Lab 0: Introduction to the FPGA Development Board

University of California, Berkeley

Department of Electrical Engineering and Computer Sciences

EECS150 Components and Design Techniques for Digital Systems

Prof. Ronald Fearing, Austin Buchan, Stephen Twigg

Table of Contents

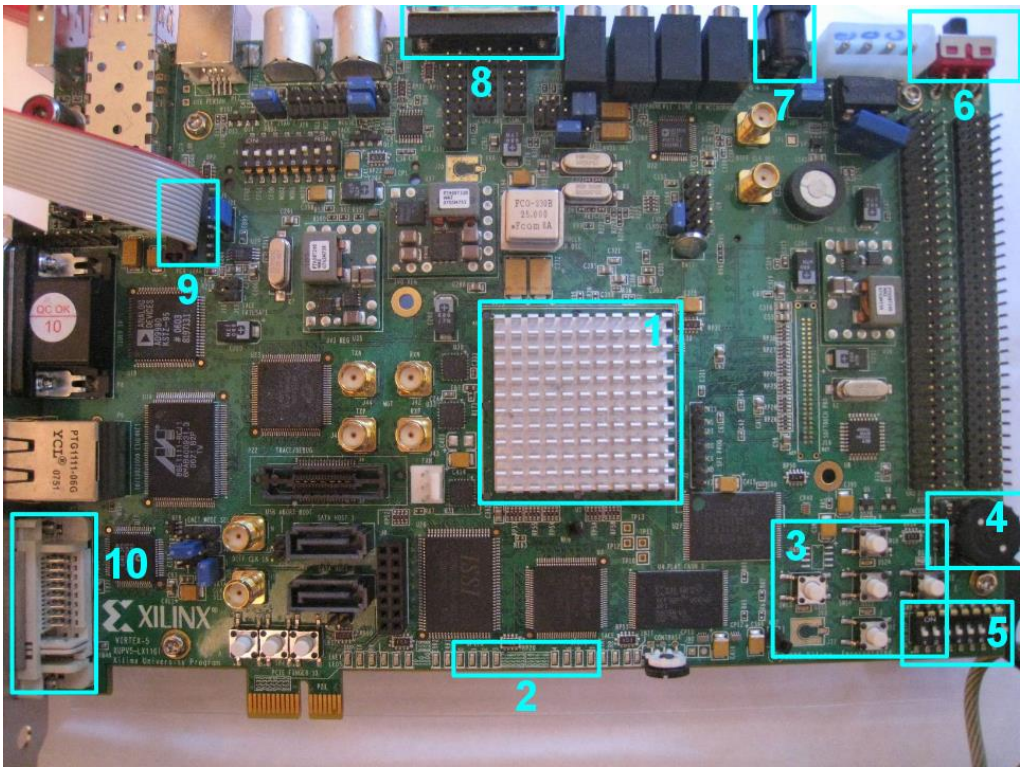
0 Before you begin

This lab is intended to both introduce the CS 150 development platform and take care of administrative tasks. You will need to complete the following steps before you can begin working on the lab. These are considered requirements for completion of the lab. For this lab only, the lab is due at the end of the section.

1. Pick up a cs150-xx class account during your scheduled lab section.
2. Log into one of the CentOS lab machines (`p380-{11..45}.eecs`). On your first time logging in you will need to select a session (GNOME or KDE) or it will fail.
3. Open a terminal and run `passwd` to set your password.
4. Register your account for the course so you appear in `glookup`. In the terminal:
   a. SSH into `cory` (`glookup`/`register` only work on Solaris machines): `ssh cory`
   b. Run `register`, enter your information, then `exit` the ssh connection
5. Make sure you have access to the CS 150 Piazza forum; if not, make sure to sign up.
6. Make sure you have card key access to 125 Cory. If not, notify a TA and we'll get you set up.
7. Read through and understand this document before you start working!

1 Development Platform

CS 150 uses the XUPV5-LX110T development board, located on the lab bench next to your workstation. The following image identifies important parts of the board:



1. Virtex-5 FPGA (covered by heat sink)
2. GPIO LEDs, numbered 0-7
3. NESW & center buttons, each with a corresponding LED
4. Rotary encoder
5. GPIO DIP switches, numbered 1-8 (but 0-7 in code).
6. Board power switch
7. Power connector
8. Serial port
9. JTAG header
10. DVI-I connector

2 Structural Verilog Introduction[1]

Throughout the semester, you will build increasingly complex designs using Verilog, a widely used hardware description language (HDL). For this lab, you will learn structural Verilog, a limited subset of Verilog that allows you to describe circuits in terms of wires, gates and modules.

2.1 Wires

Wires in structural Verilog are analogous to wires in a circuit you build by hand: they are used to transmit values between inputs and outputs.

