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

Prof. Borivoje Nikolic and Prof. Sophia Shao
TAs: Cem Yalcin, Rebekah Zhao, Ryan Kaveh, Vignesh Iyer
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 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_sp19`) 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_sp19/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.

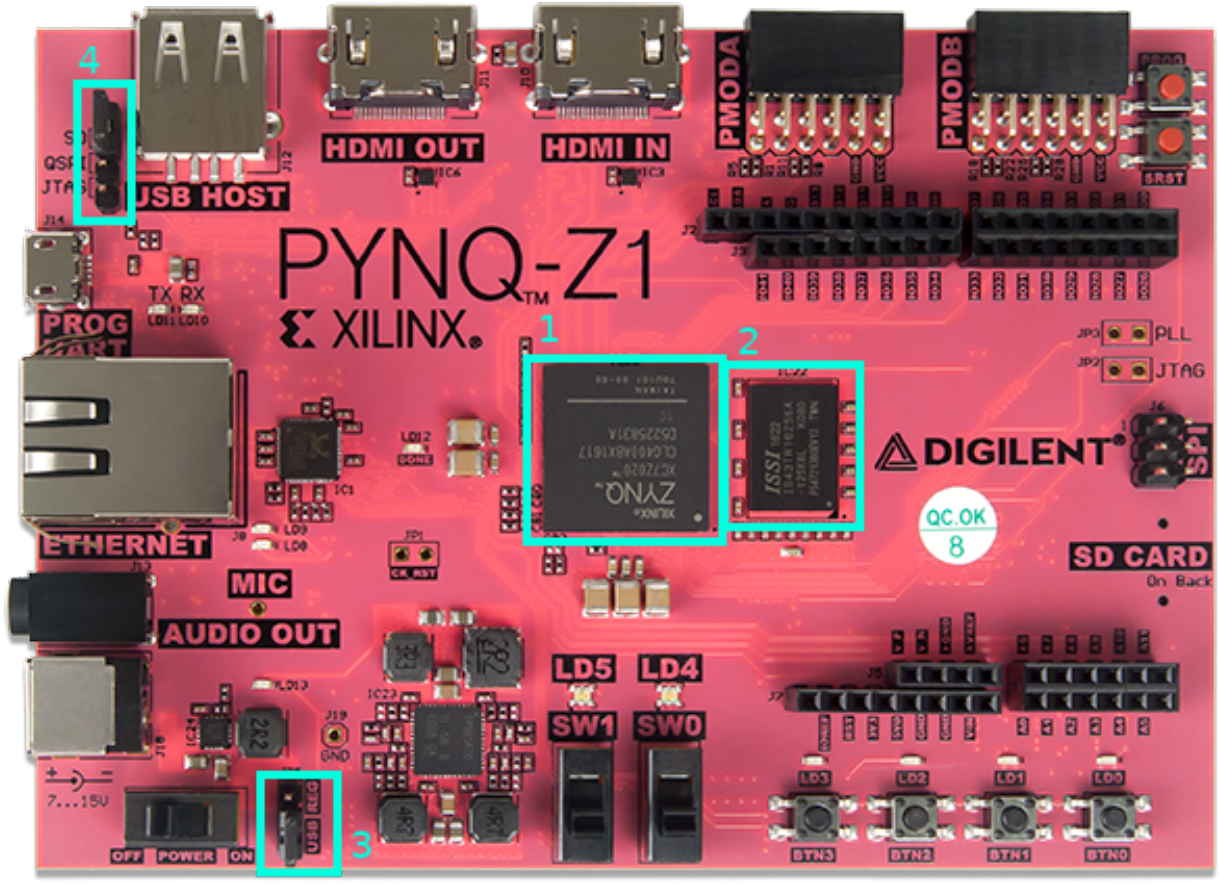
2 Our Development Platform - Xilinx Pynq-Z1

