Lab 1: FPGA Physical Layout

University of California, Berkeley

Department of Electrical Engineering and Computer Sciences

EECS150 Components and Design Techniques for Digital Systems

Written by John Wawrzynek, James Parker, Daiwei Li

Updated by Ronald Fearing, Austin Buchan, Stephen Twigg

Due September 19th, 2013

Table of Contents

0 Introduction

In the previous lab, you learned how to manipulate inputs and outputs on the FPGA using Verilog code. In this lab, you will see how the Verilog code you wrote in the previous lab maps to resources on the FPGA. Throughout the rest of the semester you will be focusing on more abstract methods of describing circuits, and the tools that are used to transform that description into actual hardware.

As you will see, being able to represent hardware in a more abstract way increases productivity and design flexibility, but also hides some of the details of the FPGA platform. Completing this lab will help to establish the underlying architecture of the FPGA in your mind, as well as help you to make optimal design decisions in the future.

1 Prelab

To help you become familiar with the FPGA that you will be working with through the semester, please read *Chapter 5: Configurable Logic Blocks (page 171)* of the Virtex-5 User Guide and answer the following 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 of LUTs does the LX110T have?
4. How do you implement logic functions of 7 inputs in a single SLICEL? How about 8?
5. What is the difference between a SLICEL and a SLICEM?

2 CAD Tools

Before we dive into the lab, let us 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. Brief descriptions of these stages follow:

2.1 Synthesis

The synthesis tool (in the case of this lab, 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 final product of the design partitioning phase is a netlist file, 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.

2.2 Translate & Map

The translate & map stages are responsible for taking the generic netlist produced by the synthesis stage and translating each component to an equivalent on the specific Xilinx *xc5vlx110t* FPGAs we have in the lab.

2.3 Placement

Placement takes a mapped design and determines the specific location of each component in the design on the FPGA. For example, each slice in the mapped designed is assigned to a specific site.

2.4 Routing

Once placement is done, all of the connections specified in the netlist must be made. In general, routing is one of the most time-consuming parts of computer-aided circuit design. Optimizing placement can help speed up the process.

3 Lab Procedure

In this lab, you will use a tool called FPGA Editor to manipulate logic and nets in the design that you wrote in Verilog in the last lab. In practice, you would not want to design your logic in FPGA editor, as it is much slower and extensible than writing Verilog code. However, this exercise is useful for visualizing the hardware and understanding the underlying processes.

3.1 Setting up

If you would like to be able to copy paste commands from this lab guide, first type the following command so that the % prompt in front of each command is ignored:

```
alias %=''
```

Grab the resources for this lab; we will be using them later in the lab.

