FPGA Lab

NAT/HEPCAT Summer School
E. Prebys and R. Hensley, UC Davis
Rev 8/22/2024

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.

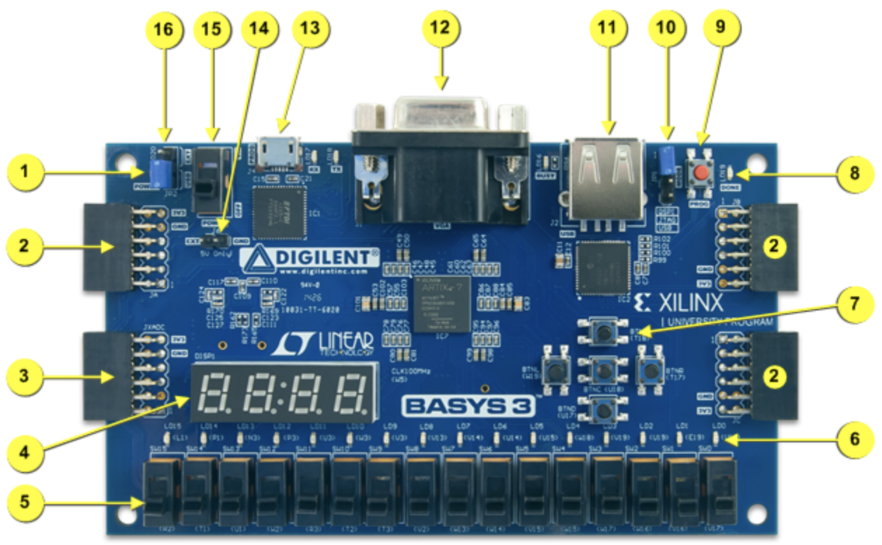
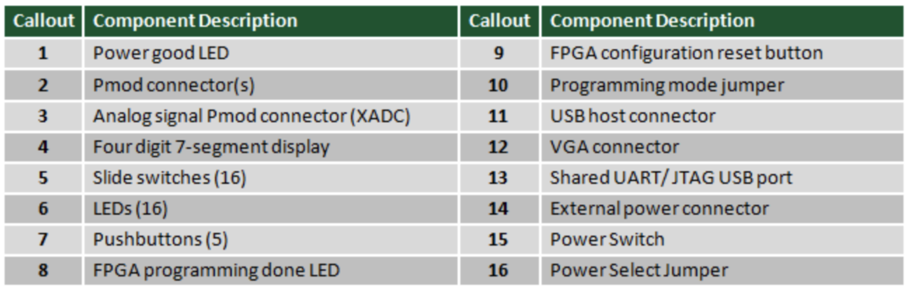


