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

Prof. John Wawrzynek, Nicholas Weaver
TAs: Arya Reais-Parsi, Taehwan Kim
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

1 Before You Start This Lab

This lab and all future labs can be done in groups of 2.

Make sure that you have gone through and completed the steps involved in Lab 0. Let the TA know if you are not signed up for this class on Piazza, don't know about the course website (http://inst.eecs.berkeley.edu/~eecs151/sp18/files/Verilog_Primer_Slides.pdf) or if you do not have a class account (eecs151-xxx), so we can get that sorted out. Go through the Verilog Primer Slides that are linked on; you should feel somewhat comfortable with the basics of Verilog to complete this lab.

To fetch the skeleton files for this lab `cd` to the git repository (`fpga_labs_sp18`) 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_sp18/docs` folder.

1.1 What happened to Lab 1?

What happened to Winamp 4?

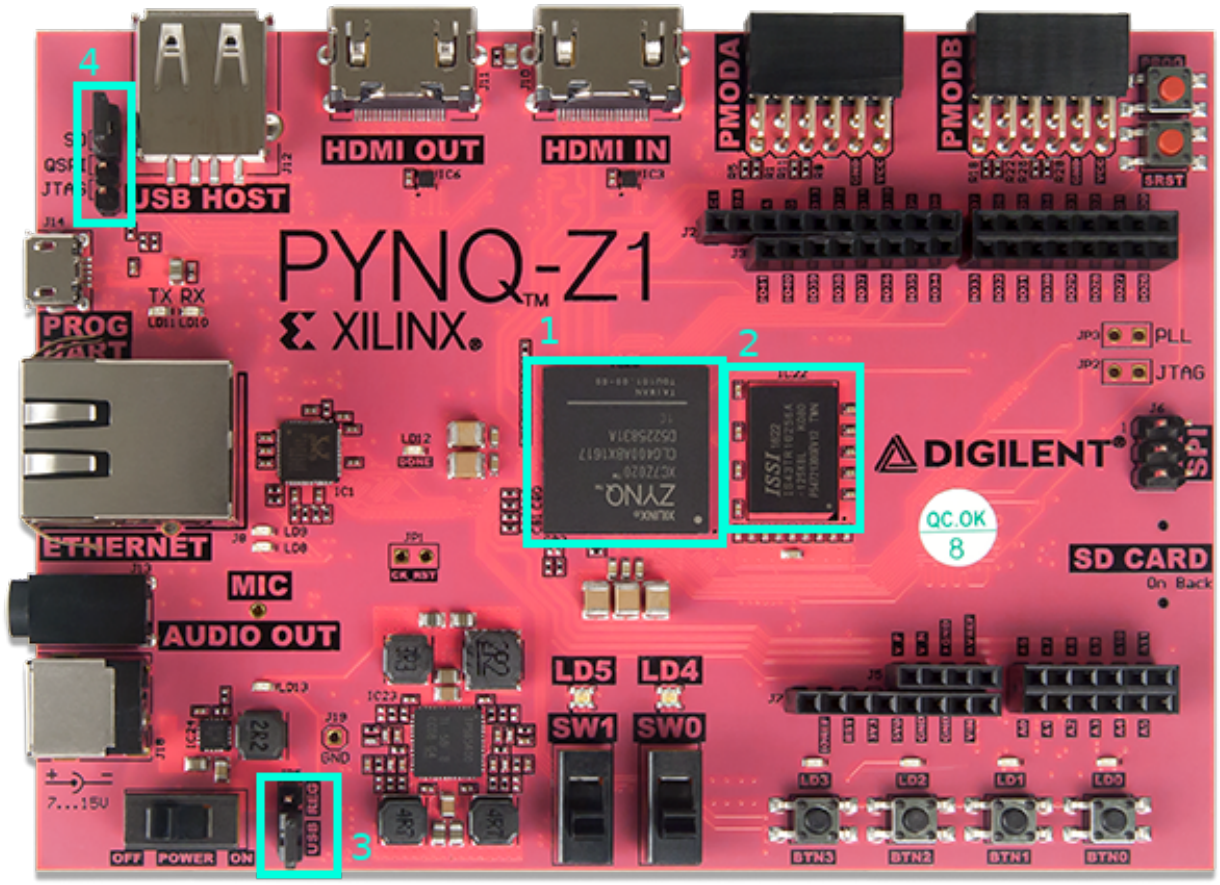
2 Our Development Platform - Xilinx Pynq-Z1

