

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 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

Prelab: Part 1

Submit your answers to this part to the Lab 7 prelab assignment on Canvas before Wednesday's 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", "p2", and "p3" are all inputs.
 - a. $\text{Out} = \text{NOT}(\text{AND}(p1, \text{NOT}(p2)))$
 - b. $\text{Out} = \text{OR}(\text{AND}(p1, p2), \text{NOT}(p3))$

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. There are 3 inputs because two of them are the bits we want to add, and one of them is the carry-in bit.

Mathematically: $i_0 + i_1 + i_2 = 2 \cdot \text{carry} + \text{sum}$

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**.

We can also use these simple rules to calculate the carry and sum bits for 3 inputs:

- An odd number of 1s in the input will result in a sum of 1. An even number of 1s will result in a sum of 0.
- If the inputs add up to a number greater than 1, the carry bit will be 1 (because it "spills over"). Otherwise, the carry bit will be 0.

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.

ESE 150 – Lab 07: Digital Logic

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.

Prelab: Part 2

1. Sign up for GitHub. Go to <https://github.com/> and choose “Sign up for GitHub”.
2. Create a GitHub account if you do not already have one

Join GitHub

Create your account

Username *

 ✓

Email address *

 ✓

Password *


 ✓

Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter.
[Learn more.](#)

Email preferences

Send me occasional product updates, announcements, and offers.

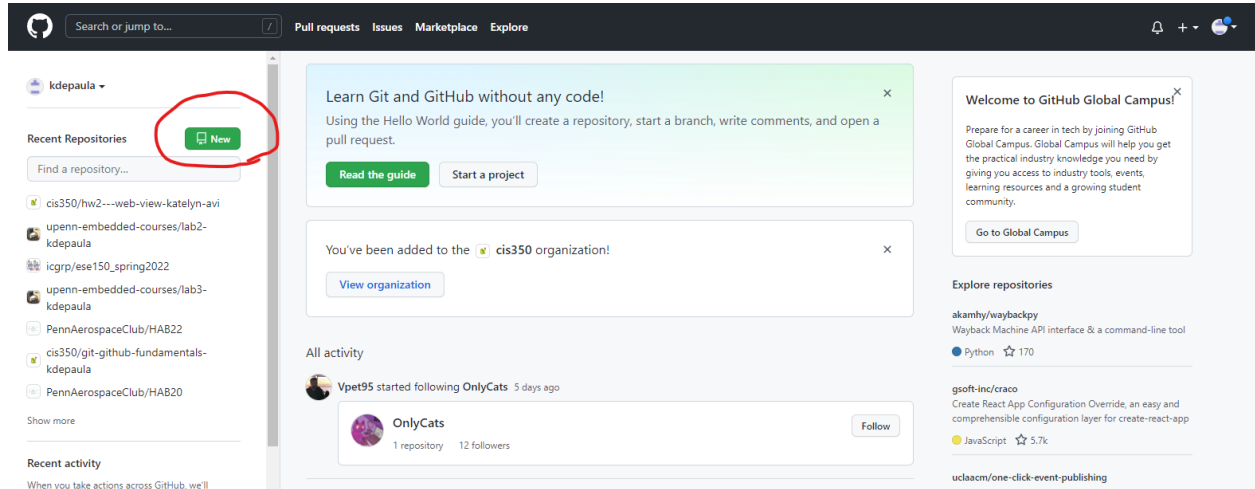
Verify your account



[Create account](#)

By creating an account, you agree to the [Terms of Service](#). For more information about GitHub's privacy practices, see the [GitHub Privacy Statement](#). We'll occasionally send you account-related emails.

3. On the main home page, click “New” under Recent Repositories



4. Create a GitHub repository. Make sure you have it set to private

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner * Repository name *

/

Great repository names are short and memorable. Need inspiration? How about [legendary-octo-parakeet?](#)

Description (optional)

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

Add a README file
This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore
Choose which files not to track from a list of templates. [Learn more.](#)