Figure 1. Basys3 board features



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

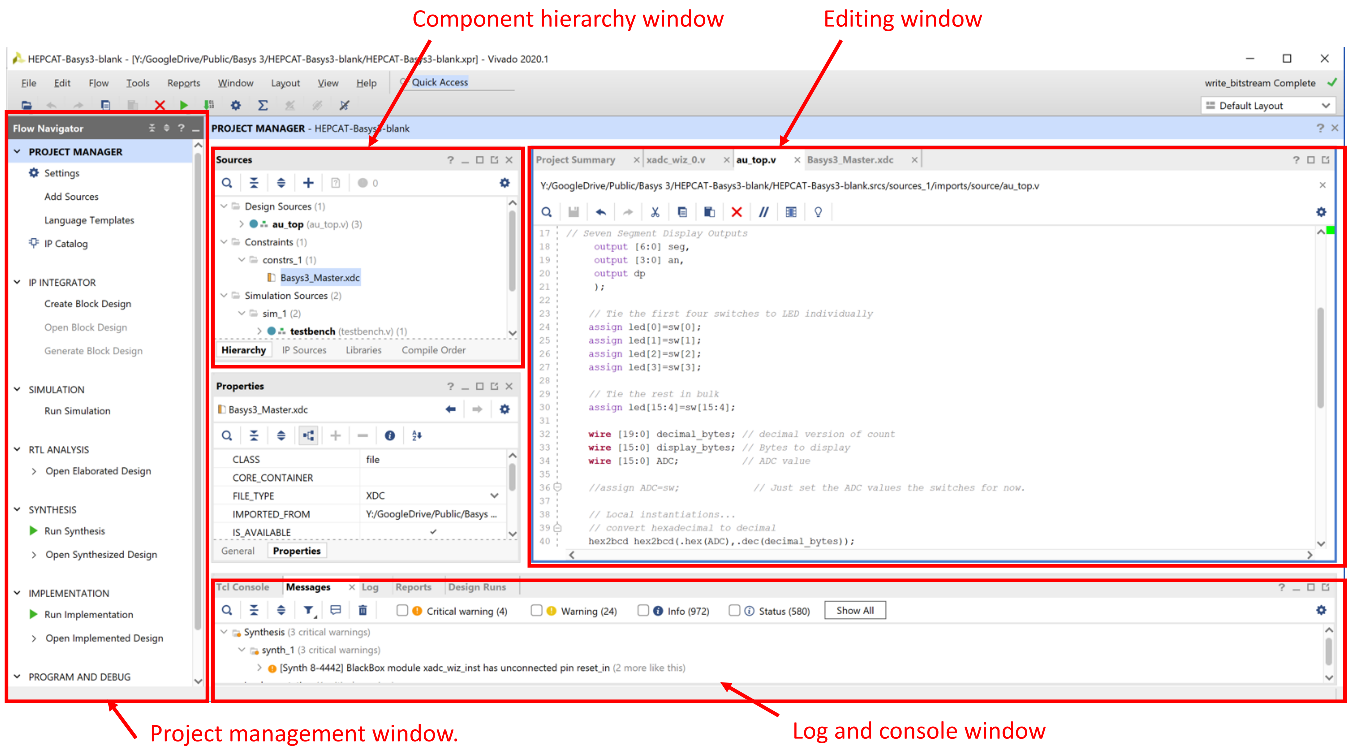
Figure 1: The features of the Basys3 development board, which uses an Artix-7 Xilinx chip.

- **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



```
Design Sources (1)
  au_top (au_top.v) (3)
Constraints (1)
  constrs_1 (1)
    Basys3_Master.xdc
Simulation Sources (2)
  sim_1 (2)
    testbench (testbench.v) (1)
```

CLASS	file
CORE_CONTAINER	
FILE_TYPE	XDC
IMPORTED_FROM	Y:/GoogleDrive/Public/Basys ...
IS_AVAILABLE	✓

Y:/GoogleDrive/Public/Basys 3/HEPCAT-Basys3-blank/HEPCAT-Basys3-blank.srcs/sources_1/imports/source/au_top.v

```
// Seven Segment Display Outputs
     output [6:0] seg,
     output [3:0] an,
     output dp
     );

    // Tie the first four switches to LED individually
    assign led[0]=sw[0];
    assign led[1]=sw[1];
    assign led[2]=sw[2];
    assign led[3]=sw[3];

    // Tie the rest in bulk
    assign led[15:4]=sw[15:4];

    wire [19:0] decimal_bytes; // decimal version of count
    wire [15:0] display_bytes; // Bytes to display
    wire [15:0] ADC;           // ADC value

    //assign ADC=sw;             // Just set the ADC values the switches for now.

    // Local instantiations...
    // convert hexadecimal to decimal
    hex2bcd hex2bcd(.hex(ADC),.dec(decimal_bytes));
```

```
Critical warning (4)   Warning (24)   Info (972)   Status (580)
Synthesis (3 critical warnings)
  synth_1 (3 critical warnings)
    [Synth 8-4442] BlackBox module xadc_wiz_inst has unconnected pin reset_in (2 more like this)
```

Figure 2: The Vivado IDE interface.