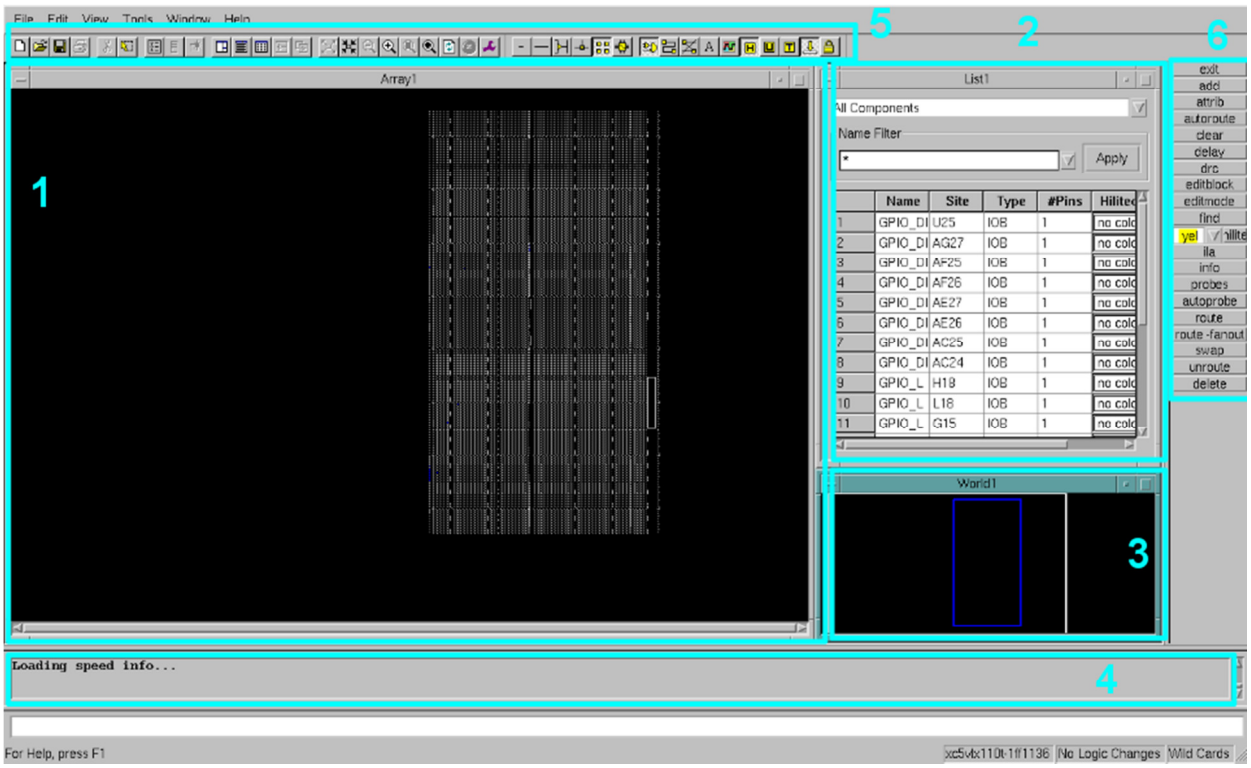
```
% wget http://inst.eecs.berkeley.edu/~cs150/sp13/lab1/lab1.tar.gz
% tar -xvzf lab1.tar.gz
```

Build the project from last week's lab0 (if you haven't done so already; you only need to make if there isn't already a build directory in lab0). You will need the build files in order to complete the first part of the lab.

```
% cd ~/lab0
% make
```

Open your design with FPGA editor. The .ncd file is the final result of the CAD tools design flow discussed above. The `.pcf` includes physical constraints; in our case, it tells us where the I/O pins are actually located on the board. By opening these files in FPGA editor, you can visualize how your design will actually be mapped to the FPGA.

```
% cd ~/lab0/build/ml505top
% fpga_editor ml505top.ncd ml505top.pcf
```