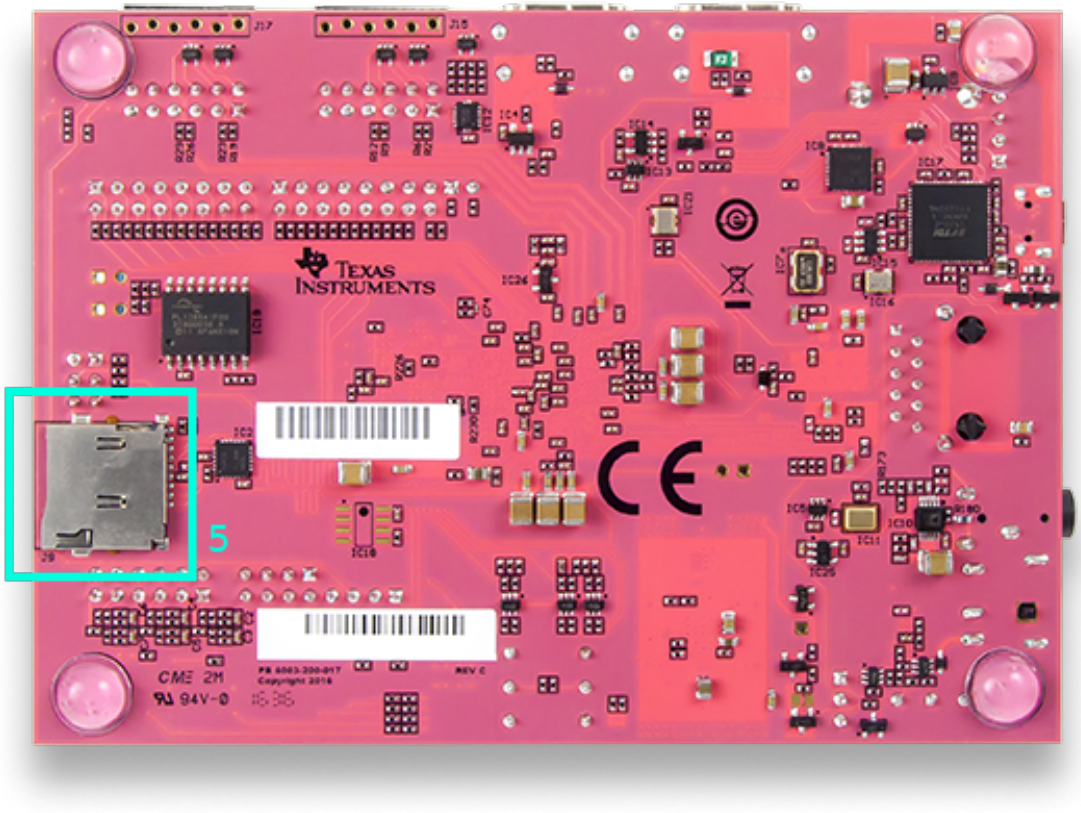
For the labs in this class, we will be using the Xilinx Pynq-Z1 development board which is built on the Zynq development platform. Our development board is a printed circuit board that contains a Zynq-7000 FPGA along with a host of peripheral ICs and connections. The development board makes it easy to program the FPGA and allows us to experiment with different peripherals.

The best reference for this board is provided by Digilent: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual (a PDF version of this manual is available in the `fpga_labs_sp19/resources` folder). Browse the documentation there to get a feel for both what features the board has and, more importantly, what information the documentation has, should you need it later.

Being a development board, the silkscreen print clearly identifies connectors of interest. You should be able to recognize the most basic IO features on the board: GPIO LEDs, slide switches, and push-buttons. You should also be familiar with other basic elements of the board: input power socket,

power switch, and the USB programming port. The following image identifies important parts of the board that may not have been obvious:





1. Zynq 7000-series FPGA. It is connected to the peripheral ICs and I/O connectors via PCB traces.
2. ISSI DRAM chip
3. Power source jumper: shorting "REG" has the board use the external power adapter as a power source; shorting "USB" has it rely on the 5 V provided by USB. The latter will work unless your design needs to power a lot of external peripherals. Since we have labs and power adaptors available, we avoid this.
4. Programming mode jumper

