ESE 150 – Lab 07: Digital Logic

LAB 07

In this lab we will do the following:

1. Investigate basic logic operations (AND, OR, INV, XOR)
2. Implement an ADDER on an FPGA
3. Implement a simple Finite-State Machine on an FPGA

Background:

In lecture we discussed the 3 basic logic operations: AND, OR, NOT (inversion). We examined each operation and learned that the 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 functions and Finite-State Machines.

ESE 150 – Lab 07: Digital Logic

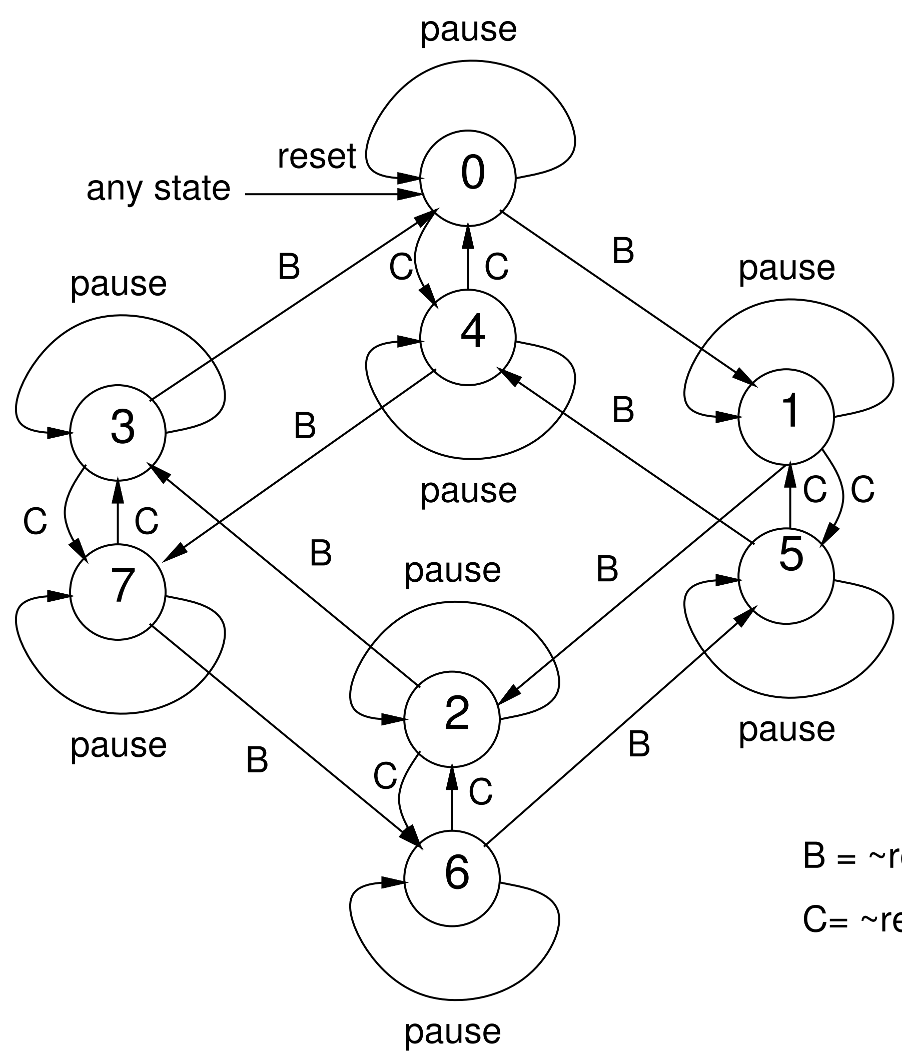
Prelab

1. Write the truth table for each of the following functions.
    a. Out = NOT(AND(p1,NOT(p2))
    b. Out = OR(AND(p1,p2),NOT(p3))
2. Write a logic expression using AND, NOR, and NOT for the following truth table for a full adder with (a, b, c) as inputs and (sum, carry) as outputs.

a	b	c	sum	carry
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

3. Write a function using if-then statements and state assignment for the FSM shown in the diagram below (next page).

    This FSM mimics a simplified remote controller with reset, pause, and rev (reverse) control. The LED's cycle 1->2->3->4->1 for forward and 4->3->2->1->4 for backward. Reset always goes back to state 0. Pause will remember the state and direction traveling. Reverse (rev) inverts the flow directions. LED1 should be on in states 0, 4; LED2 in states 1, 5; LED3 in states 2, 6; LED4 in states 3, 7.



