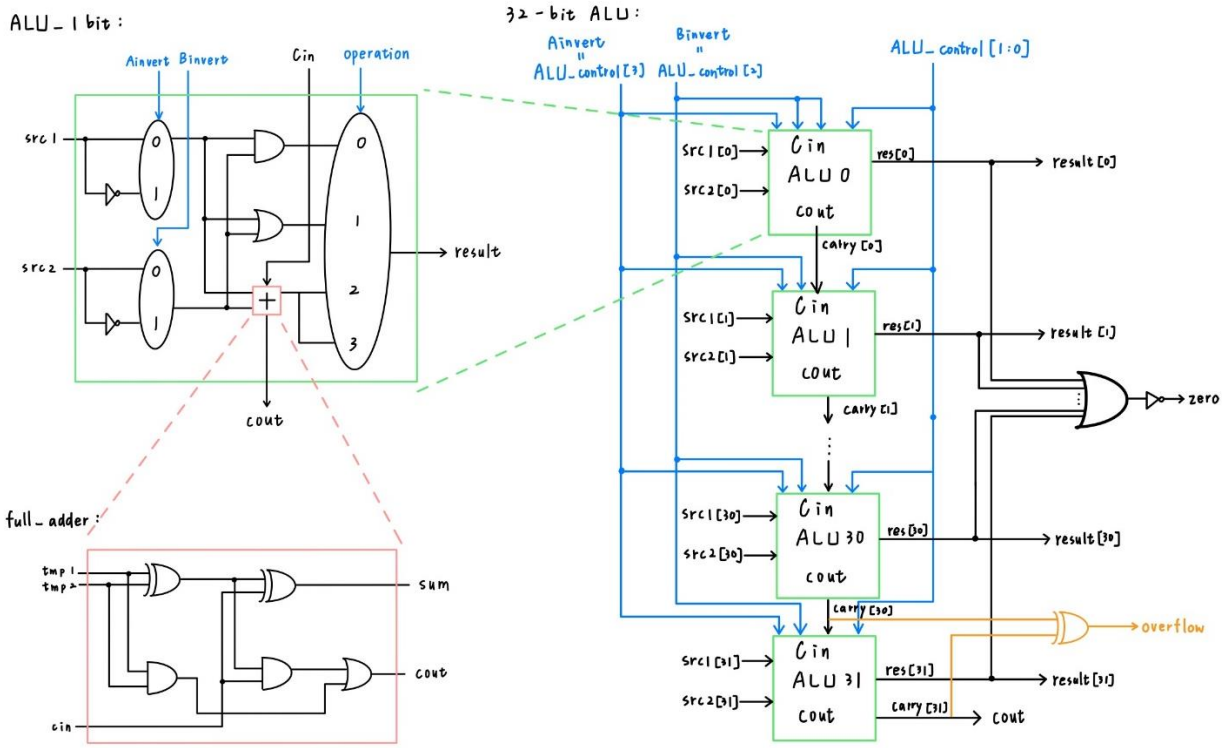
Computer Organization

Architecture diagram:



Detailed description of the implementation:

```
module full_adder(input tmp1,tmp2,cin, output sum,cout);
    wire w1,w2,w3;
    xor g1(w1,tmp1,tmp2);
    xor g2(sum,w1,cin);
    and g3(w2,w1,cin);
    and g4(w3,tmp1,tmp2);
    or  g5(cout,w2,w3);
endmodule
```

因為 ALU_1bit 需要做 add 及 sub，所以寫了一個 full adder 的 model

sum = tmp1⊕ tmp2 ⊕ cin

cout = (tmp1⊕tmp2)cin+tmp1tmp2

```
wire w1,w2,w3,w4,tmp1,tmp2;
xor g0(tmp1,src1,Ainvert);
xor g1(tmp2,src2,Binvert);
assign w3 = tmp1 & tmp2;
assign w4 = tmp1 |  tmp2;
full_adder full_adder(.tmp1(tmp1),.tmp2(tmp2),.cin(Cin),.sum(w1),.cout(w2));


always @( * ) begin

    case(operation)
    2'b00: begin
             result <=  w3;
             cout <= 0;
           end
    2'b01: begin
             result <= w4;
             cout <= 0;
           end
    2'b10: begin
             result<= w1;
             cout <= w2;
           end
    2'b11: begin
             result <= w1;
             cout <= w2;
           end
    endcase
end
```

