EECS 151/251A FPGA Lab
Lab 2: Introduction to FPGA Development + Creating a Tone Generator

Prof. Jan Rabaey
TAs: Ali Moin, George Alexandrov
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

1 Before You Start This Lab

Make sure that you have gone through and completed the steps involved in Lab 1. Let the TA know if you are not signed up for this class on Piazza, don't know about the course website (bCourses) or if you do not have a class account (eecs151-xxx), so we can get that sorted out.

To fetch the skeleton files for this lab `cd` to the git repository (`fpga_labs_fa18`) that you had cloned in the first lab and execute the command `git pull`.

You can find the documents/datasheets useful for this lab in the `fpga_labs_fa18/resources` folder. Go through the Verilog Primer Slides in `resources/Verilog`; you should feel somewhat comfortable with the basics of Verilog to complete this lab.

2 A Structural and Behavioral Adder Design and also Inspecting the Schematic

2.1 Build a Structural 14-bit Adder

To help you with this task, please refer to the 'Code Generation with for-generate loops' slide in the Verilog Primer Slides (slide 35).

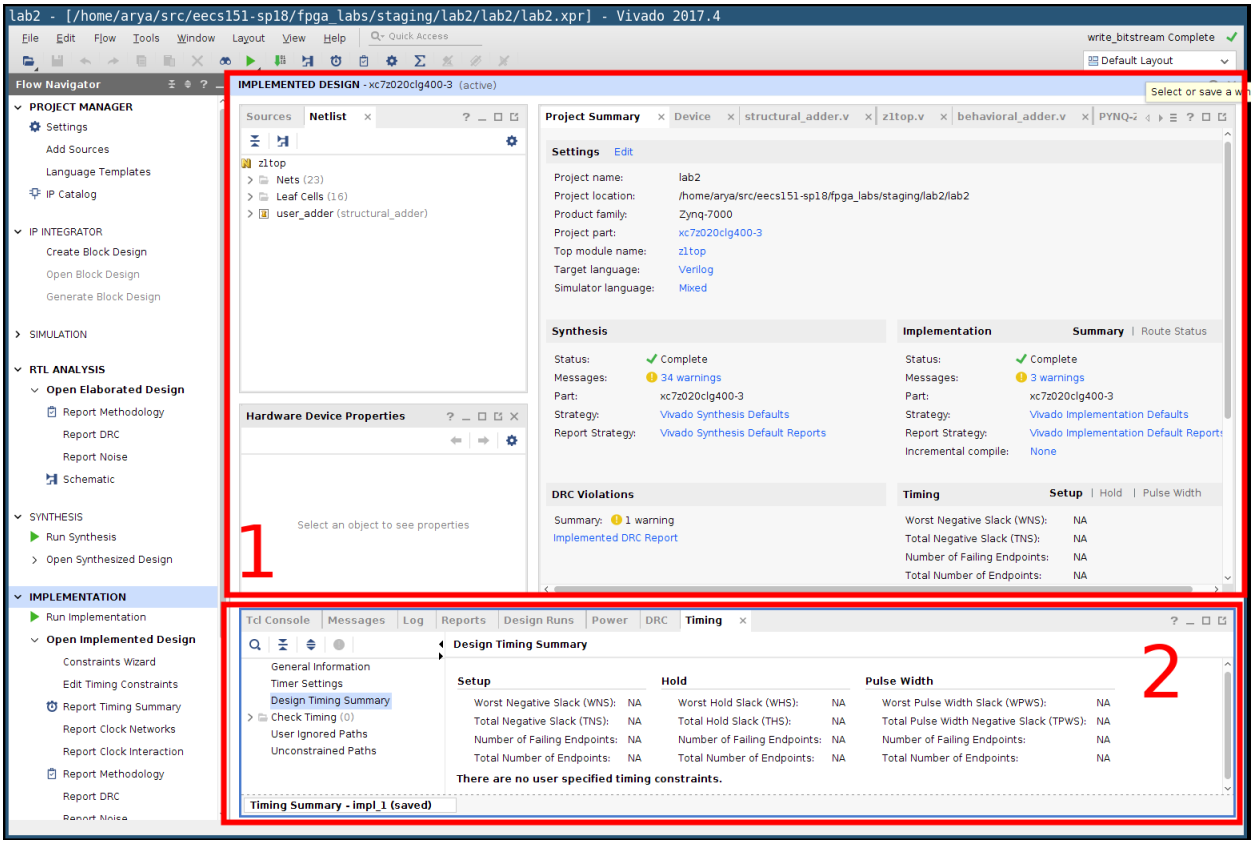
Open the Lab 2 project (`lab2.xpr`) in Vivado Design Suite from the lab repo (did you remember to `git pull`?). Begin by opening `lab2/lab2.srcs/sources_1/new/full_adder.v`; fill in the logic to produce the full adder outputs from the inputs. Then open `structural_adder.v` and construct a ripple carry adder using the full adder cells you designed earlier.

