EC605 LAB2
REPORT

Xinran Zhang zhangxr@bu.edu
Jiahao Chen tobychen@bu.edu

Task 1

We used the resource from the basic logic gates to build a full adder.
The Verilog code is attached below,

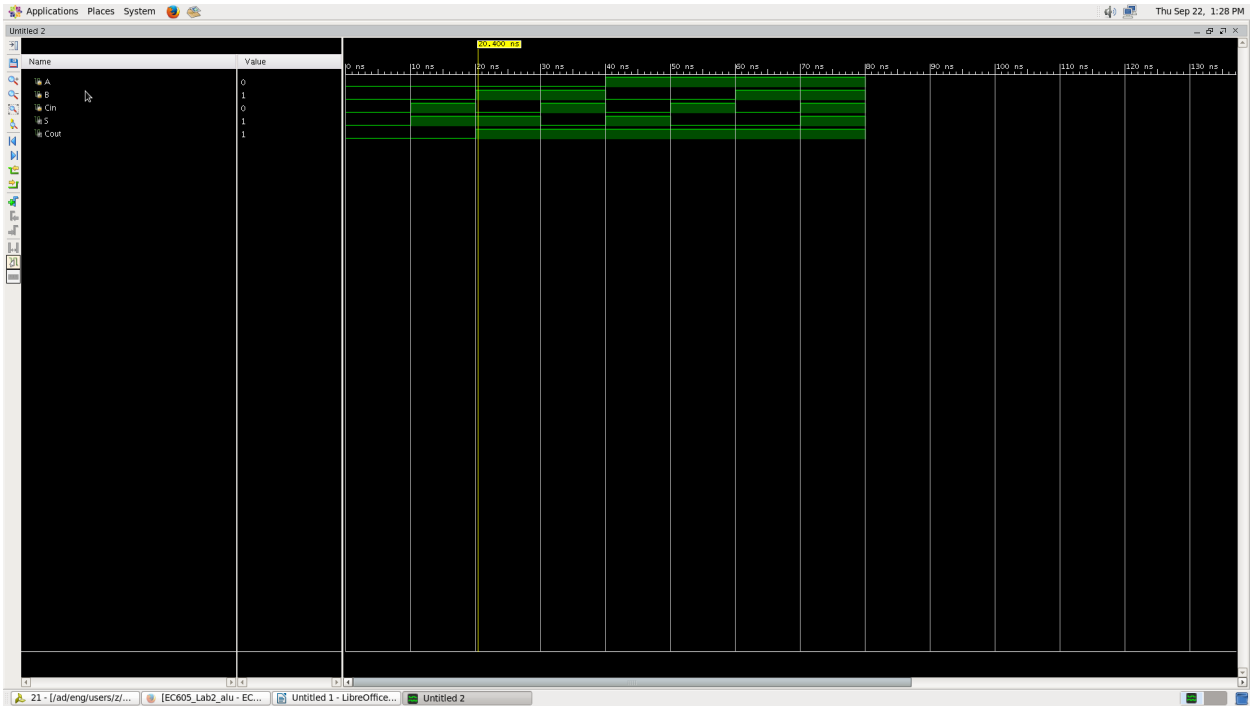
```
`timescale 1ns / 1ps


module singlebitfulladder(A,B,CI,CO,Y);
    input A, B, CI;
    output CO, Y;
    wire w1, w2, w3;

    xor G1 ( w1, A, B);
    and G2 ( w2, A, B);
    and G3 ( w3, w1, CI);
    xor G4 ( Y, CI, w1);
    or G5 ( CO, w2, w3);

endmodule
```

The testbench is listed below,

```
module Full_Adder_Test;
    reg A, B, Cin;
    wire S, Cout;

    singlebitfulladder uut(.A(A), .B(B), .CI(Cin), .Y(S), .CO(Cout));

    initial
    begin
            A = 0;
            B = 0;
            Cin = 0;

            #10;
                 {A, B, Cin} = 1;
            #10;
                 {A, B, Cin} = 2;
            #10;
```

```
                {A, B, Cin} = 3;
            #10;
                {A, B, Cin} = 4;
            #10;
                {A, B, Cin} = 5;
            #10;
                {A, B, Cin} = 6;
            #10;
                {A, B, Cin} = 7;
            #10;

            $finish;
    end
endmodule
```

The waveform is like this,



Task 2

At first, we build a hierarchy module of 2 bits full adder. Then in the process of building the ripple carry adder, we use the hierarchy to create our adder.

