on the `tone_generator` in software to check it works before putting it on the FPGA. To do this, we will need to use a Verilog testbench. A Verilog testbench is designed to test a Verilog module by supplying it with the inputs it needs (stimulus signals) and testing whether the outputs of the module match what we expect.

### 3.1 Overview of Testbench Skeleton

Check the provided testbench skeleton in `sim/tone_generator_testbench.v`. Let's go through what every line of this testbench does.

```
`timescale 1ns/1ns
```

The timescale declaration needs to be at the top of every testbench file.

```
`timescale (simulation step time)/(simulation resolution)
```

The first argument to the timescale declaration is the simulation step time. It defines the granularity of discrete time units in which the simulation advances. In this case, we have defined the simulation step time to be one nanosecond, so we can advance the simulation time by as little as 1ns at a time.

The second argument to the timescale declaration is the simulation resolution. In our example it is also 1ns. The resolution allows the simulator to model transient behavior of your circuit in between simulation time steps. For this lab, we aren't modeling any gate delays, so the resolution can safely equal the step time.

```
`define SECOND 1000000000
`define MS 1000000
// The SAMPLE_PERIOD corresponds to a 44.1 kHz sampling rate
`define SAMPLE_PERIOD 22675.7
```

These are some macros defined for our testbench. They are constant values you can use when writing your testbench to simplify your code and make it obvious what certain numbers mean. For example, `SECOND` is defined as the number of nanoseconds in one second. The `SAMPLE_PERIOD` is the sampling period used to sample the square wave output of the `tone_generator` at a standard 44.1 kHz sample rate.

```
module tone_generator_testbench();
  // Testbench code goes here
endmodule
```

`tone_generator_testbench` is a testbench module. It is not intended to be placed on an FPGA, but rather it is to be run by a circuit simulator. All your testbench code goes in this module. We will instantiate our DUT (device under test) in this module.

```
reg clock;
reg output_enable;
reg volume = 0;
reg [23:0] tone_to_play;
wire sq_wave;
```

Here are the inputs and outputs of our `tone_generator`. Notice that the inputs to the `tone_generator` are declared as `reg` type nets and the outputs are declared as `wire` type nets. This is because we will be driving the inputs in our testbench inside an `initial` block and we will be reading the output. Note we can set the initial value of `reg` nets in the testbench to drive a particular value into the DUT at time 0 (e.g. `volume`).

```
initial clock = 0;
always #(4) clock <= ~clock;
```

This is the clock generation code. The clock signal needs to be generated in our testbench so it can be fed to the DUT. The initial statement sets the value of the clock net to 0 at the very start of the simulation. The next line toggles the clock signal every 4ns, i.e. half period of 125 MHz clock.

```
tone_generator audio_controller (
```

```
    .clk(clock),
    .output_enable(output_enable),
    .tone_switch_period(tone_to_play),
    .volume(volume),
    .square_wave_out(sq_wave)
);
```

Now we instantiate the DUT and connect its ports to the nets we have declared in our testbench.

```
initial begin
    tone_to_play = 24'd0;
    output_enable = 1'b0;
    #(10 * `MS);
    output_enable = 1'b1;

    tone_to_play = 24'd37500;
    #(200 * `MS);
    ...
    $finish();
end
```

This is the body of our testbench. The `initial begin ... end` block is the 'main()' function for our testbench, and where the simulation begins execution. In the `initial` block we drive the DUT inputs using blocking (`=`) assignments.

We can also order the simulator to advance simulation time using delay statements. A delay statement takes the form `#(delay in time steps);`. For instance the statement `#(100);` would run the simulation for 100ns.

In this case, we set `output_enable` to 0 at the start of the simulation, let the simulation run for 10ms, then set `output_enable` to 1. Then `tone_to_play` is changed several times, and the `tone_generator` is given some time to produce the various tones.

The final statement is a system function: the `$finish()` function tells the simulator to halt the simulation.

```
integer file;
initial begin
  file = $fopen("output.txt", "w");
  forever begin
    $fwrite(file, "%h\n", sq_wave);
    #(`SAMPLE_PERIOD);
  end
end
```

This piece of code is written in a separate `initial begin ... end` block. The simulator treats both `initial` blocks as separate threads that both start execution at the beginning of the simulation and run in parallel.

