1. Tutorial Lab #1

a. In the Tutorial lab #1, we have installed and set up all the software, license and device support packages required. Following the course webpage document, we have understood about commonly made mistakes and questions. By completing the “Quartus Prime Introduction Using Verilog Designs” tutorial, we learned how to create new projects, and practiced useful methods to set up and implement a circuit using Verilog HDL.

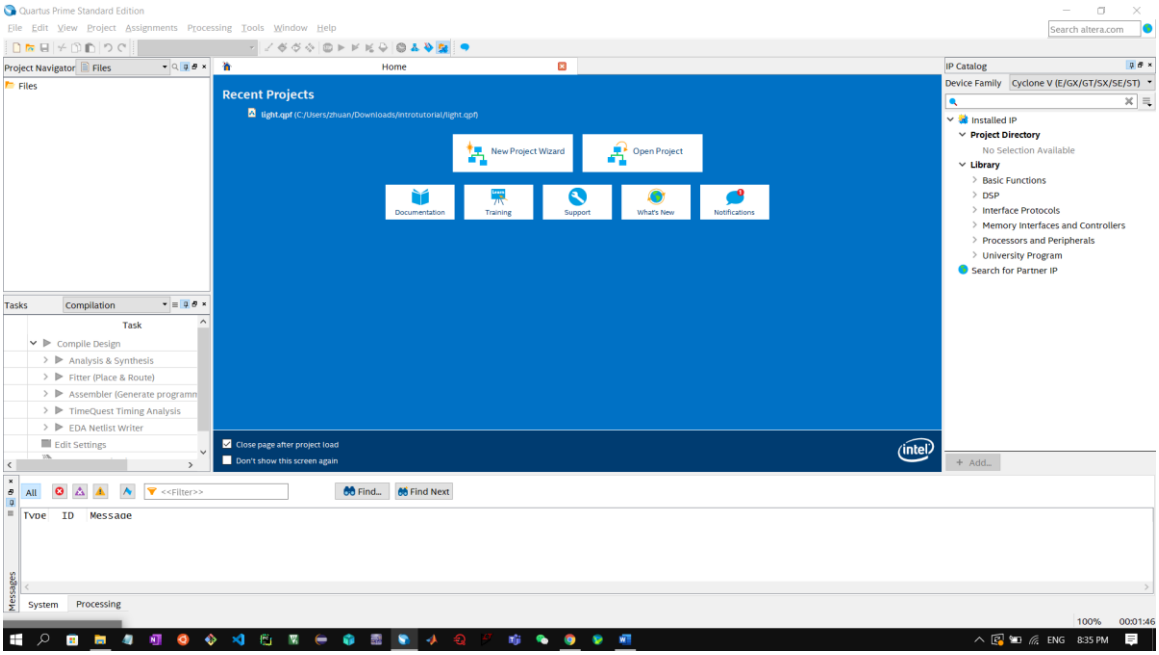

Figure 1: Setting up Software and Device Packages

b. Following the tutorial lab, the circuit written in Verilog code is successfully implemented onto the Altera DE1-SoC board by correctly mapping designed inputs and outputs to corresponding pins and error-free compilation.

