EECS 151/251A FPGA Lab
Lab 2: Simulation, Inter-module Communication, and Memories

Prof. Borivoje Nikolic
TA: 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 strongly suggest that you look through three documents that will help you better understand some Verilog constructs.

1. **labs_fa16/docs/Verilog/wire_vs_reg.pdf** - The differences between wire and reg nets and when to use each of them.
2. **labs_fa16/docs/Verilog/always_at_blocks.pdf** - Understanding the differences between the two types of always @ blocks and what they synthesize to.
3. **labs_fa16/docs/Verilog/verilog_fsm.pdf** - An overview of how to create finite state machines in Verilog, specifying their state transitions and machine outputs.

You should go through the `verilog_fsm.pdf` document before doing the last part of this lab. The first couple sections of this lab focus on simulation and it would be valuable to read the first two documents before starting.

1.1 Helpful Hint: Synthesis Warnings and Errors

At various times in this lab, things will just not work on the FPGA or simulation. To help with debugging, you can run `make synth` in the `lab2/` folder. This will just run `xst` which will only take a few seconds. Then you should run `make report`. In the window that opened, click on `Synthesis Messages` on the left under `Errors and Warnings`. Any synthesis warnings you see here are a possible alert to some issue in your circuit. If you don't understand a warning, ask a TA; it almost always reveals some issue in your RTL.

2 Lab Overview

In this lab, we will begin by taking your `tone_generator` design from Lab 1 and simulating it in software. We will learn about using ModelSim to view waveforms and debug your circuits. You will then extend your `tone_generator` to play a configurable frequency square wave and simulate it to check that you have implemented the functionality correctly. You will then construct a module

that can pull tones to play from a memory block and send them to your `tone_generator` for a specified amount of time each. Finally, you will modify your module so that it can be paused, and so it can play the tones at different tempos.

3 Simulating the tone_generator from Lab 1

3.1 Copying Your Lab 1 Code

Run `git pull` in your git cloned `labs_fa16` directory to fetch the latest skeleton files.

Begin by copying your `tone_generator` implementation into the `lab2/tone_generator.v` file. Let's run some simulations on the `tone_generator` in software. 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 and testing whether the outputs of the module match what we expect.

3.2 Overview of Testbench Skeleton

Check the provided testbench skeleton in `lab2/tone_generator_testbench.v` to see the test written for the `tone_generator`. Let's go through what every line of this testbench does.

```
`timescale 1ns/1ns
`timescale (simulation step time)/(simulation resolution)
```

The timescale declaration needs to be at the top of every testbench file. It provides information to the circuit simulator about the timing parameters of the simulation.

The first argument to the timescale declaration is the simulation step time. It defines the chunks of discrete time in which the simulation should proceed. In this case, we have defined the simulation step time to be one nanosecond. This means that 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 equal the step time.

```
`define SECOND 1000000000
`define MS 1000000
// The SAMPLE_PERIOD corresponds to a 44100 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 at a standard 44100 kHz sample rate.

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

This module is our testbench module. It is not something actually synthesized to be placed on our FPGA, but rather just a top-level testbench module to be run by our 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;
wire sq_wave;
```

Here are the inputs and outputs of our `tone_generator`. You will 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 and we will be monitoring the output.

```
initial clock = 0;
always #(30.3/2) clock <= ~clock;
```

Here is our clock signal generation code. The clock net 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 net such that it produces a 33Mhz clock signal.

```
tone_generator piezo_controller (
    .clk(clock),
    .output_enable(output_enable),
    .square_wave_out(sq_wave)
);
```

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

```
initial begin
    output_enable <= 0;
    #(500 * `MS);
    output_enable <= 1;
    #(1 * `SECOND);
    $finish();
end
```

Here is the body of our testbench. The `initial begin ... end` block specifies the 'main()' function for our testbench. It is the execution entry point for our simulator. In the `initial` block, we can assign the inputs that flow into our DUT using non-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);`.

In this case, we assign `output_enable` to 0 at the start of the simulation, then we let the simulation run for 500ms, then we set `output_enable` to 1, then we let the simulation run for one second. The final statement is called 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 blocks as separate threads that both start execution at the beginning of the simulation and that operate 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.

