**Digital Logic Lab**

In this lab we will do the following:

1. Investigate basic logic operations (AND, OR, INV)
2. Learn a bit about FPGAs (Field Programmable Gate Arrays)
3. Implement an adder on an FPGA
4. Implement an Accumulator on an FPGA

***Background:***

In lecture, we discussed the 3 basic logic operations: AND, OR, NOT (inversion). We examined each operation and learned that these operations can be implemented using a logic gate. We went further to see how we could implement any truth table in terms of these basic logic gates. We created a multi-bit adder by "cascading" full adder circuits. We saw how to store state in registers and create state-dependent logic in the form of Finite-State Machines (FSMs).

We also saw Field-Programmable Gate Arrays—programmable chips that could be configured to implement any network of gates and flip-flops.

In lab today, we'll see how to program these FPGAs to build logic, arithmetic, and stateful functions.

**FPGAs (Field Programmable Gate Arrays):**

Field-Programmable Gate Arrays contain an array of programmable gates.

In the particular FPGA we will be using, each programmable gate is called a Logic Cell (LC) and can be programmed to implement any gate of 4 inputs. The 4-input LC gate is essentially programmed by specifying its truth table. Since it is a 4-input gate, it requires $2^4$=16 bits for its programming. The LC also has provisions to support carry chain logic so that adder bits can be implemented with a single LC rather than with two. Each LC is also associated with an optional Flip-Flop (DFF) to hold state. These gates are arranged in a two-dimensional array, and programmable routing allows us to connect the inputs of any gate (LC) to the outputs of any other gate (LC) or the pins of the FPGA chip. Similarly, the programmable routing allows us to connect the output pins on the FPGA chip to the outputs of one of the gates. The particular device we will be using for this lab has 7,680 LCs on it.

While not essential for this lab, you can find the datasheet for the FPGA we will be using:
http://latticesemi.com/view_document?document_id=49312

***Prelab: Part 1***

Submit your answers to this part to the Lab 7 prelab assignment on Canvas before Monday lab session. Course staff will review before or at the beginning of the lab session.

1. Write the truth table for each of the following functions. Note that "Out" is the output and "p1" and "p2" are two inputs. "p3" is a third input.
    a. Out = NOT( AND( p1, NOT( p2 ) )
    b. Out = OR( AND( p1, p2), NOT( p3 ) )

2. A Full Adder (FA) is a useful 3-input, 2-output logic function out of which we can implement larger addition operations. The basic function of a full adder is to take in 3 input bits, count the number of ones, and produce a 2-bit output to represent the sum. Mathematically: $2 \cdot carry + sum = i0+i1+i2$

   That is, if we treat the three inputs as 1-bit values taking on 0 or 1, then we can sum them up and get a value between 0 and 3. We represent the result in a 2-bit binary number, calling the least significant bit the `sum`, and the most significant bit the `carry`.

   Complete the truth table for the Full Adder

| **inputs** | | | **outputs** | |
|---|---|---|---|---|
| **i0** | **i1** | **i2** | **carry** | **sum** |
| 0 | 0 | 0 | | |
| 0 | 0 | 1 | | |
| 0 | 1 | 0 | | |
| 0 | 1 | 1 | | |
| 1 | 0 | 0 | | |
| 1 | 0 | 1 | | |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

3. The FA can be expressed in gates. Write a logical expression in terms of AND, OR, and NOT gates for each of the two outputs (sum, carry) for the FA. You may use gates with more than 2 inputs.
4. Given two FA gates, how would you compose them to perform a 2-bit addition (take in two 2-bit values and produce one 3-bit result)? (Hint: how do we add bits of equal significance? What do we do with the carry? Why did we define the FA as having 3 inputs?)
5. Given k FA gates, how would you compose them to perform a k-bit addition (take in two k-bit values and produce one (k+1)-bit result)?
6. Explain why we need (k+1) bits to represent the result of a k-bit add.

**Part 2: Install TinyFPGA Tools on your laptop**

**Windows/OSX/Linux**:

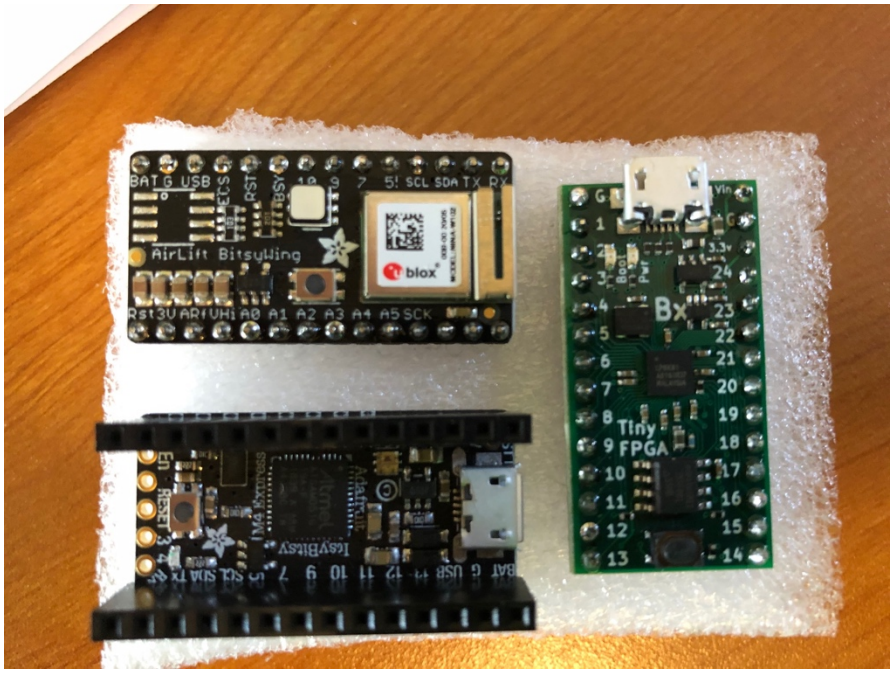
1. Follow the instructions: https://tinyfpga.com/bx/guide.html
    a. Perform software install
        **i.** When you get to "**3. Download and install Atom.**"
            1. In step 2, "setting" is not under "File"
                a. It is a tab in the middle top (probably right of "Welcome" tab)
            2. Maybe "preferences" under Atom will get you there, too.
        ii.
    b. Later, after unpacking the FPGA, you will follow the "first project tutorial" on that same page

***Lab Procedure:***

***Lab – Section 1: Working with a USB FPGA***

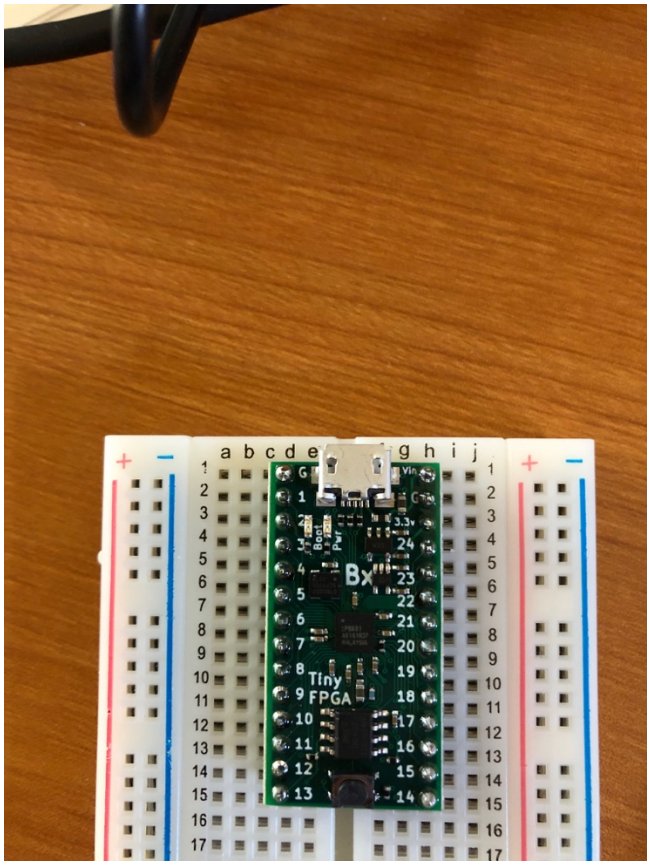
- In this section, you'll learn how to compile simple combinational logic in Verilog for an FPGA

