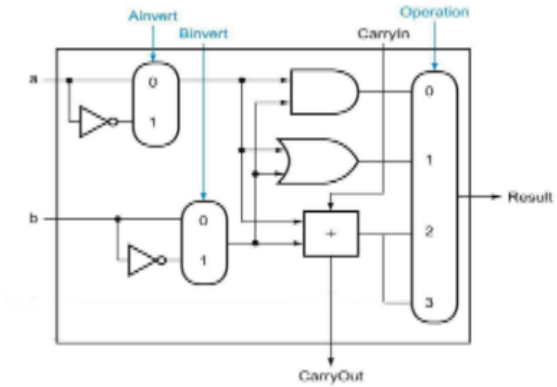
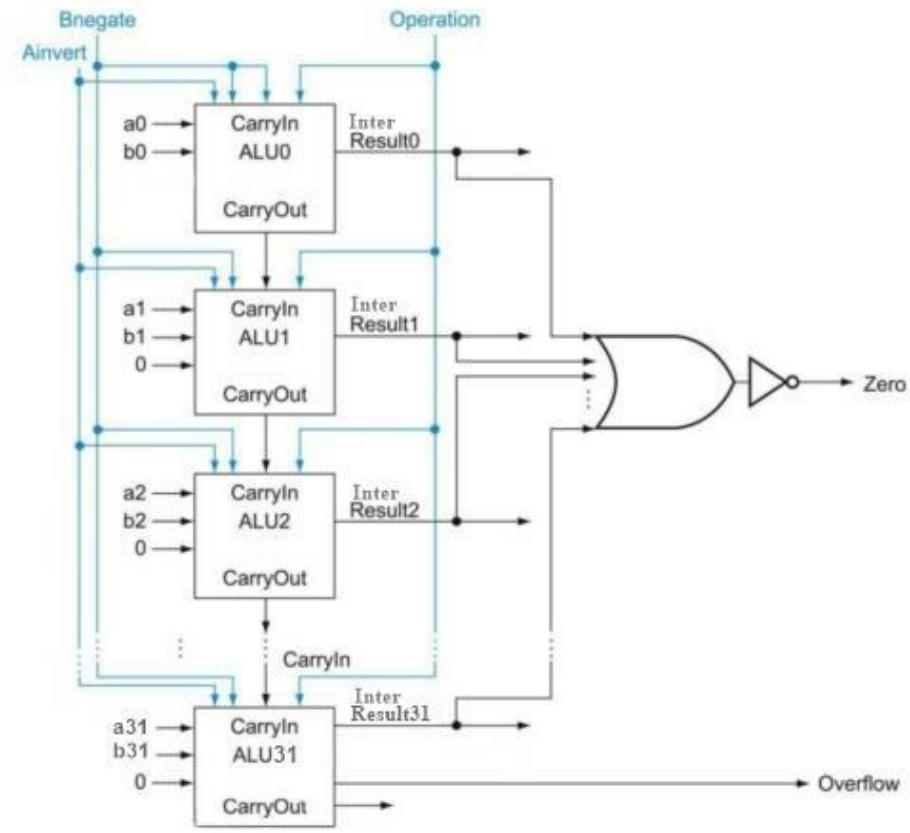
Computer Organization

Architecture diagram:

(1) 1-bit ALU



(2) 32-bit ALU



Detailed description of the implementation:

我先製作一個 **Full_Adder** 的 module 來實現 **1-bit ALU** 中的**算術運算**，把 Full_Adder 放入 1-bit ALU 的 module 中，再利用 case 選擇 ALU 輸出實現 behavioral implementation。

然後再 ALU module 中宣告 32 個 1-bit ALU。

module ALU_1bit Ai (src1, src2, Ainvert, Binvert, Cin, operation, result, cout)

其中：

1-bit ALU 的 **input**：

Src1[31:0] = src1[31:0]

Src2[31:0] = src2[31:0]

Ainvert [31:0] = ALU_control [3]

Binvert [31:0] = ALU_control [2]

Cin [0] = ALU_control[2] ; Cin [31:1] = CarryOut_wire [30:0]

operation [1:0] = ALU_control [1:0]

1-bit ALU 的 **output** 我先用 **wire** 接住值：

Result_wire [31:0] = result [31:0]

CarryOut_wire [31:0] = cout [31:0]