5. SD card slot

3 The FPGA - Xilinx Zynq-7000 7z020

To help you become familiar with the FPGA that you will be working with through the semester, please skim Chapter 21: Programmable Logic Description of the Technical Reference Manual and Chapter 2 of the Xilinx 7-series Configurable Logic Block User Guide. Pay particular attention to pages 15-25 on Slices and pages 40-42 on Multiplexers. Answer the following questions (you should be able to discuss your answers for checkoff):

3.1 Checkoff Questions

1. How many SLICEs are in a single CLB?
2. How many inputs do each of the LUTs on a Zynq-7000 FPGA have?
3. How many LUTs does the 7z020 have?
4. How do you implement logic functions of 7 inputs in a single SLICEL? How about 8? Draw a high-level circuit diagram to show how the implementation would look. Be specific about the elements (LUTs, muxes) that are used.
5. What is the difference between a SLICEL and a SLICEM?

4 Overview of the FPGA Build Toolchain

Before we begin the lab, we should familiarize ourselves with the CAD (computer aided design) tools that translate HDL (Verilog) into a working circuit on the FPGA. These tools will pass your design through several stages, each one bringing it closer to a concrete implementation. In previous years, older evaluation platforms (the ML505) used older FPGAs (a Xilinx Virtex-5 LX110T) and an older software suite (Xilinx ISE). Although there was a GUI, we had Makefiles to invoke each subsequent program in the toolchain to carry out the complete synthesis and perform analysis.

Our new boards use Xilinx's updated design software, the Vivado Design Suite. Vivado emphasizes its powerful integrated scripting capabilities (using the Tcl language) and integration with other high-level design tools (such as, for example, High-Level Synthesis - but more on that later). It also has support for equivalent Makefiles and automation through Tcl. The GUI itself has the disadvantage of being very manual to work with. Repeatedly changing and running parameters quickly becomes tedious. Our eventual goal is definitely to automate the design process as much as possible. For learning, however, the GUI has the invaluable property of guiding us through each step of the process. Read through the following sections but don't worry about the details for now. We will run through the entire tool flow on a simple project at the end of this lab.

4.1 Synthesis

To run the synthesis step in the Vivado Design Suite (that is, turn your HDL into combinational and sequential logic), select *Run Synthesis* in the *Flow Navigator* pane to the left other interface. If this has been run before, the synthesized design can be inspected by selecting *Open Synthesized Design*.

4.2 Implementation

The implementation step in the Vivado GUI is equivalent to the translation, mapping and place and route steps in the manual pipeline. Again, this takes the logical circuit synthesized previously and maps it to the physical logic devices our particular FPGA actually has. Select *Run Implementation* in the *Flow Navigator* to run it, then select *Open Implementation* to inspect its outputs.

4.3 Xilinx Design Constraints (XDC)

How do we connect one of our signals to a physical device? How do we specify special properties of the circuit that might matter for correctness and timing? The Xilinx Design Constraints file (with the `.xdc` extension) specifies necessary properties of the design (just like the old User Constraints File), and is a crucial input to the implementation phase. XDC is inspired by the Synopsis ASIC synthesis toolchain and aims to be somewhat compatible. You are writing a form of TCL for the Vivado TCL interpreter. More information can be found on page 22 of the Vivado migration guide.

Take a look at this snippet from the XDC inside `lab1/lab1.srcs/constrs_1/new/z1top.xdc`:

```
set_property -dict { PACKAGE_PIN L19   IOSTANDARD LVCMOS33 } [get_ports { BUTTONS[3] }];
```

This syntax assigns the properties `PACKAGE_PIN` and `IOSTANDARD` with the values `L19` and `LVCMOS33` (respectively) to the port `BUTTONS[3]`, a signal we defined in our Verilog source. Each of these properties has a separate consequence in the synthesis process:

- The pin to which the `BUTTONS[3]` signal should be connected to the physical pin L19 on the FPGA package.
- The logic convention (maximum voltage, what ranges constitute low and high, etc) for that port will be LVCMOS33.

