EECS 151/251A FPGA Lab
Lab 2: Introduction to FPGA Development

Prof. John Wawrzynek
TAs: Quincy Huynh, Tan Nguyen
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.

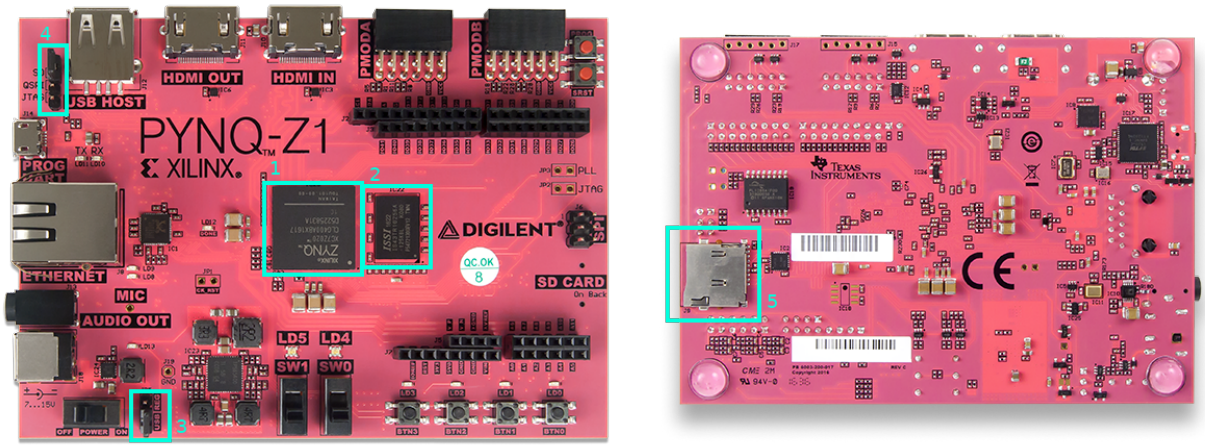
To fetch the skeleton files for this lab, `cd` to the git repository (`fpga_labs_sp20`) that you had cloned in the first lab and execute the command `git pull`.

Take some time to go through the Verilog Primer Slides and review our lecture slides on Verilog. You should understand some fundamental concepts of hardware design in Verilog, such as combinational logic and sequential logic, in order to complete this lab.

This lab will walk you through how to create and manage an FPGA project in Vivado Design suite. It is meant to be long and tedious. Don't be intimidated! As you will see, being a good hardware designer does not mean just writing Verilog or any HDL; you also need to have a firm understanding on the design tool. While it is impossible to cover many details of Vivado in one lab, the steps we show here are sufficient to get you familiar enough with the tool, and thus able to finish the labs and project later. If you already have some experience with the tool, feel free to jump straight to the lab exercises.

2 Our Development Platform - Xilinx PYNQ-Z1

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. Z-7020 System-on-Chip (SoC) of the Zynq-7000 SoC family. It comprises a hardened dual-core ARM processor and the Xilinx FPGA Artix-7 xc7z020clg400-1. The SoC connects to the peripheral ICs and I/O connectors via PCB traces.
2. ISSI 512MB off-chip DRAM.
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 to select how we want to use the ARM processor. There are two available modes: Operating-System mode (booting Linux from SD card) or Bare-metal mode. Since we are not touching the ARM processor, we avoid this for now.
5. SD card slot for inserting an SD card to boot Linux. Since we are not touching the ARM processor, we avoid this for now.

3 The FPGA - Xilinx Artix-7 xc7z020clg400-1

xc7z020clg400-1 is the part ID of our FPGA. How should we interpret it? xc7z020 is the part number which should help us to identify the specific device from a Xilinx FPGA family. clg400 is the package number which defines how many package IO pins. -1 is the speed grade.

Our FPGA is an Artix-7 Programmable Logic which is a low-end 7-series Xilinx FPGA family (the mid-end and high-end are Kintex-7 and Virtex-7 families, respectively). 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. Please also read Zynq-7000 Product Selection guide. Pay attention to slide 2. Can you identify our chip?