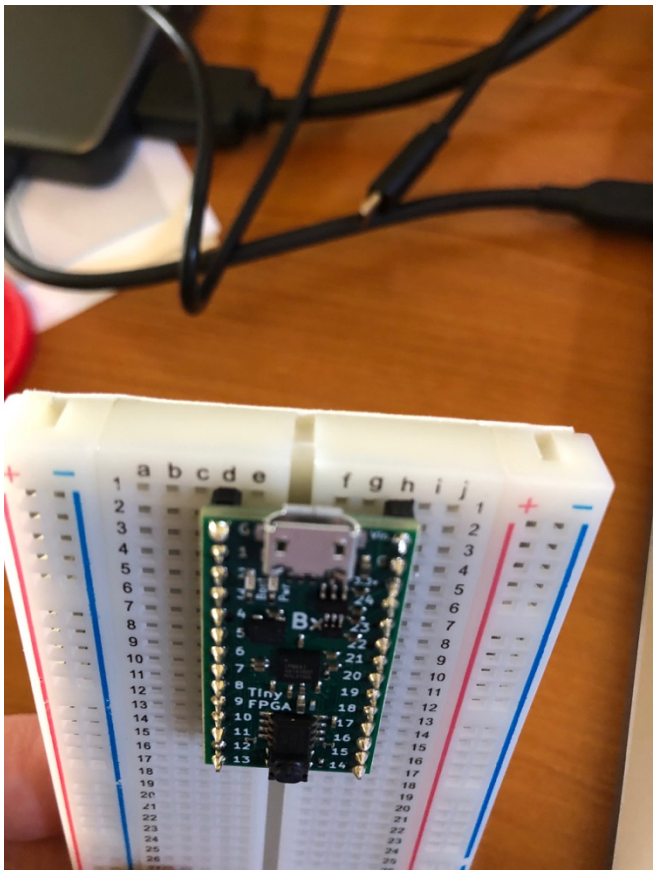
1. Your Lab kit should include a TinyFPGA Bx FPGA, PMOD switches (2 kinds), 2 PMOD cables, and a USB Extension cable.
    i. The TinyFPGA was probably in the same wrapped package as the ItsyBitsy. It's the board on the right of the bottom (no bubble wrap) picture:



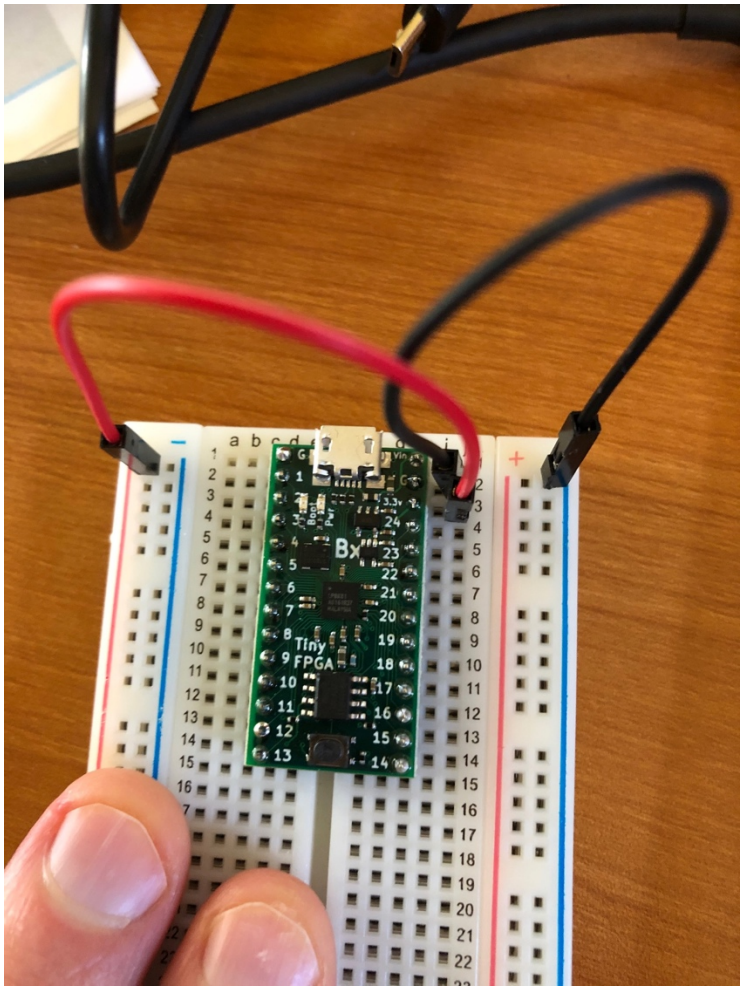

ii. The Pmod switches (bottom left), Pmod cables (right), and the USB extension (top left; you probably already used with the ItsyBitsy):


2. Put the TinyFPGA on the breadboard.
    i. Place leftmost terminal row in column D; the rightmost in column h.
    This leaves room to put the PmoDBTN directly on the breadboard as shown below.