This block of code uses two system functions `$fopen()` and `$fwrite()`, that allow us to write

to a file. The `forever begin` construct tells the simulator to run the chunk of code inside it continuously until the simulation ends.

In the `forever begin` block, we sample the `square_wave_out` output of the `tone_generator` and save it in `output.txt`. We sample this value every `` `SAMPLE_PERIOD `` nanoseconds which corresponds to a 44.1 kHz sampling rate. The `tone_generator`'s output is stored as 1s and 0s in `output.txt` that can be converted to an audio file to hear how your circuit will sound when deployed on the FPGA.

### 3.2 Running the Simulation

There are 3 RTL simulators we can use.

#### 3.2.1 VCS

If you're using the lab machines, you should use VCS:

```
make sim/tone_generator_testbench.vpd
```

This will generate a waveform file `sim/tone_generator_testbench.vpd` which you can view using `dve`. Login to the lab machines physically or use X2go and run:

```
dve sim/tone_generator_testbench.vpd &
```

### 3.3 A Choice of Simulators

As you've realised, the steps in the design and synthesis toolchain are independent programs with agreed interfaces for data flow. When it comes to simulating an HDL description of some design, several commercially available simulation tools exist. In this lab we will consider two:

1. **Vivado Design Suite** has an integrated simulator, **xsim**, that can run your Verilog testbench directly. You will have seen in previous labs the *Simulation* category in *Flow Navigator* to the left of the Design Suite application window.

2. **ModelSim** is a third-party application by Mentor Graphics. It can also synthesise a simulation of a given Verilog testbench (and its dependent modules), but is a more powerful standalone product.

Both simulators have a lot in common. They both allow you to view the waveforms of various signals in your design as a simulation, which you control, progresses. They're both driven by Tcl scripts (below), though you can also use a GUI to tweak their behaviour. Their GUIs have similar elements.

To avoid using the Vivado ISE simulator past years' labs have relied on the external ModelSim application, which is also used by the ASIC labs. For our purposes, the newer Vivado Design Suite simulator is actually pretty good. It's also built-in to the free WebPACK version of Vivado, whereas ModelSim requires a second license - a pain if you want to run simulations locally, at home. We will still emphasise ModelSim, however to help give you experience in a sandbox environment. It'll be useful to get a feel for the Vivado integrated simulator too, as a back up should you ever need

it. Understanding the features common to both or unique to one should make you more confident approaching unknown tools out in the wild.

We provide a `sim` directory in the `lab3` folder to run the ModelSim simulation from. (Vivado can also be configured to launch ModelSim directly - you can try and set that up if you want.) The next sections will introduce you to ModelSim before comparing it to the Vivado integrated option.

## 3.4 ModelSim

### 3.4.1 Using TCL scripts for ModelSim (.do files)

**ModelSim** takes commands from TCL scripts. Take a look at the `lab3/sim/tests/tone_generator_testbench.do` TCL script. Here is a quick description of what is instructs our simulator to do.

```
start tone_generator_testbench
add wave tone_generator_testbench/*
add wave tone_generator_testbench/audio_controller/*
run 10000ms
```

We begin by issuing the `start` command to the simulator. This instructs the simulator to scan a list of Verilog source files provided to it to find a module named `tone_generator_testbench`. This module name must exactly match the module name of your top-level testbench module. The simulator loads and elaborates this module so that its ready to simulate/execute.

The two `add_wave` commands are important. By default, the simulator will not log the signals in our testbench or DUT as the simulation executes. The `add wave tone_generator_testbench/*` line tells the simulator to log all signals directly inside in the `tone_generator_testbench` module. The second line tells the simulator to log the signals in a submodule of the top-level testbench module. Observe that `audio_controller` is the instance name of the `tone_generator` instance in the testbench module.

Finally, the `run (time)` command tells the simulator to jump to the `initial begin` blocks in the testbench and actually run the simulation. The time value (in our case 10000ms = 10s) gives the simulator an upper bound on the simulation time. The simulator will simulate for 10 seconds before timing out. If the simulator hits the `$finish()` function before the 10 second timeout is up, it will stop simulation instantly.