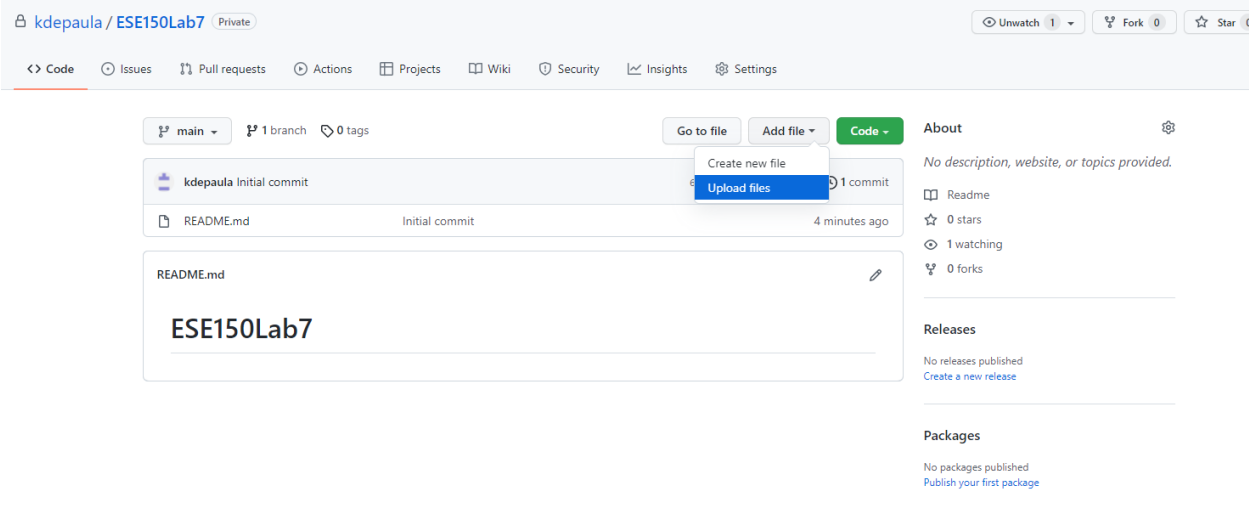
Choose a license
A license tells others what they can and can't do with your code. [Learn more.](#)

This will set `main` as the default branch. Change the default name in your [settings](#).

5. At the end of this lab, you will upload all of your code to GitHub so that you can access it from any device. To upload a file choose “Add File” within your repository (picture

ESE 150 – Lab 07: Digital Logic

below). Then you can use the Windows File Explorer to drag all of your files into the GitHub repository.

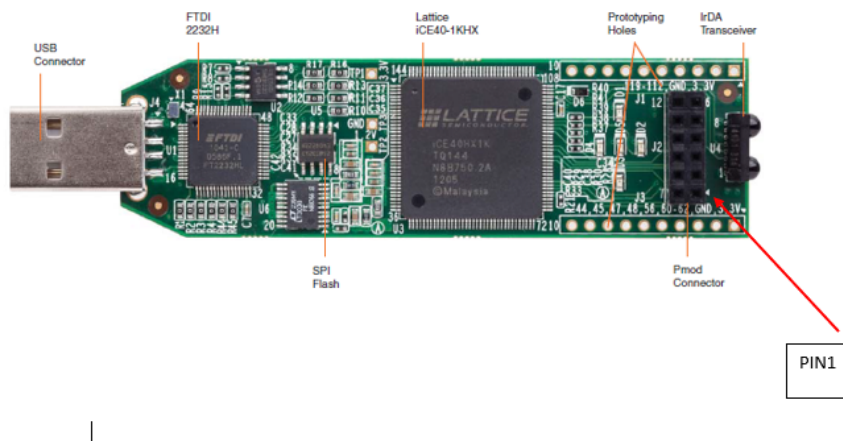


Lab Procedure:

Lab – Section 1: Working with a USB FPGA

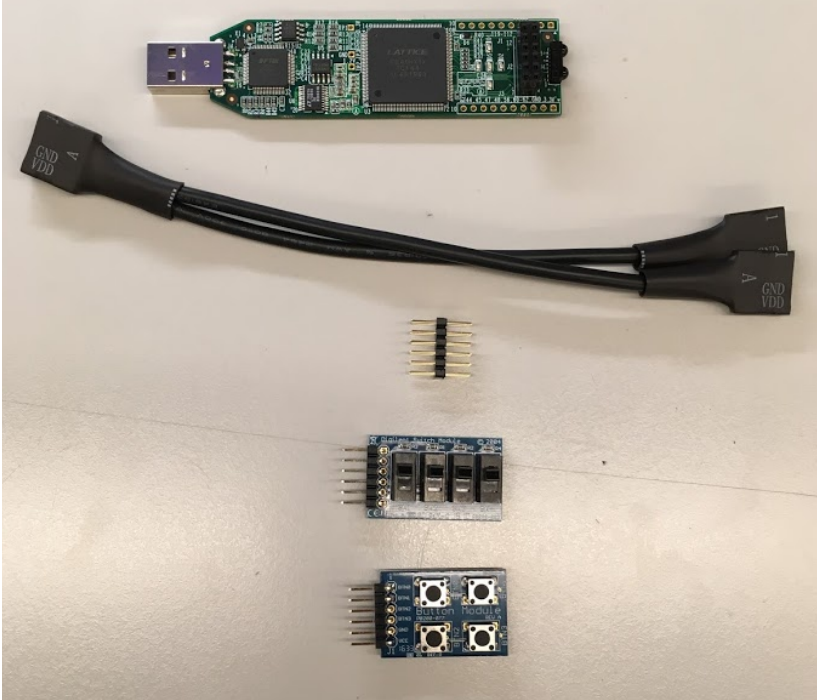
- In this section, you'll learn how to compile simple combinational logic in Verilog for an FPGA
1. Your Lab kit should include PMOD switches (2 kinds) and a PMOD Y-cable.
 2. You will need to get an an iceStick USB FPGA and a USB extension cable from the TAs to use in lab.
 3. Connect PMOD Switches 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