B = ~reset & ~rev & ~pause

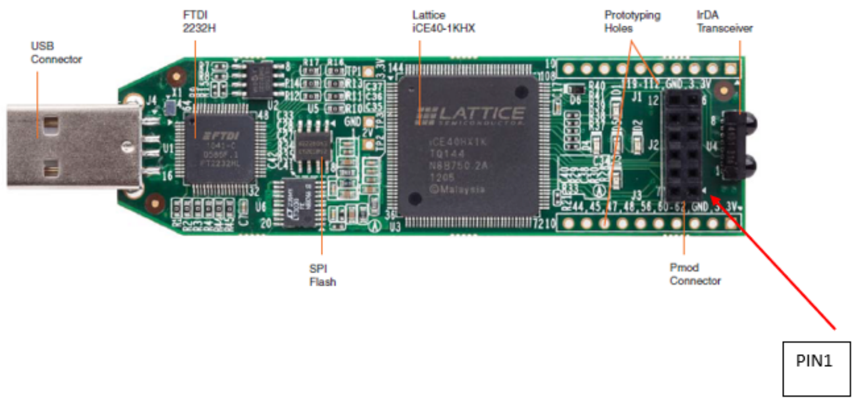
C= ~reset & rev & ~pause

Lab Procedure:

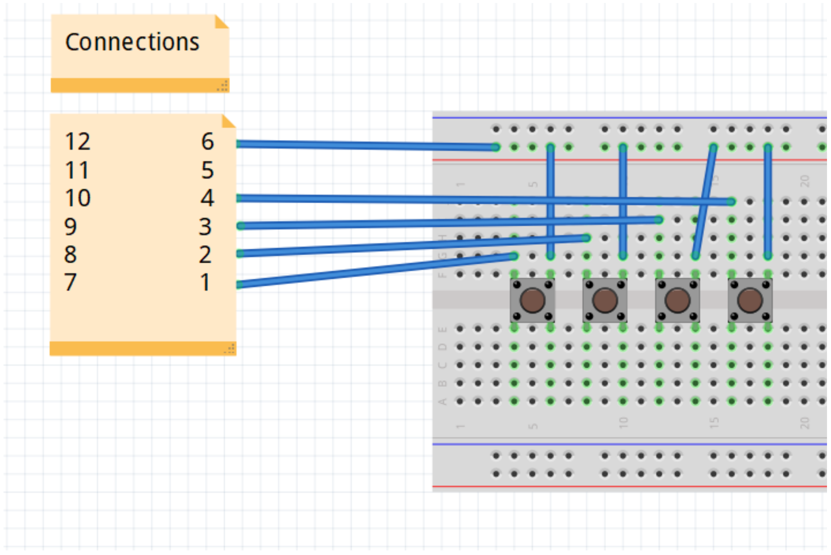
Lab – Section 1: Working with a USB FPGA

