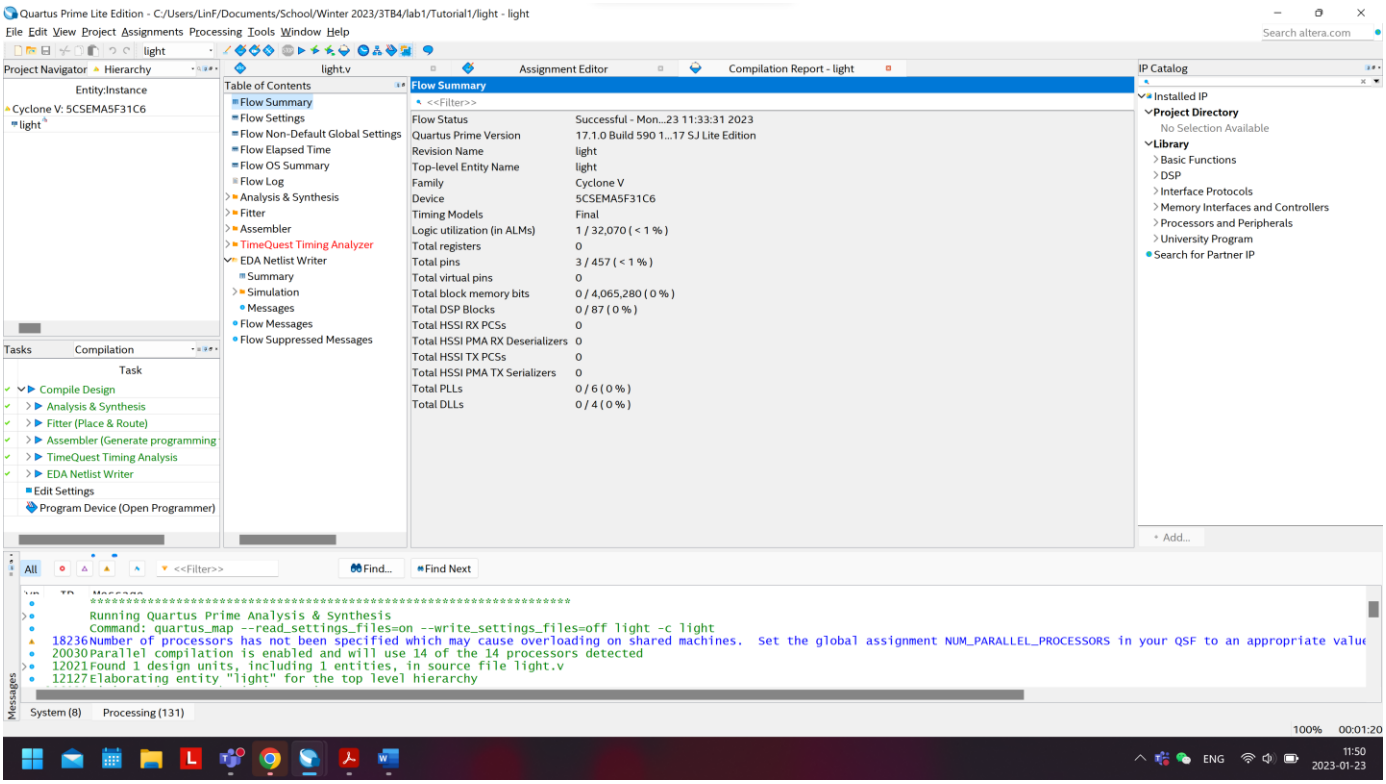
3TB4 Tutorial/Prelab 1 Report

Lin Fu ful10 400234794

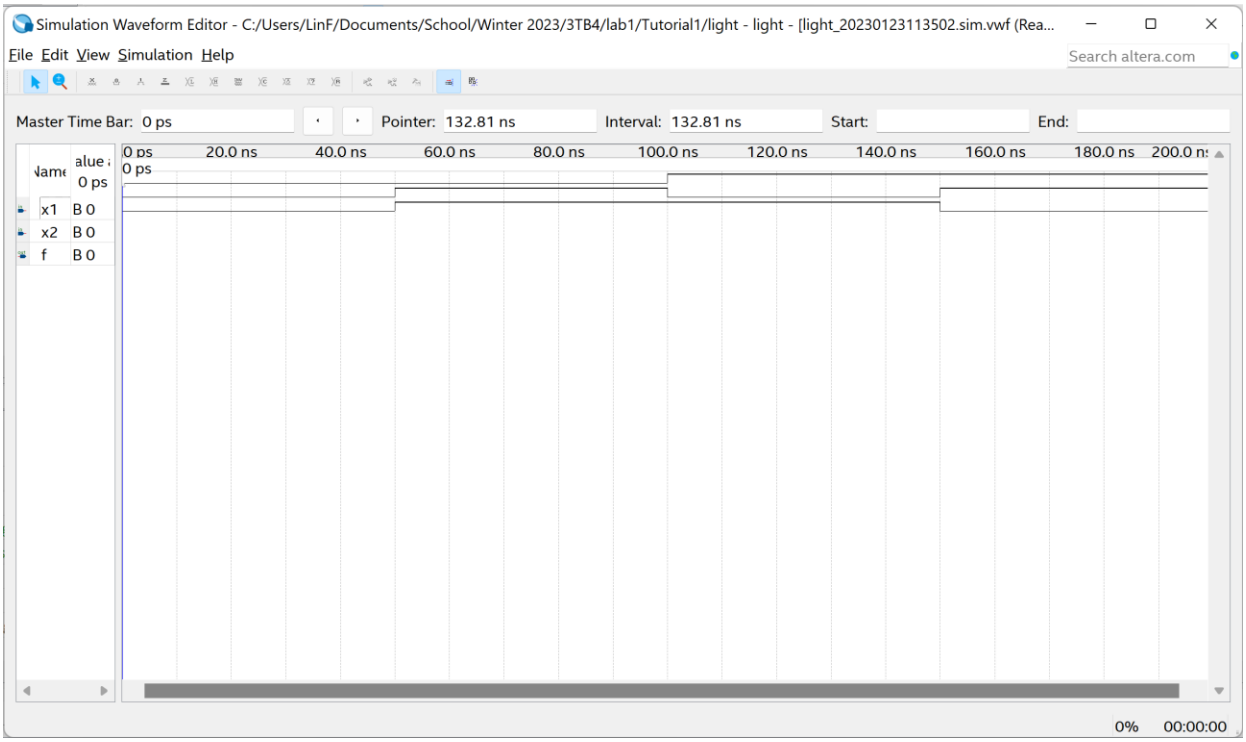
Keyin Liang liangk10 400236736

Quartus Prime Introduction Using Verilog Designs:

Complication Report:

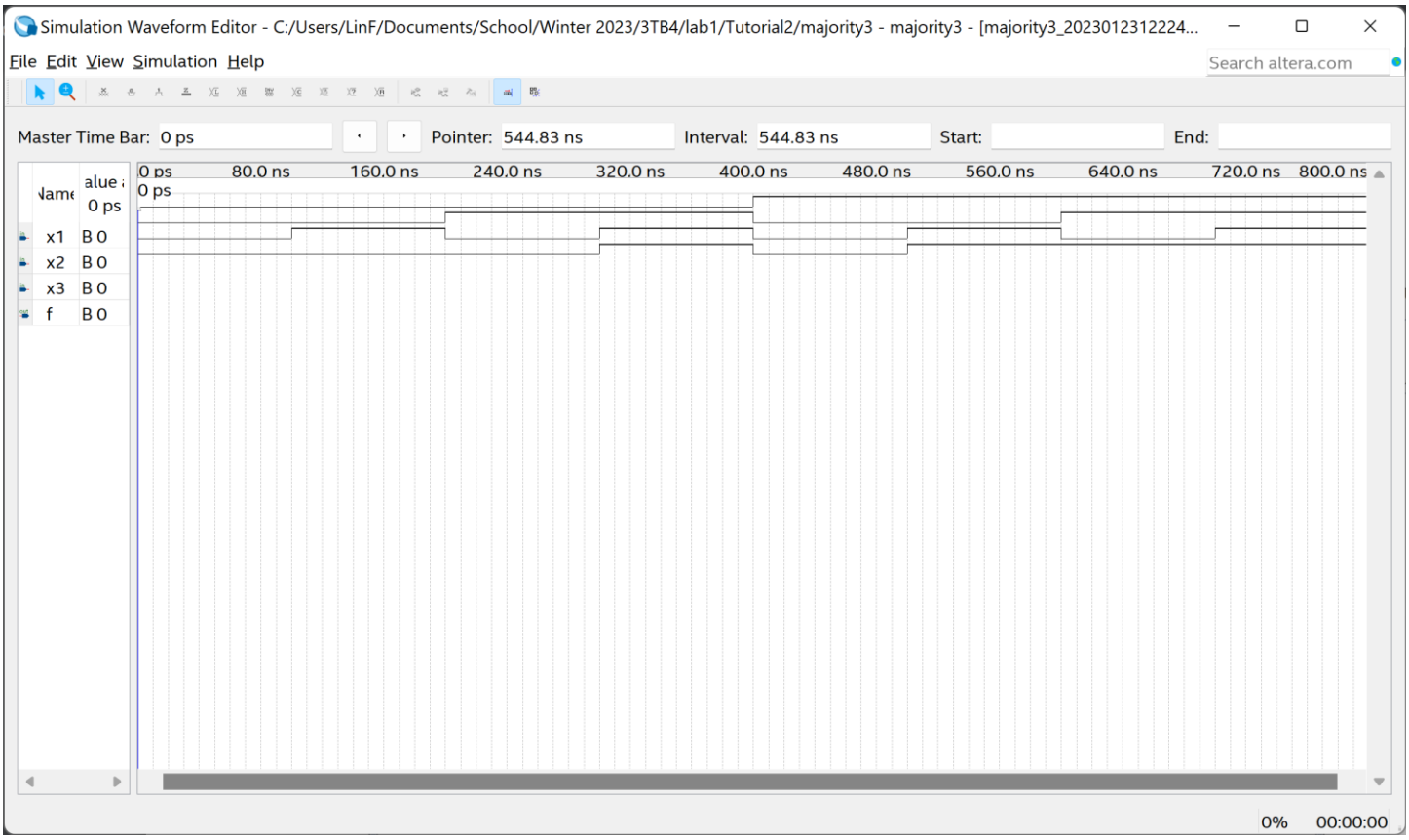


Simulation Report:

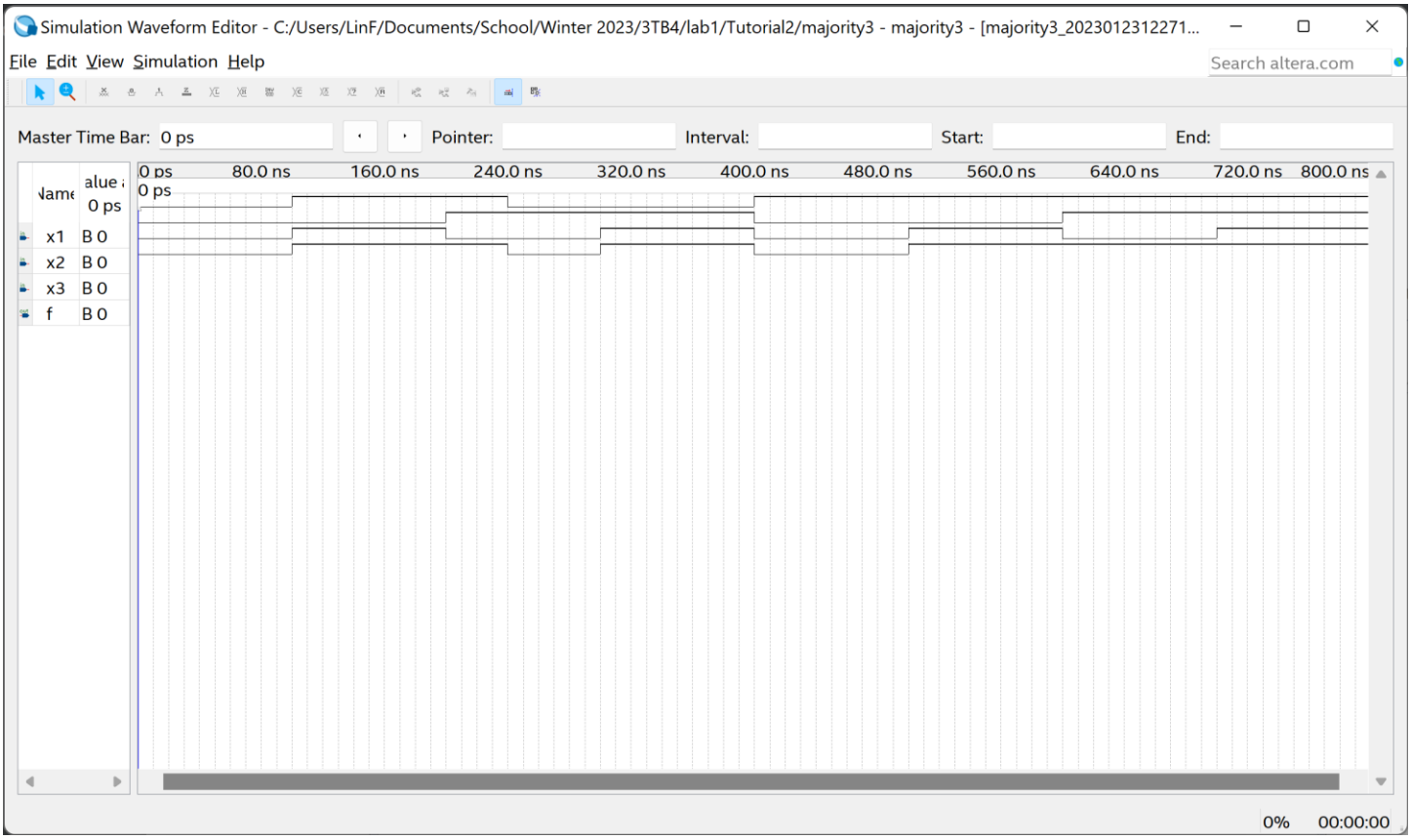


Introduction to Simulation of