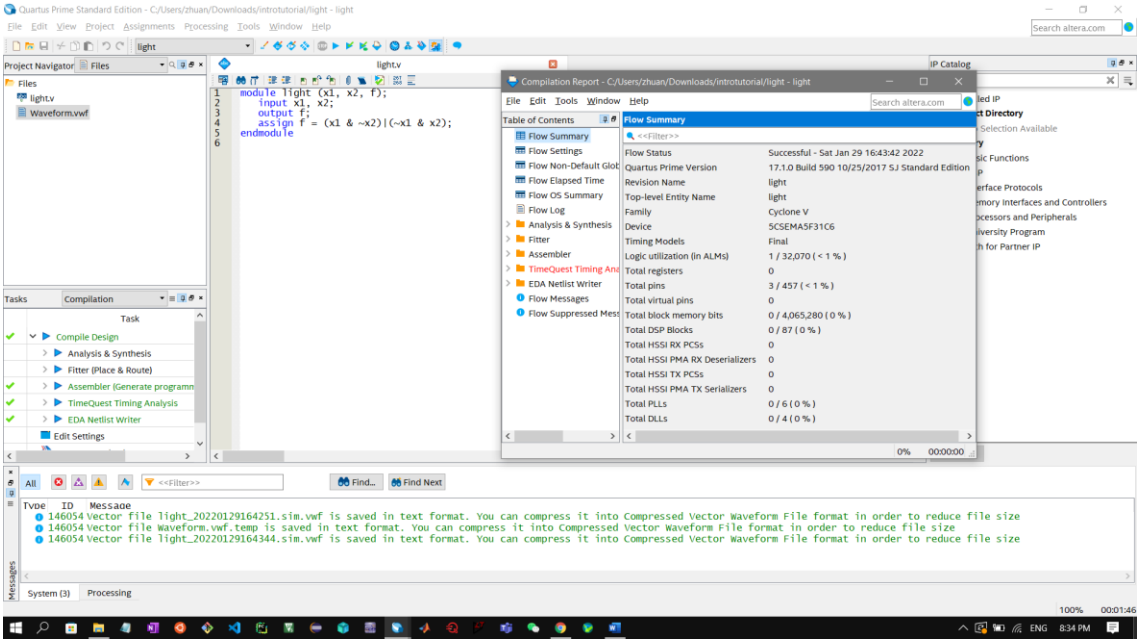

Figure 2: Successful Compilation

c. Functional simulation is used to verify the intended function of the circuit, as shown below. When the value of inputs x1 and x2 are 00, 01, 11, 10, the output f produces values of 0, 1, 0, 1, respectively. This is proved to be true by comparing the results to the actual implemented circuit on the Altera DE1-SoC board, whose working states are shown below in the pictures. When the status of switches SW0 and SW1 are 00, 01, 11, 10, the result represented by LEDR0 is OFF, ON, OFF, ON, respectively.