再把 wire 接住的值拿去做 if-else 的判斷：

1. **result** 輸出的結果分為 **SLT** 和 **Arithmetic Logic** 兩大類

   (1) **SLT**：if(a<b) result = 1; else result = 0

   因為 SLT 本身會執行減法，先把 result [31:1] = 0，再把 result_wire 的 Msb 值丟給 result 就好(result[0] = result_wire [31])。如果 src1-src2<0

的話，result_wire[31]會等於 1；如果 src1-src2>0 的話，result_wire [31]會等於 0，剛好都能當作 result[0]的輸出。

(2) **Arithmetic Logic** operation：add, sub, AND, OR, NAND, NOR

直接將 wire 的值存給 output reg (result [31:0] = result_wire [31:0])

2. Zero：if (result[31:0] == 0) zero=1;

將 result_wire 的 0~31 個 bit 做 NOR 邏輯運算，當全部 result_wire 的位元為 0 時，zero 會等於 1。

3. Cout 和 overflow：

把 cout 和 overflow 分為 Arithmetic operation 和 Logic operation 討論

(1) Logic operation：AND, OR, NAND, NOR

因為邏輯運算的結果不會產生 Carrylut 和 Overflow 的情形，所以直接令為 cout=0, overflow=0。

(2) Arithmetic operation：add, sub, SLT

算術運算(包含 SLT)就有可能會產生 cout 和 overflow，需要考慮。Cout 的想法比較簡單，直接把 wire 接到 output (cout = CarryOut_wire[31])

；overflow 則是把 cout 的 30 和 31 位元取 XOR 運算即可 。

(overflow = CarryOut_wire[30] ^ CarryOut_wire[31])

Implementation results:

1. Full_Adder

```
/****************************************************
Student Name:  劉紘華
Student ID:    0511330
****************************************************/
module Full_Adder (Sum, CarryOut, Input1, Input2, CarryIn);

  input Input1, Input2, CarryIn;
  output Sum, CarryOut;

  wire w1, w2, w3;
  xor x1(w1, Input1, Input2);
  xor x2(Sum, w1, CarryIn);
  and a1(w2, Input1, Input2);
  and a2(w3, w1, CarryIn);
  or o1(CarryOut, w2, w3);

endmodule
```

2. 1-bit ALU

```
/****************************************************
Student Name: 劉紘華
Student ID:   0511330
****************************************************/
`timescale 1ns/1ps

module ALU_1bit(
        input                           src1,       //1 bit source 1  (input)
        input                           src2,       //1 bit source 2  (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 ca, cb;
assign ca = src1 ^ Ainvert;
assign cb = src2 ^ Binvert;

wire And, Or, Sum, cout_wire;
and A1(And, ca, cb);
or O1(Or, ca, cb);

Full_Adder F1(.Sum(Sum), .CarryOut(cout_wire), .Input1(ca), .Input2(cb), .CarryIn(Cin));

always @* begin
  case(operation)
    2'b00: result = And;
    2'b01: result = Or;
    2'b10: result = Sum;
    2'b11: result = Sum;
  endcase

  cout <= cout_wire;
end

endmodule

```

3. 32-bit ALU

(Part 1) 宣告 wire 和 I/O

```
/*****************************************************
Student Name:  劉絃華
Student ID:    0511330
*****************************************************/
`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)
        );

/* Write your code HERE */
wire    CarryOut_wire[31:0];
wire    result_wire[31:0];
integer i;
```