One of the key differentiators of FPGA devices is their logic capacities. You should be aware of the device resource of your target FPGA when designing your digital circuit (it is unlike the software world that the CPU or GPU should be able to compile and run whatever code throwing at it regardless of the code size). Early FPGAs employ primitives such as LUTs or FFs (Flip-flops) for logic implementation. Then, FPGA vendors started adding hardened blocks such as fast carry adders, block memories (BRAM) and Digital Signal Processing (DSP) slices onto FPGAs to augment their capability. The carry adder macros can implement fast arithmetic and comparison operations, the BRAMs provide fast on-chip storage, and the DSP slices are able to compute multipliers very efficiently, among many other operations. State-of-the-art FPGAs also incorporate floating-point calculation capability in those hardened blocks, thus greatly enhance the performance and expand their applicability. FPGA now has evolved to a competitive computing platform, and there are many applications that map really well on to FPGA, such as networking, wireless, biology, video/image processing, high-frequency trading, or deep learning. The Zynq-7000 product line (which incorporates ARM processors next to a Programmable Logic – as in the chip we are using right now) also provides great platforms for embedded or hobbyist applications.

You will need to answer the following questions in your lab report.

3.1 Understanding your FPGA

1. How many LUTs, FFs, Block RAMs (number of 36Kb blocks), and DSP slices are on the xc7z020 FPGA?
2. How many SLICEs are in a single CLB? What does each SLICE contain?
3. What is the difference between a SLICEL and a SLICEM?
4. How many inputs do each of the LUTs have?
5. 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.

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 a circuit implemented in a Hardware Description Language (such as Verilog) into a bitstream that configures the FPGA. These tools will pass your design through several stages, starting with logic synthesis, followed by placement and routing. The final stage generates a bitstream ready to download to your FPGA. In previous years, older evaluation platform (the ML505) used older FPGAs (a Xilinx Virtex-5 LX110T) and an older software suite (Xilinx ISE).

Our new boards use Xilinx's updated design software, the Vivado Design Suite which supports modern Xilinx FPGA devices, and features improved CAD algorithms to produce higher quality-

of-result in shorter time. Vivado also has integrated scripting capability (using the Tcl language – pronounced "tickle") which allows users to write Tcl commands to interact with Vivado in a command-line fashion. It is a great alternative to the GUI mode. A good way of learning the tool without relying on the GUI is to be aware of different Tcl commands, and how to use them.

Please ask a TA if you are unclear about any of the steps below.

4.1 Set up your PYNQ-Z1

1. Plug in the power adaptor to provide mains power.
2. Connect the USB interface to a spare USB port on your workstation. Make sure you have installed Digilent cable driver if you use Vivado locally on your own machine.
3. Turn the board on.

4.2 Verilog source file

Throughout the semester, you will build increasingly complex designs using Verilog, a widely used hardware description language (HDL).

Open up the `lab2/src/z1top.v` source file. This file contains a Verilog module description which specifies input and output signals. It also describes the logic gate connecting the signals.

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. There are only 4 LEDs there; two 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!

Open up the constraint file in `lab2/constraints/z1top.xdc`. This file, which contains a bunch of Tcl commands, specifies various constraints of your digital circuit, such as IO pin mapping. Note how the signals in the Verilog correlate with the constraint commands in this file. As we will see later, the constraint file also allows you to set the clock signal (where, and how fast) for your sequential digital circuit.