Finally, inspect the `z1top.v` top-level module and see how your structural adder is instantiated and hooked up to the top-level signals. For now, just look at the `user_adder` instance of your

structural adder. As we learned in previous lab, the basic I/O options on the Z1 board are limited. How are we managing to input two 3-bit integers?

Run Synthesis, Implementation, and Generate the Bitstream for your design. (Remember that, in the GUI, you can select a later step in the pipeline and have all prerequisite steps performed automatically when prompted.) Program the board. Test out your design and see that you get the correct results from your adder. You should try entering different binary numbers into your adder with the switches and buttons and see that the correct sum is displayed on the GPIO LEDs.

If there are any problems with your design, you can view the output report in Vivado from the *Project Summary* view. The Project Summary tab opens by default, but you can bring it up again from the *Window* menu. See box 1 in the figure for an example. At the bottom of the screen (box 2 in the same figure) you can also inspect the outputs of the individual tools that make up the pipeline; there lies bountiful debugging information should you ever need it. Unfortunately, not every log or warning message is useful, but it will serve you well to compare what outputs you do see with the relative success you have with your design.



2.2 Inspection of Structural Adder Using Schematic and fpga_editor

2.2.1 Schematics and FPGA layout

Now let's take a look at how the Verilog you wrote mapped to the primitive components on the FPGA. Three levels of schematic are generated for you once you've run the pipeline (each after its prerequisite step). In the *Flow Navigator*, you can view *Schematic*s under

