FPGA Lab

INDAT Summer School

E. Prebys and R. Hensley, UC Davis
Rev 12/25/2025 (v4)

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 Basys 3 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 Basys3 board.

Once you figure out how do do this, it will be very straightforward to generalize your knowledge to much more complex applications.

Introduction

Configurable logic has become an integral part of all modern state of the art data acquisition and control systems. Field programmable gate arrays (FPGAs) not only provide extremely high logic density, but also the ability to reconfigure systems after they have been built, something which is not possible with either discrete logic or ASICs. There are several makers of FPGAs, with the two largest being Altera and Xilinx. Both have similar speed and capabilities, and which one is used depends as much on the familiarity of the designer with a particular brand as the specific needs of the application. By tradition, RF engineers tend to use Altera and data acquisition designers tend to use Xilinx.

We will be using Xilinx for this lab, specifically an FPGA from the Artix-7 family, a fairly recent generation of Xilinx chips. There are several development boards that have been produced for Xilinx chips, and for many simple applications development boards may be sufficient for smaller tasks, without the need for any custom hardware design.

For our lab, we'll be using the Digilent Basys3 development board, which is shown in Figure 1. In addition to digital inputs and outputs, the Artix-7 family has a built-in ADC, which we will access. We will also use the switches, LEDs, and 4-digit 7-segment display.

There are many ways to configure FPGAs:

- **Schematic capture:** submodules or custom library components are connected together graphically.

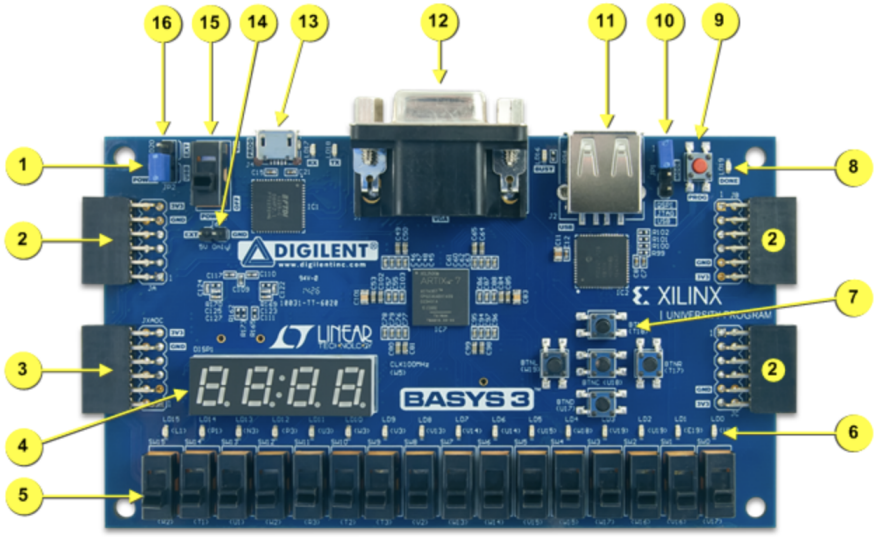


Figure 1. Basys3 board features

Callout	Component Description	Callout	Component Description
1	Power good LED	9	FPGA configuration reset button
2	Pmod connector(s)	10	Programming mode jumper
3	Analog signal Pmod connector (XADC)	11	USB host connector
4	Four digit 7-segment display	12	VGA connector
5	Slide switches (16)	13	Shared UART/ JTAG USB port
6	LEDs (16)	14	External power connector
7	Pushbuttons (5)	15	Power Switch
8	FPGA programming done LED	16	Power Select Jumper

Figure 1: The features of the Basys3 development board, which uses an Artix-7 Xilinx chip.

- **Hardware Description Languages (HDLs):** Behavior is specified using scripted code. This looks like computer code, but there are important differences. The most common are VHDL and Verilog, but Lucid is another option which is gaining popularity.
- **Parametric Design:** This method of design uses parametric "wizards" to design components for particular applications. In the Xilinx design suite, this is done through the "IP Library".
- **High Level Synthesis:** This is a growing approach in which complex behavior is specified using high level languages like C. This is particularly powerful in machine learning applications.

