EECS 151/251A FPGA Lab
Lab 1: Introduction to FPGA Development + Creating a Tone Generator

Prof. Elad Alon
TAs: Vighnesh Iyer, Bob Zhou
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

1 Before You Start This Lab

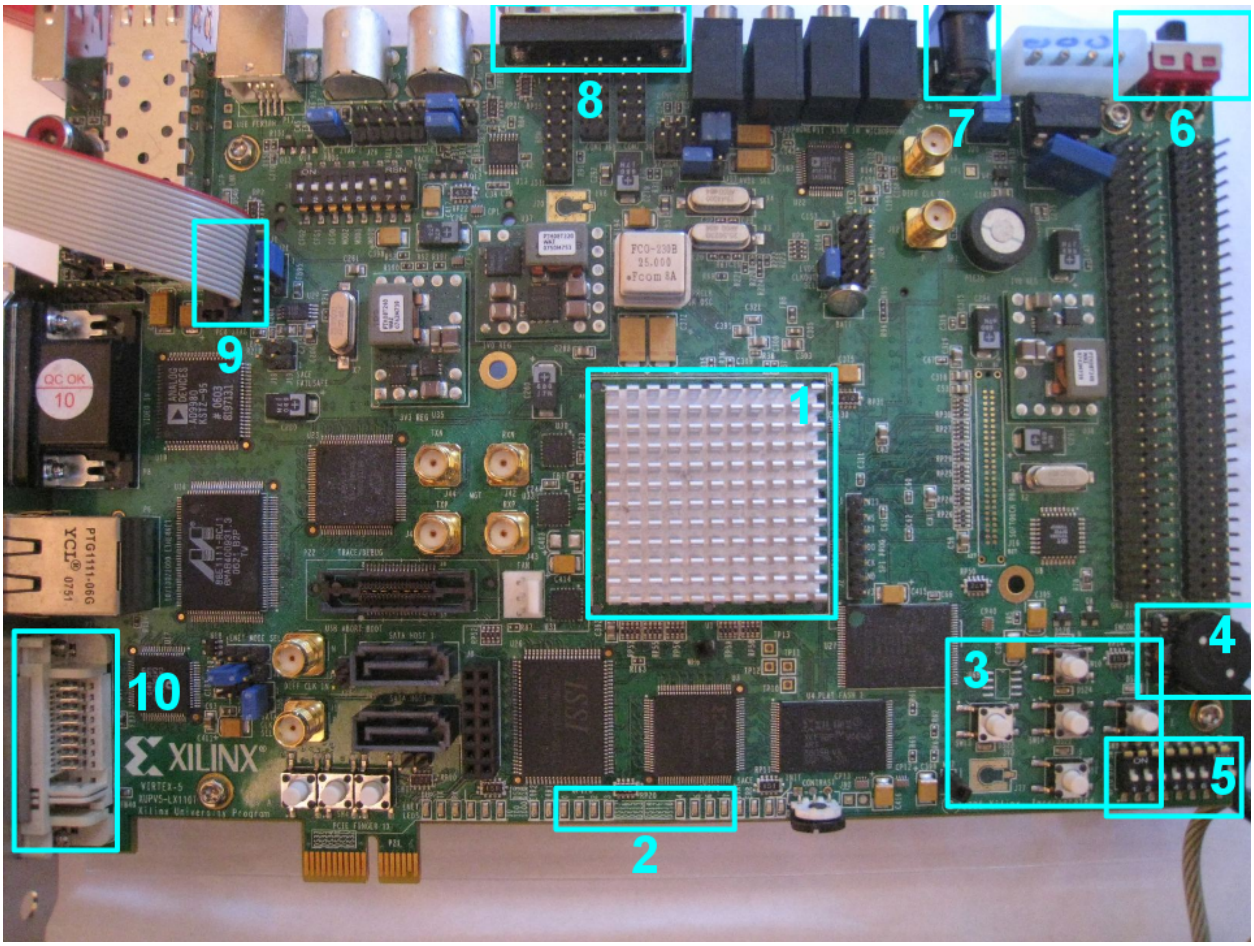
Before you proceed with the contents of this lab, be sure that you have gone through and completed the steps involved in Lab 0. There is no checkoff for Lab 0, but there will be for this lab. Let the TA know if you are not signed up for this class on bCourses and Piazza or if you do not have a class account (eecs151-xxx), so we can get that sorted out. Also, please go through the Verilog Primer slides that are linked to on Piazza; 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 (`labs_sp17`) that you had cloned in Lab 0 and execute the command `git pull`.