4.4 Bitstream generation

To generate the programming file our FPGA will understand, we invoke *Generate Bitstream* in the *Flow Navigator*.

4.5 Timing Analysis

A timing analysis report can be generated under *Synthesis* in *Flow Navigator*, by expanding *Open Synthesized Design* and selecting *Report Timing Summary*.

4.6 Design Reports

Reports are automatically generated at each step in the build flow. You should be able to discover them under each of the expanded stages in the *Flow Navigator*. The *Project Summary* window (under the *Window* menu) presents a nice summary of the reports generated through each step. You will see some examples later in the lab.

4.7 Programming the FPGA

To send the bitstream to the FPGA with the Vivado GUI, we have to use the *Hardware Manager*. This is accessible under *Program and Debug* in the *Flow Navigator*, right under *Generate Bitstream*. Once connected to your FPGA over the USB JTAG interface, you can select *Program Device* in the *Flow Navigator* (or in the *Hardware Manager* pane that opens) to perform the programming.

4.8 Toolchain Conclusion

This section was information dense. Don't worry about understanding the internals of each tool and the exact file formats they work with, especially for different Xilinx software generations. Just understand what each step of the toolchain does at a high level and you will be good for this class. You will use these kinds of tools regularly, but for now let the staff worry about making sure they work in the first place.

5 Your First FPGA Design

Finally, let's conclude with a simple example of a complete project. Throughout the semester, you will build increasingly complex designs using Verilog, a widely used hardware description language (HDL). For this lab, you will use basic Verilog to describe a simple digital circuit.

Now that you have cloned the `fpga_labs_sp19` repository, you can `cd` to the `fpga_labs_sp19/lab1` directory to see this lab's skeleton files. You will note that there is a `lab1.srcs` directory and a `lab1.xpr` file.

Past versions of this course used an FPGA development toolchain from the Xilinx ISE Design Suite. The modern alternative, and that which we are now using for this course, is the Xilinx Vivado Design Suite ("Vivado" for short). We will initially use Vivado's Integrated Development Environment (IDE) instead of any command-line tools, though you will eventually see that the framework (especially with Pynq) is ripe for automation with TCL and Python scripting.

The lab skeleton files include the project meta data file, `.xpr`, a Verilog source file for a simple top-level module, `z1top.v`, and a constraints file, `z1top.xdc`.

HDL source files like `z1top.v` (where the HDL is Verilog) describe the circuit that you want to create on the FPGA. `z1top.v` describes a circuit that is the *top-level* of your circuit: it has access to the signals that come into and out of the FPGA chip. Constraints files, such as `z1top.xdc`, allow the engineer (you!) to tailor specific properties of the synthesized design to how they wish to use their specific chip. This includes the crucial mapping between FPGA input/output pins and signal names used in circuit descriptions. We'll cover more on constraints files in the next lab, so don't worry about the details too much just yet.

5.1 Set up your Pynq-Z1

1. Plug in the power adaptor to provide mains power.
2. Insert the factory-imaged SD card to provide a boot OS for the onboard ARM (mostly for use later).
3. Connect the USB interface to a spare USB port on your workstation.
4. Turn the board on.

5.2 Open the Lab 1 project in the Vivado Design Suite

In our Centos environment, press Alt-F2 to bring up a command dialog. Type the full path to the `vivado` binary to execute it:

```
/opt/Xilinx/Vivado/current/bin/vivado
```

(You can also run this from a terminal or create a Desktop shortcut.)

Once in Vivado, open up the `lab1/lab1.xpr` project file. Look around the environment to try and get a feel for the GUI. Open up the `lab1/lab1.srcs/sources_1/new/z1top.v` source file. Again, this file contains a Verilog module description which specifies which signals are inputs into the module and which signals are outputs.