All modules are compatible at the "pin level" (inputs, outputs, and bidirectional signals), so multiple methods can be employed for a single design.

For this lab, we will use Verilog, which is and HDL with a syntax similar to C. To access the onboard ADC, we will use a component that was designed parametrically, using the IP Library.

Activities

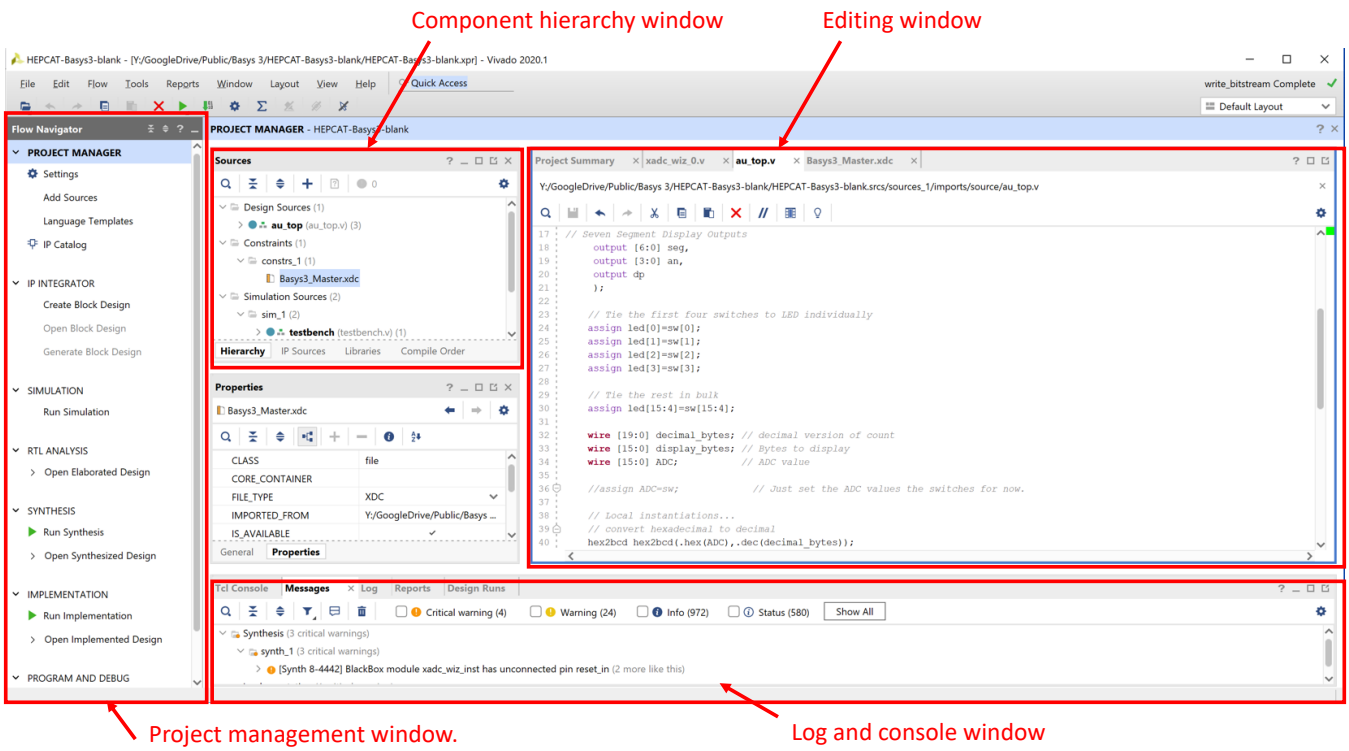


Figure 2: The Vivado IDE interface.

Go to the directory `https://tinyurl.com/INDAT-FPGAlab`, download the file `INDAT-FPGA-lab.zip`, and unpack it. The is a simple example project, which will serve as the basis for the other activities in this lab.

Template Program

Start the Vivado program, select "Open Project" and upon the file `INDAT-FPGA-lab.xpr` in the top directory you just unpacked. You should see the IDE environment shown in Figure 2.