You can find the documents/datasheets useful for this lab in the `labs_sp17/docs` folder.

2 Our Development Platform - Xilinx ML505

For the labs in this class, we will be using the Xilinx XUPV5-LX110T development board which is built on the ML505 evaluation platform. Our development board is a printed circuit board that contains a Virtex-5 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 following image identifies important parts of the board:



1. Virtex-5 FPGA (covered by heat sink). It is connected to the peripheral ICs and I/O connectors via PCB traces.
2. GPIO LEDs, numbered 0-7
3. North, East, South, West, Center user push buttons, each with a corresponding LED
4. Rotary encoder (a wheel we can rotate clockwise or counterclockwise and push horizontally)
5. GPIO DIP (dual-inline package) switches, numbered 1-8 (but referred to 0-7 in code)
6. Board power switch (toggle for a full board reset, after which you will have to reprogram the FPGA)
7. Power connector; the power cable on many of these board is sensitive to movement and can come loose easily. Make sure you seat the power cable properly.
8. Serial port
9. JTAG header (used to program the FPGA, the Xilinx programmer is connected to this header)
10. DVI-I connector (for video output)

You will also notice a device sitting to the left of the development board that looks like this:

This is a Xilinx FPGA programmer that connects to your workstation (desktop computer) over USB to receive a compiled bitstream file which it then sends to the development board and the FPGA over the JTAG interface. Before you run `make impact` to send your design to the FPGA, make sure that the LED on the programmer is glowing green. If it is glowing yellow or not glowing at all, make sure that the wired connections are solid and that the development board is powered and on.

3 The FPGA - Xilinx Virtex-5 LX110T

To help you become familiar with the FPGA that you will be working with through the semester, please read Chapter 5: Configurable Logic Blocks (pages 171-179 about SLICEs and pages 193-197 about multiplexing) of the Virtex-5 User Guide and 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 Virtex-5 LX110T FPGA have?
3. How many LUTs does the LX110T 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 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.

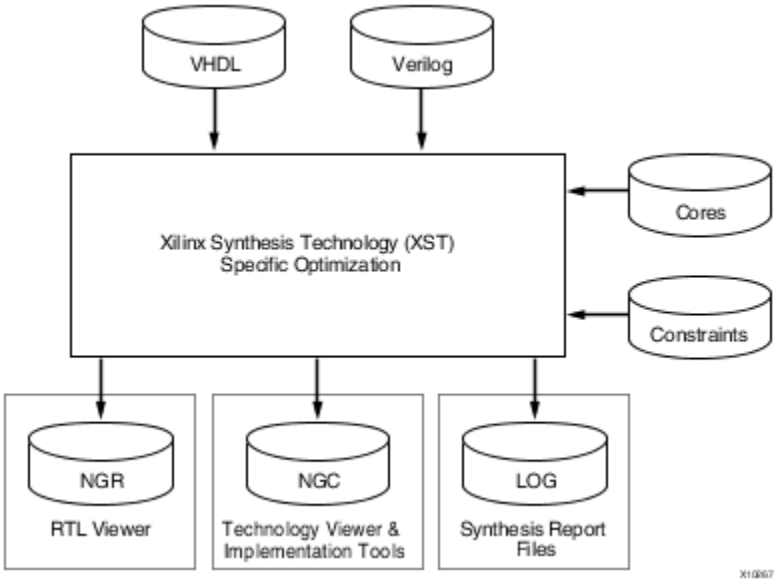
Looking at the directory structure of the `lab1` folder, you can see two main folders, `src` and `cfg`. You will also find a `Makefile`. When executing parts of the toolchain, you will want to run `make` in the `lab1` folder. The `Makefile` delegates to another `Makefile` that resides in the `cfg` folder. In the `cfg` folder you will also find other files that are used by tools during the build process. We will discuss every step of the toolchain.

