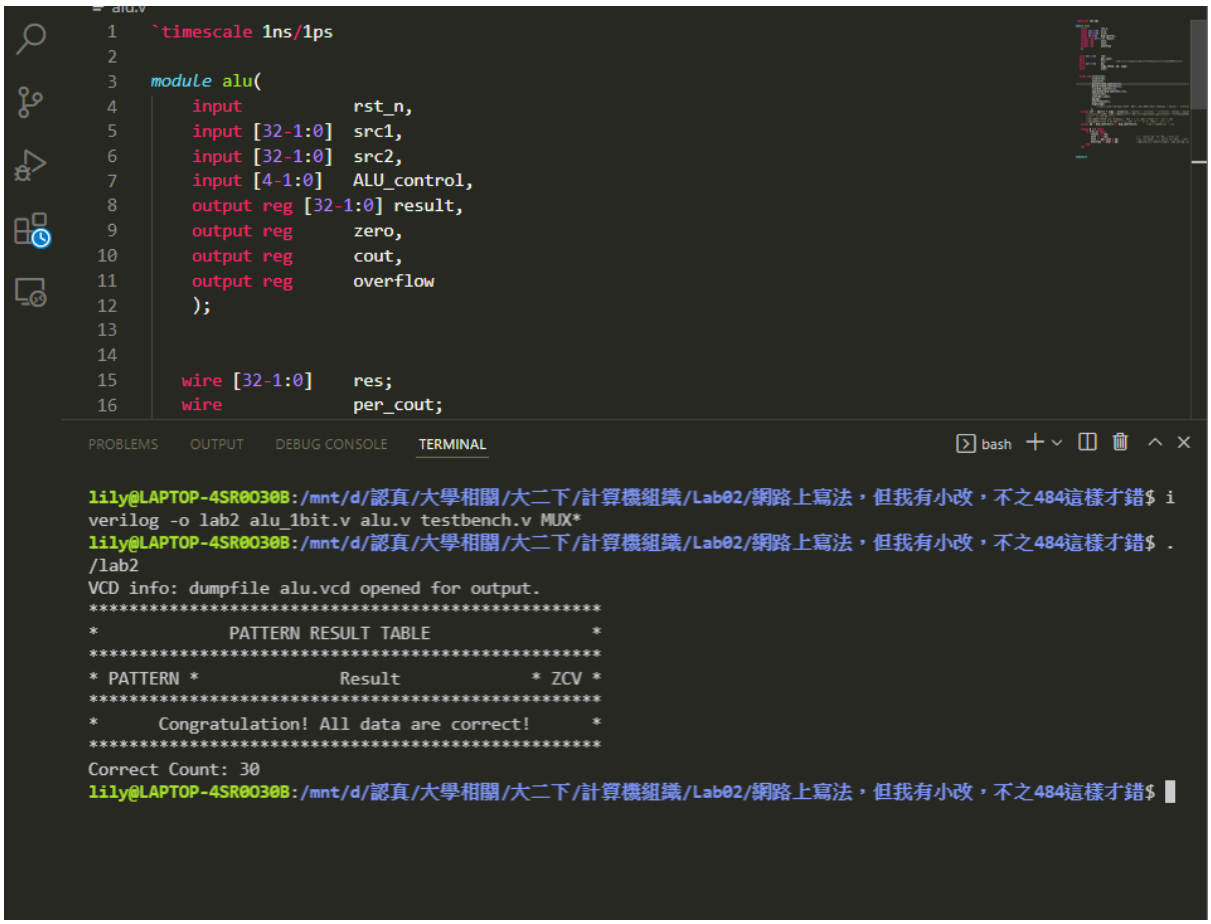
Lab02 32-bit ALU Report

109550053楊立嘉

PART 1. Result with Ubuntu

- using WSL in VS code to do implementation



PART 2 . Implementation Detail

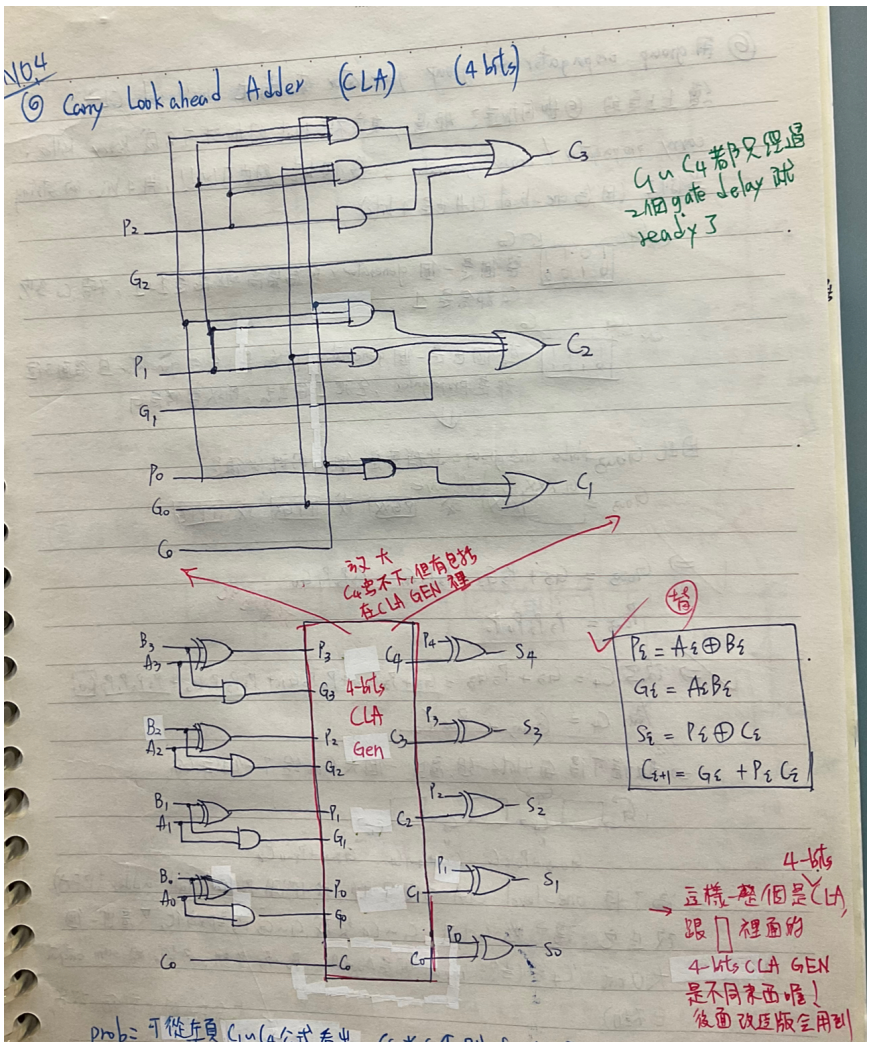
Note that we can not add more `.v` file, so I write every modules aside from 32-bits ALU in `alu_1bit.v` file. In this part, I will screenshot every modules separately and explain it in detail.

Overall of my implementation :

I finish 32-bits ALU by hierarchical structure, that is, use 32-bit carry look ahead (CLA) generator to do 32-bit ALU, because ripple carry adder (RCA) have delay from waiting previous `Cout` as `Cin` of this bit.

