Lab 5: Intro to FPGAs

UC Davis Physics 116B
Rev 2/9/2020

There's a saying when dealing with complex electronic systems: "If you can make the LED blink, you're 90% of the way there.", so in this lab you will make the LEDs blink on the Alchitry-Au prototype board. Doing so involves four distinct steps:

- Writing Verilog code to specify the desired logical behavior.
- Constraining the mapping between the internal logic signals and the pins that connect to the board.
- Compiling your code and generating the configuration (.bit) file.
- Downloading the configuration to Xilinx chip on the Alchitry board (or optionally to the onboard PROM, to be loaded at startup).

Once you figure out how do do this, it will be very straightforward to generalize your knowledge to much more complex applications in the remaining labs of this course.

Pre-lab Questions

In this lab, you will make a 24-bit counter and display the state of the bits on 24 LEDs. If we clock this counter at the full 100 MHz clock rate of the Alchitry-Au onboard clock, at what rate (in Hz) will the LED corresponding to the MSB flash; that is, how long will it take to count through the 24-range and start over?

In the next step, we will slow the counter down so it counts at a rate of 1 Hz. We will do this by implementing a separate "delay" counter that counts every clock cycle, and only update the displayed count when that counter reaches 100 million. In order to count to 100 million, how many bits to we have to allocate to the delay counter?

The Achitry-Au Prototype Board and Interface Boards

In the next few labs, we'll be using the following boards, shown in Figure 1

- Alchitry-Au Xilinx Development Board: This board has a Xinlinx Kintex-7 FPGA, a reset button, 8 display LEDs and the USB-C interface that provides power and connection to the host computer. The other boards are attached to it via the four connectors.
- Alchitry-Io I/O Board: This board provides a general communication interface: 5 buttons, 24 LEDs, 24 DIP switches, and 4 segment displays.

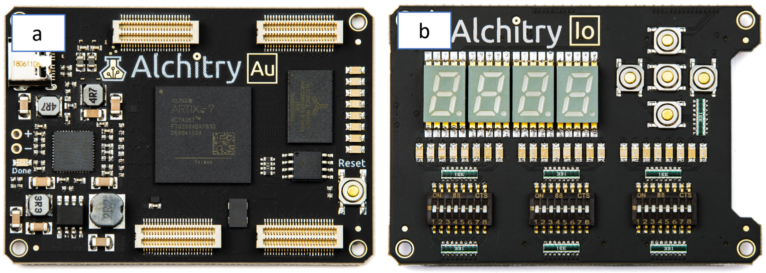


Figure 1: Prototype boards used in labs: (a) Alchitry-Au Xilinx Kintex-7 prototype board and (b) Alchitry-Io display board.