Let's see one example:

```
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.

Setting constraints is one of the most important steps in the FPGA development flow, especially if your circuit interfaces with the external world (receiving signals or sending signals via IO circuitry of the board). Please do not forget this step. You will not be able to generate a bitstream if you do not set the pin mapping to all the input/output signals of your top-level Verilog module; Vivado will complain and abort its execution. And your FPGA design will not work properly if your pin assignment is wrong. A good practice is to check the documentation of your target FPGA device carefully of which IO peripheral maps to which pin, and the appropriate logic standard unless you can get the sample board constraint file from the vendor.

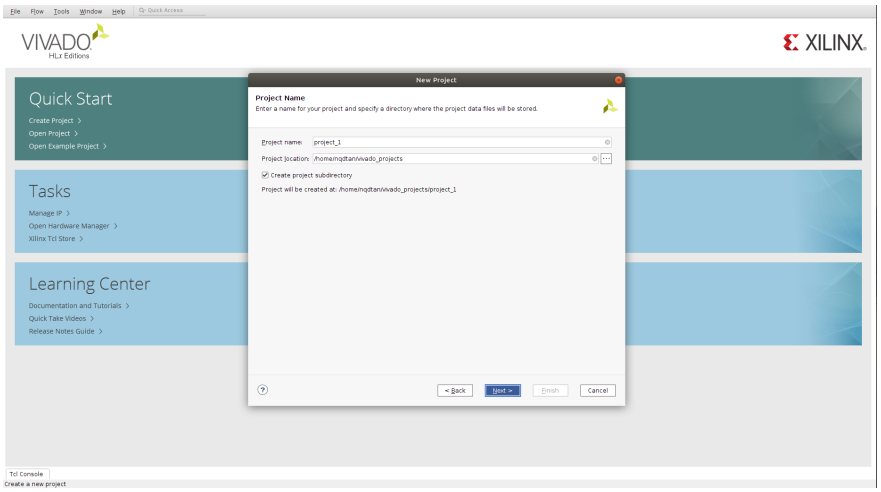
4.3 Creating Vivado Project

This section describes how to create a Vivado project using GUI mode. Project is a neat way of managing your design and constraint files, exercise different design steps, or review the design reports. In our CentOS environment, press `Alt + F2` to bring up a command dialog. Type the full path to the `vivado` binary to open Vivado GUI locally in your machine:

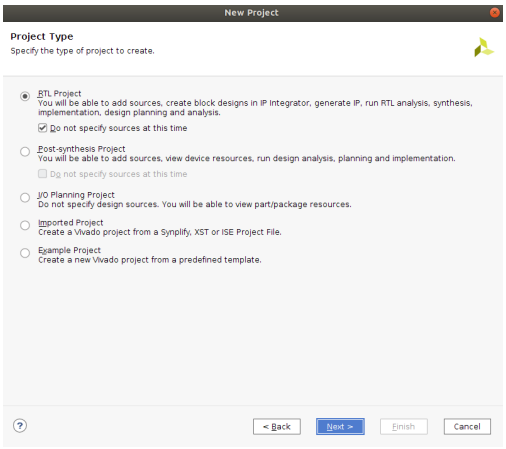
```
/opt/Xilinx/Vivado/current/bin/vivado
```

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

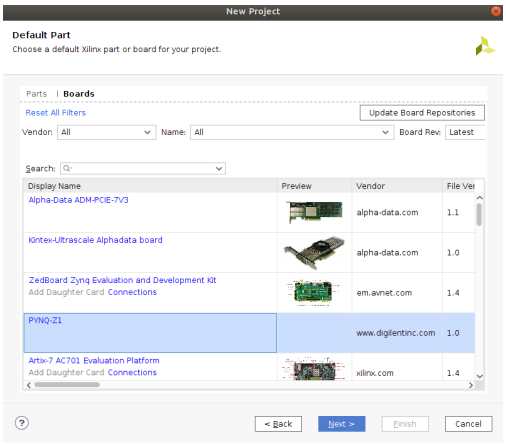
Create new project by select *File* → *New* → *Project* to launch the following wizard.



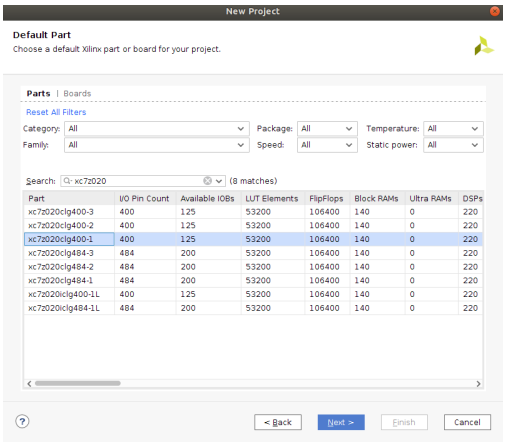
Type the project name and location. You can leave them as default. Click *Next*. In the next window, make sure to select *RTL project*, and *Do not specify sources at this time*.



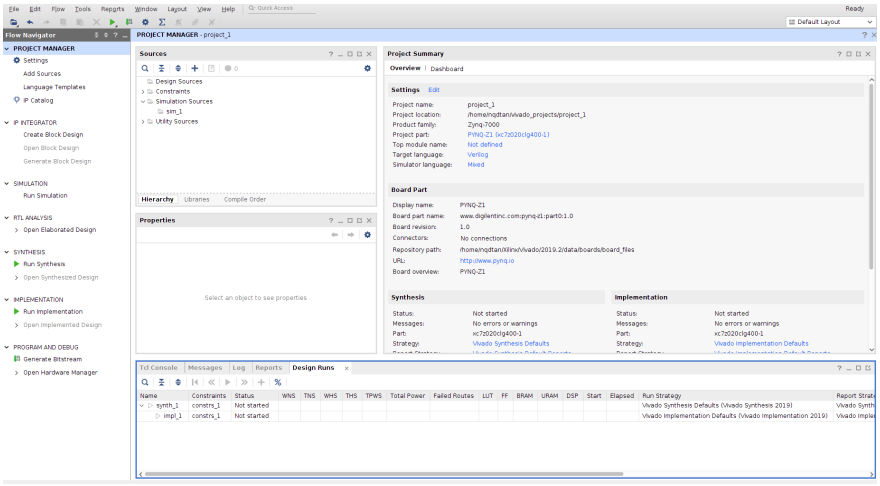