4.1 Synthesis with XST

The synthesis tool (in this case of this class, 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.

XST also takes in 'Cores' which are pre-built and synthesized digital circuit blocks that provide some commonly needed functionality; they are usually provided by Xilinx to enable rapid FPGA development. They include things like pipelined dividers and multipliers, floating point units, and system buses.

XST also takes in 'Constraints' which are given in the form of a XCF file. In this lab, we don't use any constraints or cores that are passed into XST.

The outputs of XST include a LOG file, which is a text file that you can examine to make sure the synthesis succeeded, or if it failed or emitted warnings, you can see what those issues are.

Another output of XST is a NGR file which can be viewed with Xilinx ISE (as we will see soon); this file gives a high-level schematic view of how your Verilog modules are set to be implemented on the FPGA.

The final product of synthesis is a netlist file (NGC); it is a text file that contains a list of all the instances of primitive components in the translated circuit and a description of how they are

connected.

4.2 Translation and Mapping with NGDBuild and Map

The tools that perform translation and mapping 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 vc5vlx110t FPGAs we have in the lab.

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.

4.2.1 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 contains information regarding 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. These signals were `input [7:0] GPIO_DIP` and `output [7:0] GPIO_LED`. 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.

Open the UCF `lab0/ml505top.ucf` for Lab 0 and take a look. Here you can see where the `GPIO_DIP` and `GPIO_LED` nets come from. 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 for a multi-bit signal for the same net name. 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 just a statement of the voltage level for a given pin. Here is an example for a `GPIO_LED`

```
NET GPIO_LED<0> LOC="H18";
NET GPIO_LED<0> IOSTANDARD="LVCMOS25";
```

These 2 lines give you access to a net called `GPIO_LED` 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 LED0 on the board via 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 the ML505 development board, 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 ML505 user guide and ML505 schematic to figure out what peripheral connections you want brought into your FPGA design. You can find these three files in the `labs_fa16/docs` directory. We will discuss how to use these files in the design exercise in this lab.

4.3 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 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 quite sophisticated and have long run times.

4.4 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. We use a tool called BitGen to perform the generation of the bitstream that is sent to the FPGA. This is the last step in the FPGA build process, and it produces a .bit file which can be uploaded to the FPGA via the programmer.

4.5 Timing Analysis with TRCE

The `Makefile` we use in this class performs an additional step after running BitGen. To verify that our design met all timing requirements and to see a timing analysis, we 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.

4.6 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. We use a program called xreport which ships with Xilinx ISE to produce a report detailing the status of each build tool. To run this tool, execute `make report` in the `/lab1` directory. 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.

4.7 FPGA Programming with iMPACT

Finally, to send the bitstream file you generated with BitGen to the FPGA, we use a tool called iMPACT. To execute this tool run `make impact` in the `/lab1` directory after the regular `make`

process has completed and succeeded without any errors. iMPACT will send your bitstream file to the FPGA programmer over USB, and the FPGA programmer will send the bitstream to the FPGA over JTAG. After this tool runs, your design will be configured and active on the FPGA.

4.8 Toolchain Conclusion

All of this information is dense and complex. Don't worry about understanding the internals of each tool and the exact file formats they work with. Just understand what each step of the toolchain does at a high level and you will be good for this class. We use all these tools regularly, but executing them is all handled by the staff provided `Makefile`.