Alchitry Labs IDE

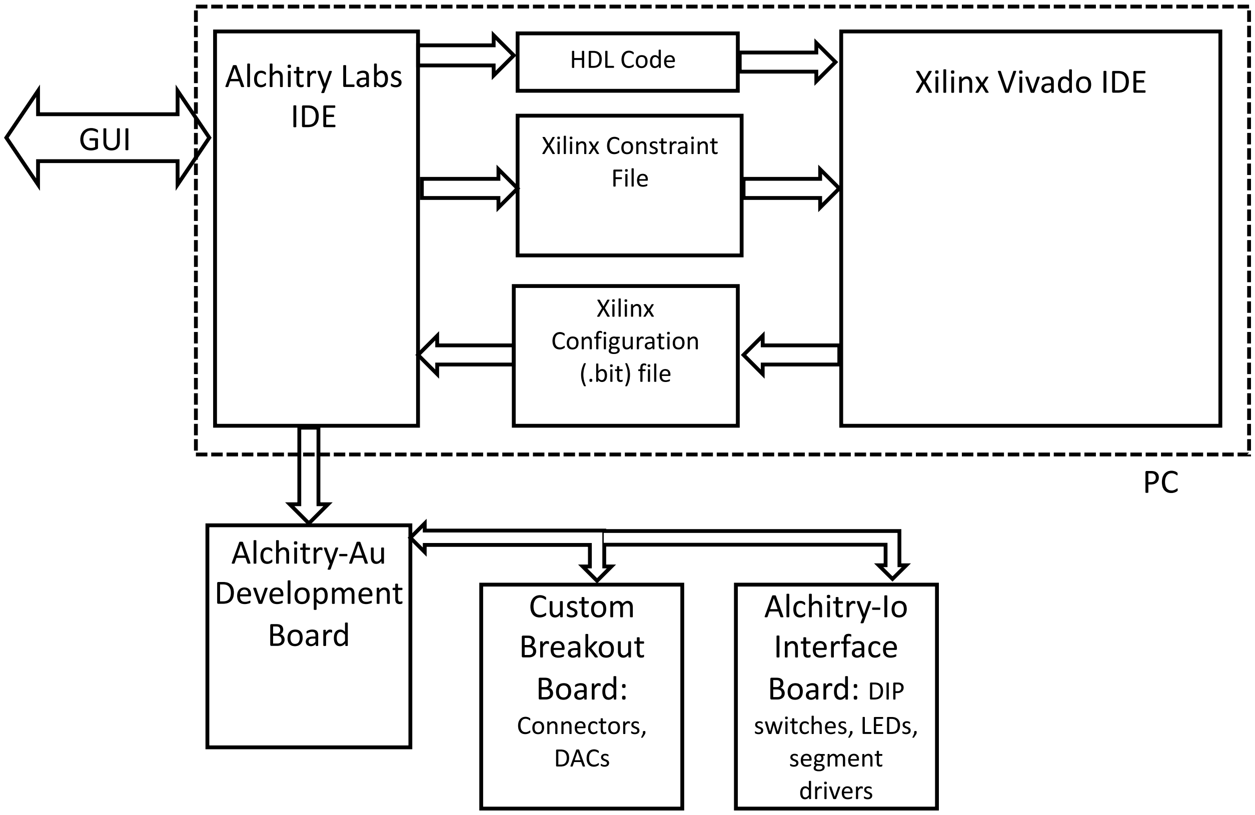

GUI
Alchitry Labs IDE
HDL Code
Xilinx Constraint File
Xilinx Configuration (.bit) file
Xilinx Vivado IDE
PC
Alchitry-Au Development Board
Custom Breakout Board: Connectors, DACs
Alchitry-Io Interface Board: DIP switches, LEDs, segment drivers

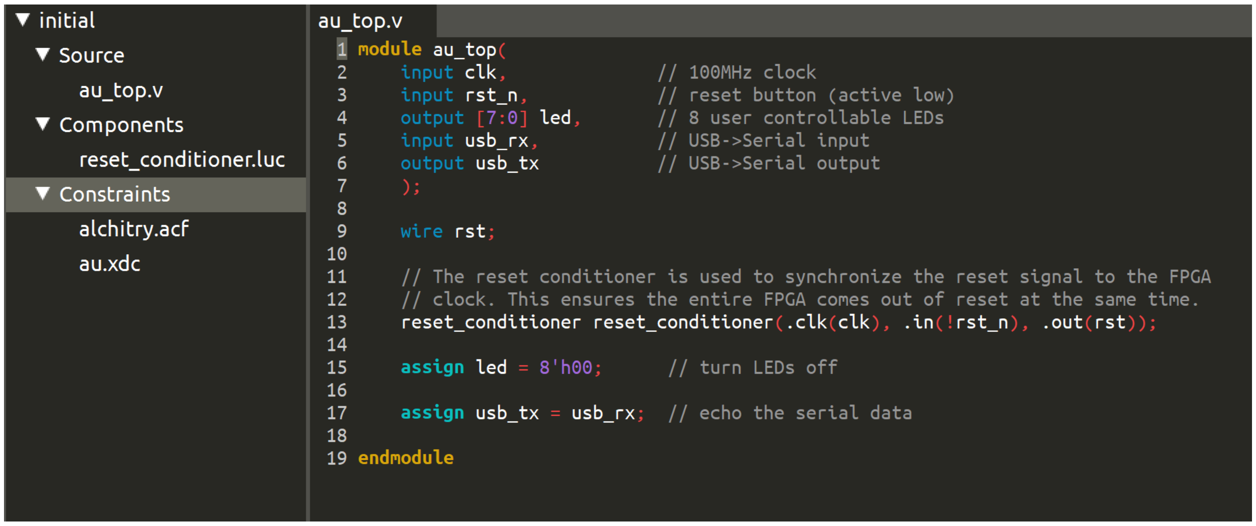
Figure 2: Flow of configuration development. The Alchitry Labs program provides the interface to Vivado, which compiles the configuration scripts to generate the configuration (.bit) file. The Vivado Lab program then loads this configuration file to the board.