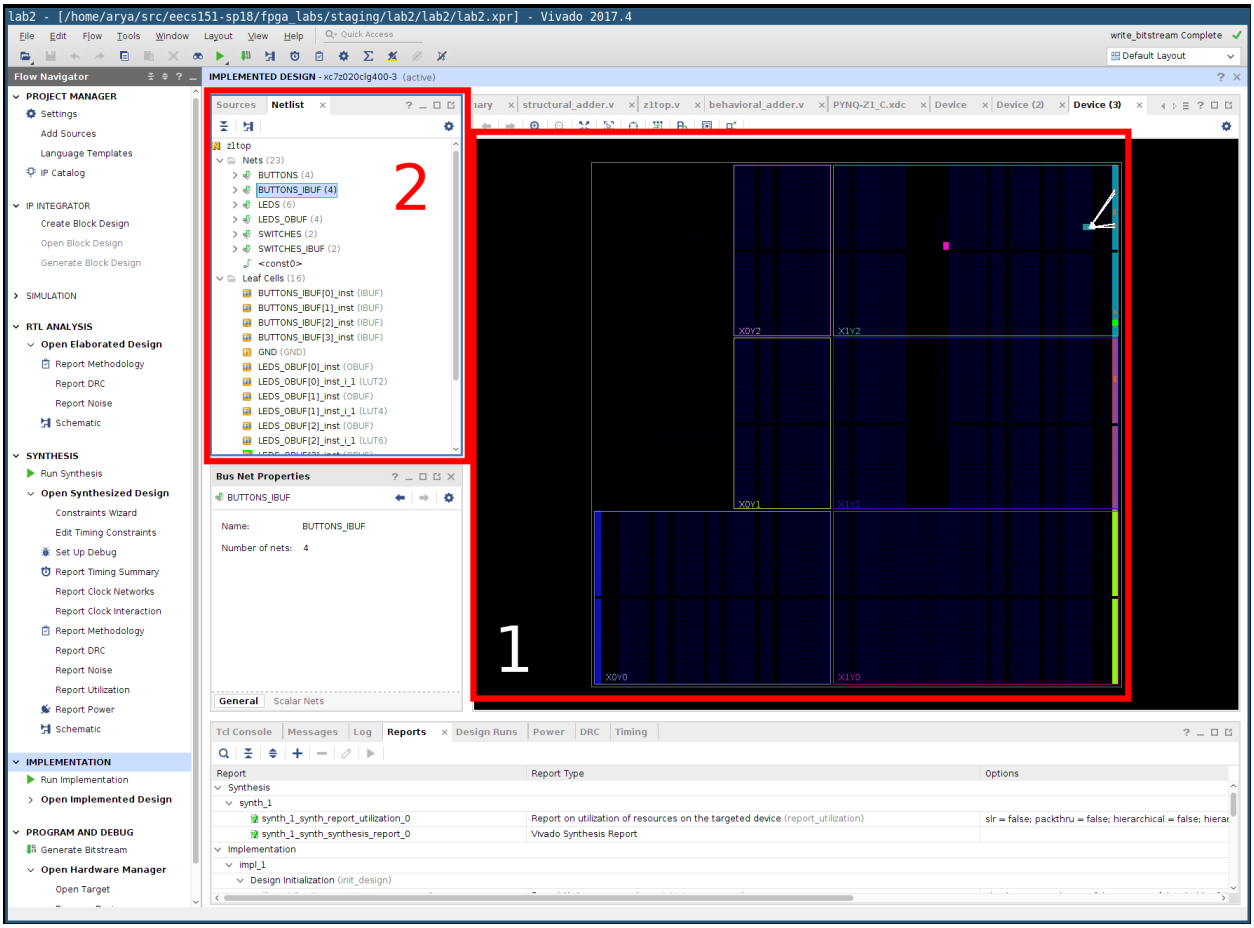
1. *RTL Analysis* → *Open Elaborated Design*
2. *Synthesis* → *Open Synthesized Design*
3. *Implementation* → *Open Implemented Design*

The first two will give you a fairly straightforward hierarchical block-level view of your design. You will find your circuit by drilling down into the `user_adder` module (that's the name you gave the instantiation of `structural_adder` in `z1top.v`). Check to see that your structural adder module is hooked up properly and looks sane. It's ok if the wires don't appear to be connected, just hover your mouse over the endpoints on the schematic and ensure that the connections are as you expect. Take note of the primitive blocks used in your circuit.

In the RTL Analysis (1) you are viewing a visualisation of the topology your RTL describes. At this point, logic elaboration is very abstract: you'll notice that your logic is expressed in terms of the logic gates you described (XOR, AND, etc). Any logic you describe in RTL is included, even if it's disconnected. In the Synthesis schematic (2) this logic has been elaborated further into what look like FPGA elements, but still at higher layer of abstraction, and with some unused signals still present. In the final schematic of the three, Implementation (3), the schematic now shows which of the elements in your nominated chip are actually targeted. Superfluous logic has been elided from the design.

Play around with the schematics. See how your logic is represented at successive stages of design.

Finally, you also look at how your circuit was placed and laid out on the FPGA. Once you've run the pipeline, open the *Window* menu and select *Design*. You'll be presented with a layout of the FPGA package as in box 1 in the figure below. It'll be hard to see with a small design, but the logic elements you've ended up using with your design will be highlighted. You can highlight your own nets in the diagram to make it easier to find them by selecting a net or signal from the Netlist pane (*Window* → *Netlist*; see box 2).



Now you can explore your design and look for the modules that you wrote. If you scroll down in the `Netlist Window` you should see various components of your logic. Some elements are mapped to LUTs: somewhere buried in their properties is the type of slice (recall that SLICELs contain the look-up tables that actually implement the logic you want). See if you can find out which nets have been assigned to LUTs, and how they are connected. Go ahead and explore several SLICELs that implement the structural adder to see how they are connected to each other and the outputs of your circuit.

2.3 Build a Behavioral 14-bit Adder

Check out `behavioral_adder.v`. It has already been filled with the appropriate logic for you. Notice how behavioral Verilog allows you to describe the function of a circuit rather than the topology or implementation.