Wires should be declared before they are used:

```
wire a;
wire b, c;  // declare multiple wires using commas
```

The wires above are scalar (i.e. represent 1 bit). They can also be vectors:

```
wire [7:0]   d;  // 8-bit wire declaration
wire [31:0]  e;  // 32-bit wire declaration
```

Wires can be assigned to other wires, concatenated, and indexed:

```
wire [31:0] f;
assign f = {d, e[23:0]}; // concatenate d with bottom 24
                         // bits of e
```

[1] *Note: This lab assumes you have basic familiarity with logic gates and Boolean algebra. Consult this document if you need to review either topic.*

In the line above, the brackets `[]` are used to index a 24-bit range of e and the braces `{}` concatenate comma-separated wires. Notice convention dictates that the high bits are specified first followed by a colon, then the low bits.

2.2 Gates (Structural Primitives)

In this lab, you may use the following primitives: `and`, `or`, `xor`, `not`, `nand`, `nor`, `xnor`. In general, the syntax is:

```
<operator> (output, input1, input2);  //for two input gate
<operator> (output, input);           //for not gate
```

For example, the following Verilog implements the Boolean equation `F = a + b`:

```
wire a, b, F;
/* … some code that assigns values to a and b */
or (F, a, b);
```

Complex logic functions can be implemented using intermediate wires between these primitive gates. This can quickly become tedious; in later labs we will explore less tedious ways to implement more complex systems.

2.3 Modules

Modules provide a means of abstraction and encapsulation for your design. They consist of a port declaration and Verilog code to implement the desired functionality. For example, consider a module that computes `y = (a + b)(c + d)`:

```
module example_module(a, b, c, d, y);
    // Port and wire declarations:
    input wire a, b, c, d;
    output wire y;
    wire a_or_b, c_or_d;
    // Logic:
    or   (a_or_b, a, b);
    or   (c_or_d, c, d);
    and  (y, a_or_b, c_or_d);
endmodule
```

There are a few things to note from this example:

•. The ports must be declared as input or output wires, but can be thought of as wires within the module.

•. Wires declared within a module (such as `a_or_b`) are limited in scope to that module.

•. Modules should be created in a Verilog file (.v) where the filename matches the module name (so the above example should be located in `example_module.v`).

Then, after creating a module, you can instantiate it in other modules:

```
example_module unique_name(
    .a(a_in),  //connects the wire a to port
    .b(b_in),
    .c(c_in),
    .d(d_in),
    .y(result));
```

In this example we assume that the port inputs `a_in, b_in, c_in, d_in`, and output `result` are valid wires in the module that this instantiation occurs in. We also assume that `unique_name` is globally unique.

The syntax `.<input/output port>(<wire>)` is used to explicitly hook up wires to the correct input/outputs of a module.

You can also write:

```
// correct order ->
example_module unique_name(a, b, c, d, result);
```

The following declaration is also perfectly valid but is not recommended since it is possible to mix up the order of the wires. The first form is also easier to read.

```
// wrong order! ->
example_module unique_name(result, a, b, c, d);
```

3 Lab Procedure

Follow these steps and make sure to save your work for checkoff as you progress. If you run into any problems during the lab (e.g. design doesn't work on the board, compiler errors) please ask a TA to look at your code.

3.1 Acquire the lab files

You can find the lab files on the class website. First create a folder on your top level called `/labs`. Then download the tar file named `lab0.tar.gz` into that directory.

To get the lab files, first `cd` to your `/labs` folder and run:

`wget http://inst.eecs.berkeley.edu/~cs150/current/lab0/lab0.tar.gz`

This should download the distribution onto your machine.

Once you have the tar file, run the following command to unpack the file.

```
tar -zxvf lab0.tar.gz
```

That should unpack the tar file contents into a folder named `/lab0`.

3.2 Existing Functionality

From the `src` directory, open `ml505top.v` in your favorite text editor (emacs, kdedit, vim, etc.). The file `ml505top.v` is the "top-level" module, which essentially means the input and output ports represent physical connections on the board. All the sub modules in the lab will be instantiated as part of this top level module. In other words, the `ml505top.v` module is the highest level in the hierarchy.

The port definition for this module contains an 8-bit input for the GPIO DIP switches (#5 in the diagram) and an 8-bit output to the GPIO LEDs (#2 in the diagram).

The skeleton files provided contain a simple example of an AND gate. This gate takes as inputs GPIO DIP switches 1 and 2, and displays the AND of these on LED 0.

Before you implement the rest of the lab, verify this functionality in hardware:

1. Run '`make`' in the `lab0` directory. This performs several steps to interpret your code and generate a configuration bitstream for the board. You will learn about these steps over the next few labs.

2. Verify that the LED on the Xilinx Platform Cable (picture) is glowing green. If it isn't, verify that the FPGA is powered on and that there a USB cable connected between the Xilinx Platform Cable and the computer.

3. After `make` completes, run '`make impact`' from the same directory. This programs the FPGA with the bitstream file created in step 1.