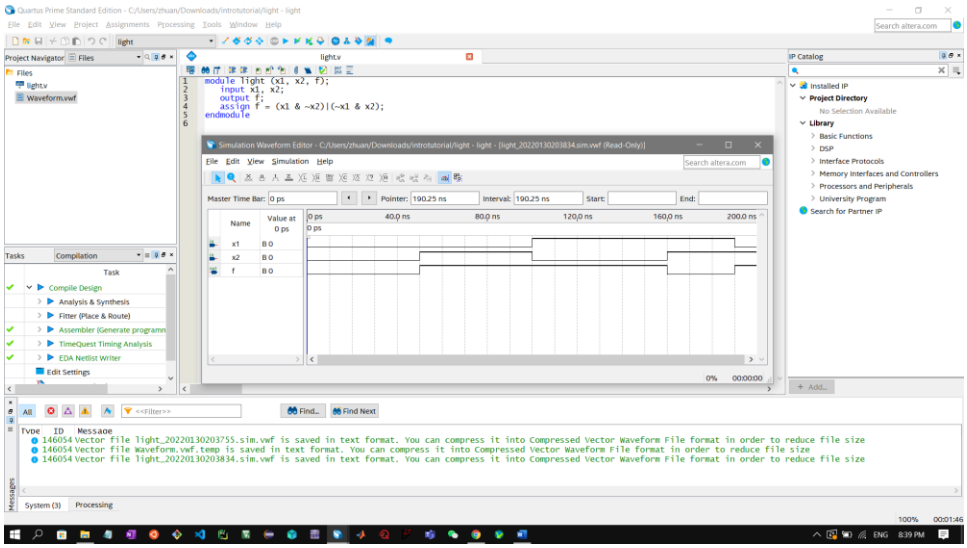

Figure 3: Functional Simulation

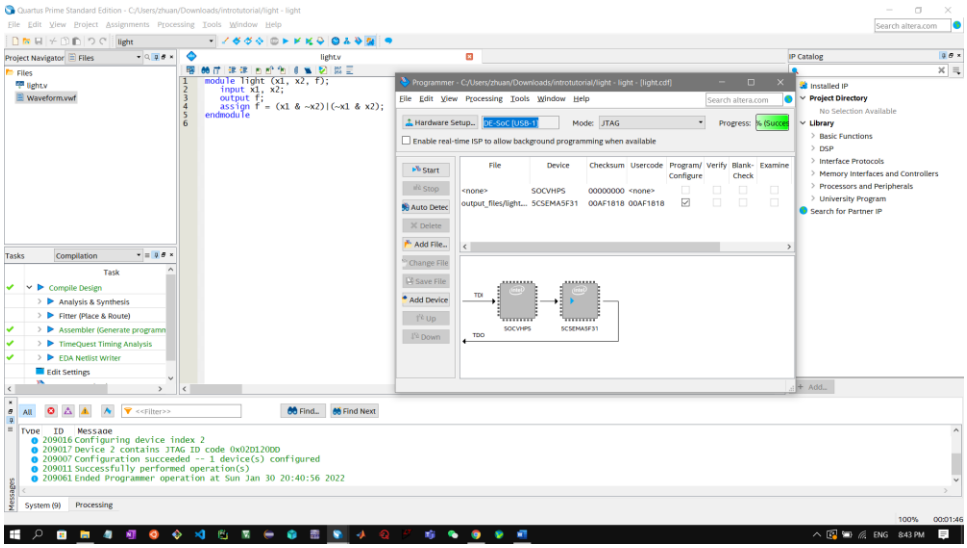

Figure 4: Implementation onto Altera DE1-SoC Board

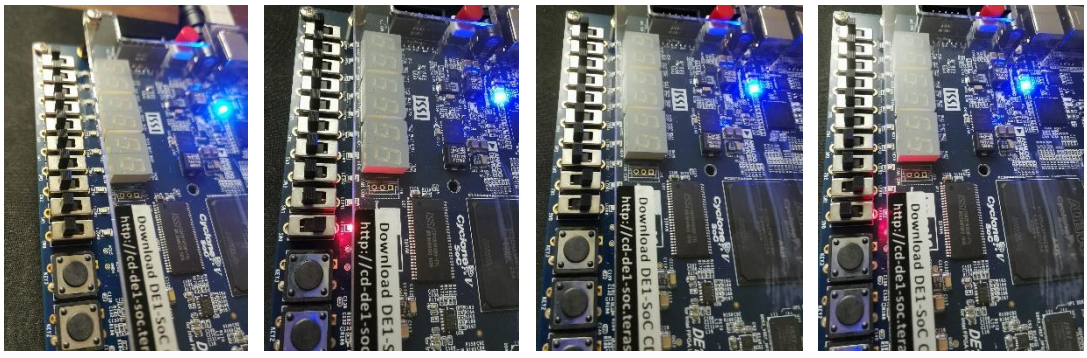

Figure 5: Real Circuit Behavior, Same as in Functional Simulation: (0,0) →0; (0,1) →1; (1,1) →0; (1,0) →1

3. Prelab Questions

a. What is the **reg** data type and what is the **wire** data type in Verilog?
Both reg and wire are parts of driver data type to describe hardware.
A reg, or register, data type is used for storing values.
A wire data type is used for connecting two points, without storing any data

b. Can the **wire** data type be used on the left side of the assignment statement in a procedural block?
The always@ procedural block cannot drive wire data type, meaning the wire datatype cannot be used on the left side of the assignment statement in a procedural block.