In `z1top.v`, you can see that the `structural_adder` and the `behavioral_adder` are both instantiated in the self-test section. A module called `adder_tester` has been written for you that will check that the sums generated by both your adders are equal for all possible input combinations. If both your adders are operating identically, both RGB LEDs will light up. Verify this on your board.

2.4 Inspection of Behavioral Adder Schematics and FPGA Layout

Go through the same steps as you did for inspecting the structural adder. View the schematics at successive levels of logic elaboration and how FPGA components are connected. Record and note down any differences you see between both types of adders in the schematic and the FPGA layouts. You will be asked for some observations during checkoff.

3 Designing a Tone Generator

Now it's time to try something new. Let's create a tone generator/buzzer on the FPGA.

Please take a look at `pynq-rm.pdf` in the `fpga_labs_fa18/resources` folder. Read about the clock sources available on the board on page 14. Clock signals are generated outside the FPGA by a crystal oscillator or a programmable clock generator IC. These clock signals are then connected to pin(s) on the FPGA so that they can be used in your Verilog design.

Take a look at the `z1top.v` module and notice the `CLK_125MHZ_FPGA` input. Next take a look at the XDC `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. Are any other clocks available? The 125 MHz clock signal we will use is actually generated by the Ethernet chip as a cost-saving manoeuvre: it actually gets disabled when the Ethernet chip is reset. We can access the signal from within our Verilog top-level module and can propagate this clock signal to any submodules that may need it.

3.1 Audio Out

As described in the Pynq Reference Manual, our evaluation boards have several other neat peripherals (and even a few expansion ports). One feature is mono (single-channel) audio out: take a look at page 18. A Sallen-Key Butterworth low-pass filter is used at the output of another standard logic interface to the FPGA. This filter "smooths out" a pulse-width-modulated (PWM) signal to generated sinusoidal signals for driving a (low-power) external speaker. Why is it a low-pass filter? To learn more about how PWM will help generate an output waveform, read on to page 19.

3.2 Enabling the Audio Out signal in the constraints file

The description of Audio Out in the Reference Manual tells us which is the interesting pin on the FPGA. Let's add the Audio Out connection to the XDC constraints file so that we can bring it in as an output from the Verilog top-level module. We are already using the master XDC file `PYNQ-Z1_C.xdc` for this board, however, so our task should be trivial.

Ask a TA if you need help for this part.

3.3 Generating a Square Wave

Let's say we want to play a 220 Hz square wave out of the Mono Audio Out port on our board. We want our square wave to 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 input we can use to time our circuit.

Find the following:

1. The period of our clock signal (frequency = 125 MHz)?
2. The period of a 220 Hz square wave?
3. How many clock cycles does it take for one period of the square wave?

Knowing how many clock cycles equals one cycle of the square wave, you can design this circuit. First open `tone_generator.v`. Some starter code is included in this file. Begin by sizing your `clock_counter` register to the number of bits it would take to store the clock cycles per square wave period. Design the logic such that a 220 Hz square wave comes out of the `square_wave_out` output.

Add an output to `z1top.v` with a matching net name to the Audio Out net you declared in the XDC. Instantiate the `tone_generator` and connect it to the Audio Out pin.

Build your design. 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. If everything works, you should be able to plug your audio-out signal into a speaker in the lab (or your own earphones) to hear a buzzing noise at 220 Hz. To stop the buzzing, just turn your FPGA off.

3.4 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/.

4 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 ~ 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.

4.1 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`.

5 Checkoff

To checkoff for this lab, have these things ready to show the TA:

1. Be able to explain the differences between the behavioral and structural adder as they are synthesized in both the high-level schematic and low-level SLICE views
2. Show the RTL you used to create your tone generator, and your calculations for obtaining the square wave at 220Hz
3. Demonstrate your tone generator on the FPGA and show that some input mutes the output noise
4. How will a higher clock frequency impact the frequency of the square wave output for a fixed `tone_switch_period`?

You are done with this lab. In the next lab, we will simulate our digital designs in software, and extend our `tone_generator` to read a song from a ROM and play it through our Audio Out.