The 2 bit adder Verilog code is listed below:

```
`timescale 1ns / 1ps
```

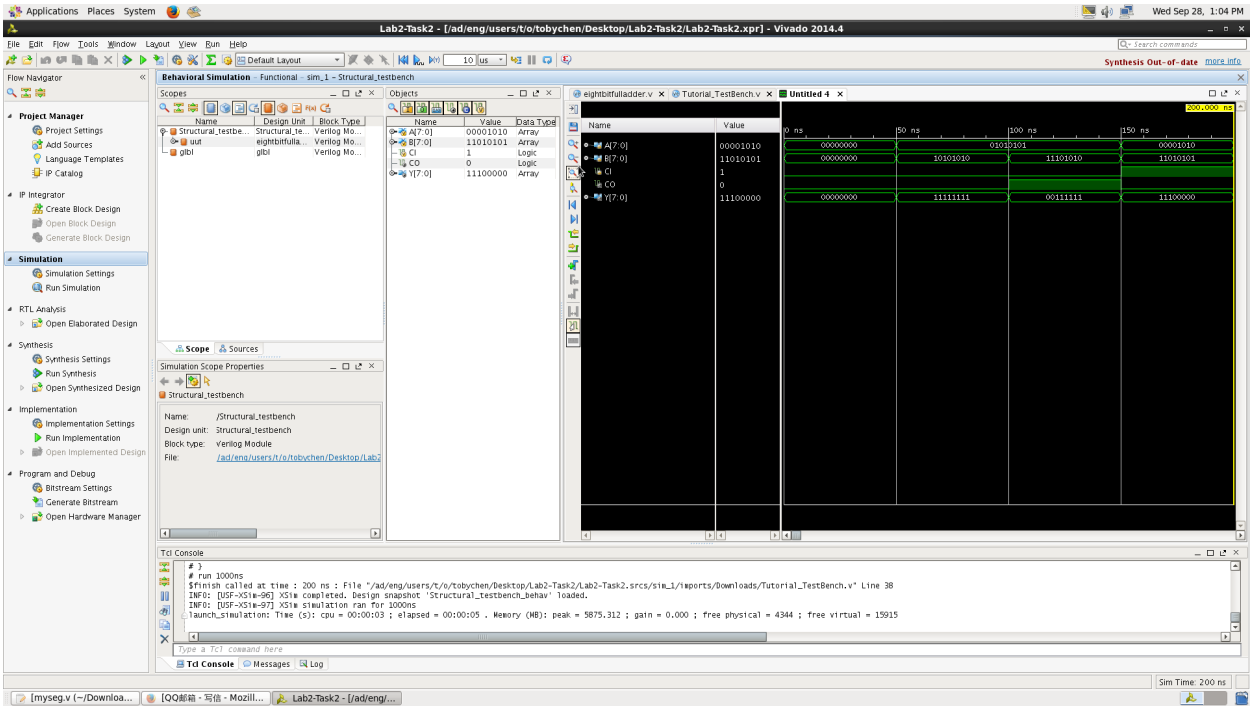
```
module eightbitfulladder( A, B, CI, CO, Y);
    input [7:0] A;
    input [7:0] B;
    input CI;
    wire [6:0] w;

    output CO;
    output [7:0] Y;


    singlebitfulladder uut1(A[0],B[0],CI,w[0],Y[0]);
    singlebitfulladder uut2(A[1],B[1],w[0],w[1],Y[1]);
    singlebitfulladder uut3(A[2],B[2],w[1],w[2],Y[2]);
    singlebitfulladder uut4(A[3],B[3],w[2],w[3],Y[3]);
    singlebitfulladder uut5(A[4],B[4],w[3],w[4],Y[4]);
    singlebitfulladder uut6(A[5],B[5],w[4],w[5],Y[5]);
    singlebitfulladder uut7(A[6],B[6],w[5],w[6],Y[6]);
    singlebitfulladder uut8(A[7],B[7],w[6],CO,Y[7]);

endmodule
```