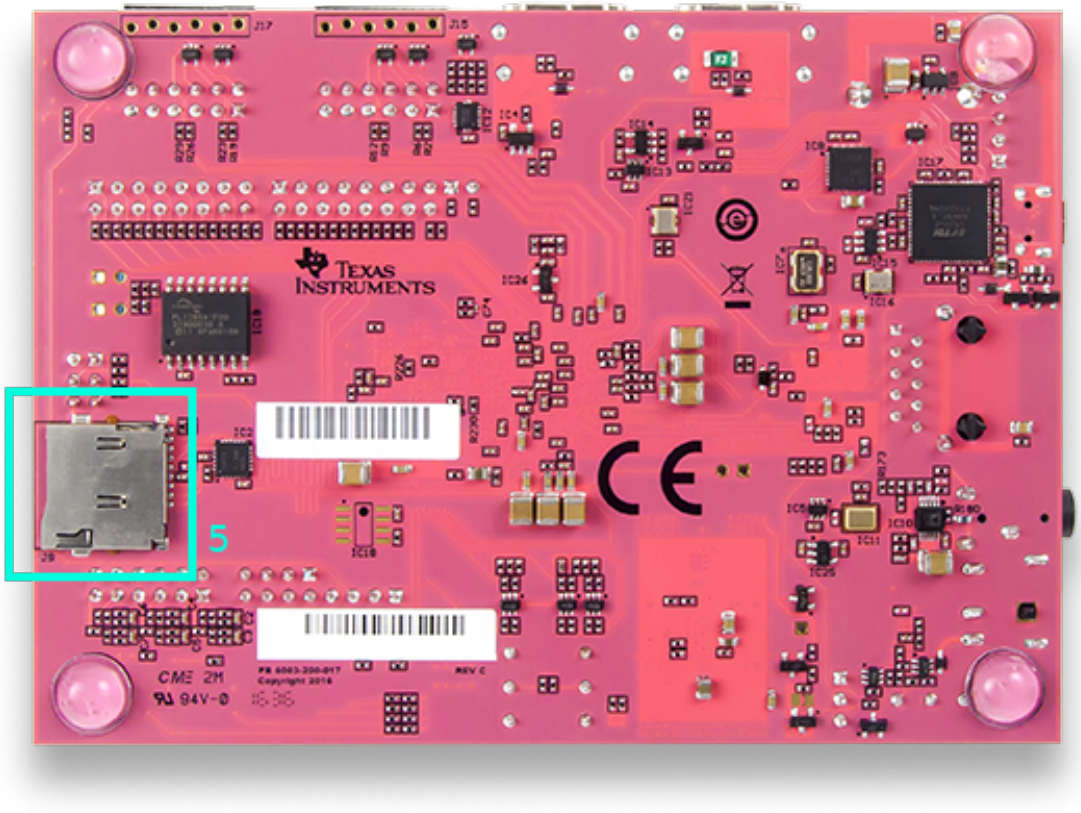
For the labs in this class, we will be using the Xilinx Pynq-Z1 development board which is built on the Zynq development platform. Our development board is a printed circuit board that contains a Zynq-7000 FPGA along with a host of peripheral ICs and connections. The development board makes it easy to 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_sp18/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 have used the most basic IO features in the last lab: 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 - you might recognise the name from your first lab) 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.

Have a quick read of the old design toolchain as described in previous years' labs, in the Appendix. There is an equivalent function in the new software suite for each of the described design steps,

described below. Seeing two software examples of the same higher-level process is didactic for understanding the abstract functions performed at each step.

Note that Xilinx publishes a large PDF on how to migrate from ISE to the Vivado suite, for reference.