The the `forever begin` block, we sample the `sq_wave` output of the `tone_generator` and save it in a file. We sample this value every `` `SAMPLE_PERIOD `` nanoseconds which corresponds to a 44100 kHz sampling rate. Your `tone_generator`'s output is stored as 1s and 0s in a file that can be translated to sound to hear how your circuit will sound when deployed on the ML505 development board and FPGA.

3.3 Using TCL scripts (.do files)

ModelSim, which is our circuit simulator, takes commands from TCL scripts. Take a look at the `lab2/sim/tests/tone_generator_testbench.do` file. 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/piezo_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 match exactly 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 when running the simulation of our circuit will not log the values of the signals in our testbench or DUT unless we tell it to. These lines tell the simulator to store and log the values of the signals directly inside the `tone_generator_testbench` module. The `*` tells the simulator to log all signals directly in the `tone_generator_module`. The second line allows us to log the signals in a submodule of the main testbench module. Observe that `piezo_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 specification (in our case 10000ms = 10s) gives the simulator an upper bound on the simulation time. The simulator won't simulate more than 10 seconds of circuit behavior. If the simulator hits the `$finish()` function before the 10 second timeout is up, it will stop simulation at that moment.

3.4 Running ModelSim

With all the details out of the way, let's actually run a simulation. Go to the `lab2/sim` directory and run `make`. After some time the simulation will finish.