The `BUTTONS` input is a signal that is 4 bits wide (as indicated by the [3:0] width descriptor). This input signal represents the logic signals coming from the momentary push-button switches on the bottom right side of your Pynq-Z1 board. You should inspect your board to find these switches and confirm that there are 4 of them. Another basic input signal is `SWITCHES`, which is 2 bits wide (as indicated by the [1:0] descriptor). Each of these two signals represents the slide switches on the Pynq-Z1, located just to the left of the momentary switches (look for SW0 and SW1).

The `LEDS` output is a signal that is 6 bits wide (as indicated by the [5:0] width descriptor). This output signal represents the logic signals coming out of the FPGA and going into the bank of LEDs at the bottom right of the Pynq-Z1, just above the buttons. Almost. There are only 4 LEDs there; 2 more are tri-color LEDs located just above the slide switches in the middle.

In this file, we can describe how the slide switches, push buttons and LEDs are connected through the FPGA. There is one line of code that describes an AND gate that takes the values of one of the buttons and one of the slide switches, ANDs them together, and sends that signal out to the first LED. Let's put this digital circuit on the FPGA!

5.3 Synthesize and Program

1. In Vivado, locate the *Flow Navigator* pane to the left of the screen. Near the bottom, under *Program and Debug*, click *Generate Bitstream*. Accept the default settings and wait. This should invoke the dependent steps in the flow: *Synthesis* and *Implementation* (among other things).
    - Selecting *Project Manager* will give you a nice overview of the progress of various background steps while this happens.
    - So will watching the *Messages* and *Logs* output.
2. When the synthesis and bitstream generation is done, select *Open Hardware Manager* and connect to your FPGA.
    - If you haven't before, or the hardware manager says no devices are connected, select *Menu* → *Open New Target*
    - You should see `xilinx_tcf` listed under Harware Targets in the top pane. In the bottom pane, two entries: `arm_dap_0` and `xc7z020_1`. That's good. *Next* → *Finish*.
3. Back in the *Flow Navigator* on the left, under *Program and Debug*, select *Program Device*. The only option to program will be the FPGA, `xc7z020_1`. The default bitstream file path should work too.
4. See if it worked! What happens when you push the BTN0 button? What about when you change SW0? Both?

Go ahead and extend this example with more AND or other gates to see them in action!