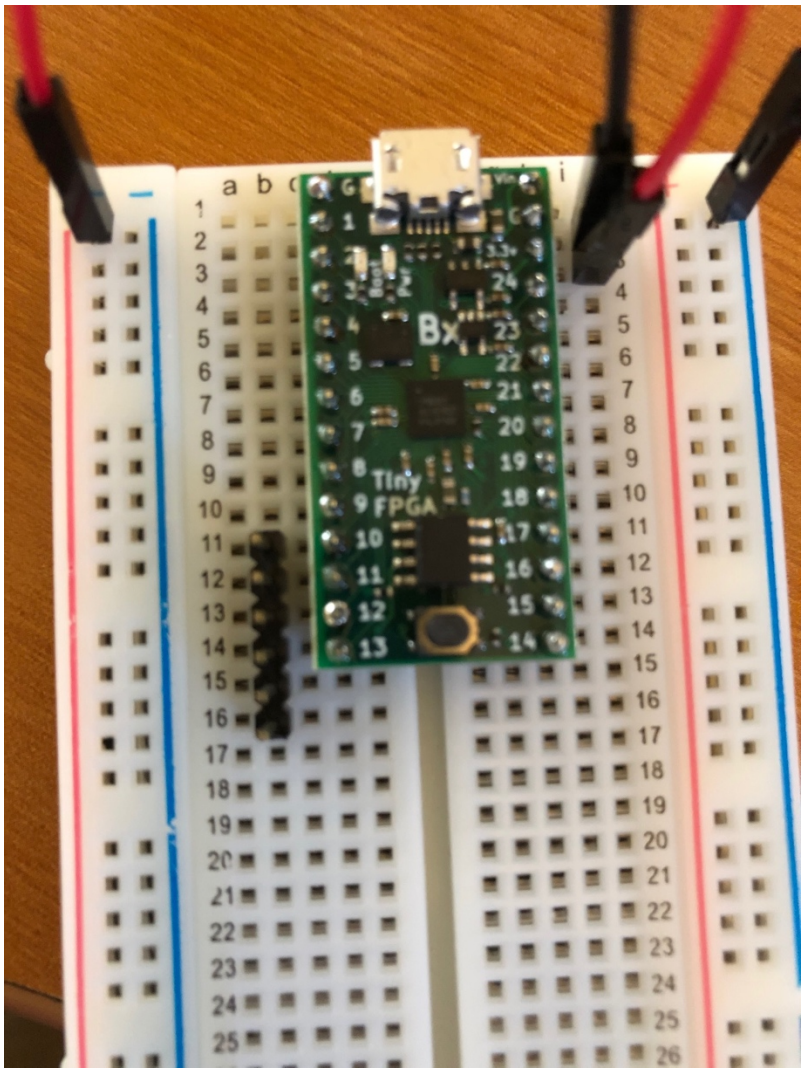

a b c d e f g h i j
Tiny FPGA


a b c d e f g h i j
Tiny FPGA

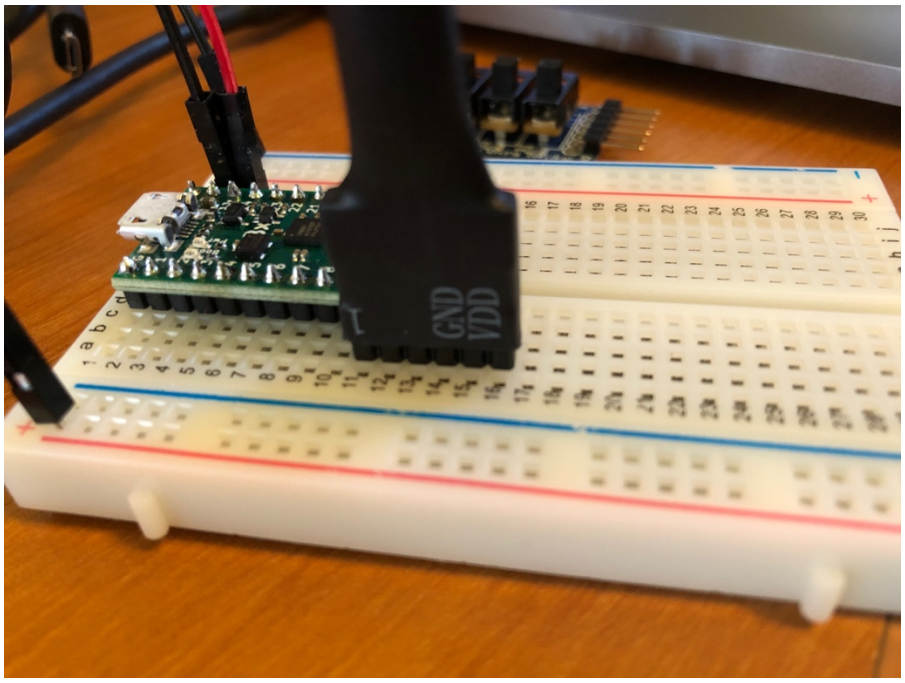
3. Follow the “first project tutorial” on the same page where you grabbed the software: https://tinyfpga.com/bx/guide.html
4. In a terminal window, create a directory for your work for this lab by entering the following commands into the terminal. If you are on a Windows computer, make sure to use PowerShell instead of cmd.
    i. `mkdir ese150logic`
    ii. `cd ese150logic`
    The `mkdir` (make directory) command will create the directory/folder. The `cd` command (change directory) will change into the new directory, like opening up a folder and looking inside.
5. Download and unpack the preliminary Verilog files you will need for this lab from the link on the course syllabus.
6. Connect breadboard power and ground
    i. Connect the G pin (second from top on right) to the rightmost ground (-) column.
    ii. Connect the 3.3v pin (third from top on right) to the leftmost power (+) column.



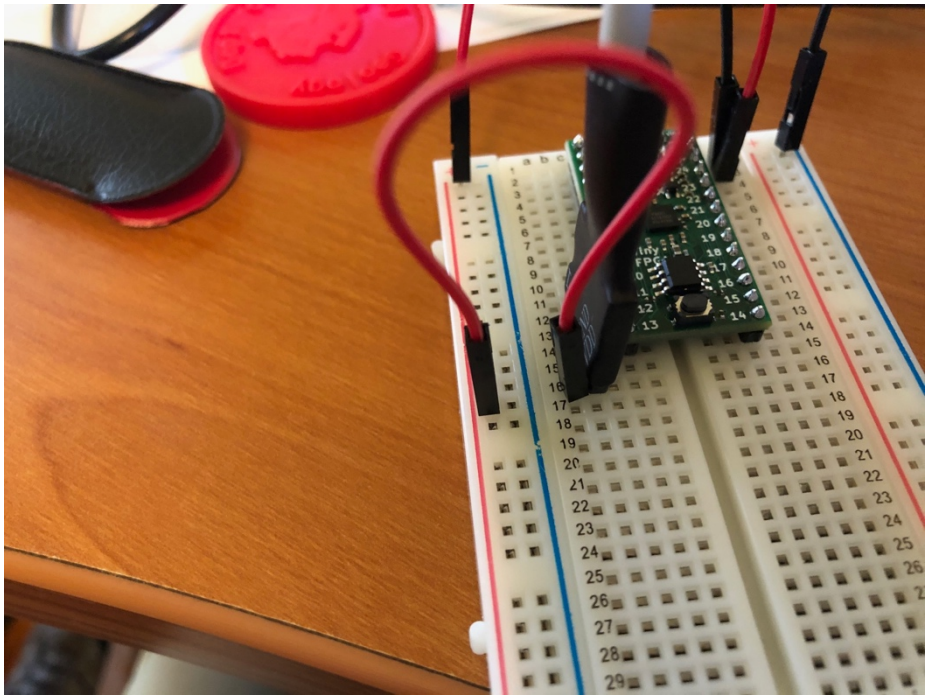
7. Connect first PMOD Switches up to the FPGA.
    i. As a side note, Pmod means “peripheral module” and refers to a standard of connections by Digilent, the company that makes the buttons and switches.
    ii. Take the 6 pin header from one Pmod cable and stick it in the bread board
        i. Aligned in row b
        ii. With the topmost pin in aligned with TinyFPGA pin 10 (left of TinyFPGA)
        iii. This guarantees that pins 10, 11, 12, and 13 are aligned with the first four header pins (which will become SWT1, SWT2, SWT3, SWT4).
        iv. The last two should then not align with any FPGA pins (these will connect to Pmod cable GND and VDD).



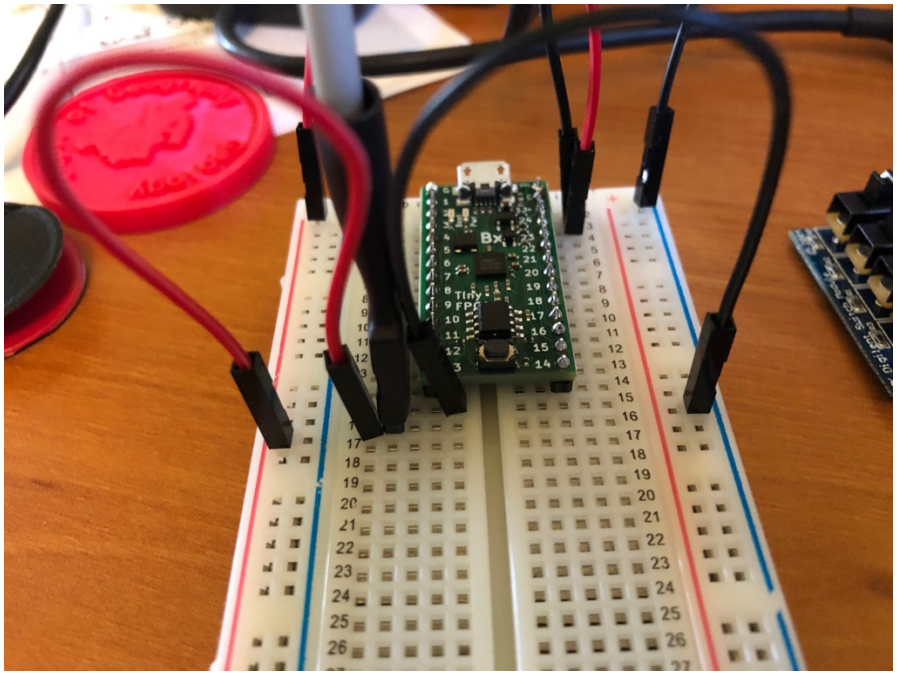
iii. Insert the cable, so that the labeled GND and VDD are away from the FPGA and VDD aligns with the bottom-most pin of the header you just inserted. The pin marked 1 aligns with FPGA board pin 10.