These inputs and outputs must be associated with particular pins on the Xilinx chip. This is done in the `Basys3_Master.xdc` file under the "constraints" tree. Examine this file. The format is a little complicated, but two lines are required for each input or output, which associate the name used in the module interface with a grid array location on the chip itself.

If you define an input or output in your file that matches something inside the constraint file, it will automatically be linked to that element on the physical FPGA board. We'll return to this constraints file later.

The source tree is in the top center, as shown in Figure 3. The source file is `indat_lab.v`. This file has all the inputs and outputs we will use for the rest of the lab, although many are not used yet:

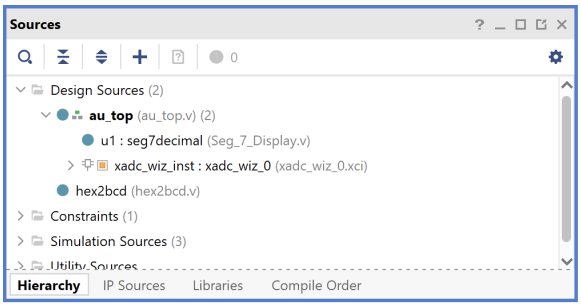


Figure 3: Source and resource window.

```
module indat_lab(
    // Part 1 Variables: connecting switches to LEDs
    input clk,              // 100MHz clock
    input [15:0] sw,        // 16 switches
    output [15:0] led,      // 16 LEDs

    // Part 2 Variables: the seven segment display
    output [6:0] seg,
    output [3:0] an,
    output dp,

    // Part 3 Variables: XADC and board header pins
    input [7:0]  JXADC,     // Analog connections
    input [7:0]  JA,        // JA header
    output [7:0] JB         // JB header);
```

The mapping of these inputs to the connectors are shown in Figure 4.

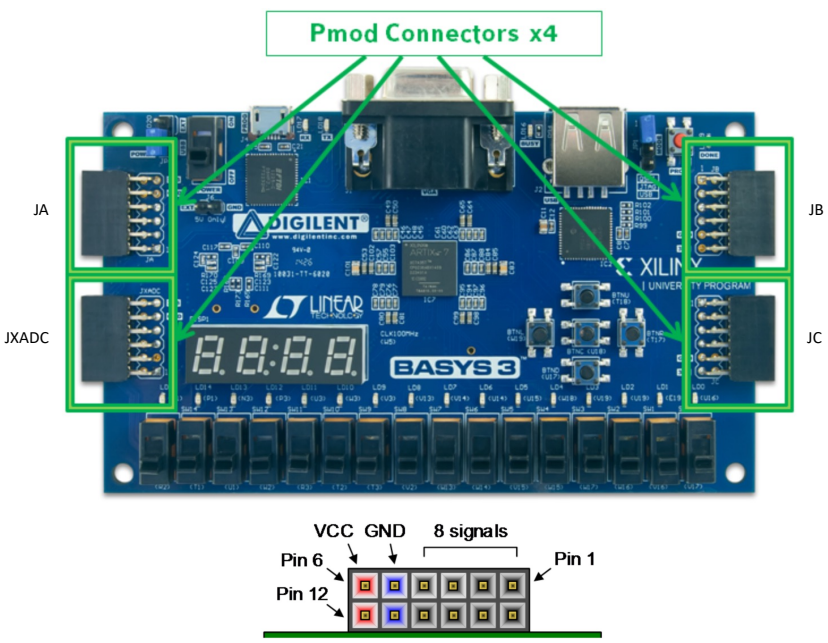
Part 1: Connecting the switches to the LEDs

The first step is to tie all the switches to the LEDs, so if you flip a switch, the LED above it will turn on. Connecting an input to an output or otherwise any two elements in your module with a simple wire can be done with an "assign" statement. Connecting the first switch to the first LED can be done like this:

```
    assign led[0] = sw[0];
```

The next three switches are `sw[1]`, `sw[2]`, and `sw[3]`. Additionally, you could also specify a range of switches like `sw[15:4]` or LEDs like `led[15:4]`. You must write the higher bit first, so [3:0] would mean "from 0 to 3" including 3, so 0, 1, 2, 3.

