Report-Lab02 Name:杜峯 ID:109550096

1. **Implement of "*alu_1bit.v*"**

```
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          set,        //1 bit set       (output)
    output reg          cout        //1 bit carry out (output)
    );
```

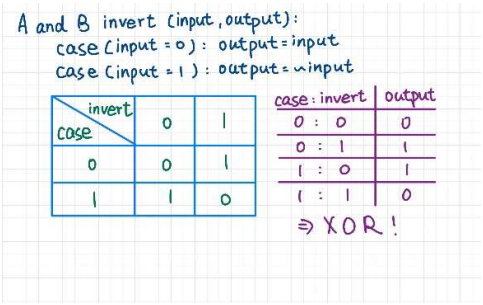
```
    output reg          result,     //1 bit result    (output)
```

Before describing the operation in this module, I have an additional output register called "*set*". This register is used in the "slt function" in "*alu.v*" and it will be described at section 2 and section 5.

```
wire src1_to_a,src2_to_b;

assign src1_to_a=src1^Ainvert;
assign src2_to_b=src2^Binvert;
```

In order to implement the "invert function", I assigned two wires with the XOR between the source input and the invert signal input.



The reason that using XOR is because if the input signal of invert is '0', the output should be as same as the source input. Relatively, if the input signal of invert is '1', the source should be inverted. This conclusion is based on the K-map in the left-side handwritten picture.

```
always @(*) begin
    case (operation)
        2'b00: begin
            result=src1_to_a&src2_to_b;
        end
        2'b01: begin
            result=src1_to_a|src2_to_b;
        end
        2'b10: begin
            {cout,result}=src1_to_a+src2_to_b+cin;
        end
        default:begin
            {cout,set}=src1_to_a+src2_to_b+cin;
            result=less;
        end
    endcase
end
```

For the main function, I used "*case*" to select which result is we want.

In the case "00", we can just assign the "and" operation with two wires "*src1_to_a*" and "*src2_to_b*" to the result.

Similar to the case "01", we can directly assign the "or" operation with two wires to the result.

Different to the above two case, I used a syntax "{}=" to implement the case "10", the "add" operation. The first bit of the summation of "*src1_to_a*", "*src2_to_b*" and "*cin*" will be assigned to the output register "*cout*", and the second bit will be assigned to the output register "*result*".

The last part of the case is for "11", the less operation. I use the register "*set*" instead of the "*result*" due to the result should be the value of the input register "*less*". The "*set*" register will record the value of the second bit of the summation of "*src1_to_a*", "*src2_to_b*" and "*cin*". It will be used in the "slt function" in "*alu.v*".

And the result of the testbench is "X1/X1/10", the carry and the sum. The first one is the result of the test case "a=1, b=1, operation=00(and)". The second one is for the case "a=1, b=1, operation=01(or)". The last one is for the case "a=1, b=1, operation=10(add)". All of the test cases are correct.

```
sum 1
carry x
===============
sum 1
carry x
===============
sum 0
carry 1
===============
```

2. Implement of "*alu.v*"

```
19  wire Ainvert,Binvert;
20  assign Ainvert=ALU_control[3];
21  assign Binvert=ALU_control[2];
22  wire [2-1:0] operation=ALU_control[1:0];
```

To implement 32-bits ALU, we will create 32 1-bit ALU. Each of them will have three control input registers, "*Ainvert*", "*Binvert*" and "*operation*". All of these are com from 4-bits "*ALU_control*". So, there are three wires to record the signal separated from "*ALU_control*".

```
24  wire [32-1:0] results;
25  wire [32-1:0] set;
26  wire cout0,cout1,cout2,cout3,cout4,cout5,cout6,cout7,cout8,cout9;
27  wire cout10,cout11,cout12,cout13,cout14,cout15,cout16,cout17,cout18,cout19;
28  wire cout20,cout21,cout22,cout23,cout24,cout25,cout26,cout27,cout28,cout29;
29  wire cout30,cout31;
30  reg cin_at_first_ALU_1bit;
31  reg less_first;
32  reg less=1'b0;
```