### 3.4.2 Running ModelSim

With all the details out of the way, let's actually run a simulation. Go to the `lab3/sim` directory and run `make CASES=tests/tone_generator_testbench.do`. After a minute or so, the simulation will finish.

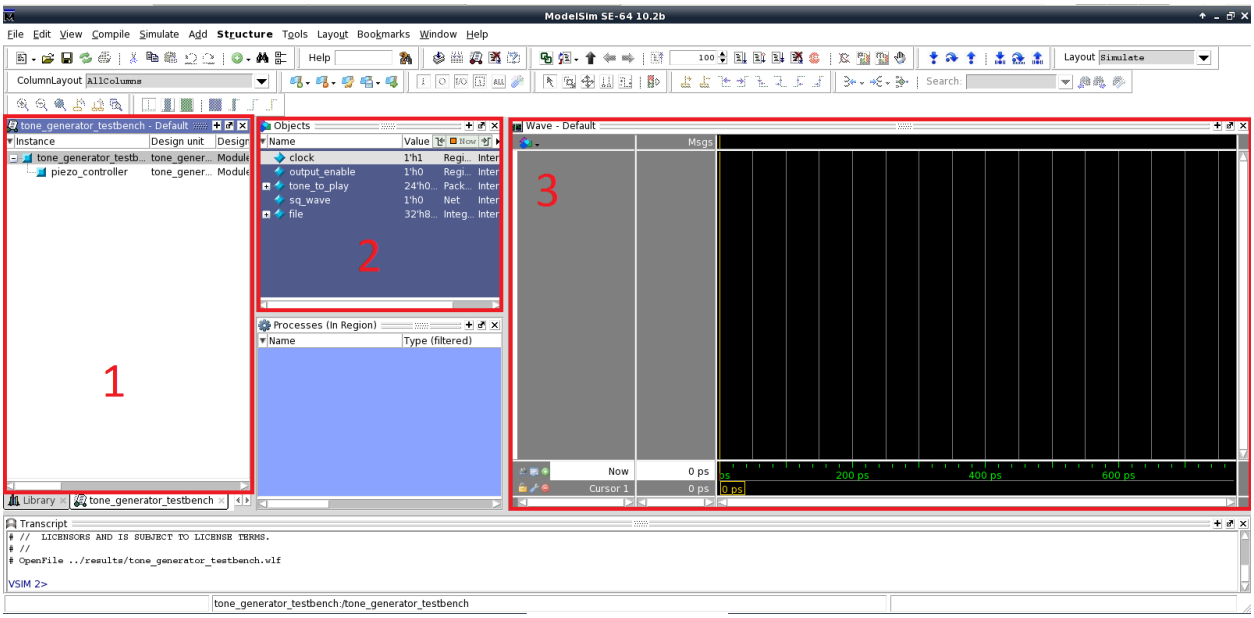
### 3.4.3 Viewing Waveforms

Let's take a look at the data that the simulator collected. Run the viewwave script like this:

```
./viewwave results/tone_generator_testbench.wlf &
```

The results of the simulation and the logged signals are stored in a `.wlf` file. This command should open that file in the ModelSim Wave Viewer.

You should see a window like this:




Let's go over the basics of ModelSim. The boxed screens are:

1. **Module Window** - List of the modules involved in the testbench. You can select one of these to have its signals show up in the object window.

2. **Object Window** - List of all the wires and regs in the selected module. You can add signals to the waveform view by selecting them, right-clicking, and doing `Add Wave`.

3. **Waveform Viewer** - The signals that you add from the object window show up here. You can navigate the waves by searching for specific values or going forward or backward one transition at a time. The x-axis represents time.

You may not see the **Waveform Viewer** when you first open ModelSim. To add signals to view, right click on the signal in the **Object Window**, and click on `Add Wave`. Add the `clock, output_enable,` and `sq_wave` signals to the waveform viewer.

Here are a few useful shortcuts:

- **Click on waveform**: Sets cursor position
- **O**: Zoom out of waveform
- **I**: Zoom into waveform
- **F**: Fit entire waveform into viewer (zoom full)

- **C**: Zoom in on cursor position
- **Middle Click + Drag Left/Right**: Zoom in on waveform section
- **Middle Click + Drag to Top Right**: Zoom out from current waveform section

You should play with these shortcuts for a few minutes. Now, zoom to fit the entire waveform in your viewer.