因為無法得知 src1,src2 是否需要 invert，由 truth table 發現，將 src1 與 Ainvert 做 xor 以及將 src2 與 Binvert 做 xor，即為所需，再將兩個 xor 的結果存成 tmp1、tmp2，之後再進行 and、or、and、sub 的運算。因為 and 跟 or 不會有 carry out， 所以 cout 直接給 0。

當進行 add 與 sub 時，因為 output 不能直接接 register，所以先用兩個 wire 將 sum 跟 cout 的值接出來後再給 result 跟 cout。

```
reg t1, t2;
wire [32-1:0]carry;
wire [32-1:0]res;

ALU_1bit alu_0(.src1(src1[0]), .src2(src2[0]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(ALU_control[2]), .operation(ALU_control[1:0]), .result(res[0]), .cout(carry[0]));
ALU_1bit alu_1(.src1(src1[1]), .src2(src2[1]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[0]), .operation(ALU_control[1:0]), .result(res[1]), .cout(carry[1]));
ALU_1bit alu_2(.src1(src1[2]), .src2(src2[2]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[1]), .operation(ALU_control[1:0]), .result(res[2]), .cout(carry[2]));
ALU_1bit alu_3(.src1(src1[3]), .src2(src2[3]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[2]), .operation(ALU_control[1:0]), .result(res[3]), .cout(carry[3]));
ALU_1bit alu_4(.src1(src1[4]), .src2(src2[4]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[3]), .operation(ALU_control[1:0]), .result(res[4]), .cout(carry[4]));
ALU_1bit alu_5(.src1(src1[5]), .src2(src2[5]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[4]), .operation(ALU_control[1:0]), .result(res[5]), .cout(carry[5]));
ALU_1bit alu_6(.src1(src1[6]), .src2(src2[6]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[5]), .operation(ALU_control[1:0]), .result(res[6]), .cout(carry[6]));
ALU_1bit alu_7(.src1(src1[7]), .src2(src2[7]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[6]), .operation(ALU_control[1:0]), .result(res[7]), .cout(carry[7]));
ALU_1bit alu_8(.src1(src1[8]), .src2(src2[8]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[7]), .operation(ALU_control[1:0]), .result(res[8]), .cout(carry[8]));
ALU_1bit alu_9(.src1(src1[9]), .src2(src2[9]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[8]), .operation(ALU_control[1:0]), .result(res[9]), .cout(carry[9]));
ALU_1bit alu_10(.src1(src1[10]), .src2(src2[10]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[9]), .operation(ALU_control[1:0]), .result(res[10]), .cout(carry[10]));
ALU_1bit alu_11(.src1(src1[11]), .src2(src2[11]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[10]), .operation(ALU_control[1:0]), .result(res[11]), .cout(carry[11]));
ALU_1bit alu_12(.src1(src1[12]), .src2(src2[12]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[11]), .operation(ALU_control[1:0]), .result(res[12]), .cout(carry[12]));
ALU_1bit alu_13(.src1(src1[13]), .src2(src2[13]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[12]), .operation(ALU_control[1:0]), .result(res[13]), .cout(carry[13]));
ALU_1bit alu_14(.src1(src1[14]), .src2(src2[14]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[13]), .operation(ALU_control[1:0]), .result(res[14]), .cout(carry[14]));
ALU_1bit alu_15(.src1(src1[15]), .src2(src2[15]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[14]), .operation(ALU_control[1:0]), .result(res[15]), .cout(carry[15]));
ALU_1bit alu_16(.src1(src1[16]), .src2(src2[16]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[15]), .operation(ALU_control[1:0]), .result(res[16]), .cout(carry[16]));
ALU_1bit alu_17(.src1(src1[17]), .src2(src2[17]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[16]), .operation(ALU_control[1:0]), .result(res[17]), .cout(carry[17]));