Next, we select the target FPGA device for the project. Click the *Boards* tab, find the PYNQ-Z1 entry and select it.



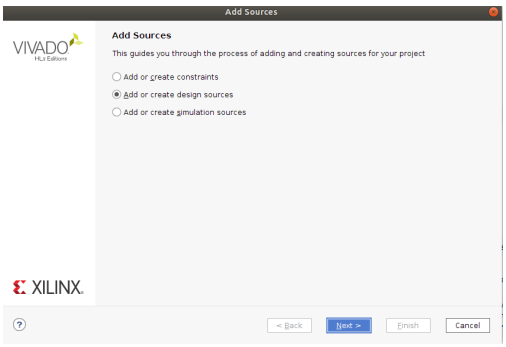
If you cannot find the PYNQ-Z1 entry, chance is your Vivado installation does not include the PYNQ-Z1 board files. Follow the instructions from Vivado board files to add the PYNQ-Z1 files to your Vivado installation. Alternatively, you can select the part ID xc7z020clg400-1 from the *Parts* tab as follows.



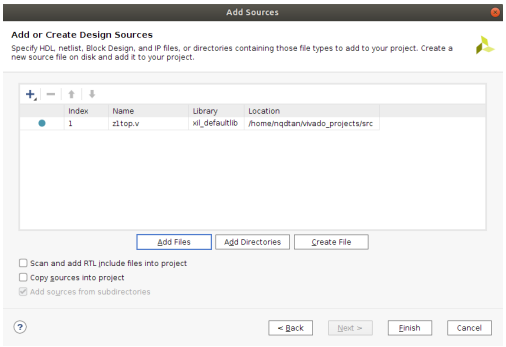
Once you are done, you should be able to see a window as follows.



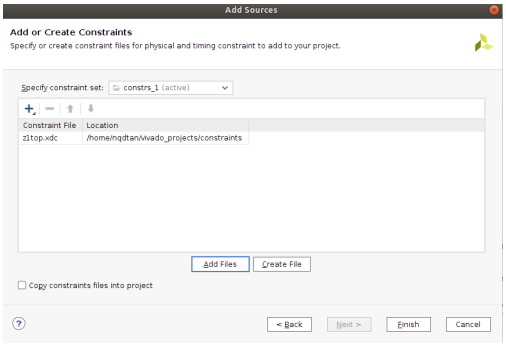
Now we start adding our design and constraint files to the project. Press `Alt + A` to add files as follows (or right click on the Sources pane and select *Add Sources*).



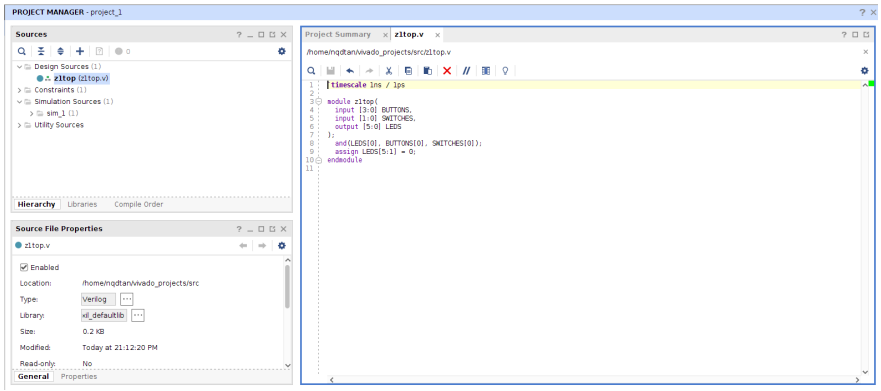
Add your Verilog source file(s).



And repeat the step to add your constraint file.



Our first Vivado project is good to go now! Note that you can also use Vivado code editor to write Verilog code. This might be a good starting point if you first learn the language since the tool is able to capture syntax errors or obvious bugs quickly which could save you from a lot of potential miseries down the road.



4.4 Synthesis

The Synthesis step performs logic synthesis which translates the input HDL code into a netlist of FPGA primitives such as LUTs, FFs, BRAMs, or DSPs. Remember that there is no logic gate at all in an FPGA! To run the synthesis step in the Vivado Design Suite, select *Run Synthesis* in the *Flow Navigator* pane on the left-hand side of the window. If this has been run before, the

synthesized design can be inspected by selecting *Open Synthesized Design*. Select *Schematic* under "Open Synthesized Design" on the left toolbar to see a circuit schematic.

If you wish to see the netlist in the form of Verilog, type the following Tcl command