Once you've written code to connect individually the first four switches to the first four LEDs, connect in bulk (using only one line of code) all of the remaining 12 switches to the remaining 12 LEDs. In the end, all sixteen switches should each be tied to one LED.



Digital Mapping (Jx=JA,JB,JC)

Jx Pin	Input/ Output
1	Jx[0]
2	Jx[1]
3	Jx[2]
4	Jx[3]
7	Jx[4]
8	Jx[5]
9	Jx[6]
10	Jx[7]

Analog Mapping

JXADC Pin	Input	Analog Input
1	JXADC[0]	vauxp6
2	JXADC[1]	vauxp14
3	JXADC[2]	vauxp7
4	JXADC[3]	vauxp15
7	JXADC[4]	vauxn6
8	JXADC[5]	vauxn14
9	JXADC[6]	vauxn7
10	JXADC[7]	vauxn15

Figure 4: The mapping of the PMOD connectors to the inputs as defined in the constraint file.

Putting code on the FPGA

There are four separate steps to convert this code into a programmable (.bit) file, as indicated in the processing flow panel at the left:

- **Synthesis:** The code is converted into primitives, associated with the actual elements within the chip.
- **Implementation:** The design is laid out for this specific chip, including connections to the I/O pins.
- **Generate Bitstream:** This design is rendered into a configuration file that can be downloaded onto the chip.
- **Hardware Manager** Once the bitstream (.bit file) is created, the hardware manager is used to put the code onto the FPGA.

In a real design, in which density or performance is an issue, the designer may intervene and optimize the design at each step; however, for this lab we will always simply run them sequentially, making the division rather cumbersome. Start by clicking "Run Synthesis", and when each step ends, you will be given the option of starting the next step. Note that the total process takes several minutes.

After the generation of the configuration file, you'll be given the opportunity to open the Hardware Manager. Make sure that the board is connect to the computer, open the Hardware Manager, and click "Open Target→Auto Connect". If successful, this should result in the tree shown in Figure 5.

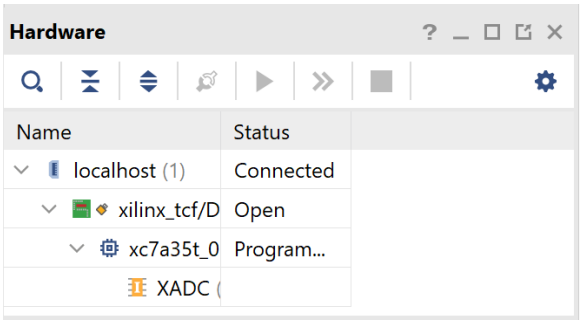


Figure 5: The hardware manager, when the board is correctly connected.

Right-click on the chip name "xc7a35t_0", select "Program device.." and accept the defaults. This will download the bit file and configure the chip.

Test this out by flipping some of the switches and making sure the LEDs turn on.

Part 2: Showing values on the seven-segment display

You will need to exit the Hardware Manager ("X" at the upper right of that window) to get back to the editing pane.

Next we want to display a value on our seven segment display. The old lab had an ADC used here, so if an ADC or voltage value is mentioned later in the lab, assume I meant this counter.

We'll be implementing a counter that counts up from zero and fills a 16-bit vector (check the Verilog help document for some examples of this).

This value is sent to the display by instantiating the `seg7decimal` module. At this point, try making a diagram on a piece of paper showing where data is flowing into which modules, and which modules are used where. Make sure to include: count, display_bytes, seg7decimal

Go through the whole process again of synthesis, implementation, generation of the bitstream, and then the hardware manager to put the bitstream on the FPGA.