The standard integrated development environment (IDE) for Xilinx chips is called “Vivado”. It’s a powerful suite of tools, but very complex. Even simple operations like those we’ll be doing in these labs require multi-step procedures with a significant learning curve. One of the main reasons we chose the Alchitry boards for this lab is that Alchitry provides a very simplified IDE to configure the Xilinx chip that shelters the user from the complexity of the Vivado suite of tools, and should allow us to use our time much more efficiently.

Figure 2 illustrates the operation of the program. The user interacts with the Alchitry Labs program, which allows them to write HDL modules and to specify pin-constraints in own mnemonic format. When the “build” button is pressed, the program translates the pin contraints into the raw pin numbers and format needed by the Vivado program, which then compiles the HDL script in the background and produces the Xilinx configuration (.bit) file. The download button on the Alchitry Labs program will then download the configuration to the Xilinx chip. One can optionally download the configuration to an onboard PROM, so it will be automatically loaded when the Alchitry-Au board powers up.

Lab Activities

Getting Started



```
module au_top(
    input clk,              // 100MHz clock
    input rst_n,            // reset button (active low)
    output [7:0] led,       // 8 user controllable LEDs
    input usb_rx,           // USB->Serial input
    output usb_tx           // USB->Serial output
    );

    wire rst;

    // The reset conditioner is used to synchronize the reset signal to the FPGA
    // clock. This ensures the entire FPGA comes out of reset at the same time.
    reset_conditioner reset_conditioner(.clk(clk), .in(!rst_n), .out(rst));

    assign led = 8'h00;      // turn LEDs off

    assign usb_tx = usb_rx;  // echo the serial data

endmodule
```

Figure 3: Default module for a new Alchitry Verilog project.

You will need the pin definitions for this Lab. All the pin definitions can be found in the file “Alchitry-pins.txt”. If it’s not on the computer desktop, you can download it from the “FPGA Files” subdirectory of the Files area of the Canvas site.

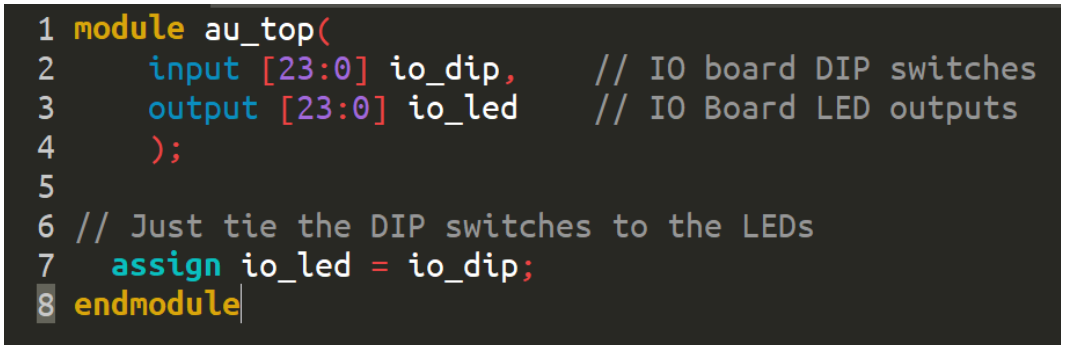
Start the Alchitry Labs program, and create a new project by selecting “Project→New Project...” from the menu at the top. Give the project a unique name, so you can return to it in later labs. Verify that the “Alchitry-Au” is selected as the board, and select “Vivado” for the language. Click “Create”. This will create the default project shown in Figure 3, which we will overwrite for our projects. As a first step, “right-click→remove” the following files, as you won’t need them for our labs:

- reset_conditioner.luc

- alchitry.acf

Do NOT remove or modify the file “au.xdc” as it contains some global directives and parameters needed by all labs that use this board.