4. The design should now be on the board. Verify that LED 0 displays the AND of GPIO switches 1 and 2.

3.3 2-1 Mux

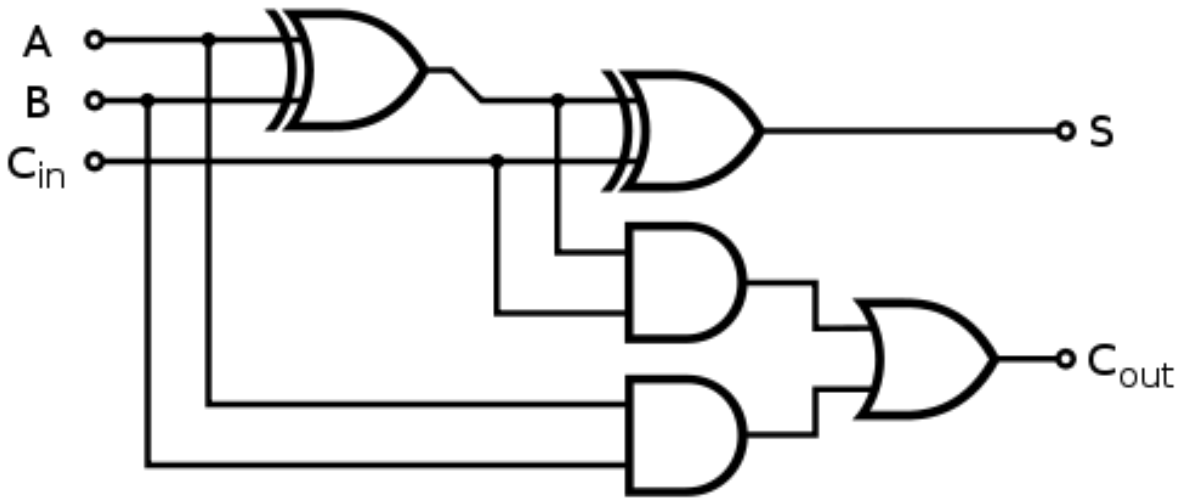
You should also see an instantiation of `Mux2_1` in `ml505top.v`. Implement this module in `Mux2_1.v` (so that it conforms to the functionality in the file's comments) and follow the steps in section 3.2 to test this on hardware.

The two inputs A and B correspond to the GPIO switches 3 and 4 on the board. The select input to the Mux corresponds to GPIO switch 5 and the output appears on LED 1. You can find these port connections in the `ml505top.v` file where the Mux2_1.v module is instantiated.

3.4 Full Adder

Next, examine the port definition of FA.v. Following the instructions in the comments of ml505top.v, write the instantiation of FA in the top file. You may find it helpful to look at the instantiation of Mux2_1 as an example.

Finally, using only structural Verilog, implement a full adder circuit in FA.v. If you are not familiar with the functionality of a full adder, the truth table and one possible gate diagram is shown below:



A	B	Cin	Sum	Cout
0	0	0	0	0

0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Again, you should be able to follow the steps in section 3.2 to test the adder on hardware. Currently the `ml505top.v` file does not have an instantiation for the `FA.v` module so you will have to do it yourself. The instantiation should be very similar to the `Mux2_1.v` module. Make sure to read the comments in the code for port specifications.

3.5 Extending the Design

Now that we've finished a single full adder, we're going to try something a little less boring and create a ripple adder. In the lab directory, you should also find an `Adder.v` module where you are to instantiate the full adders you wrote earlier in the lab. Disregard the `BehavioralAdder.v` file for now; we're only going to deal with structural Verilog for this lab.

We want you to create an 8 bit ripple adder which means that your circuit will contain 8 instantiations of the full adder module you wrote earlier. Instead of manually writing out 8 full adder instantiations, Verilog contains a `generate` structure that makes life easier by allowing you to generate multiple copies of a module.

To use the generate statement we also need a parameter which is defined for you in the `Adder.v` file.

The generate statement and parameters are declared as follows:

```
module foo(
     input [N-1:0] x;
     output [N-1:0] y;
);
     // declaration of the parameter
     parameter N = <default value>;
```

```
    // the parameter can be used to specify wire widths
    wire [N-1:0] z;
    // variable to be used in generate statement
    genvar i;
    // declaration of a generate statement
    generate for( i = 0; i < N; i = i + 1 )
        begin:<unique_name>
        /* Here, module baz has a parameter named width*/
                baz #(.width(N))
                   b(    .a(x[i]),
                         .b(y[i]),
                         .c(z[i]));
        end
    endgenerate
endmodule
```

For `<unique_name>` you can pick any legal label that is not used for a module name or instance. Your job is to use the `generate` statement to instantiate a full adder and complete the `Adder.v` module. If you are confused about the generate construct, ask your TA.

When you complete the `Adder.v` module, run the usual `make clean, make, make impact` sequence. When you finish programming the board, if you wrote your Adder.v file correctly, the GPIO LED 4 should light up. Otherwise GPIO LEDs 5-7 will light up, and you will have to fix your adder.

4 Checkoff

For checkoff, you will show a TA your implementation on hardware as well as your answers to the checkoff questions.

Please have the check off questions open in a text editor or written on paper (they are not collected so there is no need to print if you type).

Checkoff Requirements:

1. Show the adder and mux working on the board.
2. Show the 3 Verilog files you modified.
3. How many logic gates did your adder require?
4. Show your implementation of the Adder.v file and run `make impact` on the board for your TA.