c. What are the rules for module port connection?
   i. **Inputs must always be of type net internally but can be connected to a variable of type reg or net externally.**
   ii. **Outputs must always be of type net or reg internally but must be connected to a variable of type net externally.**
   iii. **Inputs must always be of type net internally and can only be connected to a variable of type net externally.**
   iv. **It is legal to connect internal and external ports of different sizes, but synthesis tools could report problems.**
   v. **Unconnected ports are allowed by using a ‘,’.**
   vi. **The net data types are used to connect structure.**
   vii. **A net data type is required if a signal can be driven a structural connection.**

d. What is continuous assignment, blocking assignment and nonblocking assignment?
   i. **Continuous assignment drives nets. They represent structural connections.**
   ii. **Blocking assignments are executed in the order they are coded; hence they are sequential, made with “=” symbol.**
   iii. **Nonblocking assignments are executed in parallel, made with “<=” symbol.**

e. What is the difference in procedural coding when implementing combination logic and sequential logic?

In procedural coding, it is more appropriate to use blocking assignments for combinational circuits in order to avoid race conditions. In sequential circuits, non-blocking assignments are appropriate as the previous value is utilized whenever the right side of a non-blocking assignment has the result of another non-blocking assignment.

f. How does one avoid inferred latches when using Verilog to describe circuits?
To avoid latches, one way is to cover all possible conditions in an “if” statement. Another way is to initialize all the variable of the combinational block as soon as it enters.

g. What is the difference between the operators “<<” and “<<<”?
“<<” is a binary logical shift, where vacated bits are filled with 0’s.
“<<<” is a binary arithmetic shift, where vacated bits are filled with 1’s or 0’s according to whether the expression is signed or unsigned.

h. How to declare an array of 6 elements of a 7-bit wire?
For example, a 7-bit wire, named wire1, is represented by 7’b1110001, then the last 6 elements of the wire can be declared as

wire1[5:0] = 6’b110001;

4. Example Modules

a. One bit data width D flip-flop:

```
//**** One bit data width D flip-flop ****//
module module1(D, clk, Q);
  input D, clk;
  output reg Q;

  always @(posedge clk)
  begin
    Q <= D;
  end
endmodule
```

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

```
//**** One bit data width D flip-flop with active low synchronous reset ****//
module module2(D, clk, synchR, Q);
  input D, clk, synchR;
  output reg Q;

  always @(posedge clk)
  begin
  if (~synchR)
    Q <= 1'b0;
  else
    Q <= D;
  end
endmodule
```

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

```
//**** One bit data width D flip-flop with active low synchronous reset and active
low enable ****//
module module3(D, clk, synchR, synchE, Q);
  input D, clk, synchR, synchE;
  output reg Q;

  always @(posedge clk)
  begin
  if (~synchR)
    Q <= 1'b0;
  else if (synchE)
    Q <= D;
  end
endmodule
```

d. D latch with synchronous enable control:

```
//**** D latch with synchronous control ****//
module module4(D, synchE, Q);
  input D, synchE;
  output reg Q;

  always @(synchE or D)
  begin
  if (synchE)
    Q <= D;
  end
endmodule
```

e. 4-to-1 multiplexer:

```
//**** 4-to-1 multiplexer ****//
module module5(a, b, c, d, sel, out);
  input a, b, c, d;
  input [1:0] sel;
  output reg out;

  always @(a or b or c or d or sel)
  begin
  case (sel)
    2'b00: out <= a;
    2'b01: out <= b;
    2'b11: out <= c;
    2'b10: out <= d;
  endcase
  end
endmodule
```

f. 4-bit counter with reset and enable controls:

```
//**** a 4-bit counter with reset and enable controls ****//
module module6(clk, enable, reset, out);
  input clk, enable, reset;
  output reg [3:0] out;

  always @(posedge clk)
  begin
  if (reset)
    out <= 4'b0;
  else if (enable)
    out <= out + 1;
  end
endmodule
```

5. Truth Table for the Seven-Segment Decoder

The names of our group members are: **Shangkun Zhuang** and **Arun Mistry.** According to the requirements specified in the lab document, the first five letters of the first names should be displayed in mapping to binary combinations 1010 to 1110, shown below.