Verilog Designs:

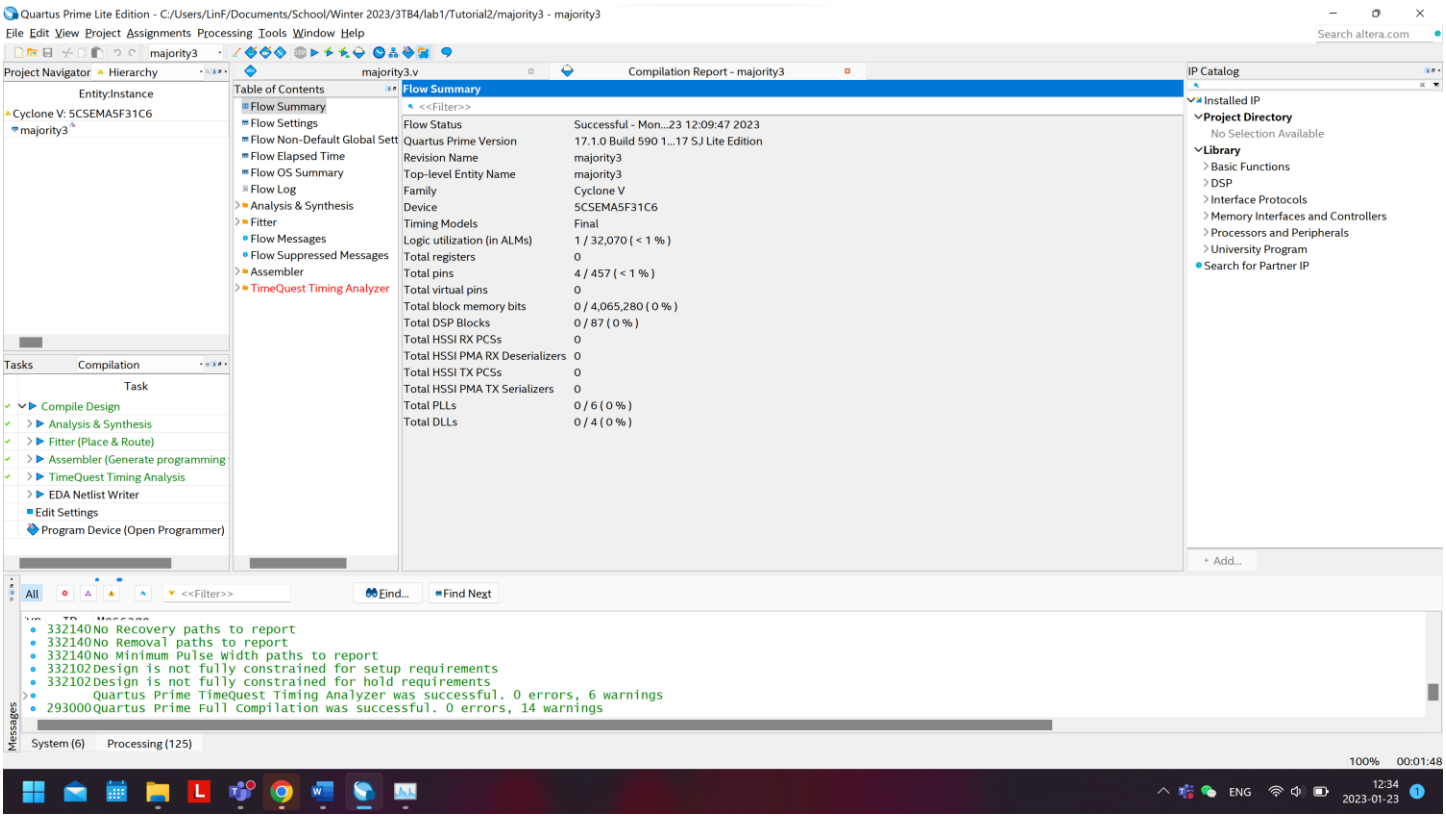
Simulation Report:



Simulation Report Modified:



Complication Report:



Prelab Questions:

1. What is the reg data type and what is the wire data type in Verilog?

The reg data type is the hardware registers that stores data, the wire data type describes the connection between elements or hardware entities.

2. Can the wire data type be used on the left side of the assignment statement in a procedural block?

No, the wire data type can only be used on the left side when outside of the procedural block (always block).

3. What are the rules for module port connection?

First, declare input, inout or output as: input/output/inout [range_val:range_var] list_of_identifiers; Inputs are type net and can be connected to a variable of type reg or net. Outputs are type net or reg and must be connected to a variable of type net. Inouts are type net and can only be connected to a variable net type.

4. What are continuous assignment, blocking assignment and nonblocking assignment?

Continuous assignment: Assign values to the nets (LHS) whenever there is a change on the RHS.

Blocking assignment: Blocking the next execution until the current statement is executed. The statements are executed in order.

Nonblocking assignments: Simultaneously execute statements.

5. What is the difference in procedural coding when implementing combinational logic and sequential logic?

Combinational logic does not require a clock to operate, whereas sequential logic does.

6. How does one avoid inferred latches when using Verilog to describe circuits?

To avoid this, make sure that all of the if statements that are used in combinational always blocks have one last else statement to catch all missing conditions.

Always make sure every variable in the procedural block gets assigned.

Use default assignments at the beginning of the procedure, so that every signal is assigned.

7. What is the difference between the operators “<<” and “<<<”?

“<<<” is the arithmetic left shift and “<<” is a logical shift. They both shift left specific bits while “<<” fills the vacated bits with 0, “<<<” fills the vacated bit positions with the value of the sign bit if the expression is signed, otherwise fills with 0.

8. How to declare an array of 6 elements of a 7-bit wire?

Wire[6:0] array[5:0]

One bit data width D flipflop:

```
module D_flipflop (input d, clk, output reg q, output q_b);
   always @(posedge clk) begin
      q <= d;
   end
   assign q_b = ~q;
endmodule
```

One bit data width D flip-flop with active low synchronous reset:

```
module D_ffwActiveLowSyncReset (input d, clk, reset, output reg q, output q_b);
   always @(posedge clk) begin
      if (~reset) begin
         q <= 1'b0;
      end
      else begin
         q <= d;
      end
   end
   assign q_b = ~q;
endmodule
```

One bit data width D flip-flop with active low synchronous reset and active low enable:

```
module D_ffwALSR_ALenable (input d, clk, reset, enable, output reg q, q_b);
   always @(posedge clk) begin
      if (~enable) begin
         if (reset == 1'b0) begin
            q = 1'b0;
         end
         q <= d;
      end
   end
   assign q_b = ~q;
endmodule
```