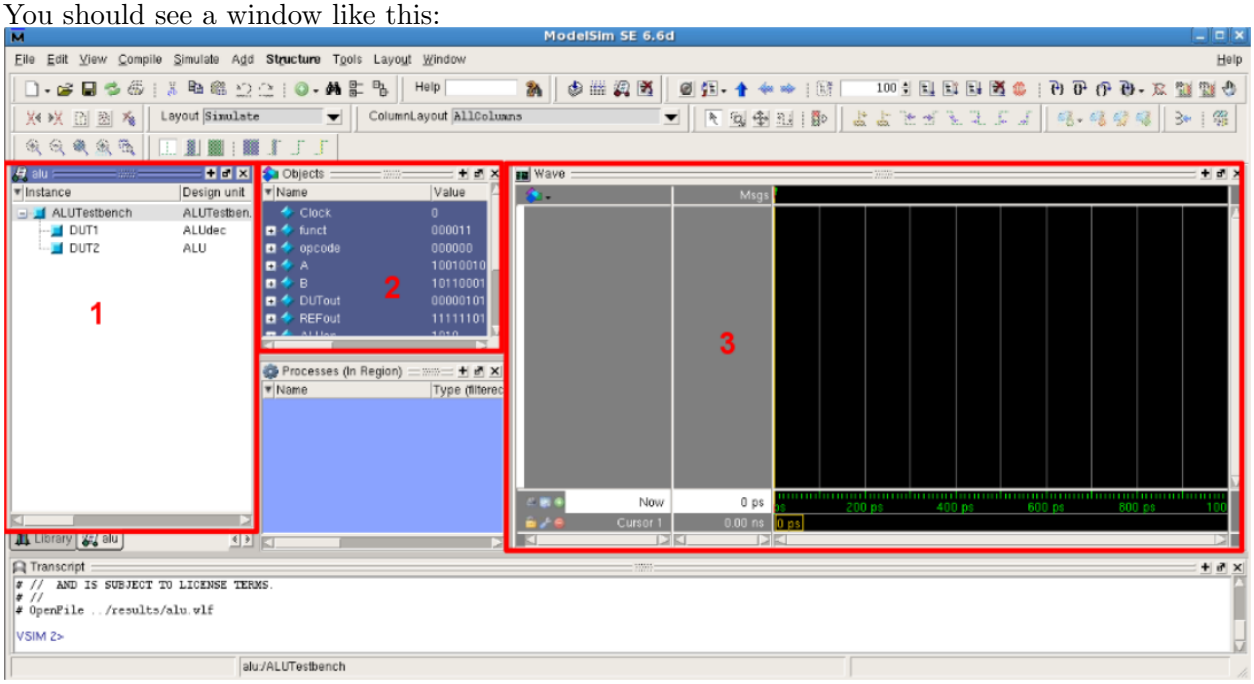
3.5 Viewing Waveforms

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

```
./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. List of the module involved in the testbench. You can select one of these to have its signals show up in the object window.

2. **Object Window** - this lists all the wires and regs in your 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.

Add the `clock, output_enable,` and `sq_wave` signals to the waveform viewer. Click anywhere on the waveform viewer to set your cursor and use the `O` and `I` keys to zoom in and out. Zoom out all the way.

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 500 ms. However, you will see that the `sq_wave` signal is just a red line. What's going on?

3.6 Fixing the Undefined clock_counter

Take a look at the `clock_counter` in your `tone_generator` module. Plot the signal in your waveform viewer. You will notice it's also a red line. Red lines in ModelSim indicate undefined signals (indicated in Verilog as the letter `x`).

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

Going back to the red line for `clock_counter`: this is caused because at the start of simulation, the 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 is 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/tone_generator.v` modify the `reg [x:0] clock_counter` line to read `reg [x:0] clock_counter = 0` instead. This implicitly tells the simulator that the initial simulation 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 default value in the same way.

Now run the simulation again.

3.6.1 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.

3.7 Listen to Your Square Wave Output

Take a look at the file written by the testbench located at `lab2/sim/build/output.txt`. It should be a sequence of 1s and 0s that represent the output of your `tone_generator`. I've written a Python script that can take this file and generate a `.wav` file that you can listen to.

Go to the `lab2/` directory and run the command:

```
python 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
```

You should hear a 440Hz square wave for 1 second after half a second of silence.

3.8 Playing with the Testbench

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

1. If you increase the clock frequency, would you expect the tone generated by your `tone_generator` to be of higher pitch or lower pitch from 440Hz? Why? Show audio evidence of this from the simulation.

4 Design a Configurable Frequency tone_generator

Let's extend our `tone_generator` so that it can play different notes. Add a 24-bit input to the `tone_generator` module 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 many clock cycles you should hold the value of your square wave before inverting it. For example a `tone_switch_period` of 37500 tells us to invert the square wave every 37500 clock cycles, which for a 33 Mhz clock translates to a 440 Hz square wave.

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 invert the square wave output. Remember to initialize any new registers declared in your `tone_generator` to their default value to prevent unknowns during simulation.

5 Simulating and Debugging Your New tone_generator

Now, extend the testbench to work with this new input signal. Add a new 24-bit reg to the testbench. Set `output_enable` to 1 at the start of the simulation. Then set the `tone_switch_period` of the DUT and run the simulation for some time (using a delay statement). Then change the `tone_switch_period` again and run the simulation for some more time.

Inspect the waveform and debug your `tone_generator` if you detect any bugs. Then use the same Python script to generate an audio file to listen to your `tone_generator`.

I suggest using `http://onlinetonegenerator.com/` to generate sample square wave tones and making sure your tones match.

Create a testbench that plays some simple melody that you define and show the TA before proceeding further.

6 Try the tone_generator on the FPGA

Modify the top-level Verilog module `ml505top.v` to include the new input to the `tone_generator`. You can tie the `tone_switch_period` to any value you want.

Run the usual `make` process and then `make impact` to put your new `tone_generator` on the FPGA. It should work as it did before.

7 Introduction to Inferred Asynchronous Memories - 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. A ROM can be broadly classed as a state element that holds some fixed data. This data can be accessed by supplying an address to the ROM after which the ROM will output the data from that address. Any memory block in general can contain as many addresses in which to store data as you desire. Every address contains some amount of data (bits), but every address should store the same amount of data. The number of addresses is called the **depth** of the memory, while the number of bits per address is called the **width** of the memory. These are important terms that are frequently used.

The synthesizer is a powerful tool that takes the Verilog you write and converts it into a low-level netlist of what structures are actually used on the FPGA. Our Verilog **describes** some digital circuits which are then **inferred** by the synthesizer. In this section, we will examine the Verilog that allows the synthesizer (XST) to **infer** a ROM. What follows is a minimal example of a ROM in Verilog: (depth of 8 entries/addresses, width of 8 bits)

```
module rom (input [2:0] address, output reg [7:0] data);
        always @(*) begin
                case(address)
                        3'd0: data = 8'h00;
                        3'd1: data = 8'hFF;
                        3'd2: data = 8'hAD;
                        3'd3: data = 8'h12;
                        3'd4: data = 8'h37;
                        3'd5: data = 8'h93;
                        3'd6: data = 8'h0A;
                        3'd7: data = 8'hC2;
                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 sequence of tones you want to play.

To generate a ROM with a script, first begin by creating a file that will define the memory contents with any name you choose. The file should contain a decimal number on each line which

represents the data stored at that address. I've provided an example file called `sample_data.txt` in the `lab2/` folder. Each line of this file is the `tone_switch_period` of a single note. The duration of each note will be specified later.