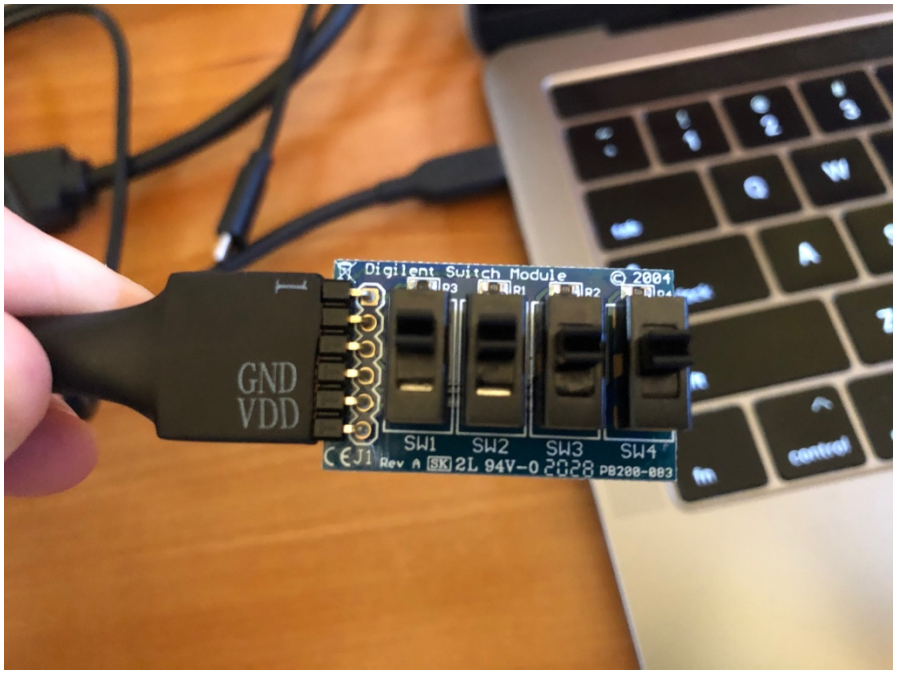

iv. Connect the bottom-most Pmod cable pin (VDD) to the left-most breadboard power (+) column (same one where you connected the 3.3v pin from FPGA board).



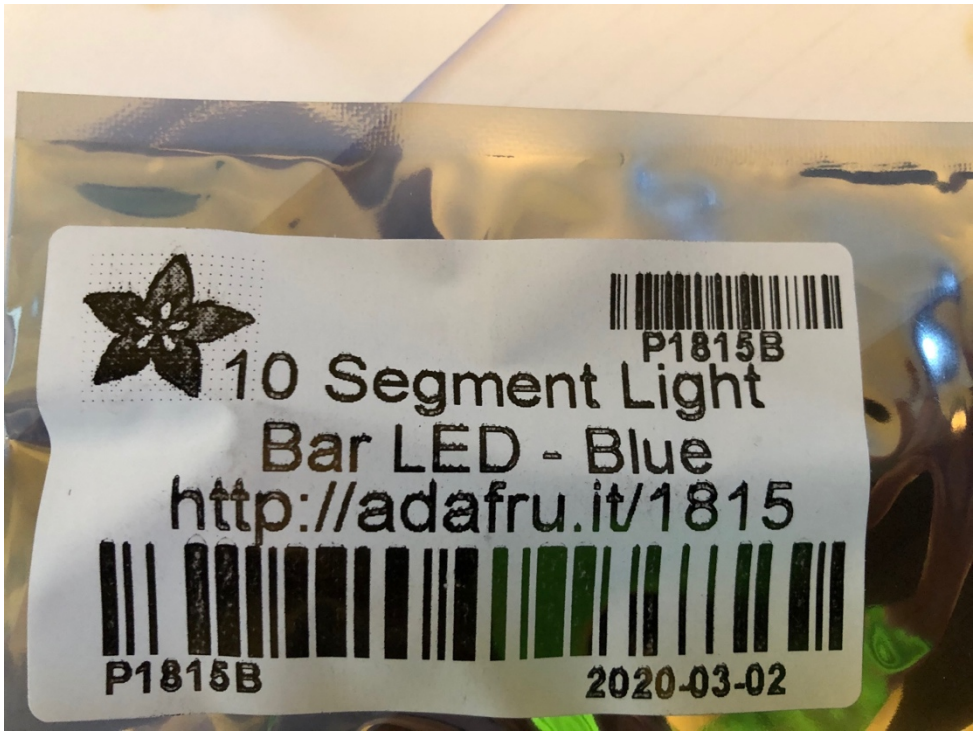
v. Connect the Pmod cable ground pin (one up from VDD, just below FPGA) to the right-most breadboard ground (-) column (same one where you connected the G pin from the FPGA board).

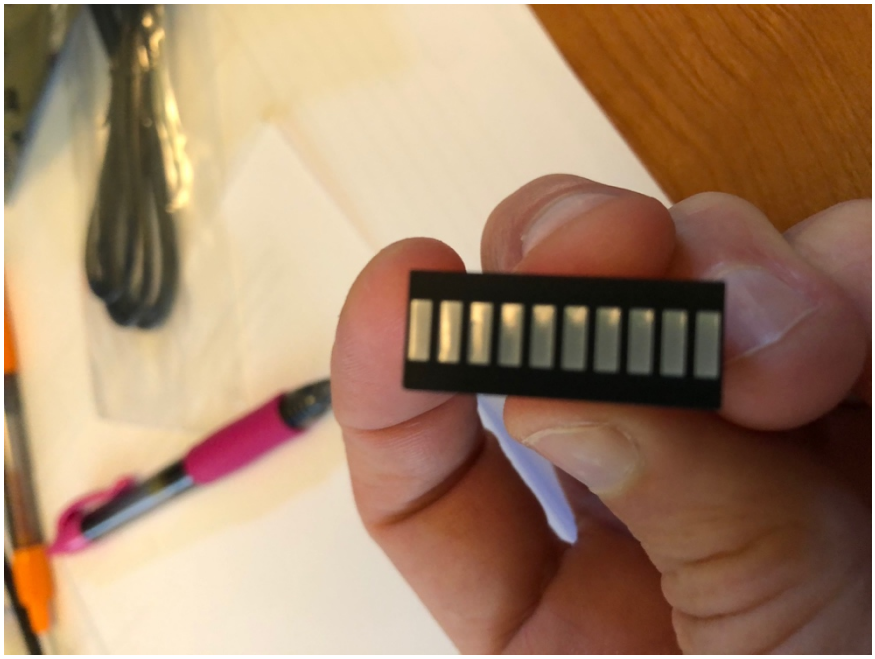


vi. Connect the PmodSWT switch to the other end of the PMOD cable.