5 A Structural and Behavioral Adder Design + Using fpga_editor and the FPGA schematic

5.1 Build a Structural 4-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). You do not need to use for-generate loops to complete this task, but they will make it easier.

Navigate to the `/lab1/src` directory. Begin by opening `full_adder.v` and 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. You can manually instantiate four instances of the full adder cell or you can use a for-generate statement. You can add as many internal wires to your module as you want.

Finally, inspect the `ml505top.v` top-level module and see how your structural adder is instantiated and hooked up to the top-level signals.

Run `make` in the `/lab1` directory and let the build process complete. Then run `make impact` to send your design to the FPGA. 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 DIP switches and see that the correct sum is displayed on the first 5 LEDs.

5.2 Inspection of Structural Adder Using Schematic and fpga_editor

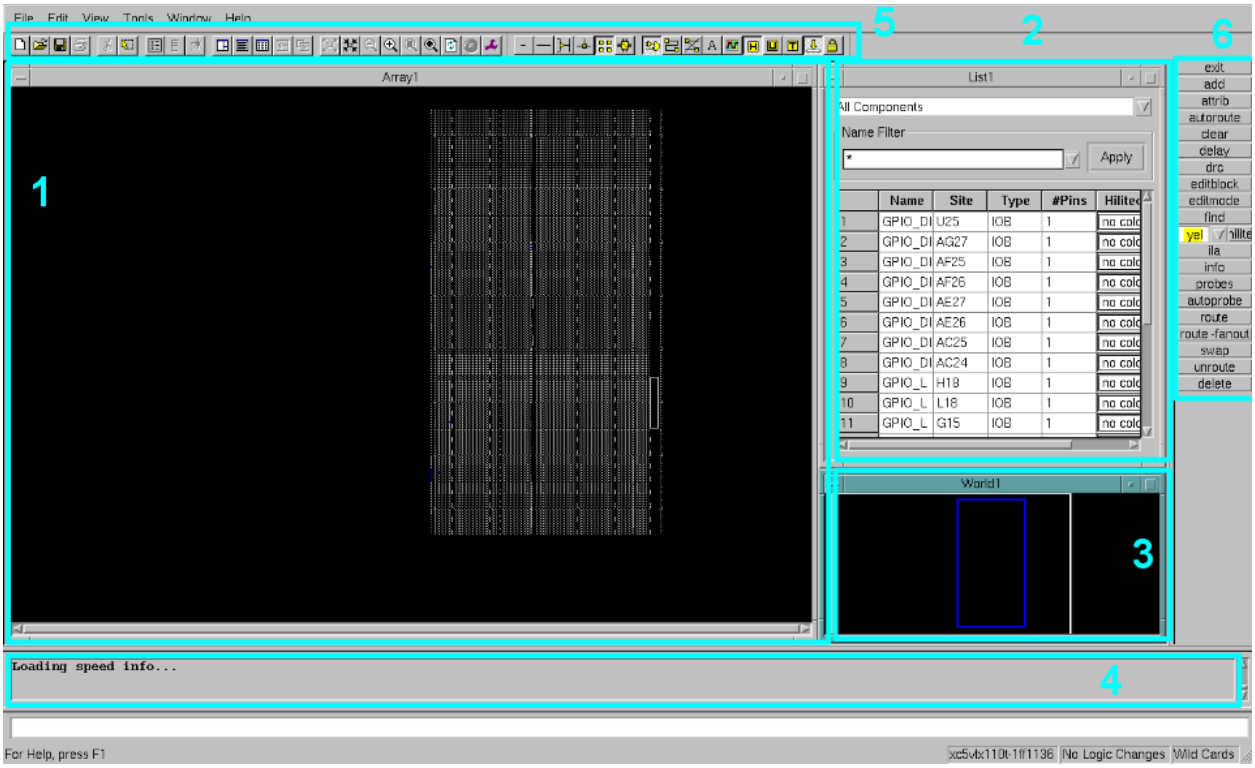
Now let's take a look at how the Verilog you wrote mapped to the primitive components on the FPGA. To do this, we will first look at a high-level schematic of your circuit using Xilinx ISE.