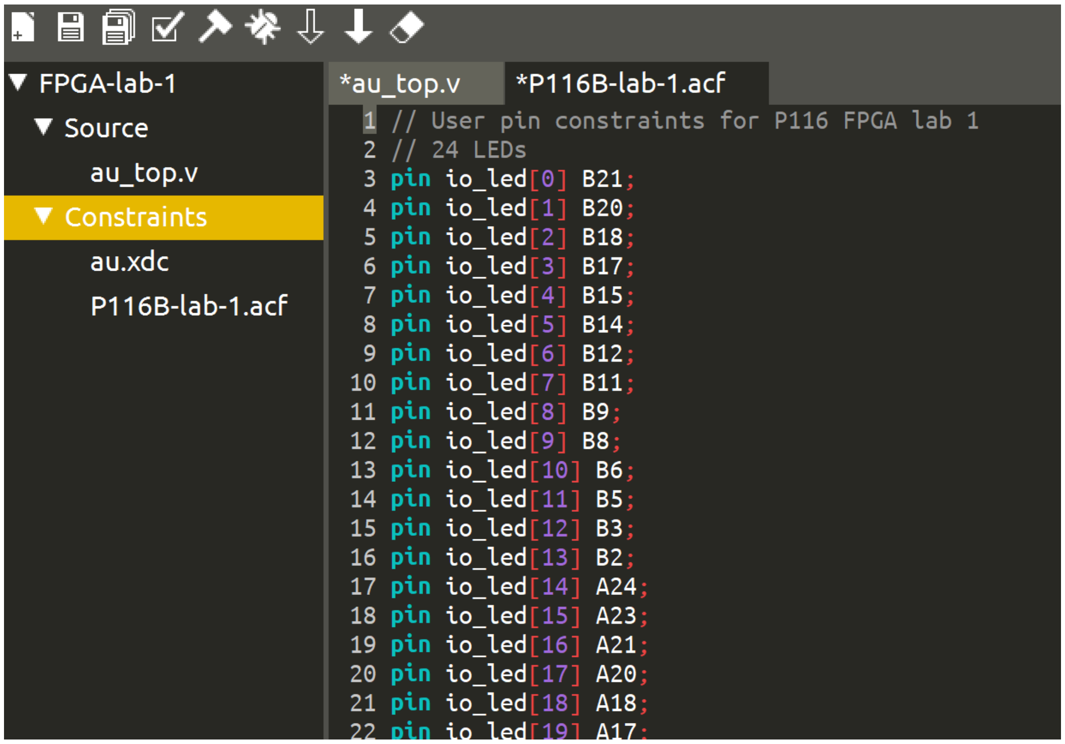
Connect DIP Switches to LEDs



```
module au_top(
    input [23:0] io_dip,     // IO board DIP switches
    output [23:0] io_led     // IO Board LED outputs
    );

// Just tie the DIP switches to the LEDs
  assign io_led = io_dip;
endmodule
```

Figure 4: Simple module to connect the DIP switches to the LEDs.



```
// User pin constraints for P116 FPGA lab 1
// 24 LEDs
pin io_led[0] B21;
pin io_led[1] B20;
pin io_led[2] B18;
pin io_led[3] B17;
pin io_led[4] B15;
pin io_led[5] B14;
pin io_led[6] B12;
pin io_led[7] B11;
pin io_led[8] B9;
pin io_led[9] B8;
pin io_led[10] B6;
pin io_led[11] B5;
pin io_led[12] B3;
pin io_led[13] B2;
pin io_led[14] A24;
pin io_led[15] A23;
pin io_led[16] A21;
pin io_led[17] A20;
pin io_led[18] A18;
pin io_led[19] A17;
```

Figure 5: Top of constraints file for this project.

