ESE 150 – Lab 07: Digital Logic

<u>Digital Logic Lab</u>

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

<u>Background:</u>

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 1280 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
The software we will be using to program the part is open source and can be found:
http://www.clifford.at/icestorm/
(we have installed these for you on computers in Ketter and Detkin; we just mention this in case you want to work with FPGAs on your own after this lab.)

Prelab

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.
   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		

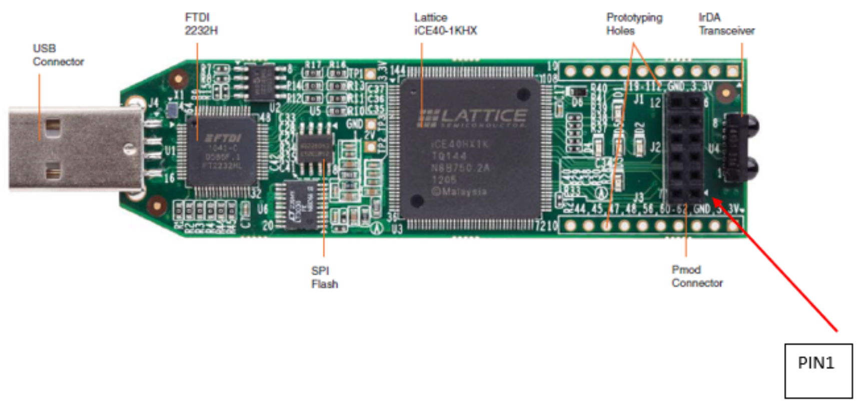
3. The FA can be expressed in gates. Write a logical expression in terms of AND, OR, and INV gates for each of the two outputs (sum, carry) for the FA. You may use gates with more than 2 inputs.
4. Given two FA gates, how would you compose them to perform a 2-bit addition (take in two 2-bit values and 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.

Lab Procedure:

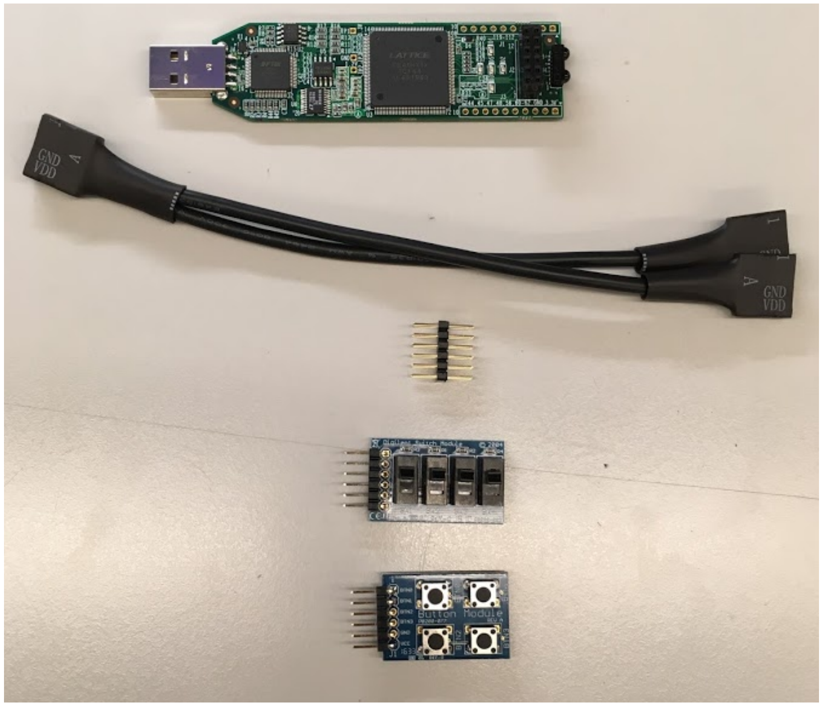
Lab – Section 1: Working with a USB FPGA

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

1. Obtain an iceStick USB FPGA, PMOD switches (2 kinds), and PMOD Y-cable.
2. Login to the Linux PC at your station using your SEAS account.
   i. Each station has both a Linux and a Windows PC. The A/B KVM switcher controls which machine is connected to the keyboard and screen. The switch should be set to Linux.
3. Bring up a terminal window.
   i. Click on icon in lower left
   ii. Look under System
   iii. Select X-terminal
4. In the terminal window, create a directory for your work for this lab
   i. `mkdir ese150logic`
   ii. `cd ese150logic`
5. Copy the files you will need for this lab into the directory you just created
   i. `cp ~ese150/logic/* .`
   Note that the above command ends with a dot (.);
   In Linux, dot refers to the current working directory, so this command is telling it to copy everything (the star (*)) from the ~ese150/logic directory into your working directory.
6. Make sure the shell script is executable
   i. `chmod +x build.sh`
7. Connect PMOD Switches up to the FPGA.
   - We will be using the following iceStick Lattice FPGA:

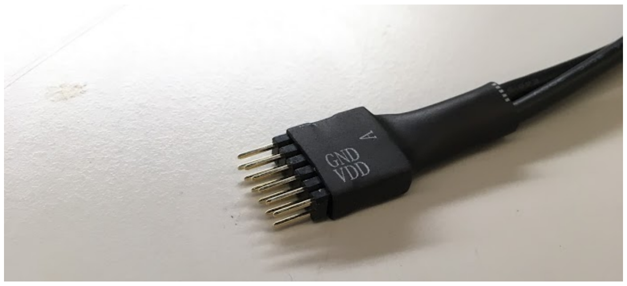


- The following parts will be needed for this lab (from top to bottom):
  - ice Stick FPGA
  - Pmod extension cable
  - Male to male headers
  - Pmod switches
  - Pmod buttons

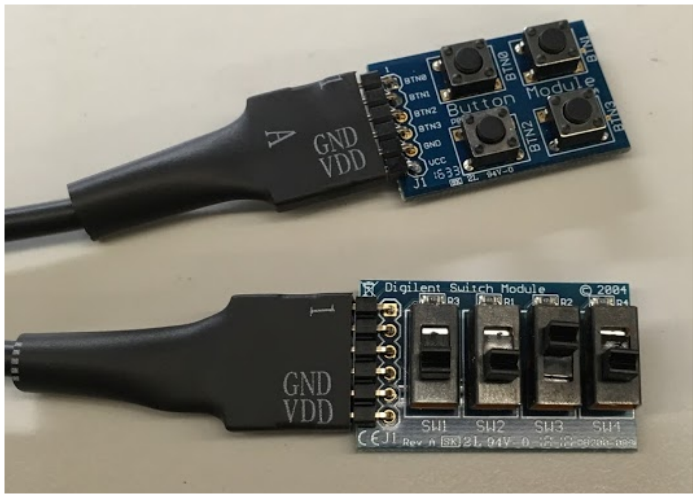


- As a side note, Pmod means “peripheral module” and refers to a standard of connections by Digilent, the company that makes the buttons and switches.

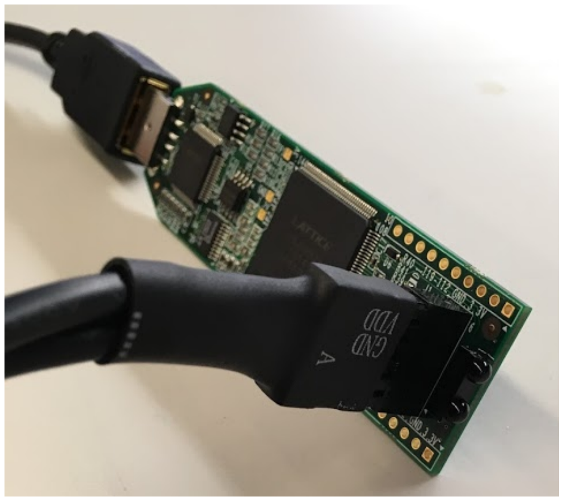
i. Connect the male to male headers to the end of the PMOD cable with 12 holes:



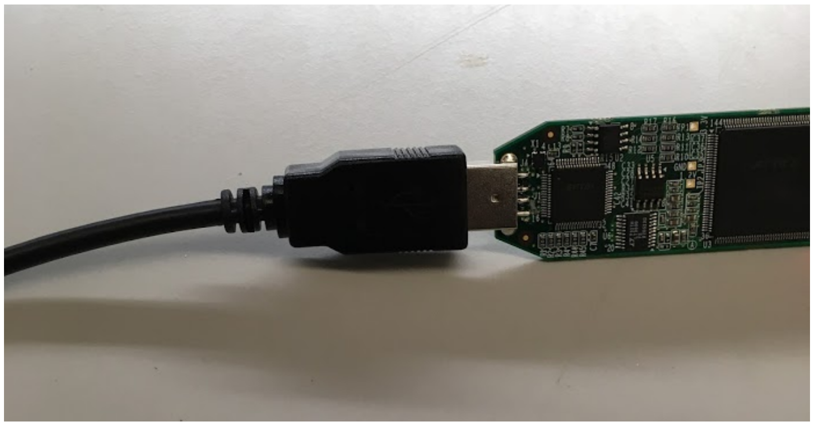
ii. Connect the Pmod buttons to the end of the Pmod extension cable that has 6 holes and is labeled A, and the Pmod switches to the other end. The buttons and switches should look as follows:



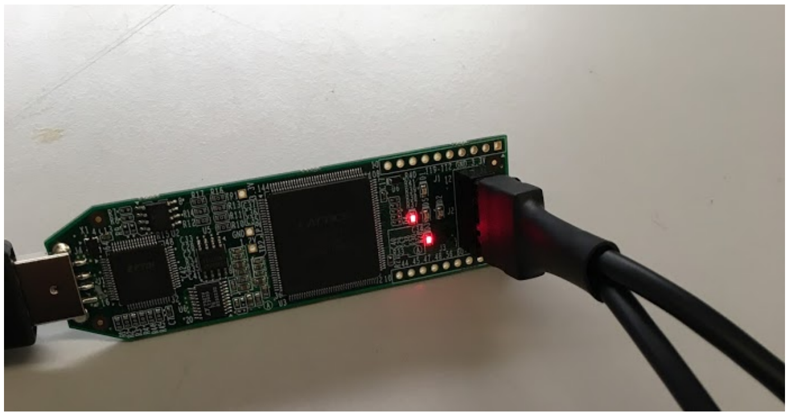
iii. Connect the Pmod extension cable with the male to male headers to the FPGA, such that the side labeled A faces **outward**:



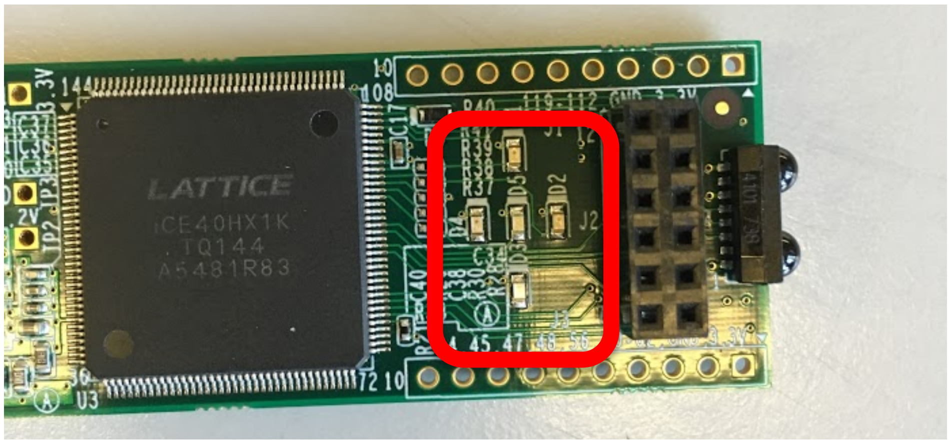
iv. Connect the USB extension cable to the FPGA. It should now look as follows:



v. Lastly, connect the FPGA to the Linux computer (the smaller of the two Dell machines) at your lab station. Now, some of the LEDs on your FPGA may turn on, if the FPGA was used and programmed in the past:



- There are five LEDs on the FPGA, labeled D1, D2, D3, D4, and D5:



8. 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     LED1,
    output     LED2,
    output     LED3,
    output     LED4,
    output     LED5,
    input      PMOD1, // input p1
    input      PMOD2, // input p2
    input      PMOD3, // input p3
    input      PMOD4,  // input p4
    input      PMOD7, // will use for section 2
    input      PMOD8, //
    input      PMOD9, //
    input      PMOD10  //
    );
```

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=PMOD1;
   assign p2=PMOD2;
   assign p3=PMOD3;
   assign p4=PMOD4;
```

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. 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 LED1 = o1;
     assign LED2 = o2;
     assign LED3 = o3;
     assign LED4 = o4;
     assign LED5 = o5;
```

9. Compile and download the section1.v Verilog file to the FPGA:
   i. Working in the same terminal window and directory where you just copied the files run the commands

      ```
      ./build.sh section1
      ```

   ii. 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.

10. Review the output of the compilation process and note the resources uses. In particular, look at the `section1.log` file that was created during the build process. You can open this in a text editor or use a terminal command:

```
more section1.log
```

Look for the following section:

```
After packing:
IOs           14 / 96
GBs           0 / 8
  GB_IOs      0 / 8
LCs           4 / 1280
  DFF         0
  CARRY       0
  CARRY, DFF  0
  DFF PASS    0
  CARRY PASS  0
BRAMs         0 / 16
WARMBOOTs     0 / 1
PLLs          0 / 1
```

This says we are using 4 LCs (Logic Cells) out of 1280 and 14 IOs out of 96. 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.

11. 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: 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. Copy section1.v to section2fa.v:

```
cp section1.v section2fa.v
```

2. Edit section2fa.v:
    a. If you are familiar with linux, you can bring up your favorite text editor.
    b. Alternately, through the menu accessed from the icon on the lower left, under Utilities, you can find "Text Editor".
3. Change the Verilog logic equations in section2fa.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 PMOD1, PMOD2, and PMOD3.
    b. Write your logic equations for sum and carry inside the always block in place of the logic that was in Section 1.
    c. Connect the output sum to LED1, output carry to LED2.
4. Compile and download your section2fa.v.
   `./build.sh section2fa`
5. Use the inputs and LEDs to verify the truth table for your full adder in section2fa.v.
    a. Debug your logic as necessary
6. Edit section2add4.v.
7. Revise the Verilog logic equations in section2add4.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;
```

We assigned a and b to the PMOD inputs for you.

```
// assign inputs to signals with meaningful names
   assign a[0]=PMOD1;
   assign a[1]=PMOD2;
   assign a[2]=PMOD3;
   assign a[3]=PMOD4;

   assign b[0]=PMOD7;
   assign b[1]=PMOD8;
   assign b[2]=PMOD9;
   assign b[3]=PMOD10;
```

Note that we can use the array notation to refer to individual bits in the a and b variables.

Also, note that b[0]=PMOD7 is physically SW1 on the Digilent Switch Module. Likewise, b[3]=PMOD10 is SW4.

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.

8. Compile and download section2add4.v to your FPGA.

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

10. 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.

<u>Lab – Section 3: Working with Verilog Arithmetic</u>

- 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. Compile and download the section3add4.v Verilog file to the FPGA
   a. Note the inputs are the same as the end of Section 2.
   b. Record resources required 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 write simple sequential logic and FSMs in Verilog

We can also write logic that includes state in registers, including FSMs in Verilog.

1. 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 12MHz clock
   reg [24:0] counter;
   reg [1:0] dec_cntr;
```

   The clock on the iceStick board runs at 12MHz. Unfortunately, if the LEDs changed at 12MHz, 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 5 million between each of the sequential logic operations. Each time the clock counter reaches 5 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.

```
// The 12MHz clock is too fast
// ...count to 6 million to divide it down to a half second
clock
   always@(posedge clk)
     begin
        counter <= counter + 1;
        if (counter == 6000000)
          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 activiated on a
different value of 2b half-second counter
   assign LED1 = (dec_cntr == 0) ;
   assign LED2 = (dec_cntr == 1) ;
   assign LED3 = (dec_cntr == 2) ;
   assign LED4 = (dec_cntr == 3) ;
```

3. Compile and download the section4fwd.v Verilog file to the FPGA
    a. Record resources required and explain them.
        i. What resources are needed to drive the LEDs from the dec_cntr?
        ii. What resources are needed for the counters? (logic compiler is being clever and may be trimming a bit from your expectation; state your expectation and what it is actually using.)
    b. Observe the LED clockwise rotation pattern
4. Copy section4fwd.v to section4bkwd.v
5. Modify the logic in section4bkwd.v to change the LED pattern to counter-clockwise rotation.
6. Compile and download the section4bkwd.v Verilog file to the FPGA
    a. Record resources required and explain them.
    b. Observe the LED rotation pattern.

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. Since we only have 5 LED outputs on our iceStick USB FPGA, we will need to share them between the low 4b of the accumulator value and the top 4b of the accumulator value.

Our complete set of inputs will be:

- 4b of input – use the 4 on-off switches (Digilent switch module, PMOD7 through PMOD10, SW1 through SW4); we call these in[3:0].
- Reset – to set the accumulator value back to 0; use a momentary switch (Button Module, PMOD1, 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, PMOD2, BTN1), which we will call p_input.
- Show high nibble – to tell the FPGA to display the top bits (bits 7—4 of the 8b accumulator value) on the LEDs. When this is set low, the LEDs should show the bottom bits (bits 3—0) ; use a momentary switch (Button Module, PMOD3, BTN2), which we will call p_high.

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 LED1) within an "always" block, don't write "assign" before the output name. For example, instead of writing "assign LED1 = accum[0]", simply write "LED1 = accum[0]".

1. Copy section5start.v to section5acc.v
2. Revise section5acc.v to behave as an accumulator as described above.
    a. Add your accumulator logic along with the counter reset as noted.
    b. Add your output select logic in the LED output section at the end as noted.
3. 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.
4. Record the LC resources needed by your design.
5. Show your accumulator to your TA for your exit ticket.
    a. TA will perform a test of a different sequence of numbers.
    b. TA will review Verilog code.
    c. TA will ask questions about the design.
6. Return your iceStick USB FPGA and PMODs and cleanup your workstation before leaving.

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)
2. Based on LC usage, how many 16-bit adders could you put on the FPGA used on the iceStick? (recall the FPGA has 1280 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).
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 iceStick 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 parts 3—5.
7. What is the minimum size part ice40 part you could use to implement this design?
   a. You may want to refer to the data sheet http://latticesemi.com/view_document?document_id=49312

HOW TO TURN IN THE LAB

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