```
write_verilog my_netlist.v
```

in the *Tcl Console* at the bottom pane. Check that the AND gate has now mapped to a 2-input LUT.

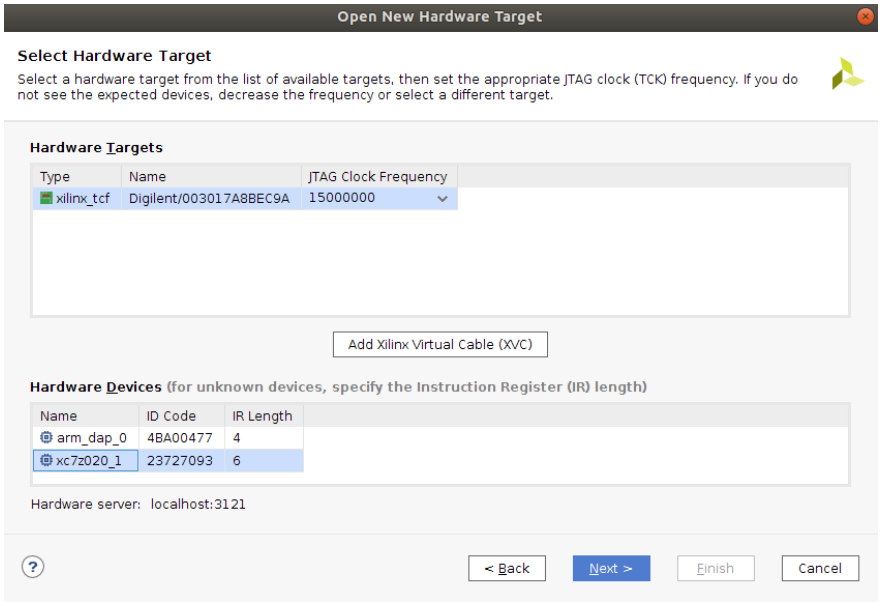
4.5 Implementation

The Implementation step performs placement and routing the design. Placement places the primitives in the netlist to the physical locations on the FPGA. Routing connects the placed blocks together using switch blocks and wires on the FPGA. The next step is timing analysis which evaluates if your design meets the target clock constraint (this only applies if your design has sequential elements). Implementation is the most time-consuming task of a hardware design flow, especially if you have a complex design. Select *Run Implementation* in the *Flow Navigator* to run it, then select *Open Implementation* to inspect its outputs.

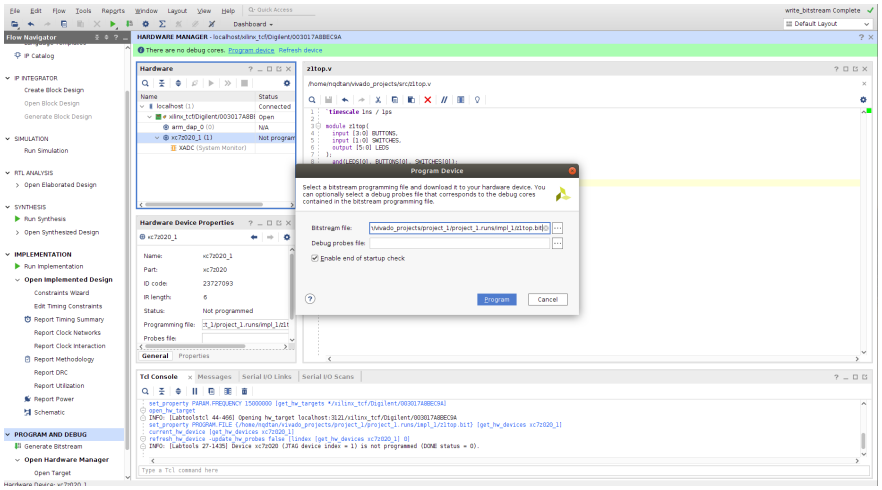
4.6 Generate Bitstream and Program FPGA

To generate the bitstream for our target FPGA, we invoke *Generate Bitstream* in the *Flow Navigator*. This step will generate a *.bit file that can be written to the configurable memory cells of the FPGA. The bitstream is compatible with the target device of your Vivado project.

When the 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*. In the Hardware Server Setting, make sure you pick *Local server* (if you are using Vivado locally). 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*.



Next, click *Program device* to load the bitstream to your connected FPGA. The "Done" LED (near the Zynq chip) should be on if you successfully configure the FPGA.



See if it worked! What happens when you push the BTN0 button? What about when you change SW0? Both?

4.7 Design Reports and Logs

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. The reports tell you the Quality of Result (QoR) of your design, such as how many LUTs, FFs, BRAMs, DSPs are used, and whether your circuit meets the target clock constraint. You can open the reports from your terminal using your favorite text editor. The following reports are

particularly important and you should pay attention to when evaluating, debugging, or optimizing your design.

`project_1/project_1.runs/synth_1/z1top_utilization_synth.rpt`: Resource utilization report after synthesis.

`project_1/project_1.runs/impl_1/z1top_utilization_placed.rpt`: Resource utilization report after placement.

`project_1/project_1.runs/impl_1/z1top_utilization_routed.rpt`: Timing summary report after routing.

You will often find the log files helpful when you want to look for more information of how your Verilog code is compiled to a bitstream. Make sure you get what you want (if not, why) by putting effort in understanding the messages as well as warnings or errors if any from the log files.

`project_1/project_1.runs/synth_1/runme.log`: Execution log of synthesis step

`project_1/project_1.runs/impl_1/runme.log`: Execution log of implementation step

5 Build Your First Combinational Digital Circuit

In this section, you will design a 3-bit adder using either structural or behavioral approach in Verilog.

5.1 Build a Structural 3-bit Adder