ALU_1bit alu_18(.src1(src1[18]), .src2(src2[18]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[17]), .operation(ALU_control[1:0]), .result(res[18]), .cout(carry[18]));
ALU_1bit alu_19(.src1(src1[19]), .src2(src2[19]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[18]), .operation(ALU_control[1:0]), .result(res[19]), .cout(carry[19]));
ALU_1bit alu_20(.src1(src1[20]), .src2(src2[20]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[19]), .operation(ALU_control[1:0]), .result(res[20]), .cout(carry[20]));
ALU_1bit alu_21(.src1(src1[21]), .src2(src2[21]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[20]), .operation(ALU_control[1:0]), .result(res[21]), .cout(carry[21]));
ALU_1bit alu_22(.src1(src1[22]), .src2(src2[22]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[21]), .operation(ALU_control[1:0]), .result(res[22]), .cout(carry[22]));
ALU_1bit alu_23(.src1(src1[23]), .src2(src2[23]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[22]), .operation(ALU_control[1:0]), .result(res[23]), .cout(carry[23]));
ALU_1bit alu_24(.src1(src1[24]), .src2(src2[24]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[23]), .operation(ALU_control[1:0]), .result(res[24]), .cout(carry[24]));
ALU_1bit alu_25(.src1(src1[25]), .src2(src2[25]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[24]), .operation(ALU_control[1:0]), .result(res[25]), .cout(carry[25]));
ALU_1bit alu_26(.src1(src1[26]), .src2(src2[26]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[25]), .operation(ALU_control[1:0]), .result(res[26]), .cout(carry[26]));
ALU_1bit alu_27(.src1(src1[27]), .src2(src2[27]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[26]), .operation(ALU_control[1:0]), .result(res[27]), .cout(carry[27]));
ALU_1bit alu_28(.src1(src1[28]), .src2(src2[28]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[27]), .operation(ALU_control[1:0]), .result(res[28]), .cout(carry[28]));
ALU_1bit alu_29(.src1(src1[29]), .src2(src2[29]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[28]), .operation(ALU_control[1:0]), .result(res[29]), .cout(carry[29]));
ALU_1bit alu_30(.src1(src1[30]), .src2(src2[30]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[29]), .operation(ALU_control[1:0]), .result(res[30]), .cout(carry[30]));
ALU_1bit alu_31(.src1(src1[31]), .src2(src2[31]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .Cin(carry[30]), .operation(ALU_control[1:0]), .result(res[31]), .cout(carry[31]));
```

```
always @( carry, result ) begin
    if(rst_n == 1) begin
        cout <= carry[31];
        zero <= (result == 0)? 1:0;
    end
end

always @( carry ) begin
    if( rst_n ) begin
        if(carry[30]^carry[31]==1) overflow <= 1;
        else overflow <= 0;
    end
    else begin
        overflow <= 0;
    end
end

always @( ALU_control, res ) begin
    if( rst_n ) begin
       if(ALU_control == 4'b0111 && res[31]==1) begin
            result[0] <= 1;
            result[31:1] <= 0;
        end
        else if (ALU_control == 4'b0111 && res[31]==0) result[31:0] <= 0;
        else result[31:0] <= res[31:0];
    end
    else begin
       result <= 0;
    end
end
```

1 個 32-bit ALU 是由 32 個 1-bit ALU 所組成的，所以一開始先宣告了 32 個 1-bit ALU，將值傳入每個 1-bit ALU，ALU_control[3] 就代表 Ainvert，ALU_control[2] 就代表 Binvert，ALU_control[1:0]即為 ALU_1-bit 的 operation，且只有減法出現時一開始的 carry_in 要為 1，而此時的 Binvert 也會為 1，所以就將 Binvert 傳入一開始的 carry_in，