Then, I have assigned two 32-bits wires, "*result*" and "*set*", to record the output of these 32 1-bit ALU. The "set" is only used for the "slt function". And the 32 cout wire is for each 1-bit ALU to output the value of "*carry-out*". In the end, there are three register called "*cin_at_first_ALU_1bit*", "*less_first*" and "*less*". The first is used to be "*carry-in*" register for the first 1-bit ALU. In general, the value of this register is '0'. But if we want to implement the "sub" operation, this value will become '1'. The second, "*less_first*", is also used for the first 1-bit ALU. This is for the "slt function". The last register, "*less*", is for others 31 1-bit ALU. The value is always '0' and it also used for "slt function".

```
34  alu_1bit first  (src1[0],src2[0],less_first,Ainvert,Binvert,cin_at_first_ALU_1bit,operation,results[0],set[0],cout0);
35  alu_1bit one    (src1[1],src2[1],less,Ainvert,Binvert,cout0,operation,results[1],set[1],cout1);
36  alu_1bit two    (src1[2],src2[2],less,Ainvert,Binvert,cout1,operation,results[2],set[2],cout2);
37  alu_1bit three  (src1[3],src2[3],less,Ainvert,Binvert,cout2,operation,results[3],set[3],cout3);
38  alu_1bit four   (src1[4],src2[4],less,Ainvert,Binvert,cout3,operation,results[4],set[4],cout4);
39  alu_1bit five   (src1[5],src2[5],less,Ainvert,Binvert,cout4,operation,results[5],set[5],cout5);
40  alu_1bit six    (src1[6],src2[6],less,Ainvert,Binvert,cout5,operation,results[6],set[6],cout6);
41  alu_1bit seven  (src1[7],src2[7],less,Ainvert,Binvert,cout6,operation,results[7],set[7],cout7);
42  alu_1bit eight  (src1[8],src2[8],less,Ainvert,Binvert,cout7,operation,results[8],set[8],cout8);
43  alu_1bit nine   (src1[9],src2[9],less,Ainvert,Binvert,cout8,operation,results[9],set[9],cout9);
44  alu_1bit ten        (src1[10],src2[10],less,Ainvert,Binvert,cout9,operation,results[10],set[10],cout10);
45  alu_1bit eleven     (src1[11],src2[11],less,Ainvert,Binvert,cout10,operation,results[11],set[11],cout11);
46  alu_1bit twelve     (src1[12],src2[12],less,Ainvert,Binvert,cout11,operation,results[12],set[12],cout12);
47  alu_1bit thirteen   (src1[13],src2[13],less,Ainvert,Binvert,cout12,operation,results[13],set[13],cout13);
48  alu_1bit fourteen   (src1[14],src2[14],less,Ainvert,Binvert,cout13,operation,results[14],set[14],cout14);
49  alu_1bit fifteen    (src1[15],src2[15],less,Ainvert,Binvert,cout14,operation,results[15],set[15],cout15);
50  alu_1bit sixteen    (src1[16],src2[16],less,Ainvert,Binvert,cout15,operation,results[16],set[16],cout16);
51  alu_1bit seventeen  (src1[17],src2[17],less,Ainvert,Binvert,cout16,operation,results[17],set[17],cout17);
52  alu_1bit eighteen   (src1[18],src2[18],less,Ainvert,Binvert,cout17,operation,results[18],set[18],cout18);
53  alu_1bit nineteen   (src1[19],src2[19],less,Ainvert,Binvert,cout18,operation,results[19],set[19],cout19);
54  alu_1bit twenty         (src1[20],src2[20],less,Ainvert,Binvert,cout19,operation,results[20],set[20],cout20);
55  alu_1bit twenty_one     (src1[21],src2[21],less,Ainvert,Binvert,cout20,operation,results[21],set[21],cout21);
56  alu_1bit twenty_two     (src1[22],src2[22],less,Ainvert,Binvert,cout21,operation,results[22],set[22],cout22);
57  alu_1bit twenty_three   (src1[23],src2[23],less,Ainvert,Binvert,cout22,operation,results[23],set[23],cout23);
58  alu_1bit twenty_four    (src1[24],src2[24],less,Ainvert,Binvert,cout23,operation,results[24],set[24],cout24);
59  alu_1bit twenty_five    (src1[25],src2[25],less,Ainvert,Binvert,cout24,operation,results[25],set[25],cout25);
60  alu_1bit twenty_six     (src1[26],src2[26],less,Ainvert,Binvert,cout25,operation,results[26],set[26],cout26);
61  alu_1bit twenty_seven   (src1[27],src2[27],less,Ainvert,Binvert,cout26,operation,results[27],set[27],cout27);
62  alu_1bit twenty_eight   (src1[28],src2[28],less,Ainvert,Binvert,cout27,operation,results[28],set[28],cout28);
63  alu_1bit twenty_nine    (src1[29],src2[29],less,Ainvert,Binvert,cout28,operation,results[29],set[29],cout29);
64  alu_1bit thirty         (src1[30],src2[30],less,Ainvert,Binvert,cout29,operation,results[30],set[20],cout30);
65  alu_1bit thirty_one     (src1[31],src2[31],less,Ainvert,Binvert,cout30,operation,results[31],set[31],cout31);
```