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 file we used in the first lab:

```
set_property PACKAGE_PIN L19 [get_ports {BUTTONS[3]}]
set_property 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 A Structural and Behavioral Adder Design and also Inspecting the Schematic

5.1 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).

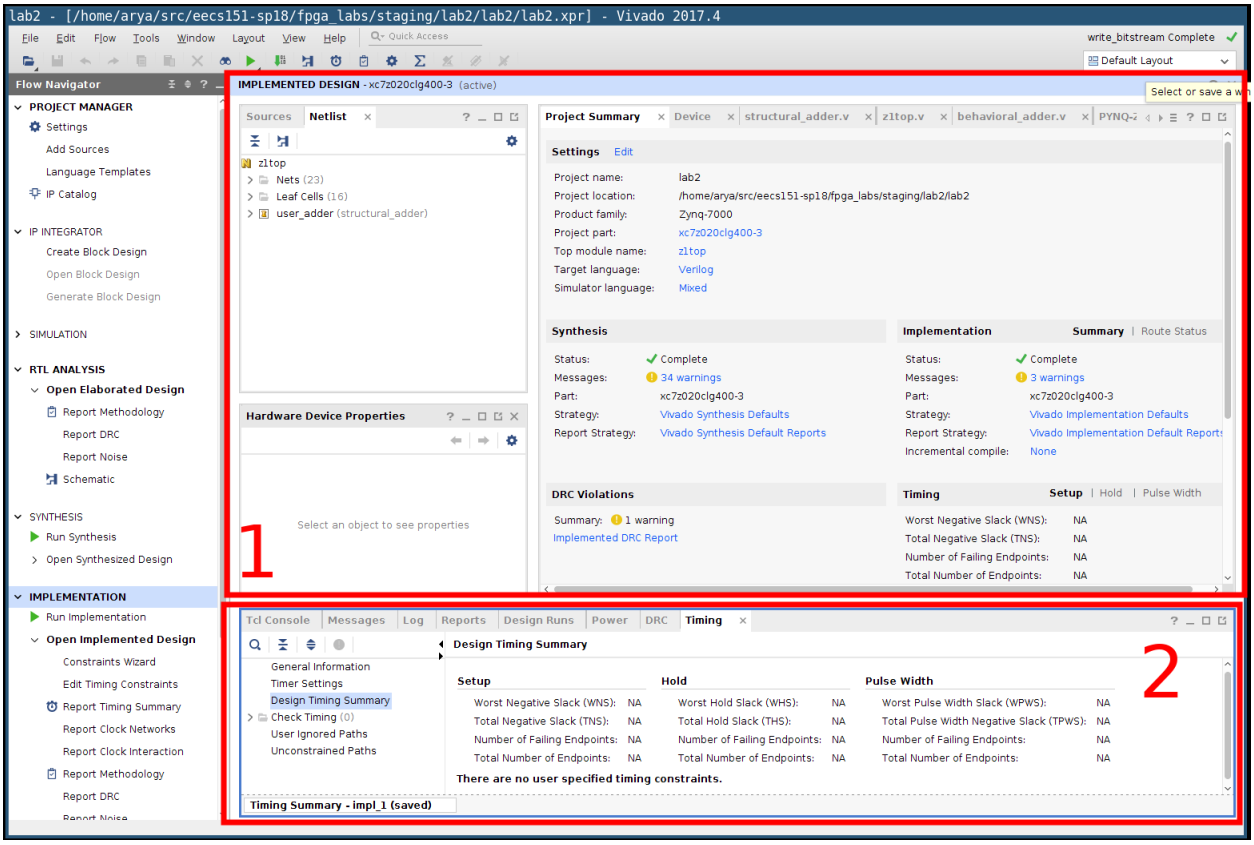
Open the Lab 2 project (`lab2.xpr`) in Vivado Design Suite from the lab repo (did you remember to `git pull`?). Begin by opening `lab2/lab2.srcs/sources_1/new/full_adder.v`; fill in the logic

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 Section 3, 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 individuals 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.



5.2 Inspection of Structural Adder Using Schematic and fpga_editor

5.2.1 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

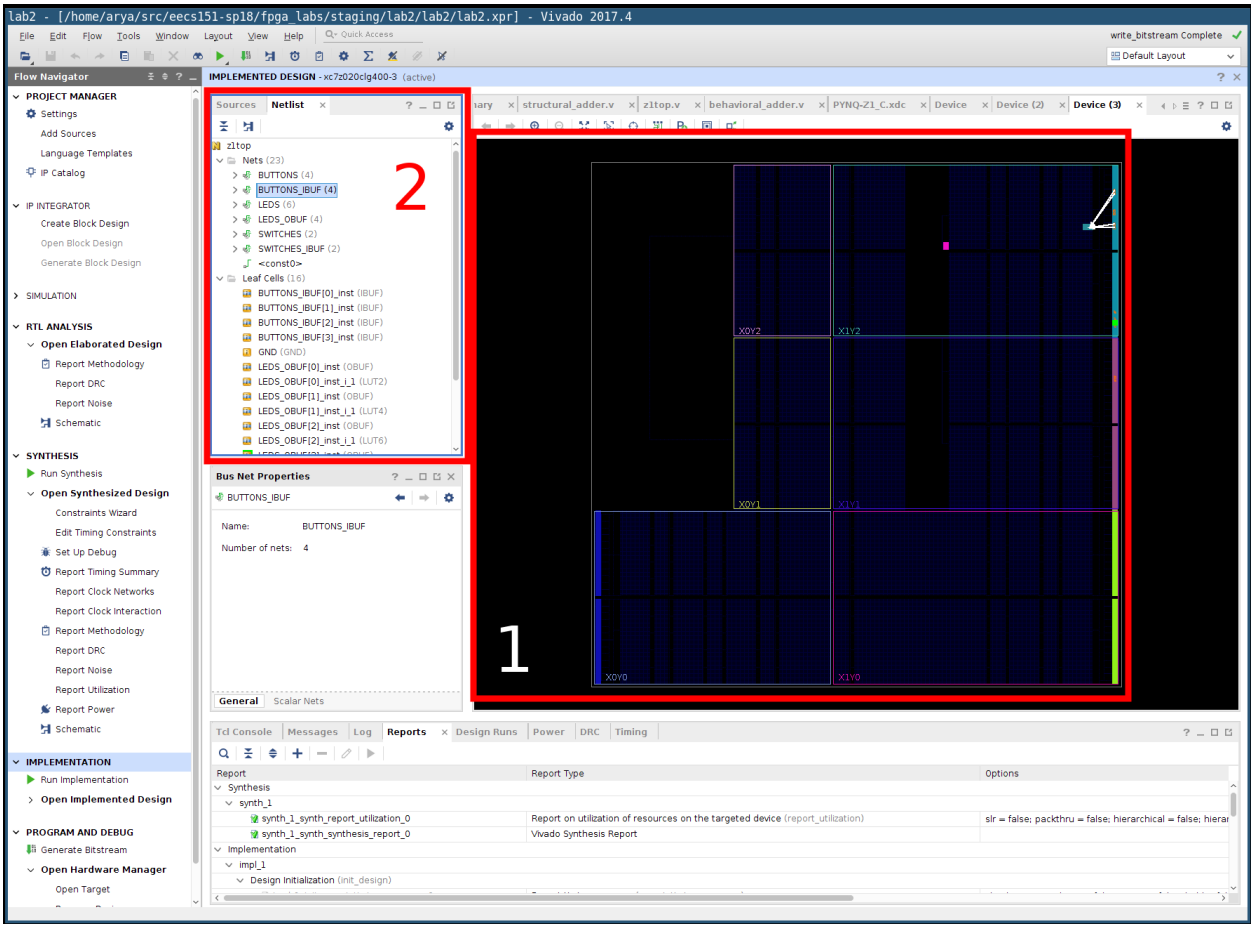
1. *RTL Analysis* → *Open Elaborated Design*
2. *Synthesis* → *Open Synthesized Design*
3. *Implementation* → *Open Implemented Design*

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 miss 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 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.

5.3 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.

5.4 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.

6 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_sp18/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.

6.1 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.

6.2 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.

6.3 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.

6.4 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/.

7 Optional: Use Remaining Switches to Control Frequency or Duty Cycle

This part is completely optional. You can use the other basic I/O devices (switches/buttons) as inputs into your `tone_generator`. See if you can use them encode a binary number that reprsents a particular duty cycle for your square or a particular set of frequencies. Show that these inputs give you control over the duty cycle or frequency of your square wave output to the audio port.