You should be able to see the clock oscillate at the frequency specified in the testbench. You should also see the `output_enable` signal start at 0 and then become 1 after 10 ms. However, you may see that the `sq_wave` signal is just a red line. What's going on?

### 3.4.4 Fixing the Undefined clock_counter

Blue lines in ModelSim indicate high-impedance (unconnected) signals. High-impedance is specified in Verilog as the letter `z`. We won't be using high-impedance signals in our designs, so blue lines in ModelSim indicate something in our testbench isn't wired up properly.

If you have a red line for your `clock_counter` at the start of your simulation, it may be because the initial value sitting inside the `clock_counter` register is unknown. It could be anything! Since we don't have an explicit reset signal for our circuit to bring the `clock_counter` to a defined value, it may be unknown for the entire simulation.

Let's fix this. In the future we will use a reset signal, but for now let's use a simpler technique. In `lab2/src/tone_generator.v` modify this line as such:

```
// Original code:
reg [x:0] clock_counter;

// Change to:
reg [x:0] clock_counter = 0;
```

This tells the simulator that the initial value for this register should be 0. For this lab, when you add new registers in your `tone_generator` or any other design module, you should instantiate them to their initial value in the same way. **Do not set an initial value for a 'wire' type net; it will cause issues with synthesis, and may cause X's in simulation.**

Now run the simulation again.

### 3.4.5 Helpful Tip: Reloading ModelSim .wlf

When you re-run your simulation and you want to plot the newly generated signals in ModelSim, you don't need to close and reopen ModelSim. Instead click on the 'Reload' button on the top toolbar which is to the right of the 'Save' button.

#### 3.4.6 Listen to Your Square Wave Output

Take a look at the file written by the testbench located at `lab3/sim/build/output.txt`. It should be a sequence of 1s and 0s that represent the output of your `tone_generator` sampled at 44.1 kHz. Use a Python script that can take this file and generate a `.wav` file that you can listen to.

Go to the `lab3/` directory and run the command:

```
python scripts/audio_from_sim.py sim/build/output.txt
```

This will generate a file called `output.wav`. Run this command to play it:

```
play output.wav
```

If `play` doesn't work, try running `aplay output.wav`.

You should hear 5 tones, played rapidly one after the other that have descending frequencies.

#### 3.4.7 Playing with the Testbench

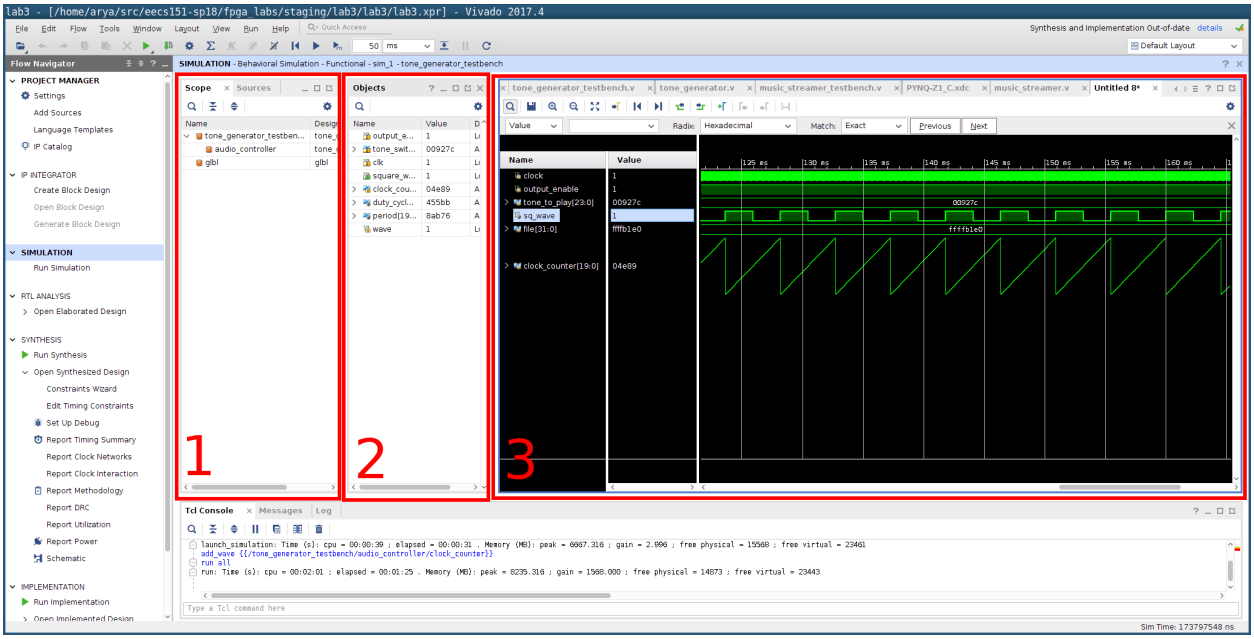
Play around with the testbench by altering the clock frequency, changing when you turn on `output_enable` and verifying that you get the audio you expect. For checkoff be able to answer the following questions and demonstrate understanding of simulation:

1. If you increase the clock frequency from 125 Mhz, would you expect the tones generated by your `tone_generator` to be of a higher or lower frequency than was generated with the 125 MHz clock? Why? Show audio evidence of this using simulation.
2. Prove that the `output_enable` input of your `tone_generator` actually works in simulation.
3. Create a testbench that plays some simple melody that you define and have its audio output file ready for checkoff.

### 3.5 Vivado's Integrated Simulator

Inside a Vivado project, expand the *Simulation Sources* category in your *Project Hierarchy* window (that's the one with all your files). At any given time only one Verilog module can be the simulation "Top", just as there can only be one synthesis "Top" for implementation. If it isn't already, right-click `tone_generator_testbench` and select *Set As Top*. Then in the *Flow Navigator* window, click *Run Simulation* in the *Simulation* category. You will get a context menu: select *Run Behavioural Simulation*.

Then you will be presented with the Vivado Simulation window:




This should be familiar to you: it looks and behaves like the ModelSim GUI you saw earlier.

1. is the **Scope Window**, which is like ModelSim's **Module Window**, and lets you select from among simulated modules;

2. is an **Object Window** just like ModelSim's, allowing you to select waveforms to view; and

3. is the **Waveform Viewer**.

Play around with it; does your intuition from ModelSim carry over?

Although the Vivado simulator can be controlled with a Tcl script like ModelSim, there is a GUI way to modify settings too. Click *Tools* menu → *Settings* to bring up the *Project Settings* dialog. Select the *Simulation* category in the left pane. (Notice that you can change the default simulator here from Vivado to, say, ModelSim.) The options defined in various tabs in the right hand pane allow you to select various simulation parameters, including initial duration.

A final note: when the *Simulation* window is open in Vivado, you can use the *Tcl Console* to execute simulation commands. (The *Tcl Console* appears just below boxes (1) - (3) in the figure above.) A lot of the commands you can use here will work like they do for ModelSim. The full list of commands is available in the user manual from Xilinx.

### 3.5.1 Where does Vivado put output.txt?

The Vivado simulator will also have produced an `output.txt` file with waveform samples. But where? In your project root directory, use `find` to *find* out:

```
lab3 $ find . -name "output.txt"
./lab3/lab3.sim/sim_1/behav/xsim/output.txt
```

### 3.6 Resetting Simulations

Running simulations, both for ModelSim and in Vivado, generates a lot of signal data for the waveforms viewers. You often don't need to keep this intermediary data (which is easily several gigabytes in this lab alone), and you may be running out of quota on the machine you're using. In order to clear intermediate state and reset the simulation, perform the following steps.

1. For **ModelSim**, navigate to the `sim` directory in which you executed `make CASES=`, e.g. **fpga_labs_fa19/lab3/sim**. There, execute `make clean`.
2. For **Vivado**, open the *Tcl Console*. It's usually available toward the bottom-left of the application window, but can also be opened from the menus. In the Tcl Console, execute `reset_simulation`.

## 4 Top-Level Wiring and Tone Generator on the FPGA

Then open `src/z1top.v` and instantiate the `tone_generator`. Connect `square_wave_out` to `aud_pwm`. Set `aud_sd` to 1, to enable the audio output.