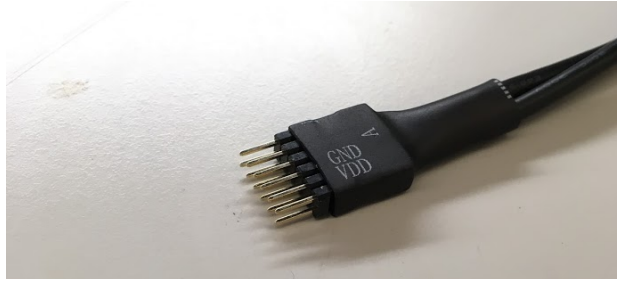
ESE 150 – Lab 07: Digital Logic



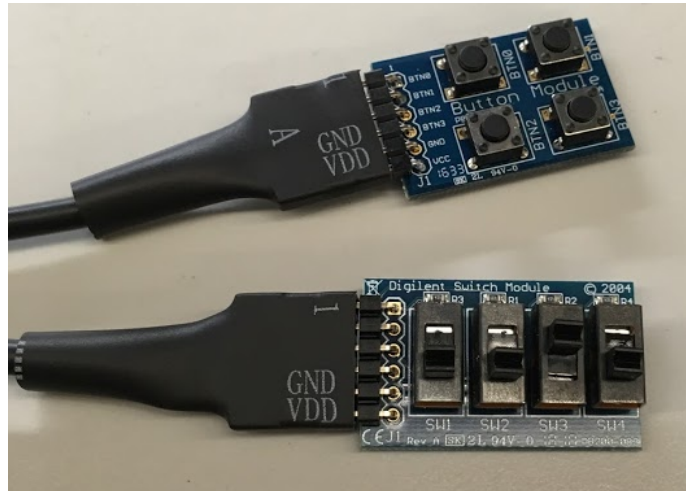
- As a side note, Pmod means “peripheral module” and refers to a standard of connections by Digilent, the company that makes the buttons and switches.

ESE 150 – Lab 07: Digital Logic

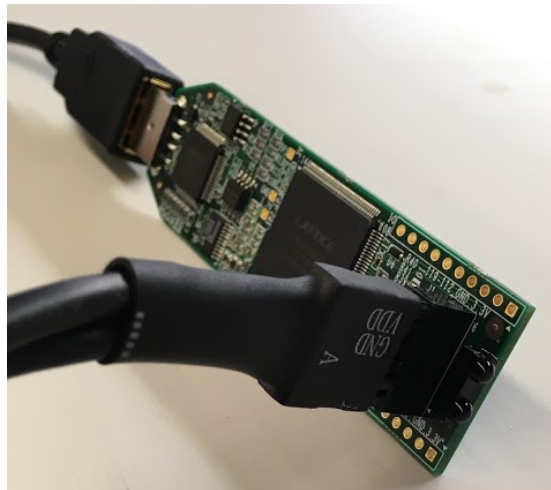
- i. Connect the male to male headers to the end of the PMOD cable with 12 holes:



- 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**:





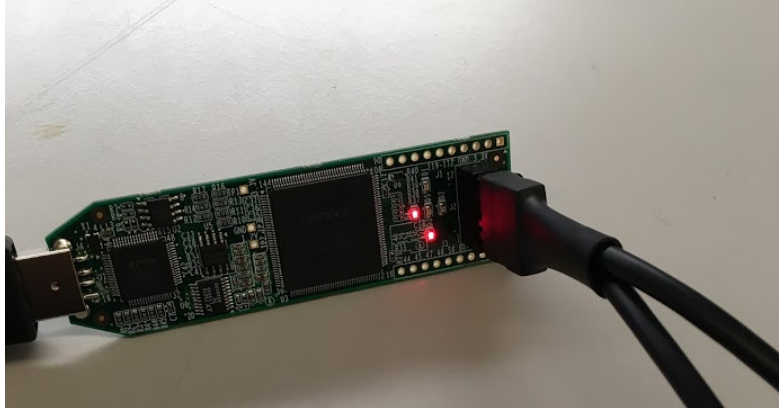
iv. Connect the USB extension cable to the FPGA. It should now look as follows:



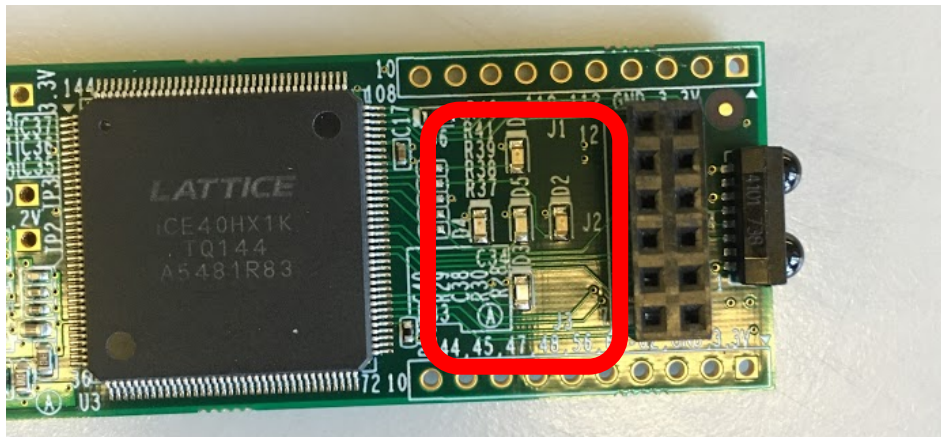
- v. Lastly, connect the FPGA to the Windows computer (the larger of the two Dell machines) at your lab station. Plug into the leftmost USB port on the main computer box.



- vi. 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:



4. In a terminal window, create a directory for your work for this lab by entering the following commands into the terminal. Use Windows PowerShell. (To find it, press the windows button and search "PowerShell"). Run the following commands in C:/Users/yourpenkey
 - 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 (ese150logic.zip) in the ese150logic folder that you will need for this lab from the link on the course syllabus.
6. Review the Verilog file top.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
reg   o1;
reg   o2;
reg   o3;
reg   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;

```

ESE 150 – Lab 07: Digital Logic

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 – update on any
change
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;
```

7. Change to the section1 directory that was created under ese150logic when you unpacked ese150logic.zip:

```
cd section1
```
8. Record the output of the following commands. Use Windows PowerShell to run the commands. In your section1 directory run:

```
apio clean
```

```
apio build –verbose
```

```
apio upload
```

**note there are two hyphens before verbose*

ESE 150 – Lab 07: Digital Logic

Scroll back and look for the following section:

```
Info: Device utilisation:
Info:         ICESTORM_LC:      6/ 1280    0%
Info:         ICESTORM_RAM:     0/   16    0%
Info:         SB_IO:           14/  112   12%
Info:         SB_GB:            0/    8    0%
Info:         ICESTORM_PLL:     0/    1    0%
Info:         SB_WARMBOOT:      0/    1    0%
```