What do we mean by "structural approach"? It builds a circuit using only Boolean logic gates and multiplexers. It's similar to how you are drawing your design in a schematic editor, but we use Verilog instead. To help you with this task, please refer to the 'Code Generation with for-generate loops' slide in the Verilog Primer Slides (slide 35).

1. Create a new project (or reuse existing project). Add the following Verilog files to your project: `lab2/src/full_adder.v`, `lab2/src/structural_adder.v`, `lab2/src/z1top_adder.v`. Also add the constraint file to your project: `lab2/constraints/z1top.xdc`.
2. Open `full_adder.v`; fill in the logic to produce the full one-bit adder outputs from the inputs.
3. Open `structural_adder.v` and construct a ripple carry 3-bit adder using the full one-bit adder cells you designed earlier. Use 'for-generate loop' construct to make your code compact.
4. Finally, inspect the `z1top_adder.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. Note how the adder operands and result are constructed from the input/output signals of the top-level Verilog module, and in turn the buttons and switches and LEDs on the FPGA.

5. Set `z1top_adder.v` as the top-level module by right-clicking `z1top_adder` in Sources → Design Sources and click 'Set as Top'. You can skip this step if it is already the top-level module.
6. Run bitstream generation as usual.
7. Program the FPGA and test out the design. Try different combinations of buttons and switches. Make sure you get correct results from your adder via the LEDs.

5.2 Inspection of Structural Adder

5.2.1 Schematics and FPGA Layout

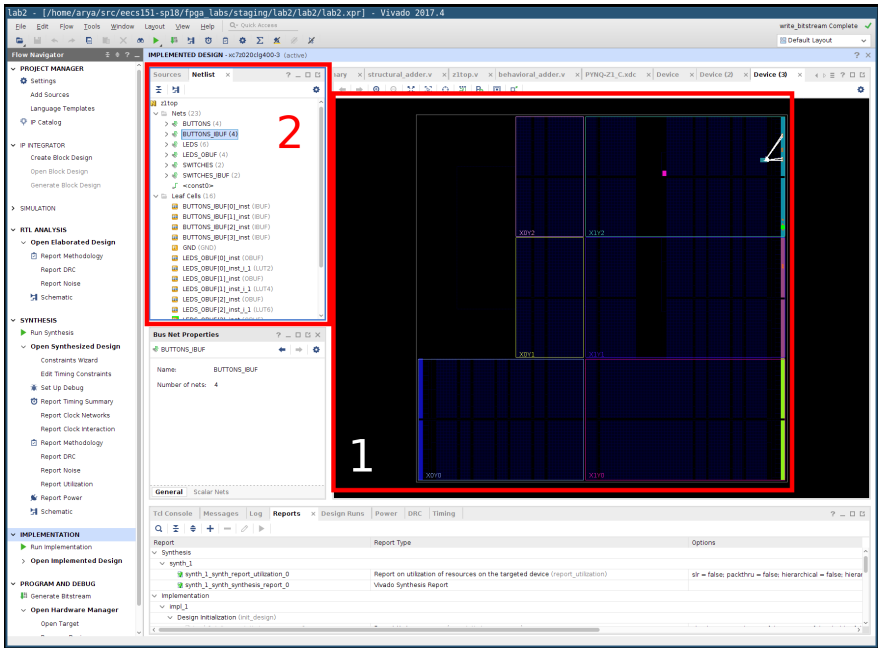
Take a look at how the Verilog you wrote mapped to the primitive components on the FPGA. Three levels of schematic are generated for you once you've run the pipeline. In the *Flow Navigator*, you can view *Schematic*s under

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_adder.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 visualization 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.

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).



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. 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 3-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. Modify your top-level module in `z1top_adder.v` to use the behavioral adder instead of the structural adder. Generate and load the bitstream to the FPGA. Verify if the behavioral adder works correctly.

Behavioral coding style allows designers to build their designs using arithmetic operators, comparators, or software-like constructs such as `if-else`, `switch-case` (to generate muxes), which greatly improves productivity.

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.

6 Build Your First Sequential Digital Circuit

In this section, you will design a 4-bit counter that increments every one second. Use the four LEDs to verify to counter's status.

6.1 Clock Sources

Look at the PYNQ Reference Manual. Read Section 11 about the clock sources available on the PYNQ. We are using the 125 MHz clock from the Ethernet PHY IC on the PYNQ board connected to pin H16 of the FPGA chip.

Look at the `lab2/src/z1top_counter.v` top-level module and its `CLK_125MHZ_FPGA` input.