Here is the 32 1-bit ALU, I created for implementing 32-bit ALU.

Looking at the first 1-bit ALU, we can find the input "*less_first*" and "*cin_at_first_ALU_1bit*". And the following 31 1-bit ALU, the "carry-in" input is the previous 1-bit ALU's "carry-out". For the

input wire of these 32 1-bit ALU, their "*Ainvert*", "*Binvert*", and "*operation*" are in the same value.

```
always @(*) begin
    if(!rst_n)begin
        result=0;
        zero=0;
        cout=0;
        overflow=0;
    end
    else begin
        // 0000 and / 0001 or / 0010 add / 0110 sub / 0111 slt / 1100 xor / 1101 xand
        case(ALU_control)
            4'b0000,4'b0001,4'b1100,4'b1101:begin
                cin_at_first_ALU_1bit=0;
                cout=0;
                result=results;
                overflow=0;
                less_first=1'b0;
                less=1'b0;
            end
            4'b0010:begin
                cin_at_first_ALU_1bit=0;
                cout=cout31;
                result=results;
                overflow=cout30^cout31;
                less_first=1'b0;
                less=1'b0;
            end
            4'b0110:begin
                cin_at_first_ALU_1bit=1;
                cout=cout31;
                result=results;
                overflow=cout30^cout31;
                less_first=1'b0;
                less=1'b0;
            end
            4'b0111:begin
                cin_at_first_ALU_1bit=1;
                cout=0;
                less_first=set[31];
                result=results;
                overflow=0;
            end
        endcase
        zero=|result;
        zero=~zero;
    end
end
```

In the main structure, it is necessary to check whether the reset operation is triggered or not. If it is triggered, set "*result*", "*zero*", "*cout*" and "overflow" to be '0'.

Then, I also used the "case" operation to determine which operation is going to be implemented. For the "**0000(and)**", "**0001(or)**", "**1100(NOR)**" and "**1101(NAND)**" operations, they are using the same structure in the 32-bit ALU. The difference between them is the "control". We just need to sent the right control bit to 32 1-bit ALU, then we can get the right result output.

In the "**0010(add)**" operation, we need to consider the "overflow". So, we implement the exclusive-or operation on "cout30" and "cout31". If they are different, we can say that this add operation is overflow. Similar to the "add" operation, we also take "cout30" and "cout31" to detect if it is overflow in the "**0110(sub)**" operation. But there is something special in this operation. The input "*cin_at_first_ALU_1bit*" is assigned to '1'. That's because when we want to implement A-B, we can transform it as A+(-B). With two's complement, -B is equal to the one's complement on B and add 1. The step of complement on B is control by the "*Binvert*" and the '1' will add by setting the value of "*cin_at_first_ALU_1bit*" as '1'. Then, the first 1-bit ALU will have a value of "carry-in" as '1'.