As a first project, you will simply connect the 24 DIP switches to the 24 LEDs on the Alchitry-Io board. To do this, modify the au_top.v module to match Figure 4. You will also need to tell the program which pins these are c connect to. To do this, create a new constraint file by mousing over the “Constraints” header to the left, and doing “Right-click→New constraint...” In the dialog box, give the file a name (do NOT use “au” or “io”) and select the “User constraints” radio button. This will create a new, empty file with a .acf suffix. Copy and paste the definitions for io_dip[23:0] and io_led[23:0] into this file from the “Alchitry-pins.txt” file you downloaded earlier. Note that it will create and error if you either use a pin you don’t define, or define a pin you don’t use, so only copy the definitions you need for each step!

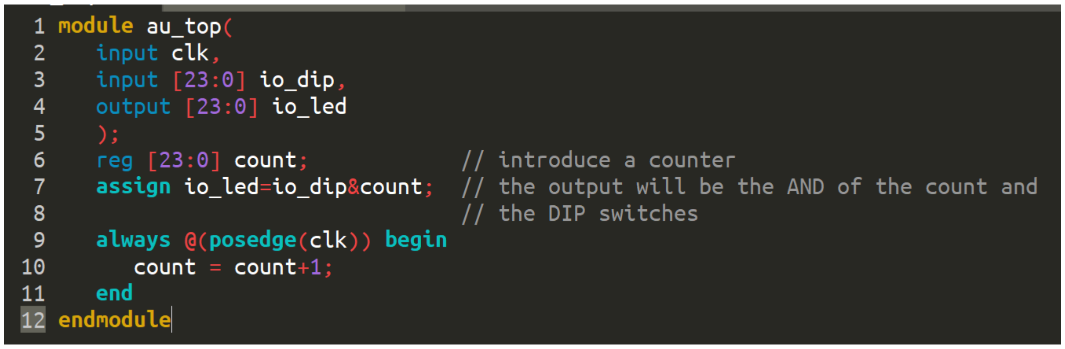
One issue with going from the full Vivado suite to the Alchitry Labs program is that the log and error files end up being a little hard to fine. Luckily, Alchitry has pretty sophisticated syntax checking, so look at both the .v file and the .acf file to *make sure none of the line numbers are highlighted in red*. Once you pass the syntax check, remaining errors are almost always a problem with the pin definitions, so take time to double-check those as well.

Once you’ve completed the project, compile it and generate the configuration file by clicking the little hammer icon at the top. This will launch Vivado in the background, initiates the multi-step process to produce the configuration file, and generates a lot of output at the bottom that scrolls by too fast to read.

If you were successful, it will display “Finished building project.” in green at the bottom. Otherwise, it will display an error in red and you’ll need to figure out what the problem was. Some troubleshooting tips can be found in the appendix.

Finally, click on the hollow arrow to download the configuration to the Alchitry board. Verify that it behaves as expected.

Fast Counter



```
module au_top(
   input clk,
   input [23:0] io_dip,
   output [23:0] io_led
   );
   reg [23:0] count;            // introduce a counter
   assign io_led=io_dip&count;  // the output will be the AND of the count and
                                // the DIP switches
   always @(posedge(clk)) begin
      count = count+1;
   end
endmodule
```

Figure 6: Top of constraints file for this project.

In this section, you will make a 24-bit counter that will count at the full 100 MHz rate of the onboard clock. To do this, modify the code as shown in Figure **??** and add the definition of the clock pin to the constraint file.