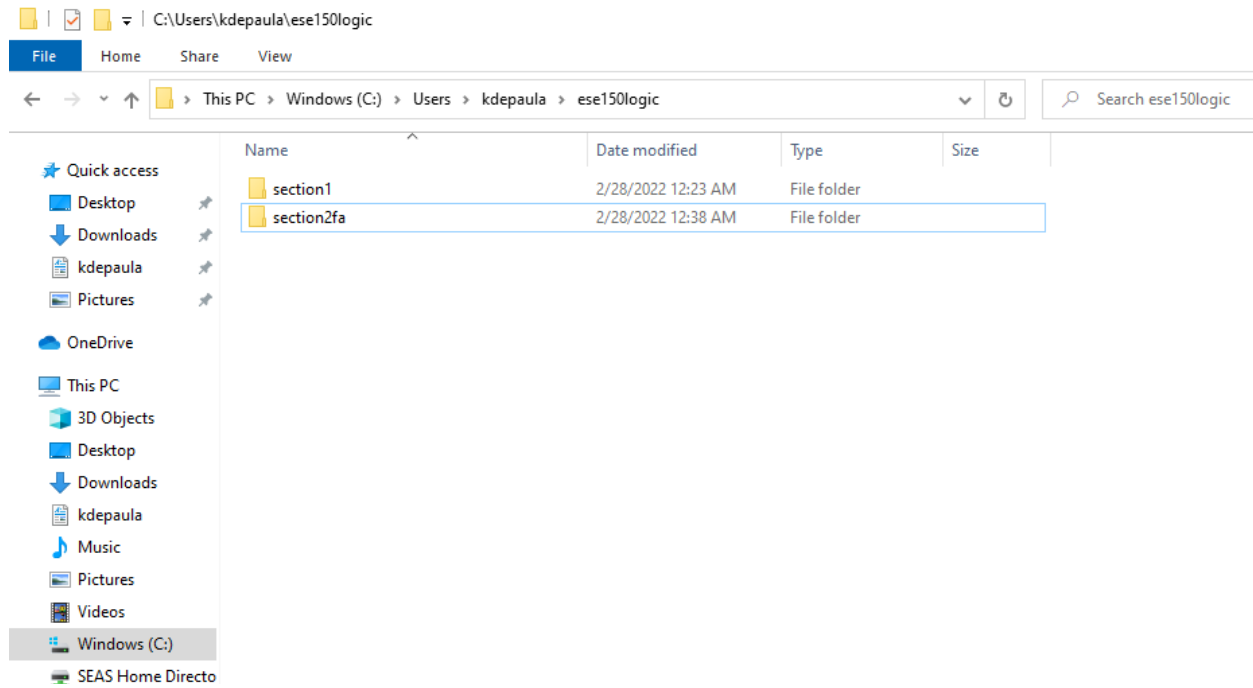
This says we are using 6 LCs (Logic Cells) out of 1280 and 14 IOs out of 112. The 4 LCs are for each of the 4 expressions we compute. None of them have more than 4 inputs, so they can each fit into a single LC. It may use another LC for the direct output.

9. Include this output of the Device utilisation in your writeup (for this and each of your following designs).
10. 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. Open the Windows File Explorer using the windows key and searching “File Explorer”. Go to C:/Users/yourpennkey/ese150logic. Copy the “section1” folder and paste it into ese150logic. Rename the folder “section2fa”



2. Open section2fa/top.v in Atom. 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. Use wire variables for p1, p2, p3 which are assigned 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 top.v
 - c. Connect the output sum to LED1 (o1), output carry to LED2 (o2).
3. Open you section2fa design, compile and upload your section2fa/top.v.
In your section2fa directory run:

```
apio clean
apio build -verbose
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 using the command line output. Scroll back and look for the “Device utilisation” report.

Lab – Section 2.2: Building a 4-Bit Adder

1. Copy and paste section1 again using the windows file explorer and rename it to section2add4
 - a. Copy over files from section1
 - b. Copy section2add4.v to section2add4/top.v (you can open top.v in notepad and copy and paste)
2. 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
reg [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.
3. Compile and upload section2add4/top.v to your FPGA.

4. Consult the output of the compilation process and note how many LCs your 4-bit adder uses.
5. 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 section2add4.v.

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

2. Copy and paste section1 files and rename that directory to “section3”, copy section3add4.v to section3add4/top.v.

In your section3 directory run:

```
apio clean
apio build --verbose
apio upload
```

- a. Note the inputs are the same as the end of Section 2.
- b. 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 won't have to submit anything for this section, but it should give you the information you need to complete Section 5.

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

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 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 6 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 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
end
```

ESE 150 – Lab 07: Digital Logic

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 LED1 = (dec_cntr == 0) ;
    assign LED2 = (dec_cntr == 1) ;
    assign LED3 = (dec_cntr == 2) ;
    assign LED4 = (dec_cntr == 3) ;
```

3. Create a directory for section4fwd, copy over section1 files, copy section4fwd.v to section4fwd/top.v, and compile and upload to the FPGA:
 - a. Watch how lights behave and relate to logic.
4. There's nothing to submit for this section, but you'll 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 $2^4-1=15$. 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]”.

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 LED assignments are made.

ESE 150 – Lab 07: Digital Logic

1. Create a directory for section 5, copy over section1 files, copy section5start.v to section5/top.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 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.
6. Return the USB FPGA and USB extension cable.
7. Add all of the Verilog files you created to the github repository you created. Test you can retrieve them from the github repository using your laptop.

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 iCEstick FPGA? (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:

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

$B[i]$ 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 an FPGA with 7680 LCs? (a larger version of the FPGA used in lab.)
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)?

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

ESE 150 – Lab 07: Digital Logic

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 the 7680 LC FPGA? If not, what is the largest dot product (number of 16-bit inputs) that will fit on the 7680 LC FPGA?

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
 - Prelab answers (refined, if necessary, from the version you submitted to pre-lab quiz)
- Each student must submit an individual lab writeup.