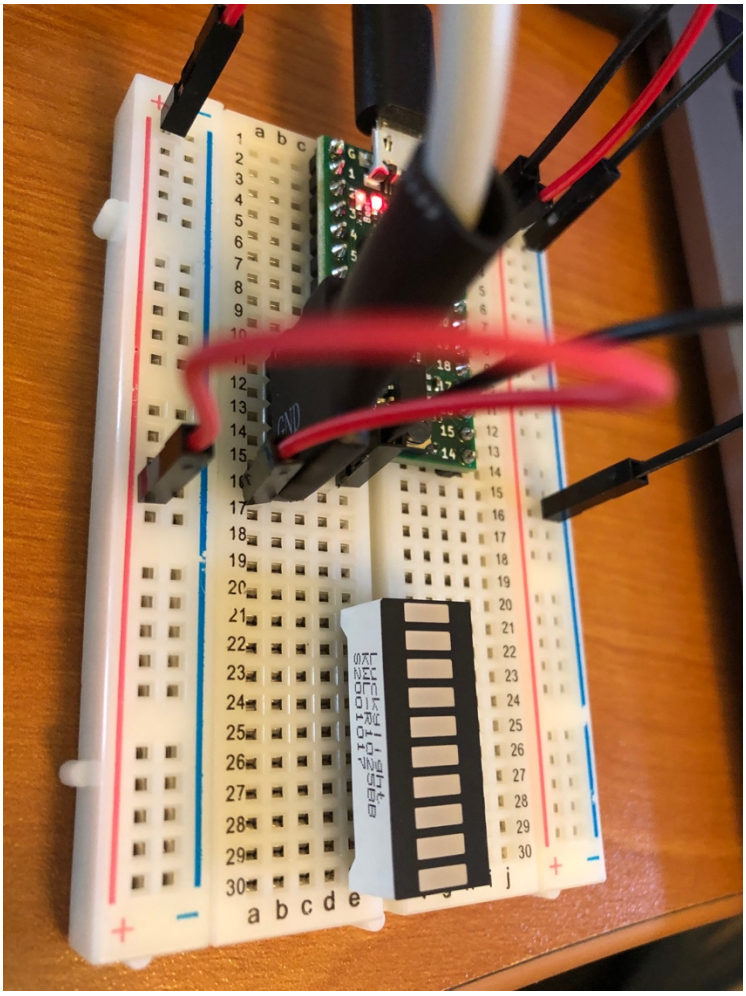


8. Connect a set of LEDs to the FPGA
    i. Find the 10 Segment Light – Bar LED

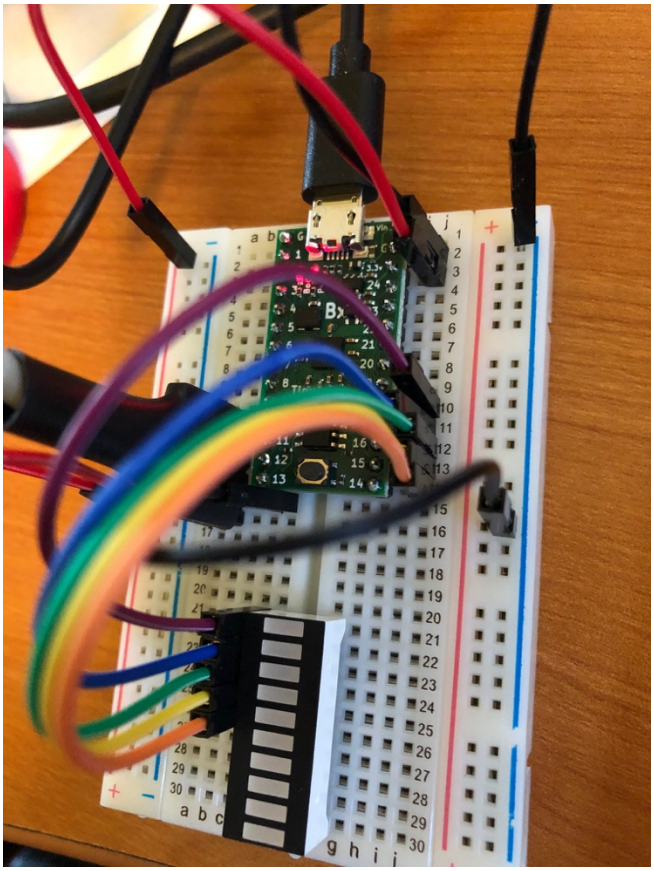


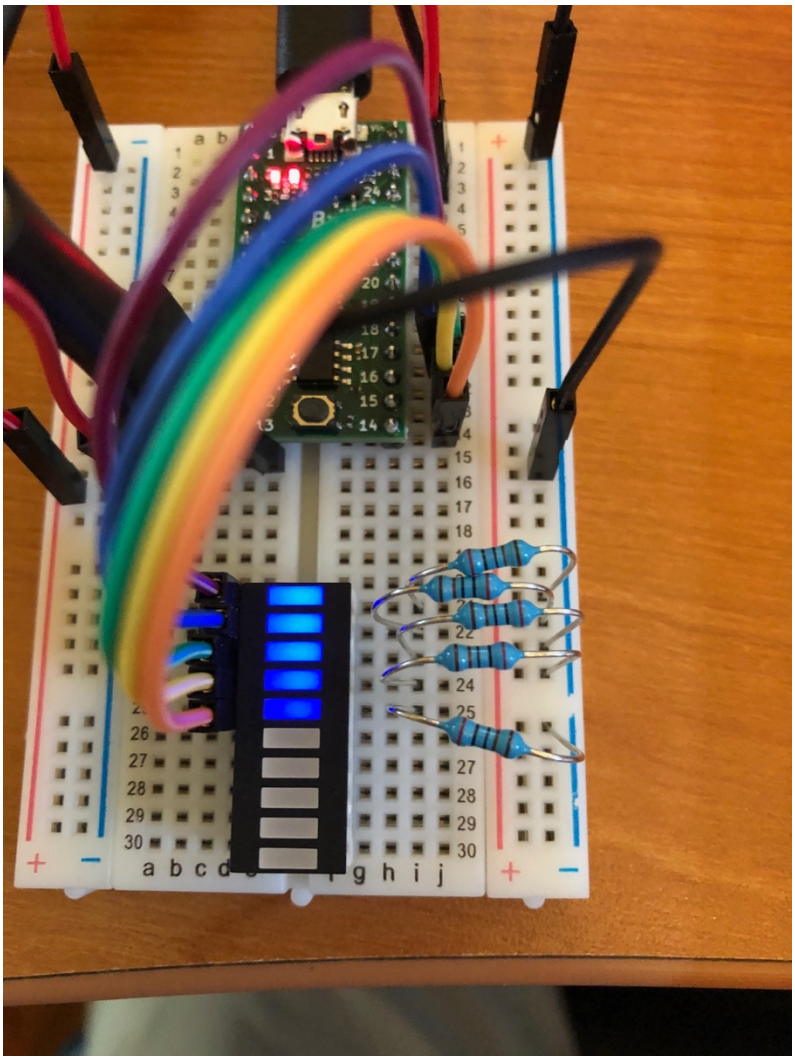
ii. Stick in bottom of breadboard



iii. Wire pins 14—18 (5 pins) to the top 5 Bar LED pins
    i. Hint, if you haven't separated all your wires, you can keep a set of 5 wires connected together for this
    ii. If you have separated them, that's fine, too.



iv. We need to limit the current through the LEDs, so the other side of each LED bar should be connected to ground through a 2K Ω resistor.
    i. Use the rightmost ground (-) strip where (same place you connected the FPGA G pin).



9. Review the Verilog file section1.v to see how it encodes combinational logic.
   The first section looks like the header signature on a C or Java function and serves a similar role. Here, it defines the input and output signals. This is the top-level for our design on the FPGA. It is defining the Inputs and Outputs for the entire FPGA. We will use this same Input/Output configuration for the entire lab. The key outputs are the LEDs, and the key inputs are on the PMOD connector, which you wired in the previous step. Also included is a clock signal (clk), which we will not use for this part of the lab.

```
`default_nettype none
module demo(
    input      clk,
    output     PIN_14,
    output     PIN_15,
    output     PIN_16,
    output     PIN_17,
    output     PIN_18,
    input      PIN_10, // input p1
    input      PIN_11, // input p2
    input      PIN_12, // input p3
    input      PIN_13  // input p4

 );
```

   Following this we declare some internal variables. These are similar to local variable declarations in C and Java. Here, the only type is “wire” meaning a combinational signal.

```
// Alias inputs
   wire   p1;
   wire   p2;
   wire   p3;
   wire   p4;

// Alias outputs
   wire   o1;
   wire   o2;
   wire   o3;
   wire   o4;
   wire   o5;
```

   Following this, we have some assignments. These are simply giving more friendly names to signals, in this case the inputs, for use with this piece of logic.

```
   assign p1=PIN_10;
```

```
assign p2=PIN_11;
assign p3=PIN_12;
assign p4=PIN_13;
```

Note that p1=PMOD1 is physically BTN0 on the Button Module. Similarly, p4=PMOD4 is BTN3. We have one more assignment which serves to directly connect one of the inputs to a signal we will connect to the output:

```
assign o5=p4; // output directly controls
```

We place the actual logic in the next section. The <= symbol is used for logic assignment (it is not a comparison operation). This logic demonstrates how Verilog expresses and (&), or (|), and invert (!) Boolean operators we introduced in the introduction.

```
    always  // combinational assignment -- always computing
    begin
       // <= is used for logic assignment
        o1<=p1 & p2;   // and together two inputs
        o2<=p1 | p2;   // or together two inputs
        o3<=!(p1 & !p2); // use a not !
        o4<=(p1 & p2) | !p3; // compound logic expression
    end
```

In the final section, we have more assignments to connect the logical outputs computed by the logical expression to the module outputs.

```
// Wire up the lights
    assign PIN_14 = o1;
    assign PIN_15 = o2;
    assign PIN_16 = o3;
    assign PIN_17 = o4;
    assign PIN_18 = o5;
```

10. Compile and download the section1.v Verilog file to the FPGA:

    i. Create a new subdirectory section1
        i. Mkdir section1
    ii. Copy the files from blink_project to this directory
        i. cp PATH_TO/blink_project/* section1/
    iii. Copy section1.v to this directory
        i. cp section1.v section1/
    iv. Load into Atom
        i. File/Open and navigate to your section1 directory
    v. Clean up the files you copied over that need to be rebuilt

i. apio/clean

vi. Build
   i. apio/build

vii. Reset the FPGA into the bootloader using the reset pin

viii. Program the FPGA
   i. apio/upload
   ii.

ix. You will see the output of the compilation and download steps scroll by. Then the LEDs will glow dim then return to a state with some on and others off. At this point, the FPGA should be programmed and ready for use.

