LAB 2 : 32-bit ALU

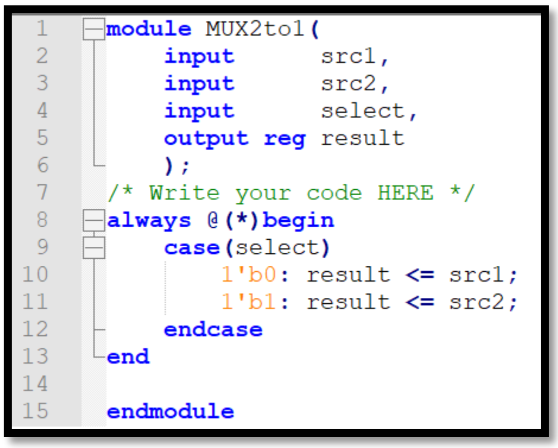
109550135 范恩宇

Part I.　Detailed description of the implementation :

1.　MUX2to1 & MUX4to1 :

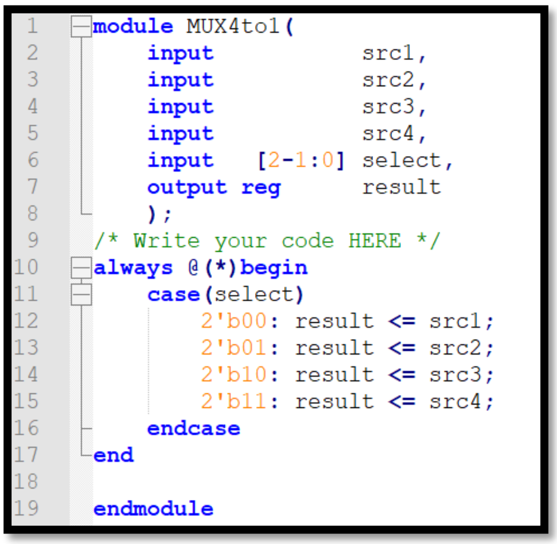
Both multiplexers are implemented with the same logic , I get the corresponding “src n” by making “select” = n (with n in binary) , then assign “src n” to “result” .

MUX 2to1 :



```
module MUX2to1(
       input       src1,
       input       src2,
       input       select,
       output reg  result
       );
/* Write your code HERE */
always @(*)begin
       case(select)
           1'b0: result <= src1;
           1'b1: result <= src2;
       endcase
end

endmodule
```