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

6.1 Build a Structural 14-bit Adder

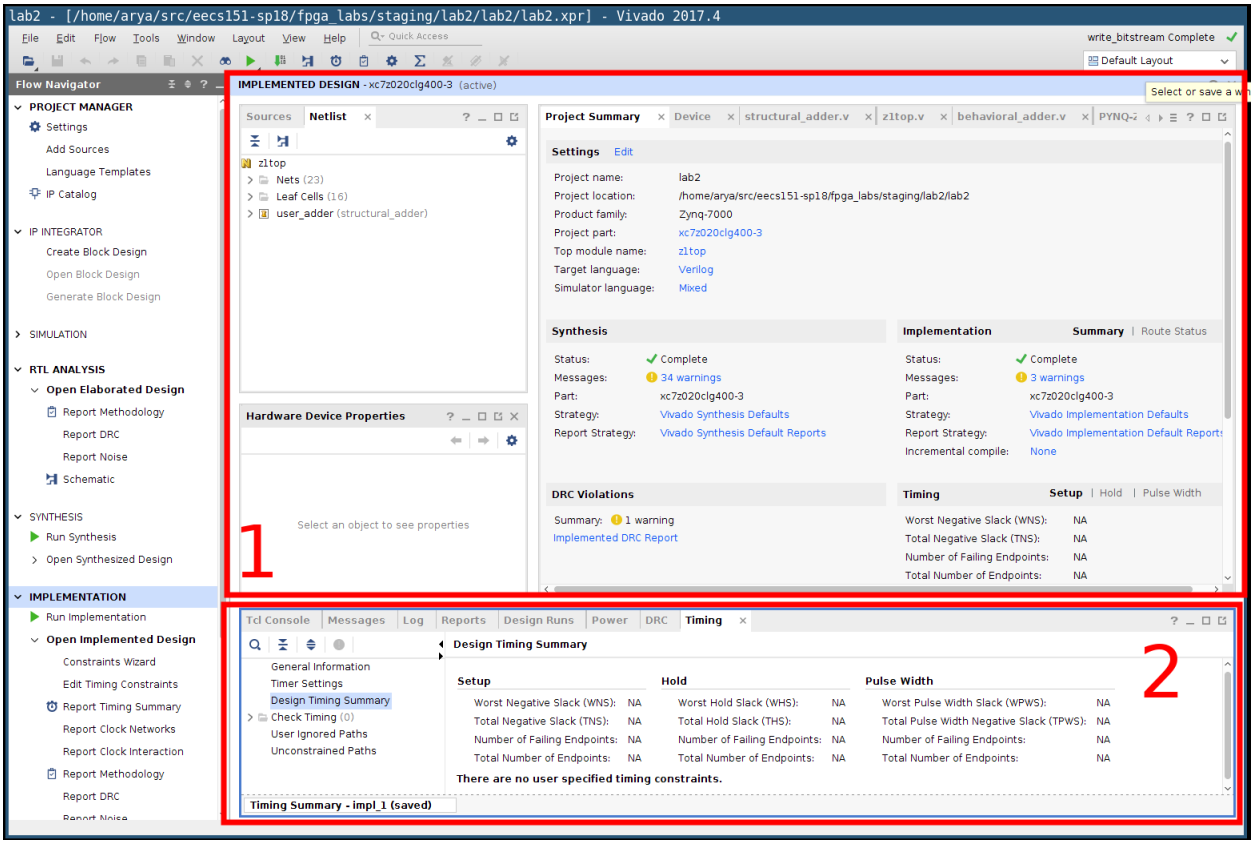
To help you with this task, please refer to the 'Code Generation with for-generate loops' slide in the Verilog Primer Slides (slide 35).

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 and a 'for-generate loop'.

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.



6.2 Inspection of Structural Adder Using Schematic and fpga_editor

6.2.1 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

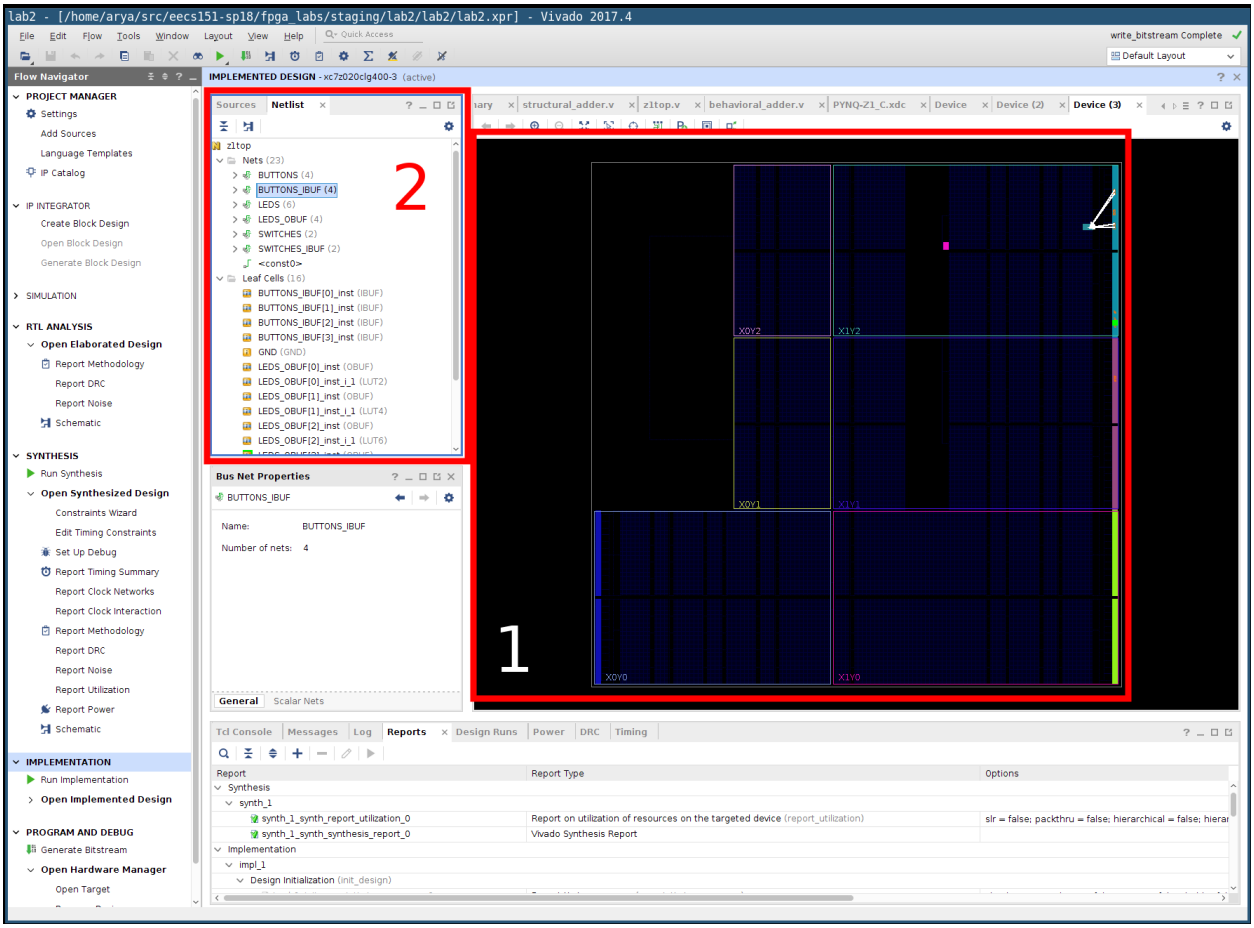
1. *RTL Analysis* → *Open Elaborated Design*
2. *Synthesis* → *Open Synthesized Design*
3. *Implementation* → *Open Implemented Design*