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

8 Checkoff

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

1. Answers for the questions in part 3.1
2. 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
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

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.

9 Appendix: The (Old) Xilinx ISE Toolchain

9.1 Invoking the toolchain

The command-line-driven ISE toolchain uses Makefiles as typical in large (old) software projects. For example, running `make` in the root project directory invokes a `Makefile` there, which itself can invoke `Makefile`s in child directories. `make` is a simple but powerful tool for generating dependent components in large projects if their dependent files have been updated.

The Makefiles are orchestrated such that every process in the FPGA synthesis and programming pipeline can be run and, when necessary, that dependencies are automatically run first. We will discuss every part of this toolchain.

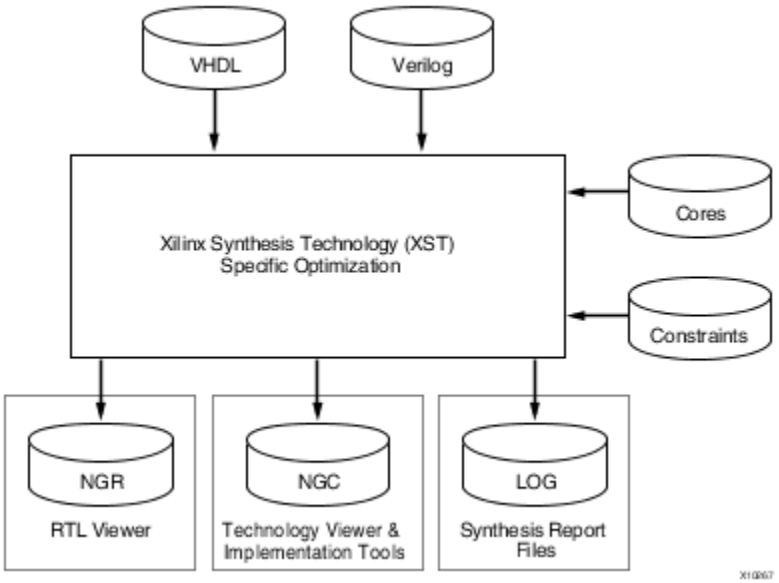
9.2 Synthesis with XST

The synthesis tool in ISE (Xilinx Synthesis Tool (xst)) is the first program that processes your design. Among other tasks, it is responsible for the process of transforming the primitive gates and

flip-flops that you wrote in Verilog into LUTs and other primitive FPGA elements.

For example, if you described a circuit composed of many gates, but ultimately of 6 inputs and 1 output, xst will map your circuit down to a single 6-LUT. Likewise, if you described a flip-flop it will be mapped to a specific type of flip-flop which actually exists on the FPGA.

The following figure shows the flow of files through XST.



XST takes in Verilog and/or VHDL files to be parsed and synthesized, "Cores" which are pre-built circuits designed by Xilinx, and "Constraints" (an XCF file). In this lab we don't supply any cores or constraints to xst.