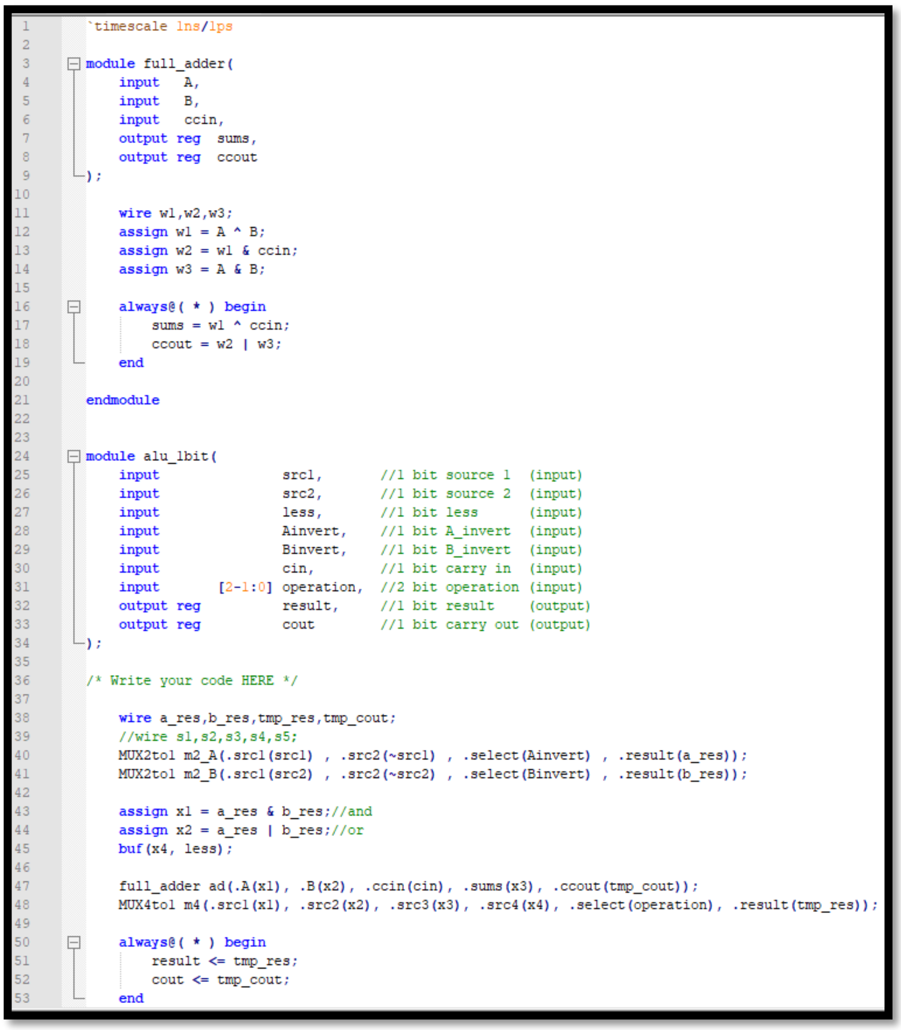
MUX 4to1 :



```
module MUX4to1(
       input               src1,
       input               src2,
       input               src3,
       input               src4,
       input       [2-1:0] select,
       output reg          result
       );
/* Write your code HERE */
always @(*)begin
       case(select)
           2'b00: result <= src1;
           2'b01: result <= src2;
           2'b10: result <= src3;
           2'b11: result <= src4;
       endcase
end

endmodule
```

2. 1-bit ALU :

Just as the original graph described , generate two outputs "a_res" and "b_res" for two 2x1-multiplexers and use "Ainvert" and "Binvert" as "select" of each . Second , assign "x1~4" as AND gate ,OR gate , full adder (also implemented in the file to generate a "tmp_cout" , similar to the one in TA' s "verilog introduction.pdf") , and "less" , then connect them to one 4x1-multiplexer and generate a "tmp_res" . Finally , assign "tmp_res" to "result" ("tmp_cout" to "cout" , too), and get the final result .



```
`timescale 1ns/1ps

module full_adder(
    input   A,
    input   B,
    input   ccin,
    output reg  sums,
    output reg  ccout
);

    wire w1,w2,w3;
    assign w1 = A ^ B;
    assign w2 = w1 & ccin;
    assign w3 = A & B;

    always@( * ) begin
        sums = w1 ^ ccin;
        ccout = w2 | w3;
    end

endmodule


module alu_1bit(
    input               src1,       //1 bit source 1  (input)
    input               src2,       //1 bit source 2  (input)
    input               less,       //1 bit less      (input)
    input               Ainvert,    //1 bit A_invert  (input)
    input               Binvert,    //1 bit B_invert  (input)
    input               cin,        //1 bit carry in  (input)
    input       [2-1:0] operation,  //2 bit operation (input)
    output reg          result,     //1 bit result    (output)
    output reg          cout        //1 bit carry out (output)
);

/* Write your code HERE */

    wire a_res,b_res,tmp_res,tmp_cout;
    //wire s1,s2,s3,s4,s5;
    MUX2to1 m2_A(.src1(src1) , .src2(~src1) , .select(Ainvert) , .result(a_res));
    MUX2to1 m2_B(.src1(src2) , .src2(~src2) , .select(Binvert) , .result(b_res));

    assign x1 = a_res & b_res;//and
    assign x2 = a_res | b_res;//or
    buf(x4, less);

    full_adder ad(.A(x1), .B(x2), .ccin(cin), .sums(x3), .ccout(tmp_cout));
    MUX4to1 m4(.src1(x1), .src2(x2), .src3(x3), .src4(x4), .select(operation), .result(tmp_res));

    always@( * ) begin
        result <= tmp_res;
        cout <= tmp_cout;
    end
```