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 *Implemented Design*, click on the *Window* menu, and select *Device*. 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.

6.3 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.

6.4 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.

7 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_sp19/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.

7.1 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. To learn more about how PWM will help generate an output waveform, read on to page 19. Aside: why is it a low-pass filter? What is the frequency response of the filter and why is it chosen as such?

7.2 Enabling the Audio Out signal in the constraints file

The description of Audio Out in the Reference Manual tells us which are the relevant pins on the FPGA. Let's add the Audio Out connection to the XDC constraints file. Uncomment lines 63 and 64 of `PYNQ-Z1_C.xdc`, which is the master XDC file we will be using throughout the semester. These lines specify that the two relevant signals (`aud_pwm` and `aud_sd`) are connected to pins R18 and T17, respectively. Additionally, add the two signals to the top-level module `z1top.v` so that they can be used in the design. (Should these be inputs or outputs?)

Ask a TA if you need help for this part.

7.3 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. Instantiate the `tone_generator` inside `z1top.v` and connect it to the Audio Out pin we defined previously. Make sure that you enable the Audio Out with the `aid_sd` signal as well.

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.

7.4 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/.

8 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.

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

8.1 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`.

9 Checkoff

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

1. Your programmed and functional board showing LED0 being controlled by BTN0 and SW0, as well as any modifications you made to the design.

2. Demonstrate your structural adder on the FPGA and show that the test passes.
3. Show the RTL you used to create your tone generator, and your calculations for obtaining the square wave at 220Hz
4. Demonstrate your tone generator on the FPGA and show that some input mutes the output noise

Also please submit a short lab report to gradescope in which you answer/show the following things:

1. Answers for the questions in section 3.1
2. Explain the differences between the behavioral and structural adder as they are synthesized in both the high-level schematic and low-level SLICE views
3. Show the RTL you used to create your tone generator, and your calculations for obtaining the square wave at 220Hz (with comments)
4. Discuss 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.

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
- Sp19: Christopher Yarp, Arya Reais-Parsi