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

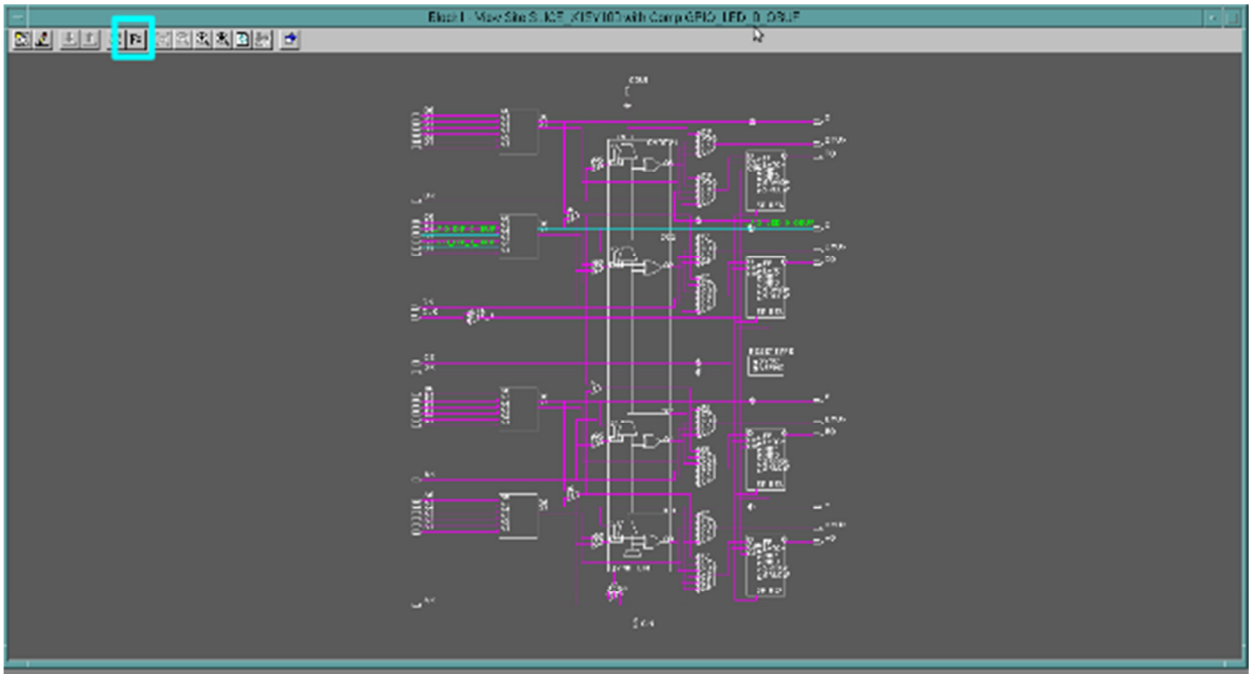
1. `Array Window` - shows a schematic of the FPGA and highlights the part of the FPGA that are currently utilized, as well as the connections between these components
2. `List Window` - lists the components and nets that are used in your design. Double-clicking items here will zoom in on them in the Array Window
3. `World Window` - this shows you where you are currently focused in the Array Window
4. `Console Output` - prints messages that often contains useful diagnostic information
5. `Toolbar` - contains useful buttons for manipulating the windows; mouse over the buttons to reveal what it does
6. `Button Bar` - contains other useful buttons for modifying the design

3.2 Modifying the Design

Now let's explore the design and look for the modules that you wrote in the previous lab. If you scroll down in the `List Window`, you will find that the Type of the last few items is *SLICEL*.

Recall from the prelab reading that *SLICEL's* contain the look-up tables (LUTs) that actually implement the logic you want. Double-clicking on the *SLICEL* named `GPIO_LED_0BUF` will take you to the corresponding *SLICEL* in your `Array Window`.

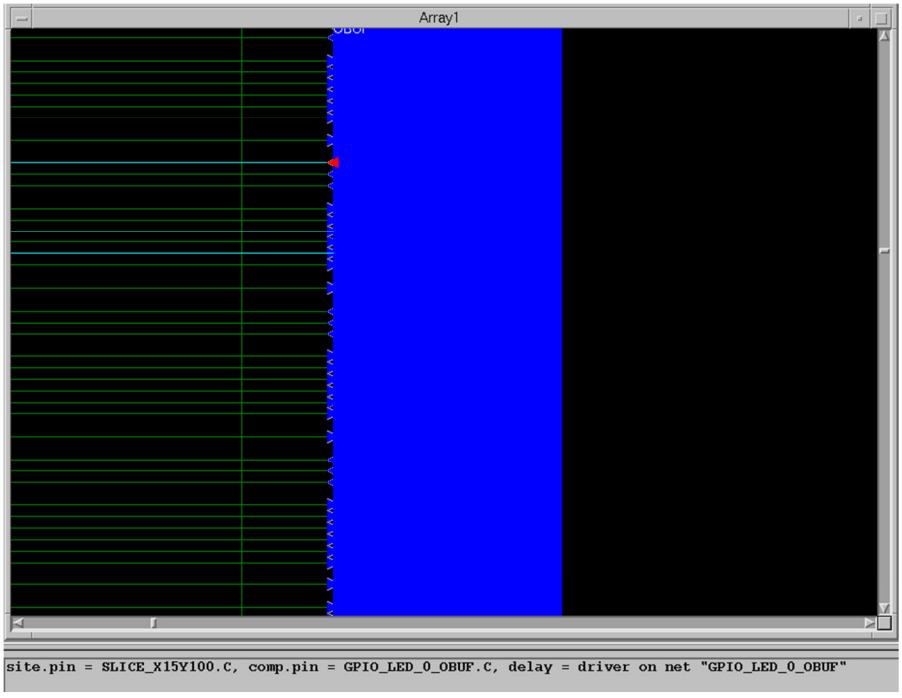
Double-click on the red rectangle (the *SLICEL*) to bring up the following Block Window:



Click the “F=” button to reveal the function of the LUT. You should see that it corresponds to an AND gate. You can deduce from this that this LUT corresponds to the following Verilog code:

```
and (GPIO_LED[0], GPIO_DIP[0], GPIO_DIP[1]);
```

We can verify that this is the case by looking at where the nets connected to the slice are going by zooming in and single clicking on the triangles attached to the slice. The screenshot below shows that the output is going to the net `GPIO_LED_0_0BUF`, which corresponds to `GPIO_LED[0]` from the Verilog code.



You can change the logic function of the LUT through FPGA editor. To do so:

1. Click the following menus: *File->Main Properties...*, set the *Edit Mode* field to *Read Write*, click *Apply* and click *Close*.

2. Go back to the Block Window, and click on the *Begin Editing* (the pencil icon), and "*F=*" again.

3. Now you may change the logic function of the LUT. Change the function of the C6LUT such that it performs an OR rather than an AND. You can do this by changing the * to a + in the function definition for C6LUT.

4. Click *Apply* (button with the down arrow), then save your changes and close the window (second button from the right).

5. Save your design and close FPGA editor.

3.3 Deploying to hardware

We will now deploy your edited design onto the hardware. Run the following command. If you need to, overwrite any conflicting files in order to generate the new `.bit` file. You may need to use `-w` to specify overwriting the file.

```
% bitgen ml505top.ncd ml505top.bit ml505top.pcf
```

This converts your design into a bitstream that can be used to program the FPGA.

The last step is to invoke Impact to program the FPGA. Unfortunately the command line interface to Impact is a little clunky, and since you will ever only use Impact to program the LX110T chip on the XUPv5, you can just use the old makefiles. Simply execute the following:

```
% make impact
```

Once the message `'5': Programmed successfully` is displayed, the FPGA has been configured with your `.bit` file.

3.3.1 Testing

Locate the DIP switches in the lower right corner, and the LEDs in the lower center of the XUPv5, your board should now be showing the results of a logical OR between switch 1 and switch 2 on LED 0. Change the positions of the two switches and note the change in LED 0.

3.4 Extending the Design

In the previous lab we built up a ripple adder in the Adder.v file. For this next part, copy over your Adder.v file into the Adder.v in your /lab1 directory. Also, copy over your FA.v file. We will experiment with the width of the adder to see how it will affect the resource usage for different adder widths.

3.5 The Build System

After you've copied over your Adder, you will want to synthesize the code into a netlist, to see how the amount of resources it takes up. The following introduces the build system that we will use for this lab and for the rest of the course.

The lab distribution contains a `Makefile` that lets you easily build and debug your designs. For those unfamiliar with `make`, you use it by invoking the program make with a target passed in as an argument. For instance, `make synth` will run the tool chain up to synthesis. Several important targets will be described later in the lab.

The `Makefile` has several variables defined at the very top of the file that let you control the behavior of the build system.

```
TOP := FPGA_TOP_ML505
UCF := src/$(TOP).ucf
PART := xc5vlx110t-ff1136-1
SRCS := $(wildcard src/*.v)
```

The most important one for this lab will be the definition of `TOP`. You use it to change the top level module for hardware synthesis. You can change it either by editing the `Makefile` (and remembering to change it back afterwards) or by invoking make like this:

```
% make TOP=<your top module here>
```

The first thing you want to do after writing new Verilog modules (like `Adder.v`) is check that it at least makes basic sense. The `synth` make target is very useful for this purpose. Run it like this:

```
% cd ~/lab1

% make synth
```

3.5.1 Synthesizing your code

The Xilinx tools that this target invokes generate a huge amount of output. Sometimes warnings and errors are obvious, sometimes they are not. The target `make report` launches a program that makes it much easier to view the output of the synthesis tool.

Do this now, and select the `Synthesis Messages` item on the left to see what messages the synthesizer generated. Errors **must** be addressed before continuing. Ask your TA if you are unsure if you should be worried about a warning you see at this stage.