We use the testbench listed below to test whether our design is right. The selection of the numbers is critical; it can help us to examine the main function efficiently. We chose the first group number to detect the zero output, the second group of number to test the original function, the third group of the number to test the CI input.

```
`timescale 1ns / 100ps

module Structural_testbench;

    reg [7:0] A;
    reg [7:0] B;
    reg CI;
    wire CO;
    wire [7:0] Y;


    eightbitfulladder uut( A, B, CI, CO, Y);

    initial
    begin

     A = 8'b00000000;
     B = 8'b00000000;
     CI = 1'b0;

     #50;
     A = 8'b01010101;
     B = 8'b10101010;
     CI = 1'b0;
```

```
        #50;
        A = 8'b01010101;
        B = 8'b11101010;
        CI = 1'b0;

        #50;
        A = 8'b00001010;
        B = 8'b11010101;
        CI = 1'b1;

        #50;

        $finish;

    end
endmodule
```

The result is like that:



Task 3

We build the 8 bit Carry Select Adder using the hierarchy of the ‘fourbitfulladder’ and ‘Gate_Level_Mux ‘ . The Verilog code is listed below,

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////
```

```
// Company:
// Engineer:
//
// Create Date: 09/22/2016 09:26:40 PM
// Design Name:
// Module Name: fourbitfulladder
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
////////

module fourbitfulladder( A, B, CI, CO, Y);
    input [3:0] A;
    input [3:0] B;
    input CI;
    wire [2:0] w;

    output CO;
    output [3:0] Y;

    singlebitfulladder uut1(A[0],B[0],CI,w[0],Y[0]);
    singlebitfulladder uut2(A[1],B[1],w[0],w[1],Y[1]);
    singlebitfulladder uut3(A[2],B[2],w[1],w[2],Y[2]);
    singlebitfulladder uut4(A[3],B[3],w[2],CO,Y[3]);


endmodule
```

Then, the whole project code is,

```
`timescale 1ns / 1ps

module eightbitcarryselect( A, B, CI, CO, Y);

        input [7:0] A;
        input [7:0] B;
        input CI;

        wire w1,w2,w3,w4;
        wire [3:0] q, a;

        output CO;
        output [7:0] Y;
```

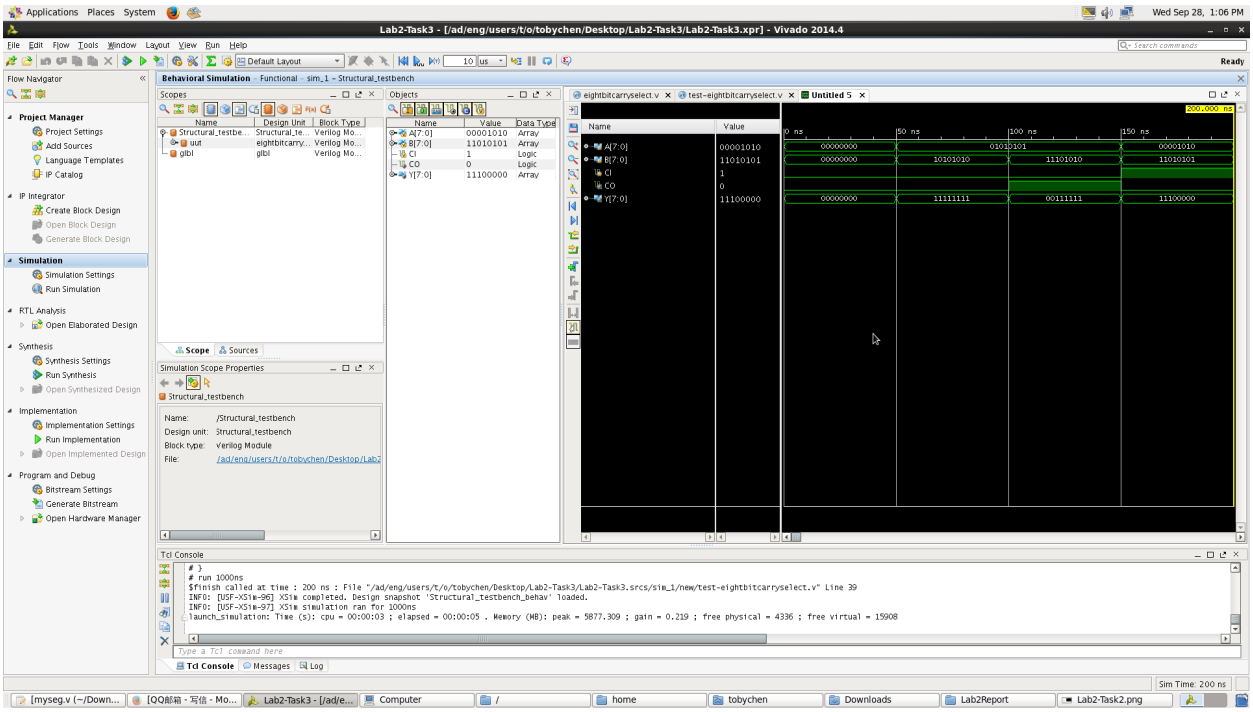
```
        fourbitfulladder uut1(A[3:0],B[3:0], CI, w1, Y[3:0]);
        fourbitfulladder uut2(A[7:4],B[7:4], 0,w2,q[3:0]);
        fourbitfulladder uut3(A[7:4],B[7:4], 1,w4,a[3:0]);
        Gate_Level_MUX uut4(q[3], a[3], w1, Y[7]);
        Gate_Level_MUX uut5(q[2], a[2], w1, Y[6]);
        Gate_Level_MUX uut6(q[1], a[1], w1, Y[5]);
        Gate_Level_MUX uut7(q[0], a[0], w1, Y[4]);