Execute the command `make schematic` in the `/lab1` directory. This will launch the ISE project navigator. Once inside ISE, go to `File -> Open` and select `ml505top.ngr`. You will recall that the NGR file is an output of XST (synthesis). Choose `Start with a schematic of the top-level block` in the dialog that pops up.

This will given you a fairly straightforward hierarchical block-level view of your design. You will find your circuit by drilling down in the following modules: `ml505top, st_add, full_adder`. Check to see that your structural adder module is hooked up properly and looks sane. It's ok if the wires aren't 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 in your circuit.

Now let's take a look at how your circuit was placed and laid out on the FPGA with the `fpga_editor`. Recall that the `.ncd` file is the final result of the FPGA toolchain flow. There also exists a `.pcf` file (Physical Constraints File) that contains the information originally present in the UCF, that is the locations of the I/O pins on the FPGA. By opening these files in the FPGA editor, you can visualize how your design will actually be mapped to the FPGA. Run these commands.

```
cd lab1/build/ml505top
DISPLAY=:0 fpga_editor ml505top.ncd ml505top.pcf
```



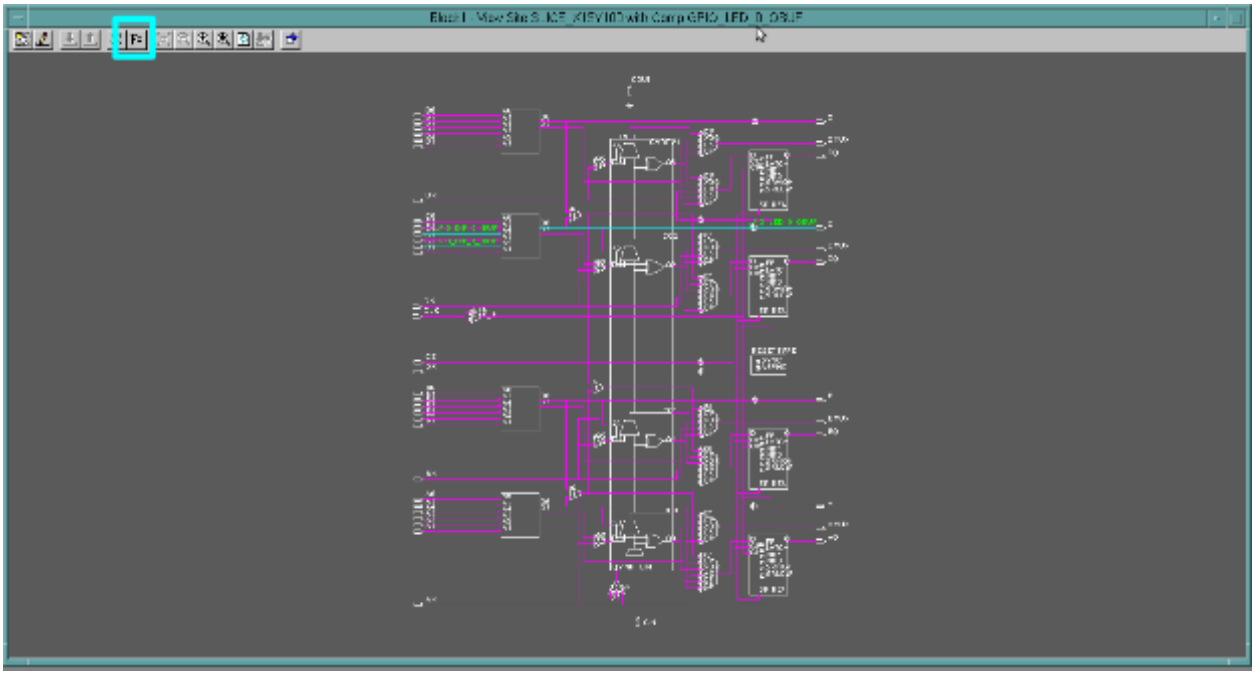
The image above shows the main view for the FPGA editor. It is split up into the following windows:

1. **Array Window** - shows a schematic of the FPGA and highlights the parts of the FPGA that are currently utilized, as well as the connections between these components.
2. **List Window** - lists the components and nets that are used in your design. Double-clicking items here will zoom in on them in the Array Window.
3. **World Window** - this shows you where you are currently focused in the Array Window.

4. **Console Output** - prints messages that often contain useful diagnostic information.
5. **Toolbar** - contains useful buttons for manipulating the windows; mouse over the buttons to reveal what is does.
6. **Button Bar** - contains other useful buttons for modifying the design.

Now you can explore your design and look for the modules that you wrote. If you scroll down in the `List Window` you will find that the type of some items is SLICEL.

Recall that SLICELs contain the look-up tables that actually implement the logic you want. Double clicking on the SLICEL named `GPIO_LED_OBUF` will take you to the corresponding SLICEL in your Array Window. Double click on the red rectangle (the SLICEL) to bring up the following Block Window:



Click on the `F=` button to reveal the function of the LUT. Go ahead and explore several SLICELs to see how they are connected to each other and the outputs of your circuit.

Finally you should run `make report` to collect resource usage information for your structural design. Record the number of LUTs used.

5.3 Build a Behavioral 4-bit Adder

Check out `behavioral_adder.v`. It has already been filled with the appropriate logic for you. Open `ml505top.v` and comment out the structural adder instantiation and uncomment the behavioral adder instantiation. Then run `make` and `make impact`. Verify that the adder circuit that you constructed by hand performs like this behavioral adder.

5.4 Inspection of Behavioral Adder Using Schematic and fpga_editor

Go through the same steps as you did for inspecting the structural adder. First run `make schematic` and then run the FPGA editor. Record and note down any differences you see between both types of adders in the schematic and the FPGA editor. 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 the `ml505_user_guide.pdf` in the `labs_fa16/docs` folder. On page 20, you can see a list of oscillators that are present on the development board. These clock signals are generated outside the FPGA by a highly accurate crystal or a programmable clock generator IC. These clock signals are then connected to pins on the FPGA so that they can be brought into your Verilog top-level module.

Take a look at the `ml505top.v` module and notice the `CLK_33MHZ_FPGA` input. Next take a look at the UCF `ml505top.ucf` and notice how the LOC for the clock net is set to AH17, just as specified in the ML505 User Guide. Basically, we have a 33Mhz clock signal that is generated on a clock generation IC on the ML505 board and is then routed to the FPGA's AH17 pin. We can access this pin from within our Verilog top-level module and can propagate this clock signal to any submodules that may need it. This is a 33 Mhz clock signal, so there are a couple other lines in the UCF that specify the period of the clock so that timing checks can be performed when place and route or trace are being executed.

6.1 Piezoelectronic Buzzers

Take a look at page 14 of the ML505 User Guide to see a block diagram of all the signals that come into our FPGA and can be monitored/driven from our Verilog design. The fourth from the top box on the left is labeled Piezo/Speaker. To get more information about this component go to page 37.

Notice how the table specifies that the piezo signal that is routed to the actual component on the board can be driven by FPGA pin G30. Keep this in mind.

Take a look at this wiki page to see how a piezoelectric speaker functions.

6.2 Finding the Piezo in the Schematic

Let's look into how the piezo is connected to the FPGA and the ML505 board. First take a look at the datasheet for the piezo on this board. You can find the datasheet in `labs_fa16/docs` with filename `AT-1220-TT-2-R.pdf`. The datasheet has a picture of how the piezo speaker looks; now

find it on the ML505 board!