(Part 2) 宣告 32 個 1-bit ALU

```
/* module  ALU_1bit (src1, scr2, Ainvert, Binvert, Cin, operation, result, cout) */
ALU_1bit A0(.src1(src1[0]), .src2(src2[0]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A1(.src1(src1[1]), .src2(src2[1]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A2(.src1(src1[2]), .src2(src2[2]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A3(.src1(src1[3]), .src2(src2[3]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A4(.src1(src1[4]), .src2(src2[4]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A5(.src1(src1[5]), .src2(src2[5]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A6(.src1(src1[6]), .src2(src2[6]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A7(.src1(src1[7]), .src2(src2[7]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A8(.src1(src1[8]), .src2(src2[8]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A9(.src1(src1[9]), .src2(src2[9]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Ci
ALU_1bit A10(.src1(src1[10]), .src2(src2[10]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A11(.src1(src1[11]), .src2(src2[11]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A12(.src1(src1[12]), .src2(src2[12]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A13(.src1(src1[13]), .src2(src2[13]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A14(.src1(src1[14]), .src2(src2[14]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A15(.src1(src1[15]), .src2(src2[15]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A16(.src1(src1[16]), .src2(src2[16]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A17(.src1(src1[17]), .src2(src2[17]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A18(.src1(src1[18]), .src2(src2[18]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A19(.src1(src1[19]), .src2(src2[19]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A20(.src1(src1[20]), .src2(src2[20]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A21(.src1(src1[21]), .src2(src2[21]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A22(.src1(src1[22]), .src2(src2[22]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A23(.src1(src1[23]), .src2(src2[23]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A24(.src1(src1[24]), .src2(src2[24]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A25(.src1(src1[25]), .src2(src2[25]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A26(.src1(src1[26]), .src2(src2[26]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A27(.src1(src1[27]), .src2(src2[27]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A28(.src1(src1[28]), .src2(src2[28]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A29(.src1(src1[29]), .src2(src2[29]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A30(.src1(src1[30]), .src2(src2[30]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
ALU_1bit A31(.src1(src1[31]), .src2(src2[31]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]),
```

(Part 3) behavirol implementation of **ZCV** and **SLT** function.

```
always @* begin
    zero <= ~(result_wire[0] | result_wire[1] | result_wire[2] | result_wire[3] | result_wire[4] | result_wire[5]
// Arithmetic: Add(0010), Sub(0110) & Conditional: SLT(0111)
    if(ALU_control[3:0] == 4'b0010 || ALU_control[3:0] == 4'b0110 || ALU_control[3:0] == 4'b0111) begin
        cout = CarryOut_wire[31];
        overflow <= CarryOut_wire[30]^CarryOut_wire[31];

    end
// Logical operation: AND, OR, NAND, NOR
    else begin
        cout = 0;
        overflow = 0;
    end


//SLT: if(src1<src2) result=1;  else result=0
    if(ALU_control[3:0] == 4'b0111) begin
        result[0] = result_wire[31];
        for(i=1; i<32; i=i+1) result[i] = 0;
    end

//other funcion: add, sub, AND, OR, NOR, NAND
    else begin
        for(i = 0; i<32; i=i+1) result[i] = result_wire[i];
    end

end

endmodule
```

4. Testbench result

```
# vsim -gui
# Start time: 19:55:34 on Apr 20,2020
# Loading work.testbench
# Loading work.alu
# Loading work.ALU_1bit
# Loading work.Full_Adder
VSIM 2> run -all
# ********************************************************
# *              PATTERN RESULT TABLE                    *
# ********************************************************
# * PATTERN *              Result              * ZCV *
# ********************************************************
# *      Congratulation! All data are correct!      *
# ********************************************************
# Correct Count:  9
# ** Note: $finish    : C:/Users/Heyward/Desktop/CO_Lab2/testbench.v(146)
#    Time: 205 ns  Iteration: 1  Instance: /testbench
# 1
# Break in Module testbench at C:/Users/Heyward/Desktop/CO_Lab2/testbench.v line 146
```

Problems encountered and solutions:

1. 遇到最難的部分是怎麼執行 SLT function，雖然知道把(src1-src2)減法結果的 Msb 拉回 result 的 Lsb 做輸出，但實作起來還是很困難。幸好後來想到把 SLT 和其他 function 分成兩類，再把 result 依不同情況執行。

   最後的解法為：

   (1) Result_wire 先接住 1-bit ALU 的運算結果。

   (2) 利用 ALU_control 判斷是否為 SLT

   (3) 如果是執行 SLT：result [31:1] = 0 ; result [0] = result_wire [31]

   (4) 其他的：直接把 wire 接給 output reg (result [31:0] = result_wire [31:0])

2. 執行減法時：src1 + src2 的 1's complement + 1，其中+1 從哪裡來我想了很久才發現，把 ALU_control[2] (Binvert) 拿去給最低為位元 ALU 的 Cin 當 input，只要執行減法時，Binvert 都會等於 1，剛好能當+1 的來源。

Lesson learnt (if any):

Verilog 好難！

差點寫不完作業之後要提早開始寫！

Comment:

這次作業最難的部分其實是不熟悉 verilog 的語法，希望之後能更清楚語法，而且完全搞懂 blocking 和 non-blocking 的邏輯意義和它們在 verilog 裡面語法區別。