Build a bitstream by running `make impl` in `fpga_labs_fa19/lab3`. Check for any warnings or errors in `build/synth/latest/synth.log` and try to fix them. Ask a TA if you need help here.

Program the FPGA by running `make program`. Plug in headphones and make sure you hear a buzzing noise at 220Hz; compare the tone with a reference generator. To stop the buzzing, you can press the `SRST` button on the top-right of the Pynq.

## 5 Enhancements

### 5.1 Switching the Wave On and Off

Now you have a tone, but it can't be toggled on and off without pulling the power to the FPGA board. Let's use the `output_enable` input of the `tone_generator` module to gate the square wave output. When `output_enable` is 0, you should pass 0 to the `square_wave_out` output, but when `output_enable` is 1, you should pass your square wave to `square_wave_out`.

Wire up the `output_enable` signal to the first slide switch (`SWITCHES[0]`) in `z1top`.

Run your design flow. Check for any warnings or errors and try to fix them. Ask a TA if you need help here. When everything looks good, program the board through the hardware manager. You should now hear a buzzing noise at 220Hz that can be turned on or off by toggling the first slide switch.

You should verify that the tone is indeed 220 Hz by comparing it to a reference tone here: `http://onlinetonegenerator.com/`.

### 5.2 Volume Adjustment

### 5.3 Design a Configurable Frequency tone_generator

Let's extend our `tone_generator` so that it can play different notes. You may start by adding a 24-bit input to the `tone_generator` called `tone_switch_period`. Note you will also have to modify your `clock_counter` to be 24 bits wide.

The `tone_switch_period` describes how often the square wave output switches from high to low or low to high. For example a `tone_switch_period` of 284091 (0 d.p.) tells us to invert the square wave output every 142045 clock cycles, which for a 125 Mhz clock translates to a $\sim 440$ Hz square wave. Here is the derivation:

$$\frac{125 \times 10^6 \text{ cycles}}{1 \text{ second}} \div \frac{440 \text{ periods}}{1 \text{ second}} = \frac{284091 \text{ cycles}}{1 \text{ period}}$$
$$284091 \text{ cycles/period} \rightarrow 142045 \text{ cycles/half-period}$$

You may have to modify the architecture of your `tone_generator` to accommodate this new input signal. You should reset the internal `clock_counter` every `tone_switch_period` cycles and should also invert the square wave output. Remember to initialize any new registers declared in your `tone_generator` to their desired initial value to prevent unknowns during simulation.

You should also handle the case when `tone_switch_period` is 0. In this case disable the tone output.

#### 5.3.1 Try the Configurable Frequency tone_generator on the FPGA

Modify the top-level Verilog module `z1top.v` to include the new input to the `tone_generator`. You should tie the `tone_switch_period` to `SWITCHES[0]` and `BUTTONS[3:0]`, left-shifted by 9 bits (effectively a multiplication by 512). This will allow you to control the `tone_switch_period` from 512 to 15872. Leave `SWITCHES[1]` to control `output_enable` initially; later, you can use it as an extra bit's worth of input. Here is a code snippet (incomplete):

```
tone_generator audio_controller (
    .output_enable(SWITCHES[1]),
    .tone_switch_period({18'd0, SWITCHES[0], BUTTONS[3:0]} << 9),
);
```

What other way(s) do you have to digitally mute your output signal?

Is the width of the bus assigned to `tone_switch_period` correct? Does it matter?

Run the usual synthesis, implementation and programming flow to put your new `tone_generator` on the FPGA. Verify that toggling the switches and buttons changes the frequency of your `tone_generator`.

# 6 Introduction to Inferred Asynchronous ROMs

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 ROM block in Verilog.

A ROM is a read-only memory. This data can be accessed by supplying an address to the ROM; after some time, the ROM will output the data stored at that address. A memory block in general can contain as many addresses in which to store data as you desire. Every address should contain the same amount of data (bits). The number of addresses is called the **depth** of the memory, 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 of the structures are actually used 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 tones that the `tone_generator` will play. You can choose the depth of your ROM based on the length of the sequence of tones you want to play.