        or G1 (w3,w1,w2);
        and G2 (CO,w3,w4);

endmodule
```

We set up the testbench in certain numbers to examine the influence of the CI and CO to the final result.
The testbench of the code is provided as follows,

```
`timescale 1ns / 1ps

module Structural_testbench;

    reg [7:0] A;
    reg [7:0] B;
    reg CI;
    wire CO;
    wire [7:0] Y;


    eightbitcarryselect uut( A, B, CI, CO, Y);

    initial
    begin

     A = 8'b00000000;
     B = 8'b00000000;
     CI = 1'b0;


     #50;
     A = 8'b01010101;
     B = 8'b10101010;
     CI = 1'b0;

     #50;
     A = 8'b01010101;
     B = 8'b11101010;
     CI = 1'b0;

     #50;
     A = 8'b00001010;
     B = 8'b11010101;
     CI = 1'b1;
```

```
        #50;

        $finish;

    end
endmodule
```

The result of the test is,



Task 4

We spent plenty of time on the ALU part of the lab. We faced two main questions. The first one is how to build the counter to change the frequency of the clock. The second one is defining how to make the seven-segments display work. Finally, we fixed these two questions out with the help of our TA.

The ALU module is,

```
`timescale 1ns / 1ps

module ALU( A, B, Y, opcode, N, Z, C, V);
    input [7:0] A, B;
```

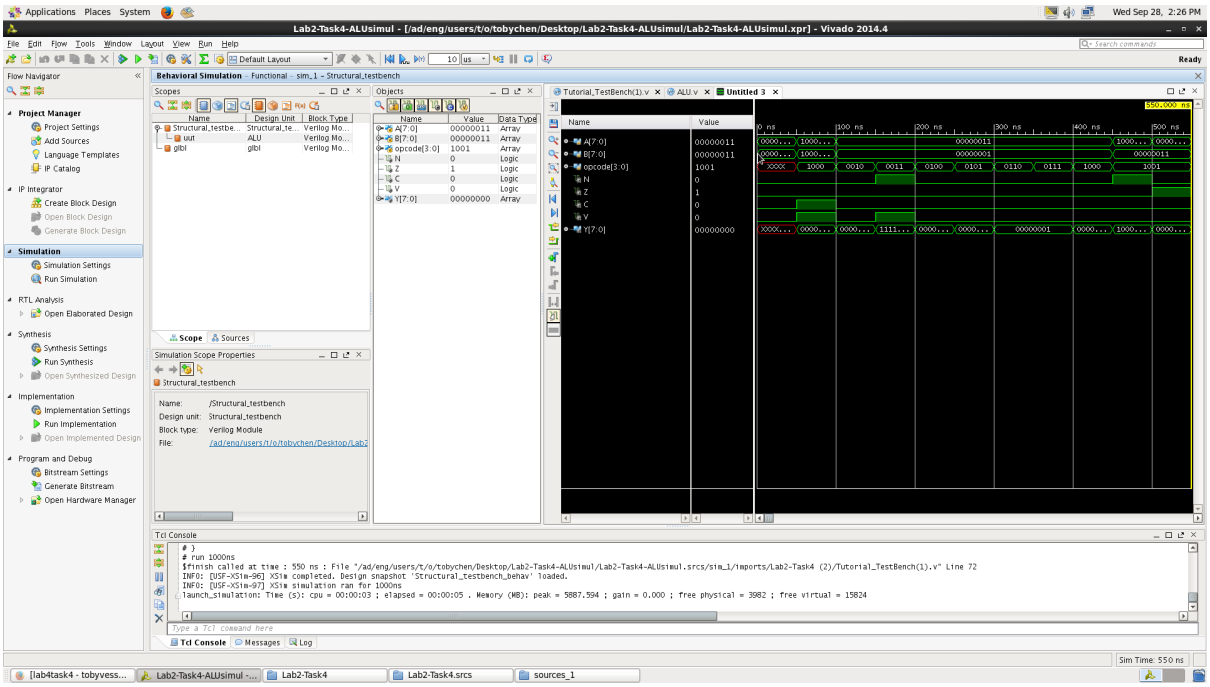
```
    input [3:0] opcode;
    reg cout;
    output reg [7:0] Y;
    output reg N, Z, C, V;

    always @( A or B or Y or opcode)
    begin


    if (opcode == 4'b0001)
        Y[7:0] = A & B;
    if (opcode == 4'b0010)
        Y[7:0] = A | B;