Go to the directory `tinyurl.com/HEPCAT-FPGAlab/`, download the file `HEPCAT-Basys3-template.xpr.zip`, and unpack it. The is a simple example project, which will serve as the basis for the other activities in this lab.

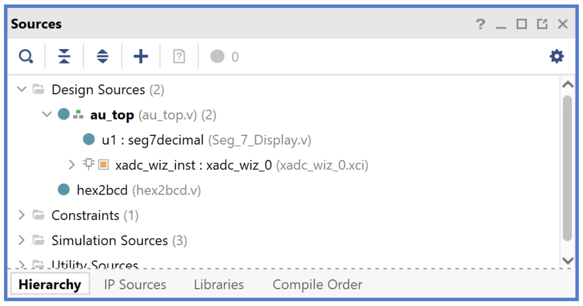
Template Program

Start the Vivado program, select "Open Project" and upon the file `HEPCAT-Basys3-template.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. We'll return to this file later.

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

```
module au_top(
    input clk,              // 100MHz clock
    input [15:0] sw,        //  switches
```



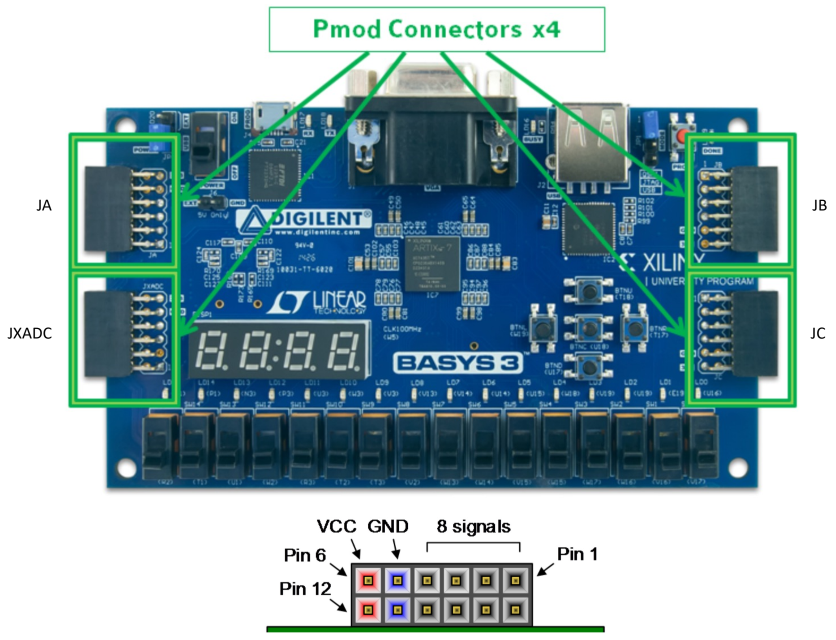
Sources

Design Sources (2)
- au_top (au_top.v) (2)
  - u1 : seg7decimal (Seg_7_Display.v)
  - xadc_wiz_inst : xadc_wiz_0 (xadc_wiz_0.xci)
- hex2bcd (hex2bcd.v)

Constraints (1)
Simulation Sources (3)
Utility Sources

Hierarchy | IP Sources | Libraries | Compile Order

Figure 3: Source and resource window.



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.

```
    input  [7:0]   JXADC,     // Analog connections
    input  [7:0]   JA,        // JA header
    output [7:0]  JB,         // JB header
    output [15:0] led,        // LEDs
// Seven Segment Display Outputs
     output [6:0] seg,
     output [3:0] an,
     output dp
     );
```

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

The simple code does the following:

- Ties the 16 switches to the 16 LEDS above them.

- Displays the 16 bit ADC conversion of the first analog input (difference between JXADC pin 1 and 7) on the 4 digit display in hexadecimal.