11. Review the output of the compilation process and note the resources used.
    To get this, you will need to run from the command line.
    In your section1 directory run:

```
apio clean
apio build –verbose-arachne
```

    Scroll back and look for the following section:

```
After packing:
IOs           10 / 63
GBs           0 / 8
GB_IOs      0 / 8
LCs           4 / 7680
DFF         0
CARRY       0
CARRY, DFF  0
DFF PASS    0
CARRY PASS  0
BRAMs         0 / 32
WARMBOOTs     0 / 1
PLLs          0 / 1
```

    This says we are using 4 LCs (Logic Cells) out of 7680 and 10 IOs out of 63. The 4 LCs are for each of the 4 expressions we compute. None of them have more than 4 inputs, so they can each fit into a single LC.

    Include this output in your writeup (for this and each of your following designs).

12. Use the input switches and LEDs to verify the truth table for the basic logic functions and the simple combinational logic in the Verilog file. Record the truth table for o4 and include with your lab report.

***Lab – Section 2.1: Writing your own combinational logic***

- In this section you'll learn how to write simple combinational logic in Verilog and implement your FA and multi-bit adder from the preclass.

1. Create a new subdirectory section2fa in parallel with section1.

```
cp section1/* section2fa/*
```

2. Edit top.v in section2fa and change the Verilog logic equations in section2fa/top.v to implement your full adder from Prelab Question 3.
    a. Declare wire variables for i0, i1, i2 and assign the inputs i0, i1, i2 to the inputs PIN_10, PIN_11, PIN_12.
    b. Write your logic equations for sum and carry inside the always block in place of the logic that was in Section1.
    c. Connect the output sum to PIN_14, output carry to PIN_15.
3. Open you section2fa design, compile and upload your section2fa/top.v.
    a. Load into Atom
        i. File/Open and navigate (select) your section2fa directory
    b. Clean
        i. apio/clean
    c. Build
        i. apio/build
    d. Reset the FPGA into the bootloader using the reset pin
        i. If you don't do this, the upload that follows will fail
        ii. Apio will warn you to reset the FPGA
    e. Program the FPGA
        i. apio/upload

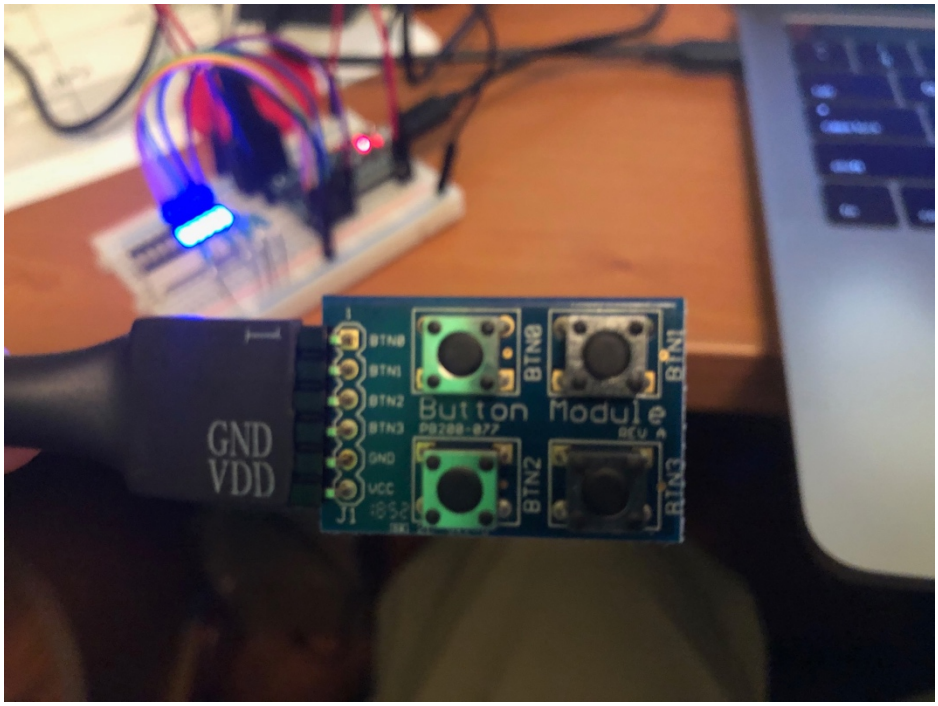
4. Use the inputs and LEDs to verify the truth table for your full adder in section2fa.v.
    a. Debug your logic as necessary.
5. Report the resources needed by your full adder
   In your section2fa directory run:

```
apio clean
apio build –verbose-arachne
```

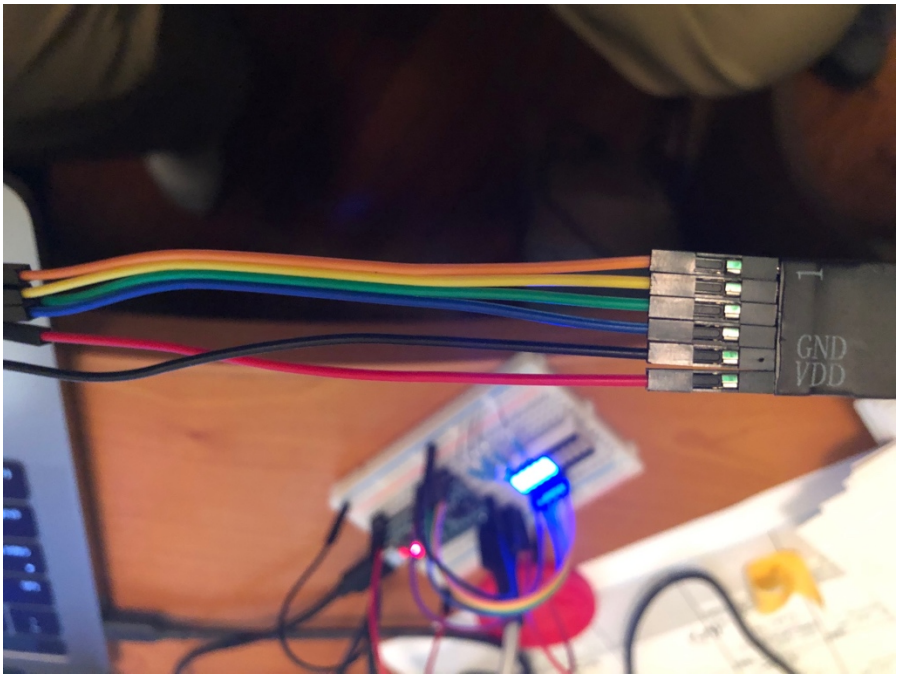
   Scroll back and look for the "after packing" report.

***Lab – Section 2.2: Building a 4-Bit Adder***

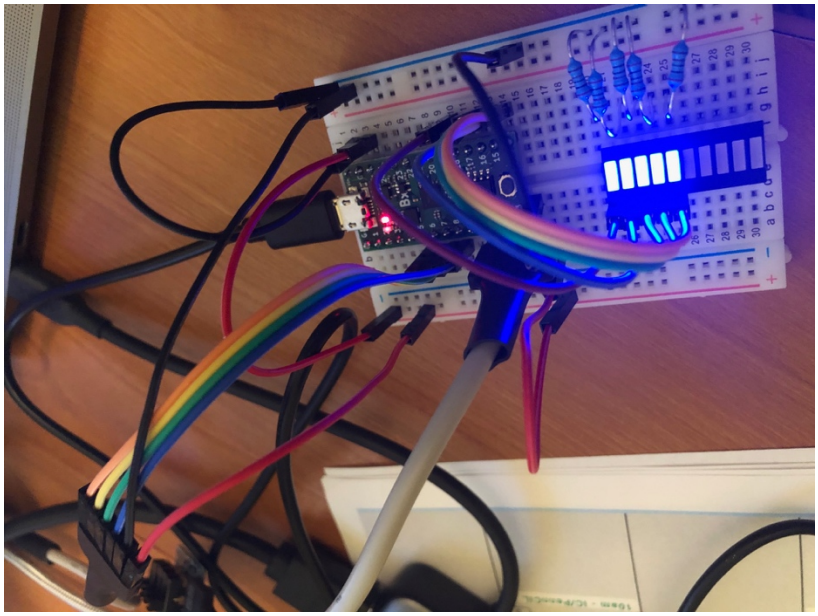
1. Connect the PmodBTN so you have 4 more inputs.
    a. Connect the PmodBTN to one end of the cable, matching the GND/VDD cable pins with the GND/VCC PmodBTN pins




    b. Connect a set of 6 wires to the other end of the cable
        i. You can leave the 4 button pins (all but GND, VDD) together as a bundle



c. Wire the 4 button pins to the breadboard to connect to FPGA pins 5—8
    i. This skips 9 before the other Pmod cable you already wired



d. Wire the GND pin to the rightmost ground (-) column on the breadboard
e. Wire the VCC pin to the left-most power (+) column on the breadboard