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

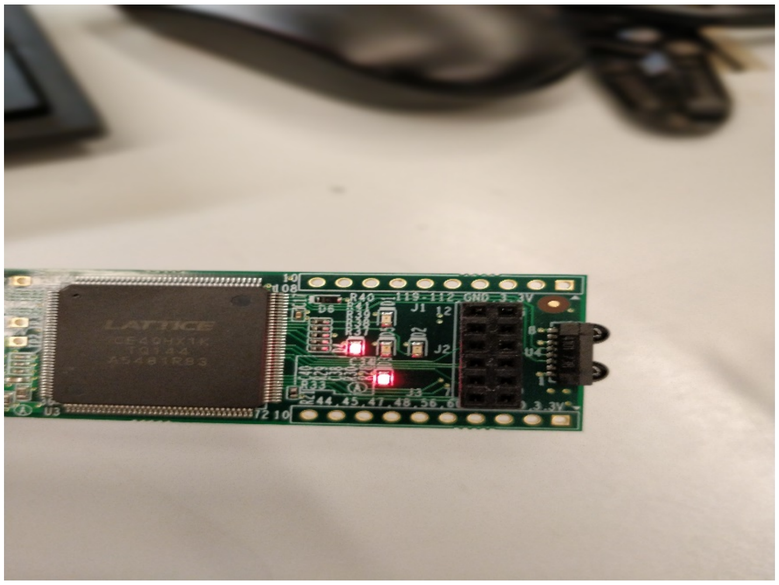
1. Obtain an iceStick USB FPGA
2. Login to the Linux PC at your station using your SEAS account.
    a. Each station has both a Linux and a Windows PC. The A/B KVM switcher controls which machine is connected to the keyboard and screen. The switch should be set to linux.
3. Bring up a terminal window.
    a. Click on icon in lower left
    b. Look under System
    c. Select X-terminal
4. In the terminal window, create a directory for your work for this lab
    a. `mkdir ese150lab7`
    b. `cd ese150lab7`
5. Copy the files you will need for this lab into the directory you just created
    a. `cp ~ese150/lab7/* .`
6. Make sure the shell script is executable
    a. `chmod +x build.sh`
7. Wire switch inputs up to the FPGA.
    a. We will be using the following FPGA:



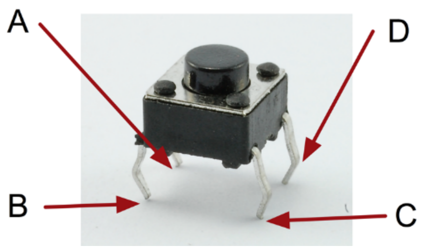
    b. The circuits connections are shown below:

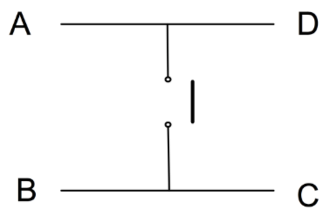


c. These are how the LEDs look, and they are denoted by D1,D2,D3,D4.



d. The push button should be connected such that (A & B) and (C & D) are not shorted.





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. They key outputs are the LEDs and the inputs

on the PMOD connector, which you wired in the previous step. Also included is a clock signal (clk), which we will be using for the sequential logic in later sections.

```
`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. In later sections, we’ll see another type for used for state.

```
// 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;
```

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 learned in class.

```
     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 up 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.
    a. Working in the same terminal window and directory where you just copied the files run the commands

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

    b. 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.