XST produces log output, an NGR file which can be viewed with Xilinx ISE (as we will see soon), and a NGC netlist file. The NGC is a text file that contains a list of primitive components to be used in the FPGA circuit and a description of how they are connected.

9.3 Translation and Mapping with NGDBuild and Map

The tools that perform translation and mapping in ISE are NGDBuild and Map respectively. These tools take the output of the synthesis tool (a generic netlist) and translates each LUT and primitive FPGA element to an equivalent on the specific Xilinx FPGAs we use (at one point it was the vc5vlx110t chip, for example).

The translation tool merges all the input netlists and design constraint information, and outputs a Xilinx Native Generic Database (NGD) file. NGDBuild takes the UCF (User Constraints File) and the NGC (from XST) files as inputs.

The mapping tool maps the logic defined by an NGD file into FPGA elements such as CLBs (configurable logic blocks) and IOBs (input/output blocks). The Map tool takes in the NGD file produced by the translation tool and produces a Xilinx Native Circuit Description (NCD) file.

9.3.1 User Constraints File (UCF)

We will take a small detour here to cover what a UCF is and how to add top-level signal connections to it.

A user constraints file is passed to the translation tool (NGDBuild) and it describes the top-level pin assignments and timing constraints. When you opened `ml505top.v` in Lab 0, you noticed that your top-level Verilog module received several input and output signals. One of those signals was `input [3:0] BUTTONS`. You added some Verilog gate primitives to drive the outputs with some logic operations done to the inputs. But how do the tools know where those signals come from? That's what the UCF is for in ISE.

Here is the syntax used to declare a top-level signal that you can sense and/or drive from your top-level Verilog module.