Once you have cleared all errors from the synthesis stage, you know your Verilog is at least syntactically correct. Unfortunately this doesn't mean much because Verilog compilers are often very willing to accept complete nonsense (and give you complete nonsense in return).

Another useful step to take is checking a diagram of the generated circuit. There is another make target for this: `schematic`. This will launch the ISE project navigator. Once inside ISE, go to *File -> Open* and select *ml505top.ngr*. Choose *Start* with a schematic of the top-level block in the dialog that pops up.

This will give you a fairly straight-forward hierarchical block-level view of your design. You will find your circuit by drilling down in the following modules: `ml505top, Adder, FA`. Check to see that your 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.

3.5.2 Verification

Once your circuit looks correct, it is time to verify it using it by pushing the design onto the board. The `ml505top` module has instantiated copies of both the `Adder` you wrote and a `BehavioralAdder`, an alternative version of the adder written in behavioral Verilog. It simply says Out = A+B and lets the tools

interpret this statement. You will learn more about behavioral Verilog in later labs; for now, you just need to know that is a higher level way to describe your circuit.

The top module compares the outputs of these two adders for all the possible inputs, and signals an error if there is a difference. In later labs, you will see other methods of testing your circuits.

To push this design to the board, start by issuing a `make` command in your lab directory. This should build your entire design all the way through to the bitfile. At this point you can issue the `make impact` command to program the board.

Once the board has been programmed, either LED 0 or LED 1 will light up. The test harness uses LED 0 to indicate success and LED 1 indicates failure. If LED 1 lights up, you can check:

a) That you wrote your generate statement correctly
b) By changing the test harness to add the same 2 constant numbers and wire the result to the unused LEDs.

Please ask a TA if you need assistance.

3.6 Circuit Analysis

Now that your circuit is fully functional, we will perform a resource analysis. This will give you a better idea of what hardware the tools inferred from your HDL.

For the Adder circuit, note down the following values for `Width` = 16 and 32. You can change the width in the Verilog file:

```
module Adder #(
 parameter Width = 16 ← Change this number
) …
```

- Number of occupied SLICEs.
- Number of SLICE LUTs
- Maximum Delay Path (look in Post-PAR Static Timing Report and write down approximately the biggest number in the Delay column)

You will be able to find these numbers by running `make report` after mapping your design. Here we actually also place and route the design so that you can visually inspect it in `fpga_editor`:

```
% make TOP=Adder
% make TOP=Adder report
```

Additionally, open up the placed and routed adder using fpga_editor and note down the logic function of any of the LUTs (doesn't matter what width). As a reminder, to open up `fpga_editor`, do the following:

```
% cd ~/lab1/build/Adder
% fpga_editor Adder.ncd Adder.pcf
```

Now do the same thing for the BehavioralAdder. Note the differences in the hardware generated and the resources used by the Adder and the BehavioralAdder.

Now we will purposely make use of some of the dedicated hardware blocks on the FPGA. Recall from the FPGA introduction lecture that the *Virtex5* has *DSP48E* slices that are can be used for efficient implementation of multiply, add, bit-wise logical operations.

However, the tools will not infer *DSP48E* slices for addition by default. You will need to change the synthesis options located in `cfg/xst.batch`. Open this file and change the line:

```
use_dsp48 auto
```

to:

```
use_dsp48 yes
```

You will not need to make any modifications to `BehavioralAdder.v` other than to change the width. Now run `make` again for `width=16` and `32` note down the maximum delay path for each. Note that no LUTs or *SLICELs* will be used since we are using a *DSP48E* block for our addition operation.

4 Checkoff

For checkoff, show a TA the items that you were asked to record in 3.6 Circuit Analysis. Please also answer the following checkoff questions:

1. Was the number of LUTs used for the `Adder` you wrote the same as how many you expected it to use? Why or why not?

2. What hardware did the `BehavioralAdder` (without changing the use_dsp48 flag) map to?

3. Why does the `BehavioralAdder` map to less LUTs than the structural `Adder` you wrote?

4. Why do you think the tools didn't choose to use the DSP48E block to perform addition by default?