D latch with synchronous enable control:

```
module D_latch (input d, en, output reg q, q_b);
   always @(*) begin
      if (en) begin
         q <= d;
      end
   end
   assign q_b = ~q;
endmodule
```

4-to-1 multiplexer:

```
module multiplexer (input a, b, c, d, s1, s2, output reg out);
   always @(*) begin
      case (s1 | s2)
         2'b00: out <= a;
         2'b01: out <= b;
         2'b10: out <= c;
         2'b11: out <= d;
      endcase
   end
endmodule
```

4-bit counter with reset and enable controls:

```
module counter (input reset, enable, output reg [3:0]count);
   always @(posedge) begin
      if (enable) begin
         count <= count + 1'b1;
         if (reset) begin
            count <= 4'b0;
         end
         else if (count == 4'b1111) begin
            count <= 4'b0;
         end
      end
   end
endmodule
```

Truth Table for 7-Seg

	Input				Output						
	A	B	C	D	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	1	0	0
L	1	0	1	0	1	1	1	0	0	0	1
I	1	0	1	1	1	0	0	1	1	1	1
N	1	1	0	0	1	0	0	1	0	0	1
F	1	1	0	1	0	0	0	1	1	1	0
U	1	1	1	0	1	0	0	0	0	0	1
OFF	1	1	1	1	1	1	1	1	1	1	1

K-map for segment 0 and 1

Segment 0

AB\CD	00	01	11	10
00	0	1	0	0
01	1	0	0	0
11	1	0	1	1
10	0	0	1	1

Segment 1

AB\CD	00	01	11	10
00	0	0	0	0
01	0	1	0	1
11	0	1	1	0
10	0	0	0	1

Segment 0 logic = $B\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D + AC$

Segment 1 logic = $B\bar{C}D + ABD + \bar{A}BC\bar{D} + A\bar{B}C\bar{D}$

```
module seven_seg_decoder(input [3:0] x, output [6:0] hex_LEDs);
   reg [6:0] reg_LEDs;

   assign hex_LEDs[0]= ( (x[2]&(~x[1])&(~x[0])) | ((~x[3])&(~x[2])&(~x[1])&x[0]) | (x[3]&x[1]) );

   assign hex_LEDs[1]= ( (x[2]&(~x[1])&x[0]) | (x[3]&x[2]&x[0]) | ((~x[3])&x[2]&x[1]&(~x[0])) | (x[3]&(~x[2])&x[1]&(~x[0])) );

   assign hex_LEDs[6:2]=reg_LEDs[6:2];


   always @(*) begin
      case (x)
         4'b0000: reg_LEDs[6:2]=5'b10000; //7'b1000000 decimal 0
         4'b0001: reg_LEDs[6:2]=5'b11110; //7'b1111001 decimal 1
         4'b0010: reg_LEDs[6:2]=5'b01001; //7'b0100100 decimal 2
         4'b0011: reg_LEDs[6:2]=5'b01100; //7'b0110000 decimal 3
         4'b0100: reg_LEDs[6:2]=5'b00110; //7'b0011001 decimal 4
         4'b0101: reg_LEDs[6:2]=5'b00100; //7'b0010010 decimal 5
         4'b0110: reg_LEDs[6:2]=5'b00000; //7'b0000010 decimal 6
         4'b0111: reg_LEDs[6:2]=5'b11110; //7'b1111000 decimal 7
         4'b1000: reg_LEDs[6:2]=5'b00000; //7'b0000000 decimal 8
         4'b1001: reg_LEDs[6:2]=5'b00100; //7'b0010000 decimal 9
         4'b1010: reg_LEDs[6:2]=5'b10001; //7'b1000111 letter L
         4'b1011: reg_LEDs[6:2]=5'b11110; //7'b1111001 letter I
         4'b1100: reg_LEDs[6:2]=5'b00110; //7'b0011001 letter N
         4'b1101: reg_LEDs[6:2]=5'b00011; //7'b0001110 letter F
         4'b1110: reg_LEDs[6:2]=5'b10000; //7'b1000001 letter U
         4'b1111: reg_LEDs[6:2]=5'b11111; //7'b1111111 OFF
      /* finish the case block */
         default: regLEDs[6:2] = 5'bx;
      endcase
   end
endmodule
```