    if (opcode == 4'b0011)
        Y[7:0] = ~A ;
    if (opcode == 4'b0100)
        Y[7:0] = A ^ B;
    if (opcode == 4'b0101)
        Y[7:0] = A << B;
    if (opcode == 4'b0110)
        Y[7:0] = $signed (A) >>> B;
    if (opcode == 4'b0111)
        Y[7:0] = A >> B;
    if (opcode == 4'b1000)
        {cout, Y} = A + B;
    if (opcode == 4'b1001)
        {cout, Y} = A - B;

    if (Y[7]==1)
        N = 1;
    else
        N = 0;

    if (Y[7:0]==0)
        Z = 1;
    else
        Z = 0;

    if (A[7]==0&&B[7]==0&&Y[7]==1||A[7]==1&&B[7]==1&&Y[7]==0)
        V = 1;
    else
        V = 0;

    if (cout==1&&(opcode==4'b1000||opcode==4'b1001))
        C = 1;
    else
        C = 0;

    end

endmodule
```

We managed different numbers to test the full function of ALU. The testbench of ALU is,

```
`timescale 1ns / 100ps

module Structural_testbench;

    reg [7:0] A, B;
    reg [3:0] opcode;
    wire N, Z, C, V;
    wire [7:0] Y;


    ALU uut( A, B, Y, opcode, N, Z, C, V);

    initial
    begin

     A = 8'b00000000;
     B = 8'b00000000;

     #50;
     A = 8'b10000100;
     B = 8'b10000011;
     opcode = 4'b1000;

     #50;
     A = 8'b00000011;
     B = 8'b00000001;
     opcode = 4'b0010;

     #50;
     A = 8'b00000011;
     B = 8'b00000001;
     opcode = 4'b0011;

     #50;
     A = 8'b00000011;
     B = 8'b00000001;
     opcode = 4'b0100;

     #50;
     A = 8'b00000011;
     B = 8'b00000001;
     opcode = 4'b0101;

     #50;
     A = 8'b00000011;
     B = 8'b00000001;
     opcode = 4'b0110;