Scroll back and look for the following section on the terminal output:

```
After packing:
IOs          10 / 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 10 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 fit into a single LC. Later we'll see the logic begin to use the flip-flops (DFF) and Carry logic (CARRY).

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.

<u>Lab – Section 2: Writing your own combinational logic</u>

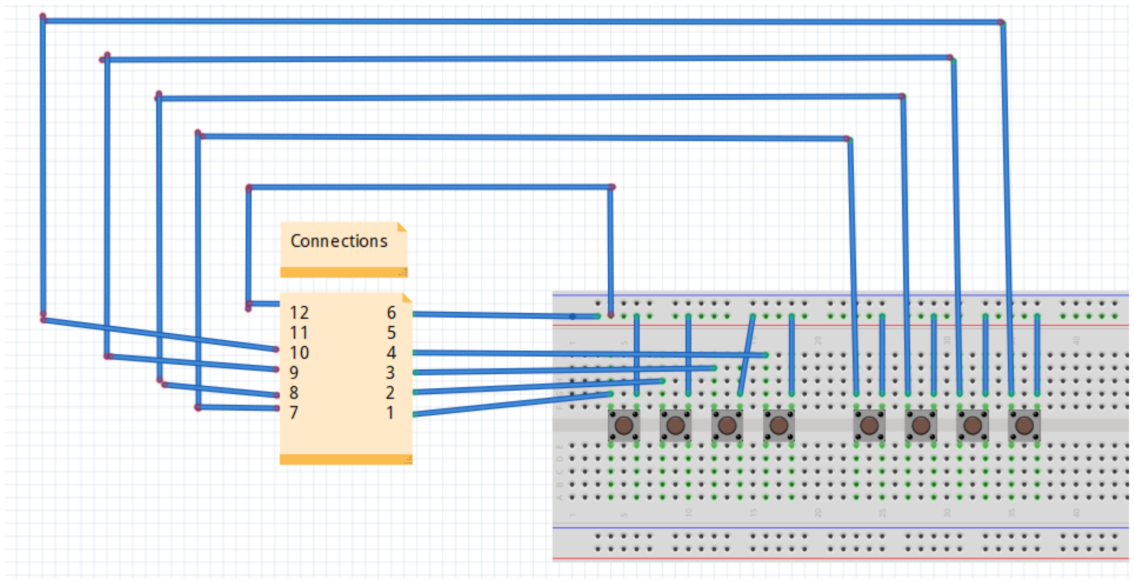
- In this section you'll learn how to write simple combinational logic in Verilog

1. Copy section1.v to 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 2.
   a. Declare wire variables for a, b, c and assign the inputs a, b, c 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.
   a. `./build.sh section2fa`
5. Record the resources required for your design.
   a. Include in report
   b. Explain the required resources (why does your logic map to these resources)
6. Use the inputs and LEDs to verify the truth table for your full adder in section2fa.v.
   a. Debug your logic as necessary
7. Edit section2add4.v.
8. 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, b, and c are each 4b values. o is a 5b value. Why do we need 5b for the output of a 4-bit adder?

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

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

We assigned a and b to the PMOD inputs for you. Following is the Circuit Diagram:



```
// 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.

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.

9. Compile and download section2add4.v
   a. Record resources required and explain them.
   b. Use the inputs and LEDs to verify the correct function of your 4-bit adder
      i. If we were to exhaustively test your adder, how many test cases (sets of input values) would there be?
      ii. 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
      iii. 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 pretty common in Verilog, so you can also write arithmetic expressions directly in Verilog.

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 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 operates 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:

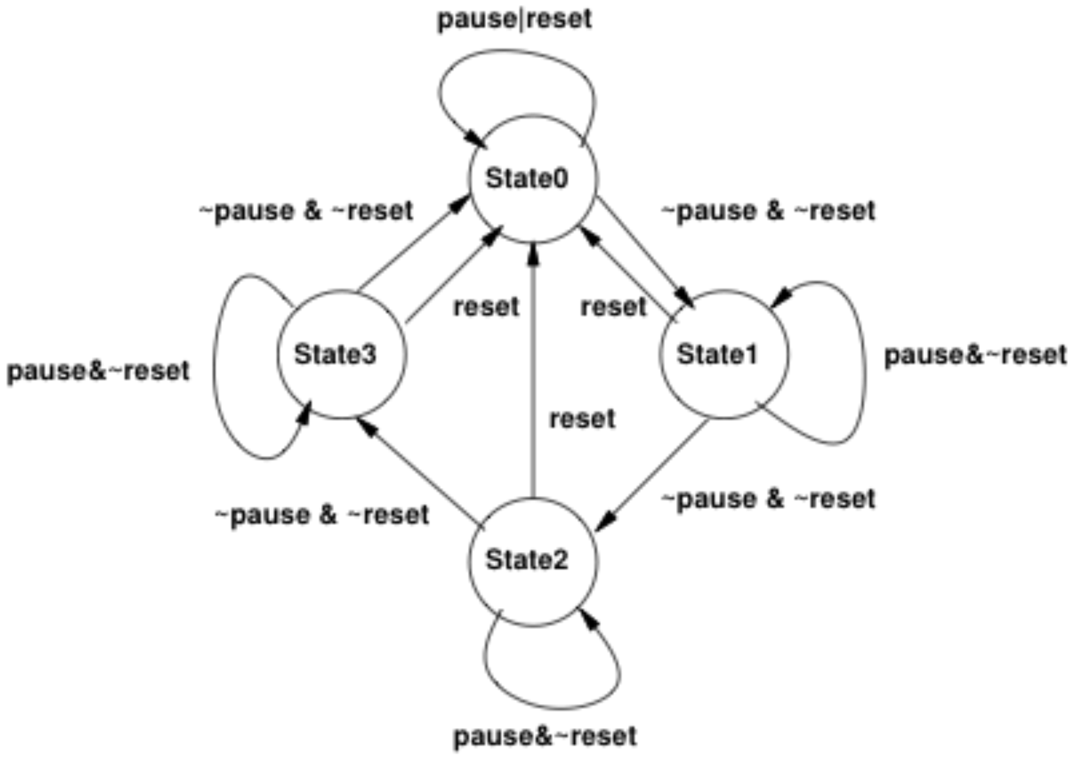
```
// Make the lights blink -- each light activiated on a different
value of 2b half-second counter
   assign LED1 = (dec_cntr == 0) ;
   assign LED2 = (dec_cntr == 1) ;
   assign LED3 = (dec_cntr == 2) ;
   assign LED4 = (dec_cntr == 3) ;
```