```
NET (net name)<bit index> LOC="(FPGA pin number)"
NET (net name)<bit index> IOSTANDARD="(voltage level)"
```

The (net name) is the signal name that is presented to your top-level Verilog module. The bit index can be set optionally to create a multi-bit signal. The LOC defines what pin coming out of the FPGA contains that signal. All the pins coming out of the FPGA's package are labeled, and this is how we can tap or drive signals from a particular pin. The second line defines an IOSTANDARD for a given net; this is a statement of the signaling standard (FPGA IO driver type) to be used for a given pin. Here is an example for `BUTTONS[3]`

```
NET BUTTONS<3> LOC="H18";
NET BUTTONS<3> IOSTANDARD="LVCMOS25";
```

These 2 lines give you access to a net called `BUTTONS` in your top-level Verilog module. This net is connected to the `H18` pin coming out of the FPGA which is routed on the PCB to one of the board LEDs (which?) using a trace.

Note that this declaration doesn't specify whether the net is an input or output; that is defined in your Verilog top-level module port declaration.

On most development boards, we can utilize what is called a master UCF file to make adding signals to your design easy. This master UCF file is used in conjunction with the user guide and schematic to figure out what peripheral connections you want brought into your FPGA design. We will discuss how to use these files in the design exercise in this lab.

9.4 Place and Route with PAR

Now we resume where we left off after the mapping tool. The map tool's NCD output file is fed into the Place and Route tool which, in ISE, is called PAR. PAR places and routes the design that was generated by Map, and it outputs another NCD file with placement and routing information. This process is often the most time consuming of any of the steps in the toolchain; the algorithms used for placement and routing are sophisticated and have long run times.

9.5 Bitstream Generation with BitGen

The fully placed and routed design from PAR as a NCD file is now ready to be translated into another file that the Xilinx FPGA programmer can understand. In ISE, the BitGen tool can perform the generation of the bitstream that is sent to the FPGA, the last step in the FPGA build process. It produces a .bit file which can be uploaded to the FPGA via the programmer.

9.6 Timing Analysis with TRCE

The `Makefile` can also perform additional steps after (or before) running BitGen. To verify that our design met all timing requirements and to see a timing analysis, we can use a tool called Trace (TRCE) which takes in output files generated by PAR and produces a timing report. It will let you know what is the maximum clock speed your design can operate at reliably.

9.7 Report Generation with ISE

A very important precaution to take after running each step of the toolchain is to verify that there are no errors or warnings that a given tool produced. xreport, which ships with Xilinx ISE, produces a report detailing the status of each build tool. The tool will give you all the warnings and errors emitted by each tool in a GUI. It will also report the resource usage of your design.

9.8 FPGA programming

Each FPGA needs a mechanism to have its configuration changed. In older classes, we used the Xilinx iMPACT programmer, which provided the interface between our USB workstations and the FPGA's programming interface (JTAG). To send the bitstream file one generated with BitGen to the FPGA, one has to use the right software for a given programmer. This is the last step: after successful programming, the FPGA should work as per our design!