The last operation, "**0111(slt)**", is also set the input "*cin_at_first_ALU_1bit*" as '1'. Because if we want to check if A<B, we can transform is into checking if A-B<0. It also uses the idea of "sub" operation. Since the result of each 1-bit ALU is set to the "*less*" wire, I construct a new register called "*set*". It is used for recording the second bit of the summation of "*src1_to_a*", "*src2_to_b*" and "*cin*" in each 1-bit ALU. Then, I assigned the output "*set*" of the last 1-bit ALU to the "less_first". It will determine whether A is less than B or not. Because the "*less*" input of other 31 1-bit ALUs are '0', the result will be like "00000000000000000000000000000000(False)" or "00000000000000000000000000000001(True)".

```
109         zero=|result;
110         zero=~zero;
```

In the end of this "always" structure, we need to check if the result is zero or not. I use the "or" operation. If there is not any '1' bit

in the result register. The "*zero*" will be '0'. After being complemented, "*zero*" will be '1', which means it is true that the result is precisely zero. Relatively, if there is one or more '1' bit in the result, the "*zero*" will be '1'. With complementing, "*zero*" will be '0', which means the result is not equal to zero.

For the output of running this part is "*Congratulation! All data are correct!*"

```
VCD info: dumpfile alu.vcd opened for output.
*************************************************
*              PATTERN RESULT TABLE             *
*************************************************
* PATTERN *              Result            * ZCV *
*************************************************
*      Congratulation! All data are correct!    *
*************************************************
Correct Count: 30
```

3. Implement of "*MUX2to1.v*"

```
 9   always @(src1,src2,select) begin
10       result=(!select) ? src1:src2;
11   end
```

For implementing the multiplexier, if the select is '0(False)', it will assign "*src1*" to the result. Otherwise, the result will be assigned as "*src2*".

4. Implement of "*MUX4to1.v*"

```
15  always @(src1,src2,src3,src4,select) begin
16      case(select)
17          2'b00:begin
18              result=src1;
19          end
20          2'b01:begin
21              result=src2;
22          end
23          2'b10:begin
24              result=src3;
25          end
26          default:begin
27              result=src4;
28          end
29      endcase
30  end
```

In the 4-to-1 multiplexier, I use the "case" to switch the value of the select. With the value of "00", "01", "10" and "11(default)", it will assign "*src1*", "*src2*", "*src3*" and "*src4*" to the result register.

5. Problems encountered and solutions

For the first problem is that I only have a little idea about Verilog. Since I didn't take the course "數位電路實驗" at previous semester. So, it takes me a lot of time to learn how to program. Before I surf some website to learn it I consider it as a monster language. But now I think may not be that hard on implement. It has several similar syntaxes to the C language. For example, the "case" in Verilog and the "switch" in C. This is not a big problem to this lab, but it really cost me a lot of time

to solve it.

The second problem is how to use the “GTKwave”. I use the WSL-Ubuntu system at windows11. I consider that the operation is the same as the normal Ubuntu system in original. But after I compile the Verilog file, I input the command just like “gtkwave alu_1bit.vcd”. It told me that there is something wrong with GTKwave in the WSL-Ubuntu system. I original contributed this problem to the version of the GTKwave I had installed. But no matter what operation I did it always told me that there is something wrong. After explore on Internet, I discover the problem. That is, GTKwave is not support WSL-Ubuntu system. In order to use GTKwave to debug, I install GTKwave in windows system and call it in the “cmd”. Then, GTKwave can show on my computer successfully.

The third problem is about how to implement “slt” operation in the 32-bits ALU. For each 1-bit ALU, it only has two output register, “*result*” and “*cout*”. The “*result*” register is directly connected by the input wire “*less*”, which causes that it cannot output the second bit of the sum operation. With the answer by TA on HackMD said that we can assign additional input/output register to the 1-bit ALU, I came up with an idea that I could use a new output register to record the sum of “*src1_to_a*”, “*src2_to_b*” and “*cin*” in 1-bit ALU. Then, we could return the “*set*” output of the last 1-bit ALU to the first 1-bit ALU as the input “*less*”. Consequently, we could have the result correctly.