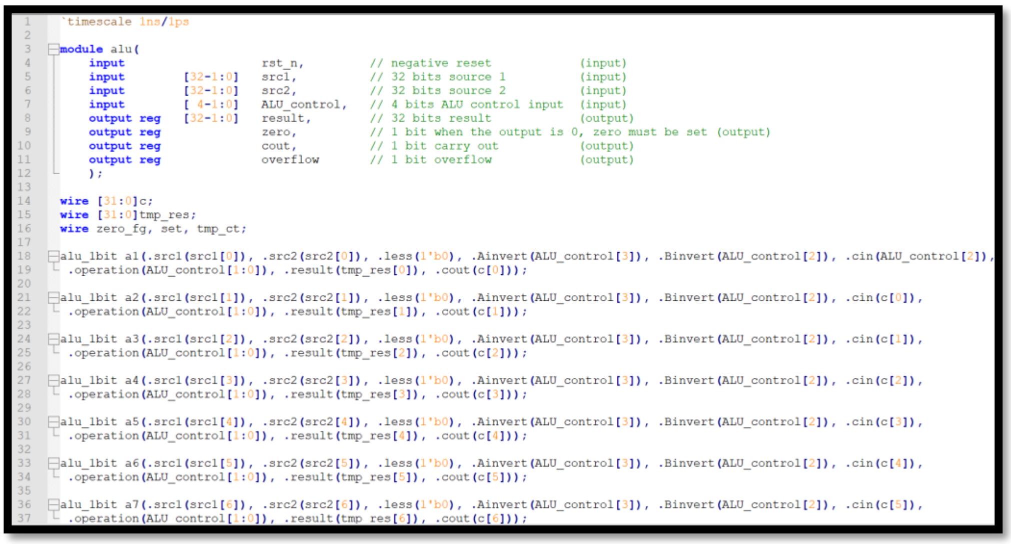
3. 32-bit ALU :

Basically just like 1-bit ALU , but connected by 32 of it . In addition , the one in the bottom deals with overflow problem . I assign its operation to be “2'b10” and make its outputs to be “set” and “tmp_ct” . Then create an initial “zero_fg” with “or” function and 32 bits of results , and deal with 3 cases .

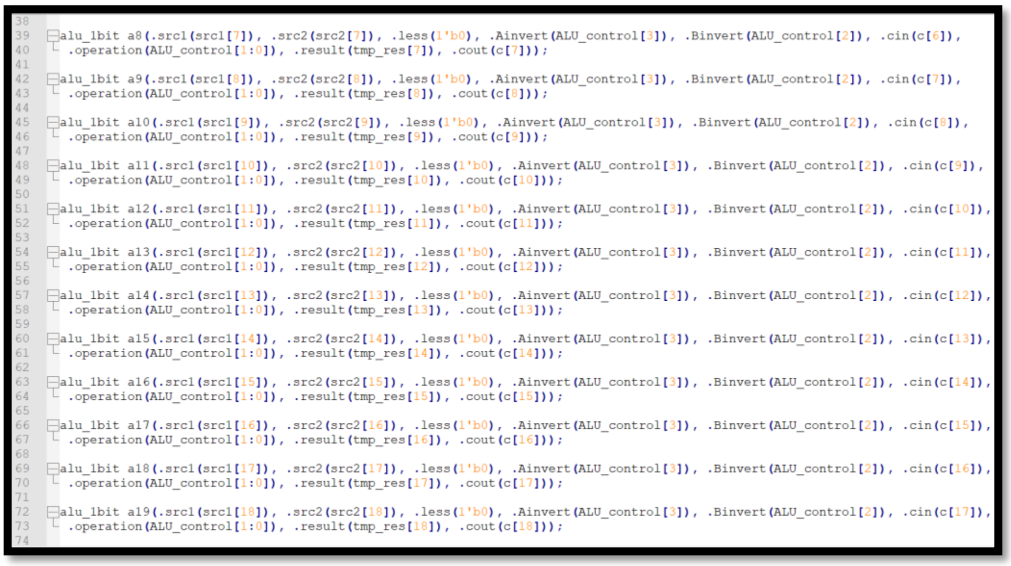
First , if the function is “addition” or “subtract” , “zero” will be “~zero_fg” , “cout” will be “cout of alu_1bit a32” , and “overflow” will be “XOR” of “cout” of “alu_1bit a31&a32” .