The former task is accomplished by the following simple code:

```
// Tie the first four switches to LED individually
// Replace this with combinatorial logic, if desired.
assign led[0]=sw[0];
assign led[1]=sw[1];
assign led[2]=sw[2];
assign led[3]=sw[3];
// Tie the rest in bulk
assign led[15:4]=sw[15:4];
```

The interface to the onboard ADC is a bit complicated and beyond the scope of this lab. It is handled by the "IP Wizard" and some code at the bottom of this module. As a result, the first analog input is continuously digitized and loaded into the ADC register. The onboard ADC has a unipolar range from 0 to 1 V and returns a 12-bit value. The output is therefore

$$\mathrm{ADCVAL} = 4096 * (\mathrm{pin1} - \mathrm{pin7})$$

where these are the voltages on the pins of the JXADC connector. This value, however, is left-shifted by 4 bits to fill a 16 bit word, which is what the XADC wizard returns as XADC_data in the example code.

This value is tied to the display by instantiating the seg7decimal module, as shown below.

```
// Tie the display bytes to the ADC output
 wire [15:0] display_bytes; // Bytes to display
 assign display_bytes=ADC;
// This instance converts a 16 bit number to a 4 digit hex display.
 seg7decimal u1 (.x(display_bytes),.clk(clk),.a_to_g(seg),.an(an),.dp(dp));
```

In order to generate a voltage at the input in the 0-1 V range, use the breadboard to connect a variable resistor to the JXADC connector as shown in Figure 5.

There are four separate steps to convert this code into a configuration (.bit) file, as indicated in the processing flow panel at the left (see Figure 6:

- **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 Bit File:** This design is rendered into a configuration file that can be downloaded onto the chip.

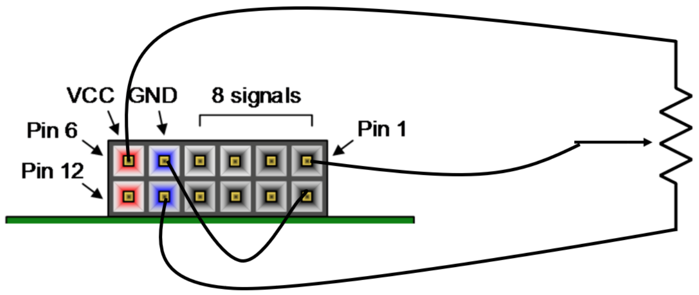


Figure 5: Setting up the ADC input. The variable resistor will act as a voltage divider to bring the input into the range of 0-1V

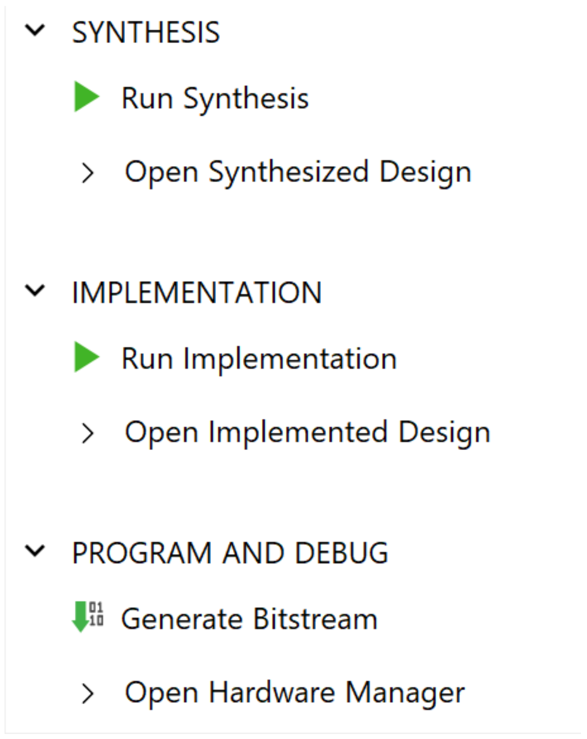


Figure 6: The steps necessary to configure the Artix-7: synthesis, implementation, generating the bitstream, and using the hardware manager to download the configuration.