INPUT				Value / Char / OFF	OUTPUT of SEGMENTS						
Bit3	**Bit2**	**Bit1**	**Bit0**		**Seg0**	**Seg1**	**Seg2**	**Seg3**	**Seg4**	**Seg5**	**Seg6**
0	0	0	0	**0**	0	0	0	0	0	0	1
0	0	0	1	**1**	1	0	0	1	1	1	1
0	0	1	0	**2**	0	0	1	0	0	1	0
0	0	1	1	**3**	0	0	0	0	1	1	0
0	1	0	0	**4**	1	0	0	1	1	0	0
0	1	0	1	**5**	0	1	0	0	1	0	0
0	1	1	0	**6**	0	1	0	0	0	0	0
0	1	1	1	**7**	0	0	0	1	1	1	1
1	0	0	0	**8**	0	0	0	0	0	0	0
1	0	0	1	**9**	0	0	0	0	1	0	0

Figure 6: Truth Table for the Common Part, the Numbers from 0 ~ 9

4-Bit INPUT				Value / Char / OFF	7-Bit OUTPUT						
Bit3	**Bit2**	**Bit1**	**Bit0**		**Seg0**	**Seg1**	**Seg2**	**Seg3**	**Seg4**	**Seg5**	**Seg6**
1	0	1	0	**S**	0	1	0	0	1	0	0
1	0	1	1	**H**	1	0	0	1	0	0	0
1	1	0	0	**A**	0	0	0	1	0	0	0
1	1	0	1	**N**	1	1	0	1	0	1	0
1	1	1	0	**G**	0	1	0	0	0	0	1
1	1	1	1	**OFF**	1	1	1	1	1	1	1

Figure 7: Truth Table for Name of Group Member Shangkun Zhuang, with First Five Character of "S H A N G"

6. K-maps Method for Segment 0 and 1 on the Truth Table

Table 1: Member Shangkun Zhuang, Segment 0

ab\cd	00	01	11	10
00	0	1	0	0
01	1	0	0	0
11	0	1	1	0
10	0	0	1	0

Table 2: Member Shangkun Zhuang, Segment 1

ab\cd	00	01	11	10
00	0	0	0	0
01	0	1	0	1
11	0	1	1	1
10	0	0	0	1

Expressions

Table 1: $\bar{a}\bar{b}\bar{c}d + \bar{a}b\bar{c}\bar{d} + abd + acd$

Table 2: $b\bar{c}d + abd + bc\bar{d} + ac\bar{d}$

7. Verilog Circuit for the Truth Table

For SHANG:

```
//**** Verilog Circuit for SHANG Truth Table ****//

module seven_seg_decoder(input [3:0] x, output [6:0] hex_LEDs);
reg [6:0] reg_LEDs;
assign hex_LEDs[0]=   ~x[3] & ~x[2] & ~x[1] & x[0] |
              ~x[3] & x[2] & ~x[1] & ~x[0] |
              x[3] & x[2] & x[0] |
              x[3] & x[1] & x[0];   /* expression for segment 0 */

assign hex_LEDs[1]=   x[2] & ~x[1] & x[0] |
              x[3] & x[2] & x[0] |
              x[2] & x[1] & ~x[0] |
              x[3] & x[1] & ~x[0];    /* expression for segment 1 */

              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'b0100100 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'b00100; //7'b0010010 decimal S
  4'b1011: reg_LEDs[6:2]=5'b00010; //7'b0001001 decimal H
  4'b1100: reg_LEDs[6:2]=5'b00010; //7'b0001000 decimal A
  4'b1101: reg_LEDs[6:2]=5'b01010; //7'b0101011 decimal N
  4'b1110: reg_LEDs[6:2]=5'b10000; //7'b1000010 decimal G
  4'b1111: reg_LEDs[6:2]=5'b11111; //7'b1111111 ALL OFF
endcase
end
endmodule
```