Second , if the function is “set less than” , “result” and “zero” will be 1&0(0&1) , if “set” equals to 1(0) . Besides , “overflow” & “cout” will both be 0 .

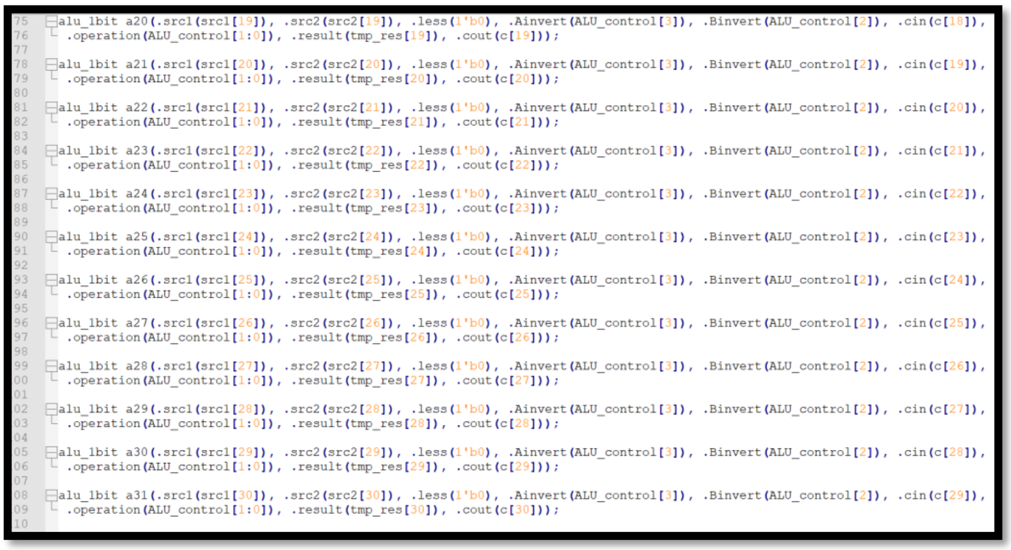
Finally , if it is none of the above , then all 4 output will all be 0 .