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 FSM in Verilog

- In this section you'll implement your own FSM logic in Verilog

1. Review the Verilog file section5ex.v to see how it encodes the simple state machine shown below with forward logic, a pause, and a reset.



The surrounding logic looks just like the previous section except that we've changed dec_cntr to a 2b state variable, state. What differs here is what we do on each half second (each time the counter hits 6 million):

```
          if (counter == 6000000)
            begin
             counter<=0; // reset counter
             if (reset == 1)
               state <= 0;
             else
             case(state) // state machine logic here
               0: begin
                    if (pause)
                         state<=0;
                    else
                      state<=1;
                  end
```

```
                1: begin
                     if (pause)
                        state<=1;
                     else
                        state<=2;
                   end
                2: begin
                     if (pause)
                        state<=2;
                     else
                        state<=3;
                   end
                3: begin
                     if (pause)
                         state<=3;
                     else
                        state<=0;
                   end
                 default: state<=0;
                endcase
               end
```

The logic first encodes the reset condition that sets the state to 0 regardless of what state the FSM is in. We handle the logic for each state separately using a case statement on state. The case statement is similar to the switch statement in C. Inside each case, we use if-else logic to write the logic for the state. We use assignment to state to control which state the machine transitions to as a result of the input (in this case the pause signal).

2. Compile and download the section5ex.v Verilog file to the FPGA
    a. Record resources required and explain them
        i. Here, you know what logic was required for the LED combinational logic and the 6 million cycle counter, so you can subtract those out to identify the resources required for the state machine logic.
    b. Validate operation (hold reset for 0.5s to get started)
3. Copy section5ex.v to section5fsm.v
4. Revise section5fsm.v to implement the FSM from prelab 3
    a. Inputs: reset, pause, rev
    b. Outputs: LED1, LED2, LED3, LED4
5. Compile and download the section5fsm.v Verilog file to the FPGA
    a. Record resources required and explain them.
    b. Validate operation

6. Demonstrate your FSM to your TA, show your code, and answer a few questions for exit checkoff.
7. Return your iceStick USB card and switches and cleanup.
8. Make sure to log out
    a. Icon lower left
    b. Under Power/Session
    c. Select logout

Postlab

1. How many resources will be required for a two input, 16-bit adder (adds together two 16b inputs to produce one 17b output)
2. How many 16-bit adders could you put on the FPGA used on the iceStick?
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)
4. What other logic do you need besides adders for the multiplier? How many FPGA resources will this require?
5. How many of these combinational 16-bit multipliers can you place on the FPGA used on the iceStick?
6. How many resources 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)?
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
    - Postlab answers
- Each student must submit an individual lab writeup