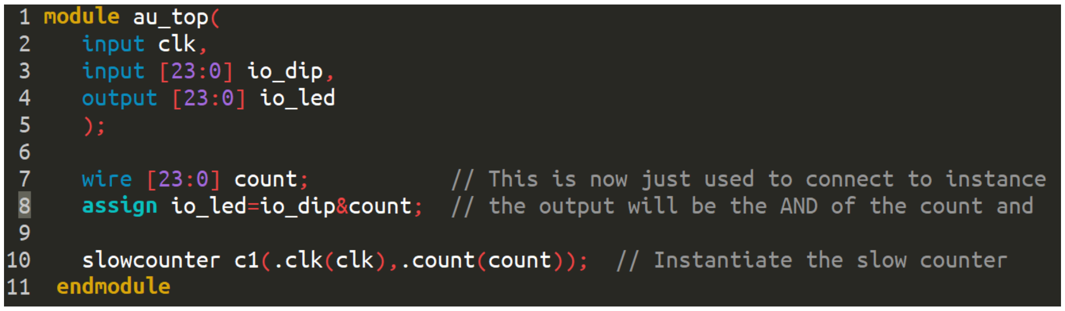
Compile and download this configuration. Does it behave as expected? Is the rate consistent with what you calculated in the pre-lab questions? What is the role of the DIP-switches in this case?

Slow Counter

For this activity, you're on your own. You will slow down the counter so it counts at 1Hz. To do this, define and extra counter in the module, called "delay". Inside the "always" block, increment this count every clock cycle. When it reaches 100 million, increment "count" by 1 and reset "delay" to 0.

Compile and download this to verify that it works as expected.

Instantiation



```
module au_top(
   input clk,
   input [23:0] io_dip,
   output [23:0] io_led
   );

   wire [23:0] count;            // This is now just used to connect to instance
   assign io_led=io_dip&count;  // the output will be the AND of the count and

   slowcounter c1(.clk(clk),.count(count));  // Instantiate the slow counter
endmodule
```

Figure 7: Instantiating the counter.

For this part, you will move the counting logic into a separate module, and then "instantiate" that module in the top level design.

From the Source header at left, do "Right-click→New source...". Give it a name "slowcounter" and select the "Verilog source" radio button. Give this an input called "clk" and a 24-bit output *register* called "count". Cut the counting logic out of au_top.v and move it into slowcounter.v. Finally, instantiate slowcounter in the top module, as shown in Figure 7. Why did I have to define "count" as a wire instead of a register here?

Compile and download this to verify that it works as expected. Once you've got it working, click the solid arrow to download the configuration to the onboard PROM. Unplug the Alchitry board and plug it back in to verify that this is now the default configuration at startup.

Lab Writeup

For your lab report, include answers to all questions, and a screen shot of the Verilog script for your final versions of au_top.v and slowcounter.v.

Be sure to save your files for use in later labs!

Appendix: Troubleshooting

One thing we give up using Alchitry Labs instead of Vivado is that Vivado produces very detailed error logs that highlight problems in yellow or red. Alchitry Labs dumps output to the console window, and it's easy to miss the details of errors if the program fails to compile.

It has a pretty good syntax checker that will find mistakes in basic syntax, as well as missing or extra pin definitions. When you think you're ready, click the little "check" button to verify that your program passes this basic test.

On the other hand, there are a lot of things that the syntax checker can miss. One trick is to select, copy, and paste the text from the console window into a text editor, so you can search for "ERROR" and "WARNING".

Some of the most common errors that get past the syntax-checker are:

- Incorrectly spelled variable names. The syntax-checker will not check this, but you wil get an "undeclared" error when you compile.
- Putting vector dimensions after the name instead of before it; eg. writing `reg A[7:0];` instead of `reg [7:0] A;`. This is a very common mistake, particularly for people used to programming in C, and it generates a very confusing error message.
- Using nets (wire, etc) and variables (reg, integer, etc) in the wrong places. Remember that nets can't be used on the left hand side of equations in concurrent statements and nets can't be used on the left hand side of equations in procedural statements (inside always blocs).
- Wrong pin assignment in the constraint file. This will either generate a "null" assignment error or a conflict error. This shouldn't happen if you always copy the pin definitions from the file that has been provided.