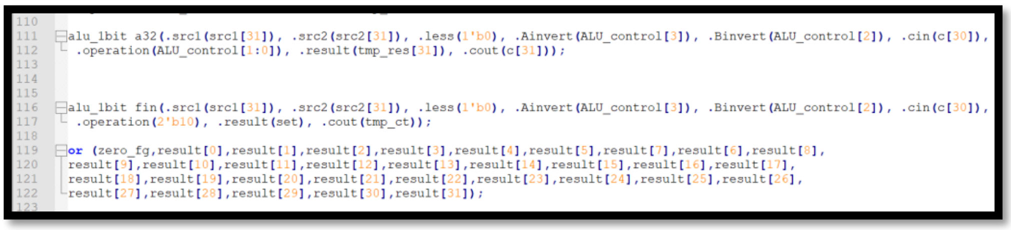
```verilog
`timescale 1ns/1ps

module alu(
    input                   rst_n,         // negative reset            (input)
    input        [32-1:0]   src1,          // 32 bits source 1          (input)
    input        [32-1:0]   src2,          // 32 bits source 2          (input)
    input        [ 4-1:0]   ALU_control,   // 4 bits ALU control input  (input)
    output reg   [32-1:0]   result,        // 32 bits result            (output)
    output reg              zero,          // 1 bit when the output is 0, zero must be set (output)
    output reg              cout,          // 1 bit carry out           (output)
    output reg              overflow       // 1 bit overflow            (output)
    );

wire [31:0]c;
wire [31:0]tmp_res;
wire zero_fg, set, tmp_ct;

alu_1bit a1(.src1(src1[0]), .src2(src2[0]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(ALU_control[2]),
 .operation(ALU_control[1:0]), .result(tmp_res[0]), .cout(c[0]));

alu_1bit a2(.src1(src1[1]), .src2(src2[1]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[0]),
 .operation(ALU_control[1:0]), .result(tmp_res[1]), .cout(c[1]));

alu_1bit a3(.src1(src1[2]), .src2(src2[2]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[1]),
 .operation(ALU_control[1:0]), .result(tmp_res[2]), .cout(c[2]));

alu_1bit a4(.src1(src1[3]), .src2(src2[3]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[2]),
 .operation(ALU_control[1:0]), .result(tmp_res[3]), .cout(c[3]));

alu_1bit a5(.src1(src1[4]), .src2(src2[4]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[3]),
 .operation(ALU_control[1:0]), .result(tmp_res[4]), .cout(c[4]));

alu_1bit a6(.src1(src1[5]), .src2(src2[5]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[4]),
 .operation(ALU_control[1:0]), .result(tmp_res[5]), .cout(c[5]));

alu_1bit a7(.src1(src1[6]), .src2(src2[6]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[5]),
 .operation(ALU_control[1:0]), .result(tmp_res[6]), .cout(c[6]));
```