     #50;
     A = 8'b00000011;
     B = 8'b00000001;
     opcode = 4'b0111;
```

```
        #50;
        A = 8'b00000011;
        B = 8'b00000001;
        opcode = 4'b1000;

        #50;
        A = 8'b10000011;
        B = 8'b00000011;
        opcode = 4'b1001;

        #50;
        A = 8'b00000011;
        B = 8'b00000011;
        opcode = 4'b1001;


        #50;


        $finish;

    end
endmodule
```

The result of ALU testing,



To make the code can work on the board, we added several parts in it, for example, the counter, the display part. Verilog code is listed below,

```
`timescale 1ns / 1ps
```

```
module Top_module(clk, rst, switches, btn, leds, seven_seg, seven_enable);
input clk, rst;                          // Needed for syncronous inputs
input [15:0] switches;             // Input switches
input [1:0] btn;                   // Buttons used to save switches to
registers
output [3:0] leds;                       // Display N, V, C, Z
output reg [6:0] seven_seg;        // Display outputs to seven segment
output reg [3:0] seven_enable;     // Select which seven segment to
display
reg [15:0] counter;
reg [7:0] A, B;
reg [3:0] SEL;
wire [7:0] Y;
wire [3:0] ones, tens, hundreds;
wire [6:0] seven_seg_ones, seven_seg_tens, seven_seg_hundreds;
/********* ADD MODULE DECLARATIONS HERE ************/
ALU uut1( A, B, Y, SEL, leds[3], leds[2], leds[1], leds[0]);
bcd_converter uut3(Y, ones, tens, hundreds);
seven_segment_display uut4(ones, seven_seg_ones);
seven_segment_display uut5(tens, seven_seg_tens);
seven_segment_display uut6(hundreds, seven_seg_hundreds);
/********* END MODULE DECLARATIONS HERE ************/

/*************** INPUT REGISTERS *****************/
/* This always block saves switch inputs into      */
/* registers when buttons are pressed.             */
always @ (posedge clk or posedge rst)
begin
    if (rst)
    begin
        A <= 0;
        B <= 0;
        SEL <= 0;
    end
    else
    begin
        if (btn[0])
            {A, B} <= switches;
        if (btn[1])
            SEL <= switches[3:0];
        counter = counter + 1'b1;
    end
end

always @(counter[13])
begin
case(counter[13:12])
0:begin
seven_enable=4'b1110;
seven_seg=seven_seg_ones;
end
1:begin
seven_enable=4'b1101;
seven_seg=seven_seg_tens;
```

```
end
2:begin
seven_enable=4'b1011;
seven_seg=seven_seg_hundreds;
end
default: seven_enable=4'b1111;

endcase
end

/********* End always block **********/
endmodule
/************ End module *************/

/************ EXAMPLE BCD MODULE HEADER *************/
/* This function would be used to convert a binary  */
/* number into a decimal number split into 3 digits */

module bcd_converter(binary, ones, tens, hundreds);
input [7:0] binary;
output reg [3:0] ones, tens, hundreds;
integer a;

always @(binary or hundreds or tens or ones)
begin
ones = binary % 'd10;
tens = binary % 'd100 /'d10;
hundreds = binary / 'd100;

end

endmodule
/************ End module *************/

/********* EXAMPLE SEVEN SEGMENT DISPLAY MODULE **********/
/* This function would be used to convert a decimal digit*/
/* into a proper seven segment display value. Refer to   */
/* reference manual for seven segment led configuration  */

module seven_segment_display(digit, seven_seg_out);
input [3:0] digit;
output reg [6:0] seven_seg_out;

always @(*)
begin
case(digit)
4'd0 : seven_seg_out = 7'b1000000;
4'd1 : seven_seg_out = 7'b1111001;
4'd2 : seven_seg_out = 7'b0100100;
4'd3 : seven_seg_out = 7'b0110000;
4'd4 : seven_seg_out = 7'b0011001;
4'd5 : seven_seg_out = 7'b0010010;
4'd6 : seven_seg_out = 7'b0000010;
4'd7 : seven_seg_out = 7'b1111000;
```

```
4'd8 : seven_seg_out = 7'b0000000;
4'd9 : seven_seg_out = 7'b0010000;
endcase
end
endmodule
/************ End module *************/
```