```
module z1top_counter (
    input CLK_125MHZ_FPGA,
    ...
);
```

Next take a look at the constraints in `lab2/constraints/z1top_counter.xdc` and notice how the LOC for the clock net is set to H16, just as specified in the PYNQ-Z1 Reference Manual.

```
set_property -dict { PACKAGE_PIN H16 IOSTANDARD LVCMOS33 } [get_ports { CLK_125MHZ_FPGA }];
```

We can access the clock signal from our Verilog top-level module and can propagate this clock signal to any submodules that may need it.

Also note that we added the following constraint on the clock signal.

```
create_clock -add -name CLK_125MHZ_FPGA -period 8.00 -waveform {0 4} \
[get_ports { CLK_125MHZ_FPGA }];
```

This constraint informs Vivado that our input clock signal has a period of 8 ns (125 MHz) which matches the actual clock frequency from pin H16. This is the target clock constraint (or often referred as frequency constraint) of your design. Vivado will try to optimize your design to meet this constraint, and the post-routing timing analysis will report whether it is the case. Please do not forget to set a clock constraint when you design a sequential circuit.

6.2 Build a 4-bit Counter

Your circuit receives an input clock signal of 125 MHz (8 ns). Remember that

$$Latency\ (sec) = Frequency \times Clock\ cycles$$

How many clock cycles are equivalent to one second? In other words, how many clock cycles do you need to wait before incrementing your 4-bit counter? You may use an additional counter to keep track of this.

You are required to use the library of registers that we provide you (`lib/EECS151.v`). Please instantiate a REGISTER module whenever you want to build a register. Please do *not* write

any sequential always block (e.g., `always @(posedge clock)`) or non-blocking assignment in your code.

1. Create a new project (or reuse existing project). Add the following Verilog file to your project: `lab2/src/z1top_counter.v`.

   Also add the constraint file to your project: `lab2/constraints/z1top_counter.xdc`.

2. Open `z1top_counter.v`; fill in the necessary logic to implement the counter. Some initial code has been provided to help you get started.

3. Set `z1top_counter.v` as the top-level module by right-clicking `z1top_counter` in Sources → Design Sources and click 'Set as Top'. You can skip this step if it is already the top-level module.

4. Generate the bitstream, program the FPGA and test out the design.

5. Check the timing report. Does your circuit meet the target timing constraint?

7 Lab Deliverables (due: 11.59PM, Feb 6th, 2020)

7.1 Lab 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 a working 3-bit structural adder on the FPGA.

3. Demonstrate a working 4-bit counter that increments every second on the FPGA.

7.2 Lab Report

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. What is the resource utilization (in terms of LUTs) of your 3-bit structural adder? What about the 3-bit behavioral adder?

3. What is the resource utilization (in terms of LUTs and FFs) of your 4-bit counter? Include a screenshot of how those LUTs and FFs are placed and routed after *Implementation* in your report. Also include `z1top_counter.v` code in your report (copy-paste or screenshot is fine).

A Using Vivado in command-line mode (Tcl mode)

You can launch Vivado in command-line mode without opening the GUI as follows.