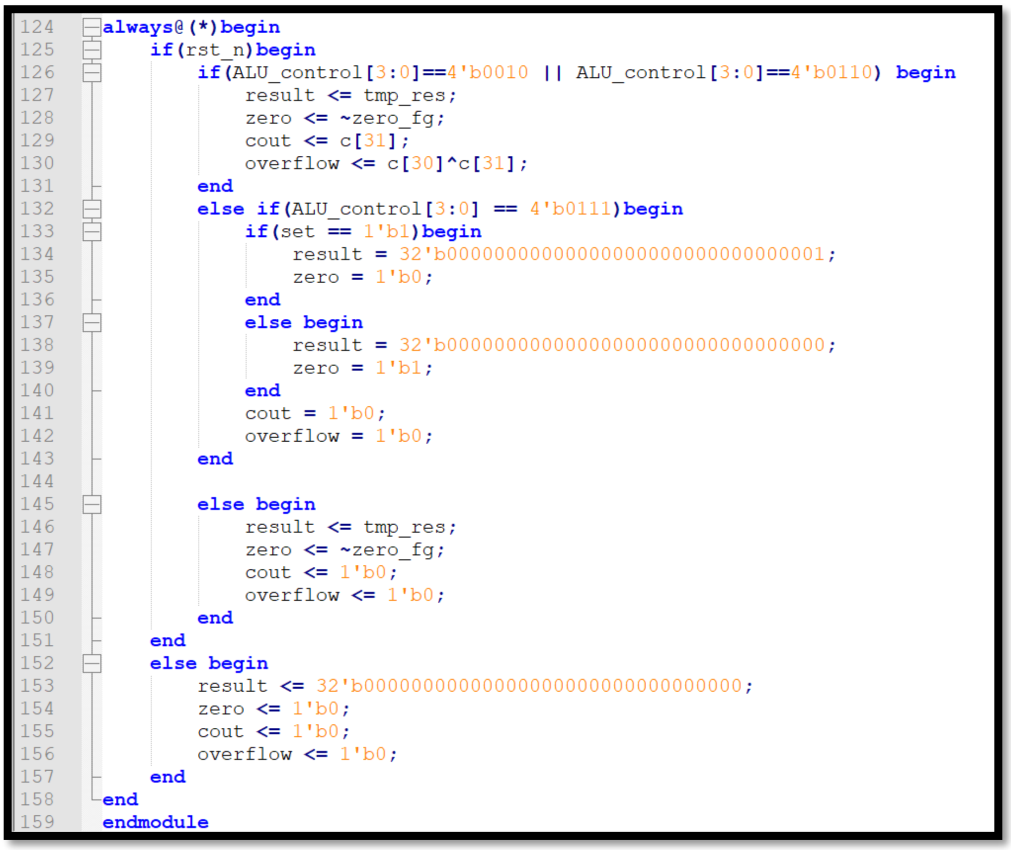
```verilog
alu_1bit a8(.src1(src1[7]), .src2(src2[7]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[6]),
 .operation(ALU_control[1:0]), .result(tmp_res[7]), .cout(c[7]));

alu_1bit a9(.src1(src1[8]), .src2(src2[8]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[7]),
 .operation(ALU_control[1:0]), .result(tmp_res[8]), .cout(c[8]));

alu_1bit a10(.src1(src1[9]), .src2(src2[9]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[8]),
 .operation(ALU_control[1:0]), .result(tmp_res[9]), .cout(c[9]));

alu_1bit a11(.src1(src1[10]), .src2(src2[10]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[9]),
 .operation(ALU_control[1:0]), .result(tmp_res[10]), .cout(c[10]));

alu_1bit a12(.src1(src1[11]), .src2(src2[11]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[10]),
 .operation(ALU_control[1:0]), .result(tmp_res[11]), .cout(c[11]));

alu_1bit a13(.src1(src1[12]), .src2(src2[12]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[11]),
 .operation(ALU_control[1:0]), .result(tmp_res[12]), .cout(c[12]));

alu_1bit a14(.src1(src1[13]), .src2(src2[13]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[12]),
 .operation(ALU_control[1:0]), .result(tmp_res[13]), .cout(c[13]));

alu_1bit a15(.src1(src1[14]), .src2(src2[14]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[13]),
 .operation(ALU_control[1:0]), .result(tmp_res[14]), .cout(c[14]));

alu_1bit a16(.src1(src1[15]), .src2(src2[15]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[14]),
 .operation(ALU_control[1:0]), .result(tmp_res[15]), .cout(c[15]));

alu_1bit a17(.src1(src1[16]), .src2(src2[16]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[15]),
 .operation(ALU_control[1:0]), .result(tmp_res[16]), .cout(c[16]));

alu_1bit a18(.src1(src1[17]), .src2(src2[17]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[16]),
 .operation(ALU_control[1:0]), .result(tmp_res[17]), .cout(c[17]));

alu_1bit a19(.src1(src1[18]), .src2(src2[18]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[17]),
 .operation(ALU_control[1:0]), .result(tmp_res[18]), .cout(c[18]));
```



```verilog
alu_1bit a20(.src1(src1[19]), .src2(src2[19]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[18]),
 .operation(ALU_control[1:0]), .result(tmp_res[19]), .cout(c[19]));

alu_1bit a21(.src1(src1[20]), .src2(src2[20]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[19]),
 .operation(ALU_control[1:0]), .result(tmp_res[20]), .cout(c[20]));

alu_1bit a22(.src1(src1[21]), .src2(src2[21]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[20]),
 .operation(ALU_control[1:0]), .result(tmp_res[21]), .cout(c[21]));

alu_1bit a23(.src1(src1[22]), .src2(src2[22]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[21]),
 .operation(ALU_control[1:0]), .result(tmp_res[22]), .cout(c[22]));

alu_1bit a24(.src1(src1[23]), .src2(src2[23]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[22]),
 .operation(ALU_control[1:0]), .result(tmp_res[23]), .cout(c[23]));

alu_1bit a25(.src1(src1[24]), .src2(src2[24]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[23]),
 .operation(ALU_control[1:0]), .result(tmp_res[24]), .cout(c[24]));

alu_1bit a26(.src1(src1[25]), .src2(src2[25]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[24]),
 .operation(ALU_control[1:0]), .result(tmp_res[25]), .cout(c[25]));

alu_1bit a27(.src1(src1[26]), .src2(src2[26]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[25]),
 .operation(ALU_control[1:0]), .result(tmp_res[26]), .cout(c[26]));

alu_1bit a28(.src1(src1[27]), .src2(src2[27]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[26]),
 .operation(ALU_control[1:0]), .result(tmp_res[27]), .cout(c[27]));

alu_1bit a29(.src1(src1[28]), .src2(src2[28]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[27]),
 .operation(ALU_control[1:0]), .result(tmp_res[28]), .cout(c[28]));

alu_1bit a30(.src1(src1[29]), .src2(src2[29]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[28]),
 .operation(ALU_control[1:0]), .result(tmp_res[29]), .cout(c[29]));

alu_1bit a31(.src1(src1[30]), .src2(src2[30]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[29]),
 .operation(ALU_control[1:0]), .result(tmp_res[30]), .cout(c[30]));
```