In a real design, in which density or performance is an issue, the designer may intervene and optmize 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 7.

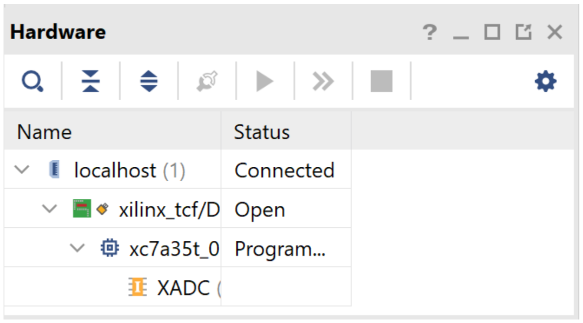


Figure 7: The hardware manager, when the board is correctly connected.

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 the configuration by adjusting the variable resistor until the display comes within the range 0000 to FFF0. Note that the lowest order digit is always 0, because this is only a 12-bit ADC.

Adjust it until it is about the middle of the range (800) and use the scope to verify that this is about .5 V (note: the scope does not automatically detect that these are x10 probes, so you will have to use the "CONF" button to select this).

Modifying the code

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

One of the things we'll do in this section is to convert the ADC value to decimal, which would require a 5-digit display to display the largest values. Since we're only interested in the highest order 12 bits anyway, we can solve this problem by right shifting the output when we load the data from the XADC wizard. The easiest way to do this is to change the line near the bottom to

```
if(ready==1) ADC=XADC_data>>4;
```

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

- Make LED0 the logical AND of the first four switches.
- Make LED1 the logical OR of the first four switches.
- Make LED2 come on if ADC is below half the range (`assign led[2]=(ADC<12'h800);`
- Make LED3 come on if ADC is greater than half the range (inverse of LED2).
- Tie the first output of the JB connector (JB[0]) to the state of LED3; ie, assert it if the signal is more than half the range.

Also, 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. 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.

To convert the display to decimal, replace the assignment of `display_bytes` directly to `ADC` with a conversion, using the `hex2bcd` module, included with the template but not yet used:

```
module hex2bcd (
    input [15:0] hex,  // hexadecimal representation
    output [19:0] dec // decimal representation
  );
```

Follow the instantiation example is `seg7decimal` but use a different instantiation name (eg "u2"). Don't worry about the fact the number of bits don't match exactly. Verilog is much more forgiving about that sort of thing than VHDL (which can often lead to more coding errors!)

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.

For the next steps, we'll use the function generator on the scope. This function generator has variable waveforms and frequency, but unfortunately has a fixed amplitude of $\pm \approx 1$ V. In order to make the amplitude variable, replace V_{cc} in Figure 5 with the output of the function generator, using the BNC to alligator clip cable and some jumpers. Remember to ground the black lead.

Configure the generator to produce a 1 Hz square wave, using a combination of the arrow keys and OK button at the top. Adjust the variable resistor so that the input crosses the .5V threshold and verify that LEDs 2 and 3 are alternating at 1Hz at this point.

Pulse Counter

Now we're going to implement a 16 bit counter

```
    reg [15:0] count=0;
```

Set up the logic inside the `always` block to increment this counter every time the ADC value crosses half the full scale value (this will involve creating another register that always holds the previous value of the ADC). Note that although Verilog is similar to C, it does not understand "++", so you will have to explicitly say count = count + 1.

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 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.

As a final step, increase the pulse rate to 1 MHz and use the two scope probes to measure the time between when the input square wave goes high and output `JB[0]` is asserted. Note that almost all of this time is related to the input and output time. The internal logic propagation is much faster.

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.