Use the provided python script to generate a ROM Verilog file (run in `lab2/`):

```
python rom_generator.py src/rom.v sample_data.txt 128 24
```

This will generate a ROM in `lab2/src/rom.v` using the data from `sample_data.txt` with a width of 24 bits and a depth of 128 entries/addresses. Try it out and make sure you understand what the script is doing. Since `sample_data.txt` doesn't contain 128 lines, the undefined memory addresses are filled with zeros.

You can edit `sample_data.txt` to fill it with your own melody. We will explore ways of generating a ROM/RAM from an initialization file using Verilog alone in a future lab.

You might want to try instantiating your memory in `ml505top.v` and putting your design on the FPGA to see how the memory works.

```
module ml505top (
        input  CLK_33MHZ_FPGA,
        input  [7:0] GPIO_DIP,
        output PIEZO_SPEAKER,
        output [7:0] GPIO_LED
);
        tone_generator piezo_controller (
                .clk(CLK_33MHZ_FPGA),
                .output_enable(GPIO_DIP[0]),
                .square_wave_out(PIEZO_SPEAKER)
        );

        rom memory (
                .address(GPIO_DIP[6:0]),
                .data(GPIO_LED[7:0])
        );
endmodule
```

You can toggle the DIP switches and see the 8 LEDs light up with the lowest byte of the data in the address specified by the switches.

8 Design of the music_streamer

Open up the `music_streamer.v` file. This module will contain an instance of the ROM you created earlier and will address the ROM sequentially to play tones. The `music_streamer` will play each note for a predefined amount of time by sending each switch period defined in the ROM to the `tone_generator`.

Read this entire section, then... You should calculate what 1/5th of a second is in terms of 33 Mhz clock cycles. Verify with the TA that you got the right answer. Then, before writing any Verilog, draw a schematic of the music_streamer circuit. Show your circuit to the TA before proceeding.

Let's proceed incrementally to integrate and design the `music_streamer`.

Begin by instantiating the `music_streamer` module in `ml505top.v`. Connect its `tone` output to the `tone_switch_period` input of the `tone_generator`. Connect its `clk` input to the global clock signal. You can leave the `tempo and pause` inputs unconnected for the time being.

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. Next, write the RTL that will increment the address supplied to the ROM every **1/5th seconds**. The data coming out of the ROM should be fed directly to the `tone_generator`. 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.

9 Simulating the music_streamer

To simulate your `music_streamer` add an instance of it to your `tone_generator_testbench`. Wire it up to the `tone_generator` just like you had it in `ml505top.v`. Then make sure you drive `output_enable` to the `tone_generator` to 1 at the start of simulation. Then insert a delay statement to let the simulation run for as long as you desire. Finally run `make` in the `lab2/sim` directory.

Inspect your waveform to make sure you get what you expect. It will likely be helpful to add a line to the `.do` file in `lab2/sim/tests` so that you can inspect the internal signals of your `music_streamer`. Then, run the Python script to generate a `.wav` file of your simulation results and listen to your `music_streamer` in action.

10 Verify your Code to 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 (meaning generate a 0Hz wave), that the `tone_generator` doesn't produce any oscillating output. Verify this in simulation. We will be using this functionality in the next lab when playing sheet music with the `music_streamer`.

11 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 GPIO_DIP 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.**

12 Optional: Adding Tempo Variations and Pausing to the music_streamer

In the next lab, we will be making our `music_streamer` more full-featured. If you have time now, you can implement some of these features.

Connect a GPIO_DIP switch to the `pause` input of the `music_streamer`. When this switch is turned on, your module should pause the music at the current note and should cut the output to the piezo speaker. When the switch is turned off, your module should resume playback.

Connect a pair of GPIO_DIP switches to the `tempo` input of the `music_streamer`. When these switches are toggled, your `music_streamer` should play the notes faster or slower. Basically, you can define four different tempos that hold each note for a different amount of time. We choose a standard 1/5th of a second for this lab, but you can vary it from 1/10, 1/7, 1/5, 1/3 to change tempos of your music on the fly.

13 Conclusion

You are done with lab 2! Please write down any and all feedback and criticism of this lab and share it with the TA. This is a brand new lab and I welcome everyone's input so that it can be improved.