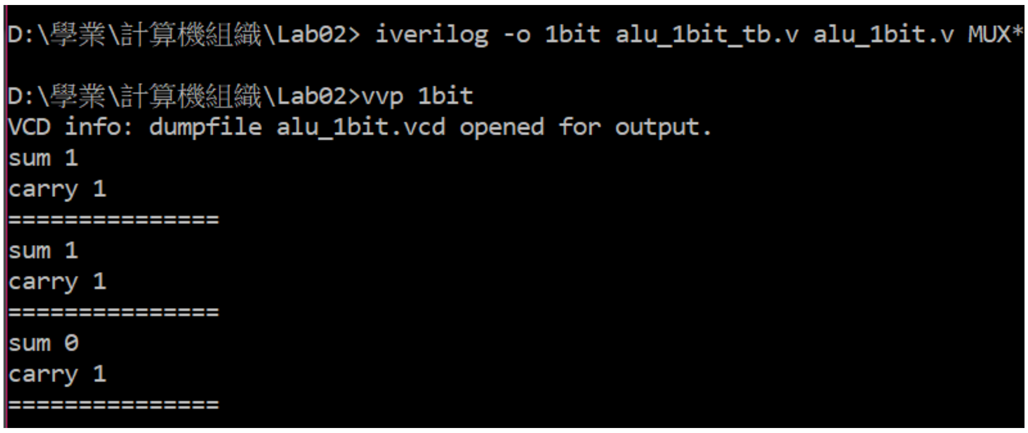
```verilog
alu_1bit a32(.src1(src1[31]), .src2(src2[31]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[30]),
 .operation(ALU_control[1:0]), .result(tmp_res[31]), .cout(c[31]));



alu_1bit fin(.src1(src1[31]), .src2(src2[31]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(c[30]),
 .operation(2'b10), .result(set), .cout(tmp_ct));

or (zero_fg,result[0],result[1],result[2],result[3],result[4],result[5],result[7],result[6],result[8],
result[9],result[10],result[11],result[12],result[13],result[14],result[15],result[16],result[17],
result[18],result[19],result[20],result[21],result[22],result[23],result[24],result[25],result[26],
result[27],result[28],result[29],result[30],result[31]);
```



```
always@(*)begin
    if(rst_n)begin
        if(ALU_control[3:0]==4'b0010 || ALU_control[3:0]==4'b0110) begin
            result <= tmp_res;
            zero <= ~zero_fg;
            cout <= c[31];
            overflow <= c[30]^c[31];
        end
        else if(ALU_control[3:0] == 4'b0111)begin
            if(set == 1'b1)begin
                result = 32'b00000000000000000000000000000001;
                zero = 1'b0;
            end
            else begin
                result = 32'b00000000000000000000000000000000;
                zero = 1'b1;
            end
            cout = 1'b0;
            overflow = 1'b0;
        end

        else begin
            result <= tmp_res;
            zero <= ~zero_fg;
            cout <= 1'b0;
            overflow <= 1'b0;
        end
    end
    else begin
        result <= 32'b00000000000000000000000000000000;
        zero <= 1'b0;
        cout <= 1'b0;
        overflow <= 1'b0;
    end
end
endmodule
```