因為 output 不能直接接 register，所以先用兩個 wire res[31:0]、carry[31:0] 接出 output，之後再傳給 result[32-1:0]，cout 則是只有 carry[31]=1 時才會等於 1，zero 則是 result 的每個 bit 都為 0 時才會等於 1。

Overflow 有 4 種可能發生的情況，可以發現在第 31 個 bit 的 carry_in 與 carry_out 不同時會發生 overflow，因此將這兩個 bit 做 xor 即可知道是否 overflow。

當 ALU_control 為 4’b0111 時代表要做 slt ，res[31]為 src1[31] – src2[31]的結果，即為最高位元相減的結果，如果是 1 就代表 src1 < src2， 就將 result 的第 31 個 bit 改成 1， 0 ~ 30 個 bit 改成 0，否則就將 result 的 32 個 bit 都改成 0 。

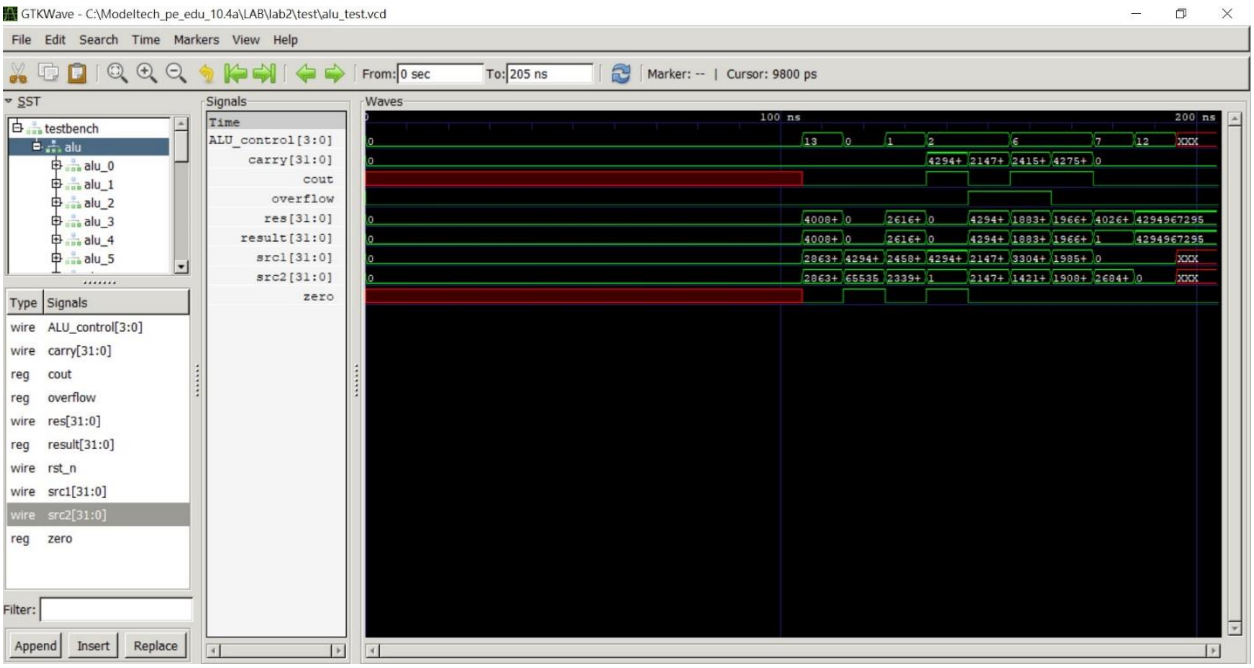
Implementation results:

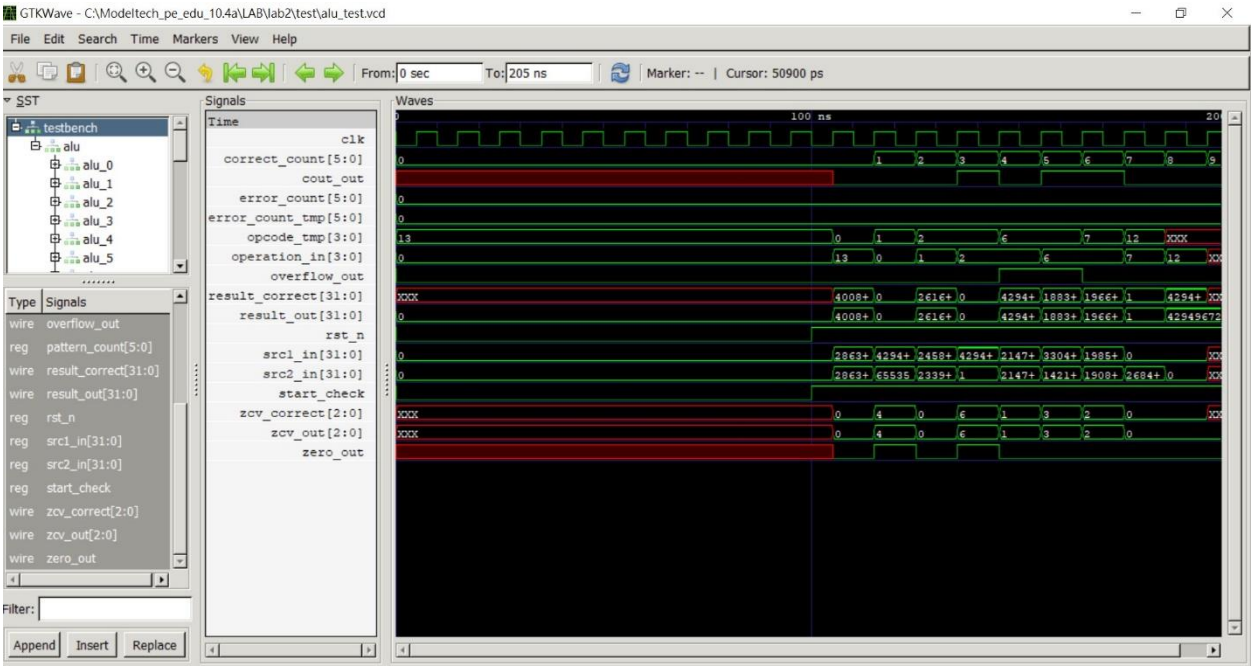
```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.18363.720]
(c) 2019 Microsoft Corporation. 著作權所有，並保留一切權利。

C:\Modeltech_pe_edu_10.4a\LAB\lab2\test>iverilog -o test.vvp ../*.v

C:\Modeltech_pe_edu_10.4a\LAB\lab2\test>vvp test.vvp
VCD info: dumpfile alu_test.vcd opened for output.
*****************************************************
*            PATTERN RESULT TABLE                   *
*****************************************************
* PATTERN *              Result              * ZCV *
*****************************************************
*      Congratulation! All data are correct!        *
*****************************************************
Correct Count:  9

C:\Modeltech_pe_edu_10.4a\LAB\lab2\test>gtkwave
```





Problems encountered and solutions:

(1) 一開始在寫 1-bit ALU 的時候不知道不能在 always block 裡面 call model， 造成一開始 compile 一直不能過，後來才發現要寫在外面。

(2) 一開始寫 slt 的時候我寫成把每個 bit 都拿來比較，因此出來的答案一直是錯的，後來才發現只要比最高位元就好了。

(3) 寫完之後 simulate 時會有類似無限迴圈的狀況，詢問助教後才知道不建議在給值之後又在後面更動，否則可能會有這種情況發生。

Lesson learnt (if any):

(1) wire 跟 register 的差別

(2) assign 只能寫在 always 外面，而且只能對 wire 做 assign

(3) 通常都會用 nonblocking 而非 blocking

Comment:

透過這次的作業讓我開始熟悉 verilog ，因為之前完全沒有寫過 verilog ，所以一開始的時候不知道要從哪裡下手，覺得很恐慌，但後來上網查了一些 verilog 的語法，好像有點感覺後才逐漸開始完成這次的作業，過程中雖然遇到蠻多小問題，但在跟同學討論後貌似都有解決。