2. Create a directory for section2add4
    a. Copy over files from section1
    b. Copy section2add4.v to section2add4/top.v
3. Note that the Verilog now includes 4 more inputs

```
// assign inputs to signals with meaningful names
assign a[0]=PIN_10;
assign a[1]=PIN_11;
assign a[2]=PIN_12;
assign a[3]=PIN_13;

assign b[0]=PIN_5;
assign b[1]=PIN_6;
assign b[2]=PIN_7;
assign b[3]=PIN_8;
```

    a. Note that we can use the array notation to refer to individual bits in the a and b variables.
    b. Also, note that b[0]=PIN_10 is physically SW1 on the Digilent Switch Module. Likewise, b[3]=PIN_13 is SW4. And a[0]=PIN_5 is BTN0 on the Digilent Button module and a[3]=PIN_8 is BTN3. Yes, Digilent isn't consistent between these two modules whether they start numbering at 0 or 1.

4. Revise the Verilog logic equations in section2add4/top.v to produce a 4-bit adder:
    a. We have setup the inputs and outputs for you. This shows that you can declare multi-bit variables in Verilog similar to arrays in C or Java. Here, a and b are each 4-bit values. c and o are 5-bit values.

```
// Alias inputs
wire  [3:0] a;
wire  [3:0] b;
wire  [4:0] c; // you will likely use

// Alias outputs
wire   [4:0] o;
```

    b. Create your adder by replicating the full adder logic equations you have already written for each set of inputs and connecting the carry out (c[i]) between the bits of the full adders. Treat the carry input to your circuit (c[0]) as 0.

5. Compile and upload section2add4/top.v to your FPGA.

6. Consult the output of the compilation process and note how many LCs your 4-bit adder uses.

7. Use the inputs and LEDs to verify the correct function of your 4-bit adder:
    a. If we were to exhaustively test your adder, how many test cases (sets of input values) would there be? (that is, how large would the truth table be?)
    b. Test at least the following cases: 0+1, 0+2, 0+4, 0+8, 1+0, 2+0, 4+0, 8+0, 1+15, 2+15, 4+15, 8+15, 15+15, 5+2, 2+5, 7+1, 1+7.
    c. Test 4 more “random” cases.

***Lab – Section 3: Working with Verilog Arithmetic***

- In this section, you'll learn how to write simple arithmetic in Verilog

Arithmetic is common in Verilog, so you can also write arithmetic expressions directly.

1. Review the Verilog file section3add4.v to see how it encodes a simple addition.

Here, we simply tell it to perform addition on the multi-bit variables using the multi-bit addition (+) operator. The rest of the code in section3add.v is the same as the setup you saw for section2add.v.

```
always  // combinational assignment -- always computing
   begin //
     o<=a+b;
     end
```

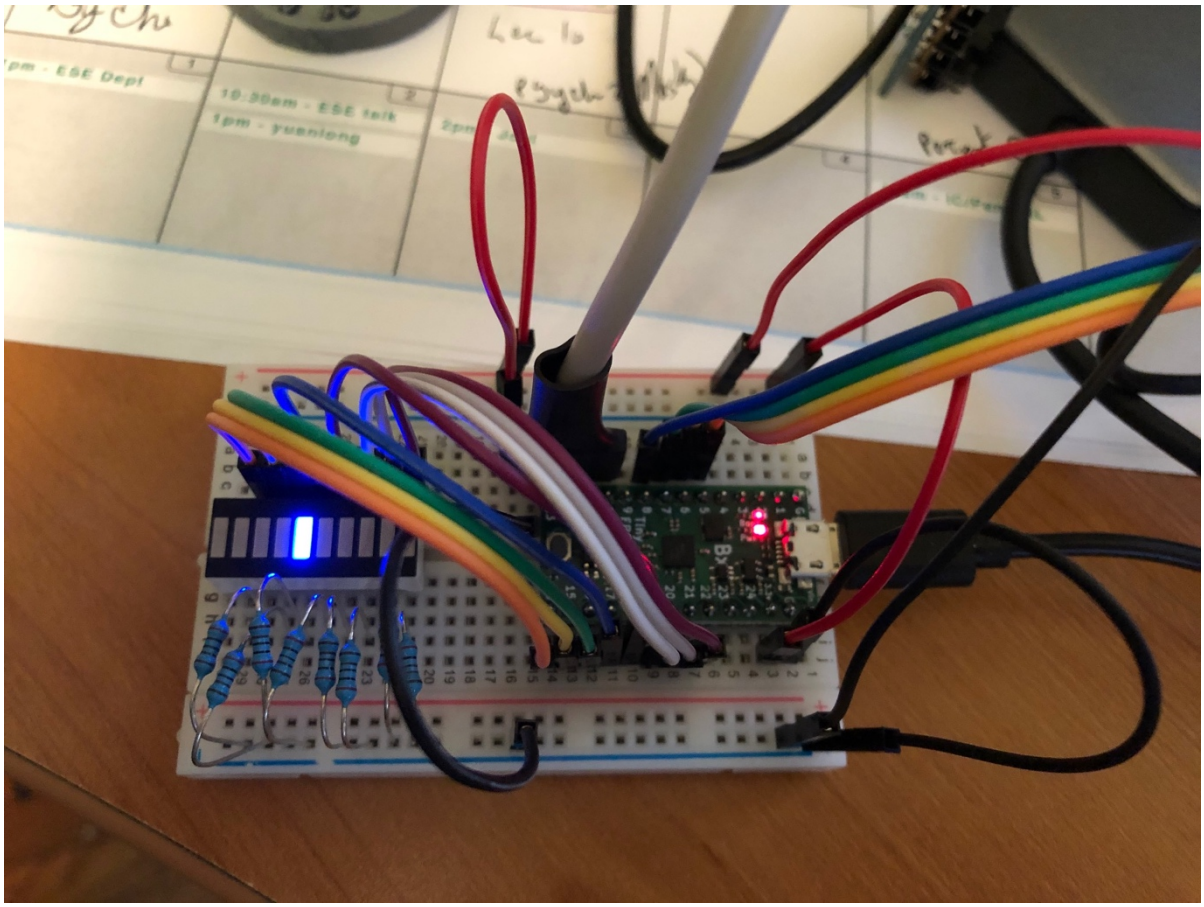
2. Create a directory for section3add4, copy over section1 files, copy section3add4.v to section3add4/top.v, and compile and upload to the FPGA:
   a. Note the inputs are the same as the end of Section 2.
   b. Record resources required (LCs and CARRY) and explain them. Note that it now uses CARRY logic resources.
   c. Use the inputs and LEDs to verify the correct function of this 4b adder. Perform the same tests as you did at the end of Section 2.

***Lab – Section 4: Working with State in Verilog***

- In this section, you'll learn how to maintain state in Verilog.
- You only need o submit the resource summary for this section, but the example here should give you the information you need to complete Section 5.

In Verilog, we can write logic that includes state in registers.

1. Add support for 3 more bar LEDs to your breadboard:
    a. Add 2K Ω resistors to the next 3 LEDs on the bar graph.
    b. Move the existing 5 wires down by 3 to match with the new resistors.
    c. Add 3 more wires to connect the top 3 LEDs (opened up by moving the existing wires down 3) to the next 3 FPGA pins 19, 20, 21.
        i. This wire relocation is so that the LEDs will be in a sensible order related to the FPGA pin numbers.