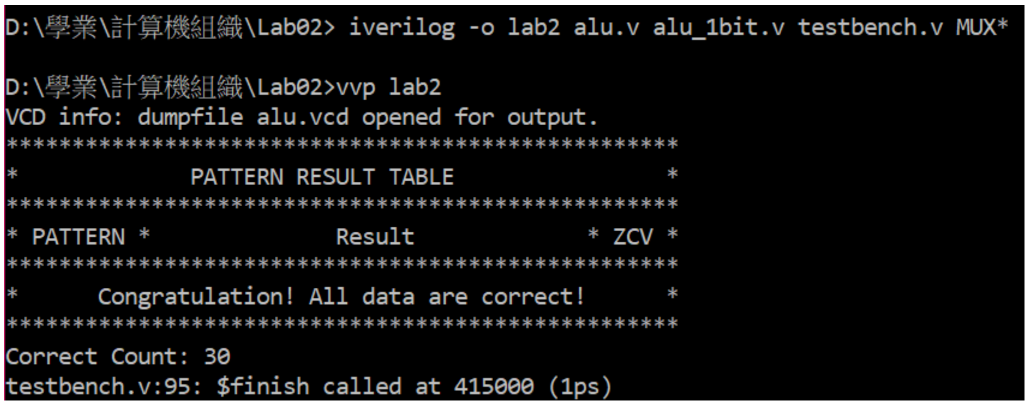
Part II. Implementation results :

1. 1-bit ALU :



```
D:\學業\計算機組織\Lab02> iverilog -o 1bit alu_1bit_tb.v alu_1bit.v MUX*

D:\學業\計算機組織\Lab02>vvp 1bit
VCD info: dumpfile alu_1bit.vcd opened for output.
sum 1
carry 1
===============
sum 1
carry 1
===============
sum 0
carry 1
===============
```

2. 32-bit ALU :



```
D:\學業\計算機組織\Lab02> iverilog -o lab2 alu.v alu_1bit.v testbench.v MUX*

D:\學業\計算機組織\Lab02>vvp lab2
VCD info: dumpfile alu.vcd opened for output.
***************************************************
*              PATTERN RESULT TABLE               *
***************************************************
* PATTERN *              Result              * ZCV *
***************************************************
*      Congratulation! All data are correct!      *
***************************************************
Correct Count: 30
testbench.v:95: $finish called at 415000 (1ps)
```

Part III. Problems encountered and solutions :

First problem I encountered is just like some other students , I had not used Verilog before . So it did take me for a while to get familiar with this language . The unfamiliarity make me forget to add certain wires frequently at first . However after I knew verilog more , I found out that it is literally using the language to represent the original diagram , which is easier than my expectation .

Second one is for 32-bit ALU . In the beginning , I had a hardtime dealing with the last 1-bit ALU , having no idea how to implement it in a different way as the slide says . However , after observing the original diagram , I found out that I only have to assign its operation as "2'b10" and two outputs as "set" and "tmp_ct" , which is quite intuitive .