Once you have located the piezo speaker, let's see the schematic of the board to see how it is driven by the FPGA. Open the schematic file in the `labs_fa16/docs` folder called `ml50x_schematics.pdf`. Go to page 3 of the schematics and take a look at the Bank 15 block. The third pin from the top on the right side is labeled `PIEZO_SPEAKER`. Notice also how `G30` is listed next to the net indicating that is connected to the G30 pin on the FPGA. Bank 15 is an I/O block on the FPGA; general purpose I/O connections on the FPGA are organized in many banks, each with some special connections and voltage inputs.

Clicking on the `PIEZO_SPEAKER` label will take you to page 18 of the schematic. Here you can see the piezo net going into an IC (SN74LVC1G126). This IC just acts as a buffer for the signal coming from the FPGA allowing a greater current to be sent into the piezo speaker. From this IC you can see the signal go through a 10uF capacitor before going into the piezo (SP1 in the schematic).

6.3 Adding the Piezo signal to the UCF File

Now let's add the piezo connection to the UCF so that we can bring it in as an output from the Verilog top-level module. To do this, we will use the master UCF file. Find this in the `labs_fa16/docs` folder with the filename `master_xupv5-lx110t.ucf`. Open the master UCF and search for 'piezo'. You will find on line 368 the net declaration for the `PIEZO_SPEAKER` signal. Copy this line into your `ml505top.ucf` file. Specify the IOSTANDARD for this net in the UCF as `LVCMOS18` since the bank (Bank 15) it is connected to has a Vcco of 1.8V.

Ask a TA if you need help for this part.

6.4 Generating a square wave

In your `ml505top.v` file comment out your adders and uncomment the `tone_generator` instantiation. Add an input to the top-level module with a matching net name to the net you declared in the UCF. Connect the `tone_generator` instance to the piezo input.

Now perform some quick calculations. Let's say we want to play a 440 Hz square wave into the piezo speaker. We want our square wave to have a 50% duty cycle, so for half of the period of one oscillation the wave should be high and for the other half, the wave should be low. We have a 33 Mhz clock input we can use to time our circuit and wave generation.

Find the following:

1. The period of our clock signal (frequency = 33 Mhz)?
2. The period of a 440 Hz square wave?

3. Approximately how many clock cycles pass during one period of the square wave (round to even number)?

Now, 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.

Next, modify the `always @ (posedge clk)` block to increment the `clock_counter` on every cycle until it hits your calculated number of cycles, upon which the counter should be set to zero.

Finally, use an assign statement or an always @ (*) block to set `square_wave_out` to either 1 or 0 depending on whether the `clock_counter` is below or above the middle of the number of clock cycles that pass during one period of the square wave.

Now `make` and `make report`. Check for any warnings or errors and try to fix them. Ask a TA if you need help here. When everything looks good run `make impact`. You should now hear a buzzing noise at 440Hz.

6.5 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 full square wave to `square_wave_out`. You can accomplish this by adding a wire in your module which stores the square wave signal, and then using an assign statement to assign `square_wave_out` based on the value of `output_enable`.

Now `make` and `make report`. Check for any warnings or errors and try to fix them. Ask a TA if you need help here. When everything looks good run `make impact`. You should now hear a buzzing noise at 440Hz that can be turned on or off by toggling the 0th DIP switch.

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

This part is completely optional and is self-directed. You can use the remaining 7 DIP switches as inputs into your `tone_generator` and can use those switches to encode a binary number that represents a particular duty cycle for your square wave or a particular range of frequencies. Show that these switches give you control over the duty cycle or frequency of your square wave output to the piezo speaker.

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 440Hz
4. Demonstrate your tone generator on the FPGA and show that the 0th DIP switch controls the buzzing of the piezo speaker

You are done with this lab. In the next lab, we will extend our `tone_generator` to read a song from a block RAM and play it out over the piezo speaker. We will explore state machines and simulation/functional verification.