2. Review the Verilog file section4fwd.v to see how it encodes a simple clockwise rotation of the LEDs.

We now use the reg type instead of wire to denote that these variables are registers (flip flops). They will hold state and can be controlled to only change their values at clock edges. We declare these as multi-bit values.

```
// Manage 16MHz clock
   reg [24:0] counter;
   reg [2:0] dec_cntr;
```

The clock on the TinyFPGA BX board runs at 16MHz. Unfortunately, if the LEDs changed at 16MHz, we wouldn't be able to track them. So, we start by slowing the rate of advance down to 0.5 seconds. We do this by counting to 8 million between each of the sequential logic operations. Each time the clock counter reaches 6 million, we reset it and increment the counter for the LEDs. Since this is sequential logic, we only want the logic to operate in response to a clock edge. We specify that by telling the always block to operate on the positive clock edge, when the clock goes from low to high.

```
// The 16MHz clock is too fast
// ...count to 8 million to divide it down to a half second clock
     always@(posedge CLK)
       begin
          counter <= counter + 1;
          if (counter == 8000000)
            begin
           counter<=0; // reset counter
           dec_cntr <= dec_cntr + 1; // count half seconds
         end
       end
```

We use combinational logic to select LEDs based on values of the dec_cntr:

```
// Make the lights blink -- each light activated on a
different value of 2b half-second counter
   assign PIN_14 = (dec_cntr == 0) ;
   assign PIN_15 = (dec_cntr == 1) ;
   assign PIN_16 = (dec_cntr == 2) ;
   assign PIN_17 = (dec_cntr == 3) ;
   assign PIN_18 = (dec_cntr == 4) ;
   assign PIN_19 = (dec_cntr == 5) ;
   assign PIN_20 = (dec_cntr == 6) ;
   assign PIN_21 = (dec_cntr == 7) ;
```

3. Create a directory, setup files, Compile and upload `section4fwd.v`
    a. Watch how lights behave and relate to the logic.
    b. Make sure the LEDs sequence from bottom to top. This also serves as a sanity check on your wiring. If you see the LEDs firing one at a time, but not in the expected order, use that as a guide to rewire appropriately.
4. Capture the resource usage and include in your report.
5. You will need to be able to understand how to use registers for Section 5. Hopefully, this is a useful example.

***Lab – Section 5: Implement an accumulator in Verilog***

- In this section you'll implement an accumulator in Verilog

An accumulator is a unit that keeps a sum of all the inputs that it has been given since being reset. (Note that the large piece of ENIAC in the first floor ENIAC Suite is labeled "Accumulator 18".) Since it remembers the previous sum, it must maintain state in registers.

We will build an 8b unsigned accumulator with 4b unsigned inputs. That is, the accumulator can store values between 0 and $2^8$-1=255 and take as inputs values between 0 and $2^4$-1=15. We use the 8 LEDs wired up on the bar LED to display the accumulator value.

Our complete set of inputs will be:

- 4b of input – use the 4 on-off switches (Digilent switch module, PIN_10 through PIN_13, SW1 through SW4); we call these in[3:0].
- Reset – to set the accumulator value back to 0; use a momentary switch (Button Module, PIN_5, BTN0), which we will call p_reset.
- Read-input – to take in the current value of the 4b input and add it to the accumulator value; use a momentary switch (Button Module, PIN_6, BTN1), which we will call p_input.

One challenge is to make sure that each p_input button press results in only a single addition of the input in[3:0] to the accumulator. To do that, we want to demand that we only take a valid keypress if p_input was previously 0. We use the previous_p_input register to hold the previous value of p_input.

We have setup the input and outputs for you in section5start.v. This includes the counter from section4fwd.v so that keypresses are considered only every 0.1 seconds.

HINT: If you want to set the output values (such as PIN_14) within an "always" block, don't write "assign" before the output name. For example, instead of writing "assign PIN_14 = accum[0]", simply write "PIN_14 = accum[0]".

NOTE: the if statement in Verilog only works in always blocks. You won't able to use it in the section outside the always block where the output PIN_* (LED) assignments are made.

1. Make a directory section5acc for your accumulator.
2. Copy over files from section1 into section5acc.
3. Copy section5start.v to section5acc/top.v
4. Revise top.v to behave as an accumulator as described above.
    a. Add your accumulator logic along with the counter reset as noted.
5. Clean/build/upload your design.
6. Test your design on a number of summation sequences.
    a. Reset the accumulator and add a 1 for 20 times; use the show high nibble to check full counter value.
    b. Reset the accumulator and add a 15 for 13 times. What result should the accumulator hold? Use the show high nibble to check full counter value.
    c. Reset the accumulator and add the integers from 1 to 6. What result should the accumulator hold? Use the show high nibble to check full counter value.
    d. Create a sequence of 6 random integers between 0 and 15. Note their sum. Reset the accumulator and add the integers. Use the show high nibble to check full counter value.
7. Record the LC resources needed by your design.
8. Show your accumulator to your TA for your exit ticket.
    a. TA will direct you to demonstrate a test of a different sequence of numbers.
    b. TA will review Verilog code.
    c. TA will ask questions about the design.

***Postlab***

1. How many LCs will be required for a two input, 16-bit adder (adds together two 16b inputs to produce one 17b output)?
   Hint: review the LC counts you found in Sections 2 and 3 for the single FA and the 4-bit adders. Use the observations you made in the prelab to generalize this to 16-bit.
   We realize the CAD tools don't always provide optimal mappings, so we're looking for a rough estimate of what this should take.
2. Based on LC usage, how many 16-bit adders could you put on the FPGA used on the TinyFPGA? (recall the FPGA has 7680 LCs)
3. How many 16-bit adders do you need to implement a combinational 16-bit multiplier (multiplies two 16b values to produce one 32b output)?

Recall that you can multiply two numbers by summing shifted copies of the multiplicand. For 16b numbers:

$$multiply(A,B) = \sum_{i=0}^{i=15} B[i] \times 2^i \times A$$

B[$i$] represents the $i$th bit of B, similar to the syntax you used in Verilog.

Assume the shift (shown as multiplication by $2^i$) comes for free (it is just expressing how you wire up the adder gates).

An example for a 4-bit multiplier:

| | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| p0[6:0]= | 0 | 0 | 0 | B[0]&A[3] | B[0]&A[2] | B[0]&A[1] | B[0]&A[0] |
| p1[6:0]= | 0 | 0 | B[1]&A[3] | B[1]&A[2] | B[1]&A[1] | B[1]&A[0] | 0 |
| p2[6:0]= | 0 | B[2]&A[3] | B[2]&A[2] | B[2]&A[1] | B[2]&A[0] | 0 | 0 |
| p3[6:0]= | B[3]&A[3] | B[3]&A[2] | B[3]&A[1] | B[3]&A[0] | 0 | 0 | 0 |

product [7:0] = p0+p1+p2+p3

4. What other logic do you need besides adders for the multiplier? (Hint: what does the multiplication by B[i] require?) How many LCs will this additional logic require? (per operation? For the entire 16b by 16b multiplication?)
5. How many of these combinational 16-bit multipliers can you place on the FPGA used on the TinyFPGA USB FPGA?
6. How many LCs will it require perform a combinational 16-point dot product on 16-bit inputs (input is 16 16-bit inputs for vector A and 16 16-bit inputs for vector B, output is one 36-bit output)?

$$dotproduct(A,B) = \sum_{i=0}^{15} A[i] \times B[i]$$

Here, A and B are vectors of 16b values (not 16b values as used earlier); A[i] and B[i] each represent a 16b value, so the multiplication of A[i] by B[i] is a multiplication like you developed in steps 3—5.

This solution should be entirely combinational – do not use an accumulator and sequential additions.

7. Does this fit on your TinyFPGA? If not, what is the largest dot product (number of 16-bit inputs) that will fit on your TinyFPGA?

**HOW TO TURN IN THE LAB**

- Upload a PDF document to canvas containing:
  - All tables completed
  - All code you wrote (.v files)
  - All resource summaries
  - Answers to all questions (highlighted in yellow)
  - Postlab answers
- Each student must submit an individual lab writeup.