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 `lab3/`.

```
python scripts/musicxml_parser.py musicxml/Twinkle_Twinkle_Little_Star.mxl music.txt
python scripts/rom_generator.py music.txt ./lab3.srcs/sources_1/new/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.

Take a look at `music.txt` and `src/rom.v`. 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.

# 7 Design of the music_streamer

Open up the `music_streamer.v` file. You will need to modify this module to contain an instance of the ROM you created earlier and logic to address the ROM sequentially to play notes. The `music_streamer` will play each note in the ROM for a predefined amount of time by sending it to the `tone_generator`.

We will play each note for 1/25th of a second. Calculate what that is in terms of 125Mhz clock cycles.

Now let's begin the design of the `music_streamer` itself. Instantiate your ROM in the `music_streamer` and connect the ROM's `address` and `data` ports to wire or reg nets that you create in your module. The `last_address` port outputs the last address in the ROM (depth).

Next, write the RTL that will increment the address supplied to the ROM every **1/25th of a second**. The data coming out of the ROM should be fed to the `tone` output. The ROM's address input should go from 0 to the depth of the ROM and should then loop around back to 0. You don't have a reset signal, so define the initial state of any registers in your design for simulation purposes. Also hook up the `rom_address` output to the ROM address currently being accessed.

Now that you have implemented `music_streamer`, create an instance of it in the module `z1top.v`. Use the instance name `streamer` to match the expected name in the `.do` file. Instantiate a `tone_generator` and wire `SWITCHES[1]` to `output_enable`, `CLK_125MHZ_FPGA` to `clk`, and `aud_pwm` to `square_wave_out`. Assign `aud_sd` to 1. Connect the `tone` output of the `music_streamer` to the `tone_switch_period` input of the `tone_generator`. Connect the `music_streamer`'s `clk` input to the global clock signal. Finally, connect its `rom_address` output to the `LEDS`s by routing the top 6 bits of address.

# 8 Simulating the music_streamer

To simulate your `music_streamer` open up the `lab3/src/music_streamer_testbench.v`. In contrast to the `tone_generator_testbench` where the `tone_generator` was instantiated in isolation, in this testbench we are instantiating our entire top-level design, `z1top`. This testbench is referred to as a system-level testbench, which tests our entire design using top-level I/O, in contrast to the `tone_generator_testbench` which is a block-level testbench. This is similar to the difference between unit and integration tests in software development.

You can see that this testbench just runs a simulation for 2 seconds and then exits. You might have to modify the `music_streamer_testbench.do` file to match the name of your module instances in `z1top.v`.

To execute the testbench, run `make CASES=tests/music_streamer_testbench.do` in `lab3/sim`. This may take several minutes to complete. You may have to run `make clean` before running `make` if ModelSim has cached build artifacts.

Inspect your waveform to make sure you get what you expect. Verify that there are no undefined signals (red lines, x) Then run the Python script to generate a `.wav` file of your simulation results and listen to your `music_streamer`. It should sound like the first few seconds of the song that was

loaded on the ROM.

# 9 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. Verify this in simulation, and prove the correct functionality during checkoff.

# 10 Try it on the FPGA!

Now try your `music_streamer` on the FPGA. You should expect the output to be the same as in simulation. The `SWITCHES[1]` switch should still work to disable the output of the `tone_generator`. Show your final results, simulation, and the working design on the FPGA to the TA for checkoff.

# 11 Saving your Work Using Git

At the beginning of this lab, you were asked to copy part of your solution lab 2 into this project. As we continue onto new labs and eventually onto the project, you will be asked to build off of your solution to previous labs. Given that your lab solutions will be important later in the term, we want to make sure you have a safe place to store your solutions. To do that, we have provided you with a private git repository on GitHub.

As was mentioned in lab 1, git is a revision control system that can keep track of files in your lab assignments and project. You have already been using git to get each week's lab assignments. Starting with this lab, you will also be using git to keep track of your solutions to the lab assignments.

Now let's get your solution committed into the git repository!

## 11.1 Adding New Files

First, git does not automatically track new files that you create. These files can be added to git using the `git add` command. If you created any new files during this lab that you want to save, run the following command (replacing `<path to file>` with the path to the file you wish to add):

```
git add <path to file>
```

## 11.2 Checking Git Status

We can check to see what git believes the state of the repository is. Run the following command in the lab3 directory:

```
git status
```

You should see a list of many files that you have modified over the course of labs 1, 2, and 3. This command also shows files that git is not tracking, which are listed under the "Untracked files"

section. Check to make sure that files you want to save are not listed in the "Untracked files" section. If they are, add them using `git add`.

### 11.3 Committing Your Solution

We now need to commit your changes to the repository. Committing a file saves any changes made to the file since the last commit into the repository. There are several methods to do this:

- You can use the command `git commit <file/directory list>` to commit files that are in the list specified by `<file/directory list>`. You should replace `<file/directory list>` with the list of files and/or directories you want to commit. Use spaces between the different files/directories in the list.
- You can use the command `git commit -a` to commit every file that has been changed and added with `git add`.
- You can use the `git add` command to "stage" files for commit. Run `git add` for each file you would like to commit. Then run `git commit` to commit the staged files.

Each of these command will bring up a text editor (probably vim) with a summary of the the actions git is about to take. It will ask you to type a commit message at the top of the document, save it, and quit the text editor. If you are comfortable with vim, feel free to enter your commit message, save the file, and exit. Otherwise, enter `:q!` and press return. This will exit vim without saving. Git will give you a message saying that the commit was aborted because of an empty commit message. You can enter a commit message from the command line by adding the flag `-m "commit message"` to any of the above commands. You should replace "commit message" with some text explaining what you are committing.

### 11.4 Setting Up Your Private GitHub Repository - Do this Once Per Clone

In this class you will interacting with 2 GitHub repositories :

- a public repository maintained by the lab staff which contains the skeleton code for each lab
- a private repository where you will store your solutions to the lab

Earlier in lab 1, you cloned the public repository maintained by the lab staff. This repository allows you to pull new updates that the lab staff post. Your private repository can be viewed via a web browser at `https://github.com/EECS150/sp19-eecs151-####` where `###` is the three characters in your eecs151 instructional account. Make sure you can access the web page for your repository. Because it is private, you will need to be signed into GitHub to see it. You may also need to accept an invitation sent to the e-mail address you registered with GitHub.

Currently, git on your computer is only aware of the public repository. We need to execute some commands to tell it about your new private repository.

In the `fpga_labs_sp19` directory, run the following commands, replacing `###` with the 3 characters in your eecs151 instructional account:

```
git remote rename origin staff
```

```
git remote add origin git@github.com:EECS150/sp19-eecs151-###.git
```

You will need to repeat these steps each time you clone the `fpga_labs_sp19` repository.

### 11.5 Pushing Updates to your Private GitHub Repository

Now that you have your files committed into the git repo, you need to push your updates to the GitHub server. To do this, run the following command:

```
git push origin master
```

If you gave a password with your ssh key, enter it when prompted. You should then see messages indicating that git is pushing your updates to your private repository!

Go to `https://github.com/EECS150/sp19-eecs151-###` and check that your files made it.

### 11.6 Pulling from GitHub

From this point forward, you will use this command to get new lab content from the course staff:

```
git pull staff master
```

If you are on a different computer (ex. a home computer) and want to pull any of the solutions you committed and pushed to your private repo, use the following command:

```
git pull origin master
```

## 12 Checkoff

1. Show the RTL you used to create your tone generator, and your calculations for obtaining the square wave at 220Hz
2. Demonstrate your tone generator on the FPGA and show that some input mutes the output noise
3. Section 3 - Answer the questions in section 3.4.7 and show any relevant simulations. Play an audio file that was generated using the `tone_generator_testbench` that plays some melody you define.
4. Section 7 - Prove that if the ROM contains an entry for a `tone_switch_period` of 0, that the square wave doesn't oscillate.
5. Section 8 - Show the working `music_streamer` on the FPGA.
6. Section 9 - Show that your solution has been committed and pushed to your private github repository.
7. Show the RTL you used to create your tone generator, and your calculations for obtaining the square wave at 220Hz (with comments)

8. Discuss how will a higher clock frequency impact the frequency of the square wave output for a fixed `tone_switch_period`?

## 13 Conclusion

You are done with lab 3! Please write down any and all feedback and criticism of this lab and share it with the TA.

## 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