```
/opt/Xilinx/Vivado/current/bin/vivado -mode tcl
```

Then you can enter Tcl commands to interact with Vivado. Here is a minimal set of Tcl commands from creating Vivado project until prgramming FPGA.

```
# Create Vivado project "project_1" under the directory "my_project_dir"
# with the target FPGA device "xc7z020clg400-1"
create_project project_1 my_project_dir -part xc7z020clg400-1

# Set a specific board configuratio (PYNQ-Z1)
set_property board_part www.digilentinc.com:pynq-z1:part0:1.0 [current_project]

# Add design and constraint files
add_files -norecurse src/z1top.v
add_files -fileset constrs_1 -norecurse constraints/z1top.xdc

# Update project
update_compile_order -fileset sources_1

# Set top-level module
set_property top z1top [current_fileset]

update_compile_order -fileset sources_1

# Run Synthesis;
# Check my_project_dir/project_1/synth_1/runme.log for Synthesis output
launch_runs synth_1

# Run Implementation;
# Check my_project_dir/project_1/impl_1/runme.log for Implementation output
launch_runs impl_1

# Run the whole flow (including Synthesis and Implemention if not yet run)
# until bitstream generation;
# Check my_project_dir/project_1/impl_1/runme.log for Implementation output
launch_runs impl_1 -to_step write_bitstream

# Establishing connection with local connected FPGA (only need to run once
# if Vivado can connect the FPGA)
open_hw_manager
connect_hw_server -url localhost:3121 -allow_non_jtag
open_hw_target

# Specify the bitstream for configuration
set_property PROGRAM.FILE {my_project/project_1.runs/impl_1/z1top.bit} \
[get_hw_devices xc7z020_1]
```

```
current_hw_device [get_hw_devices xc7z020_1]
refresh_hw_device -update_hw_probes false [lindex [get_hw_devices xc7z020_1] 0]

# Load the bitstream to FPGA
program_hw_devices [get_hw_devices xc7z020_1]
refresh_hw_device [lindex [get_hw_devices xc7z020_1] 0]
```

As an optional exercise, you can try those scripts above to see if you can generate bitstream and configure your FPGA. You can also put all the Tcl commands into one file and run it in batch mode as follows.

```
/opt/Xilinx/Vivado/current/bin/vivado -mode batch -source my_script.tcl
```

This will invoke Vivado to run all the Tcl commands written in `my_script.tcl`.

You can learn more about different Tcl commands in Vivado from this extensive document Vivado Tcl commands. You might also be interested to learn about Vivado non-project mode which provides more flexilibity in what specific steps you want to run, or reports you want to generate. Keep in mind that the Tcl commands for non-project mode are slightly different from project-mode. Here is an additional resource to learn about the tool Vivado Design flows Overview Tutorial.

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
- Fa19: Cem Yalcin, Rebekah Zhao, Ryan Kaveh, Vighnesh Iyer