Part 3: Modifying the code

Adding new functions to the switches and LEDs

First, change the first four LEDs so they are defined as follows:

- Make LED0 the logical AND ("`&&`") of the first four switches (`assign led[0] = ...`).
- Make LED1 the logical OR ("`||`") of the first four switches (`assign led[1] = ...`).
- Make LED2 come on if the count value is below half of the max value (hex "8000", which you can write as "`16'h 8000`", which means a 16-bit hexadecimal number with value "8000", half the maximum value of "FFFF")

- Make LED3 come on if count is *greater* than half the range (inverse of LED2: `~led[2]`).

Then, as a new, fifth instruction to the code,

- Tie the first output of the JB connector (JB[0]) to the state of LED3. This will effectively assert it if the signal is more than half the range.

Part 4: Convert hexadecimal display to decimal display

Next, we will convert the display from hexadecimal to decimal by converting it to "binary coded decimal" (BCD), in which four bits are used for each *decimal* digit 0-9 (so four digits would require a total of 16 bits)[1].

To convert the display to decimal, we need to add something in between the "count" register and the "display_bytes", which right now are connected directly to each other. We can instantiate the `hex2bcd` module, included with the template but not yet used and pass count into it.

Follow the instantiation example of `seg7decimal` but use a different instantiation name (eg "u3").

Repeat the configuration steps from the last section, and verify that that the digital display is now in decimal and that everything behaves as expected. What do you expect the maximum value of the decimal display for our setup to be?

Part 5: Implementing a button to reset the counter

Since we're implementing a counter, it's a good idea to be able to reset it. We'll do this using one of the push buttons. To do this, go into the `Basys_Master.xdc` file and uncomment the two lines defining "btnC" (central push button) and add a `btnC` input to the module definition. Then add some logic to your always block to reset the counter when `btnC == 1`.

Synthesize and download your configuration, and verify that it works as expected. Increase the pulse frequency and verify that the pulse count rate increases. Try lowering the amplitude until it stops counting.

Part 7: Final Project

If you have time, try to select one of the following challenge problems (or multiple if you are enjoying this lab), and implement them. Alternatively, if you have another idea you want to try, now is the time to try it out. I'll add your idea into labs for future years.

[1] This is how older computers did math and it's how calculators still work. In general it takes more BCD bytes to display a number than hexadecimal bytes, but since the maximum for 12 bits is 4095, it will still fit in a four digit display.

- Try making the decimal display increment or decrement with button pushes, including an ability to reset the counter with a different button push.
- Try showing the binary value of the count value as binary on the LEDs.
- Try using the buttons to move a light on the LED array left or right. Maybe the center button can spawn a new permanent light on whatever spot you're at, or something like that.
- Try making some kind of game with the LEDs, switches, and buttons. It's up to you to decide what kind of game to make, but maybe I can give you ideas. You could defeat enemies on the LEDs, or play a math quiz game with the display and the switches.
- Try making a binary adder using switches and LEDs. Add the binary values from the first five switches, plus next five switches, and display those on the last six LEDs. Or, you could display the output on the display and use eight switches for each number.
- Try making your own project! Whatever you decide could be fun and interesting to try out. If you're not sure if it would be feasible, let me know and I'll tell you if there's a simple way it could be done.

Looking Under the Hood

To get some idea of the potential of modern FPGAs, we'll look at some details of our implementation and see how little of the chips resources we used.

In your final design, expand the "Open Implemented Design" tree at the right. First, click on "Open Implemented Design" at the top. This will show you the physical layout of the chip, indicating usage. As you can see, we used *very* little of it.

Now, click on "Report Utilization". Examine both the "Hierarchy" and the "Summary" pages in the log window. Because we've connected to so many things, we used a reasonable fraction of the I/O ports, but only around 1% of everything else. What part of our configuration used the most resources? Is this surprising?

To put this in perspective, we used the Artix XC7A35T chip, which is actually at the lower end of the Artix-7 family in terms of resources. The largest chip in that family, the Artix XC7A200T, has roughly *seven times* the logic resources, and the largest Xilinx chips currently available have roughly an order of magnitude more than that!

Finally, click on the schematic to see how the design is laid out in FPGA primitive components. You'll need to zoom in to see any detail. Click on the instantiated components to see how each of them is laid out. In particular, look at the details of the hex to BCD conversion.