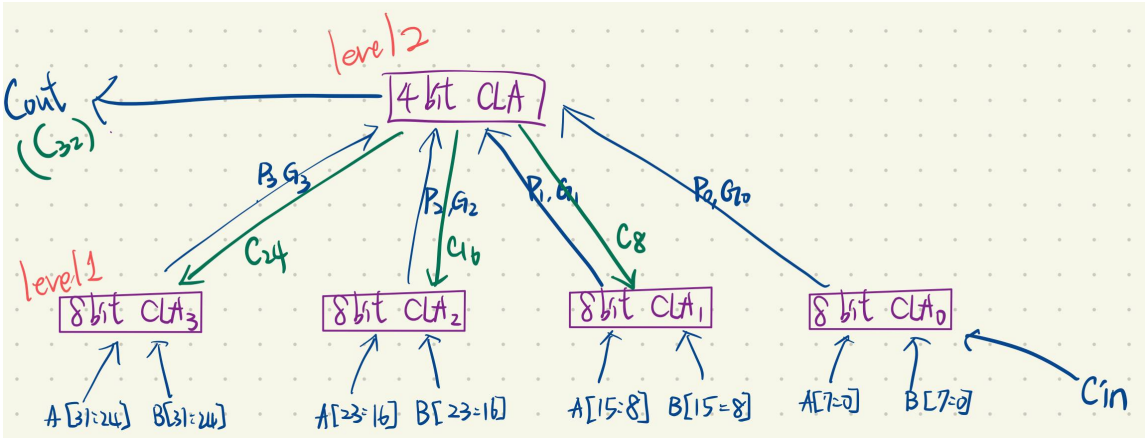
As study in class, I record propagator `P` and generator `G` for every 8-bit ALU, and of course small `p_i` and small `g_i` for every 1-bit ALU. Also implement 8-bit carry lookahead generator (`cla8` module) for eight 1-bit ALU in level1, 4-bit carry lookahead generator (`cla4` module) for four 8-bit ALU in level2.

- Picture below shows the circuit for `cla4` module, and `cla8` module just extended by `cla4.`



$$
\begin{aligned}
c1 &= g0 + (p0 \cdot c0) \Rightarrow 2 \text{ gate delay} \\
c2 &= g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0) \text{ 再OR起来} \Rightarrow 2 \\
c3 &= g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0) \Rightarrow 2 \\
c4 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\
&\quad + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0) \Rightarrow 2
\end{aligned}
$$

- Picture below shows the total hierarchical structure of my implementation.



- Summary of all 6 modules :
  1. alu_1bit : 1-bit ALU, level 1 in my hierarchical structure
  2. cla4 : 4-bit carry lookahead generator
  3. cla8 : 8-bit carry lookahead generator
  4. alu_8bit : 8-bit ALU, level 2 in my hierarchical structure
  5. alu32 : “fake” 32-bit ALU, just for adjustment to TAs’ input output in `alu.v` , this module input the separated `ALU_control` parameter, that is, `Ainvert`, `Binvert` and `operation` ,and .return calculating results and information I needed.
  6. alu : “real” 32-bibt ALU, return all `zcv` info. The only module that is implemented in `alu.v` .

- about basic instruction set :

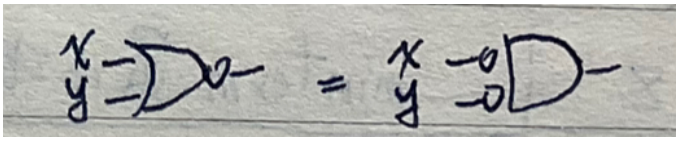
(2.) Basic instruction set (70%)

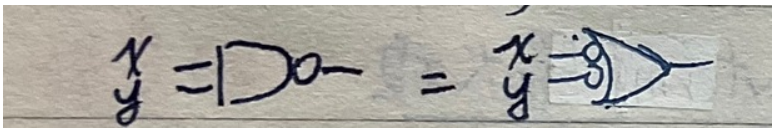
ALU action	Function	ALU control input
and	AND	0000
or	OR	0001
add	Addition	0010
sub	Subtract	0110
slt	Set less than	0111
nor	NOR	1100
nand	NAND	1101

1. First implement 1-bit ALU, so I can do AND, OR and ADD.
2. SUB :

   Use `Binvert` to control subtract function, because $A - B = A + (-B) = A + \bar{B} + 1$. Plus one is input from Cin.
3. NOR、NAND :

   Like the picture below shows, doing NOR on A and B just like doing AND on $\bar{A}$ and $\bar{B}$; doing NAND on A and B just like doing OR on $\bar{A}$ and $\bar{B}$.





4. SLT :

   Just like professor taught in class, we use wire `set` (int my code, is called wire `slt`) to control SLT signal.

   - If $A < B$ then `slt` is set, that is :
     - If A and B have opposite sign, one positive and one negative, then just check if A is negative one to male sure $A < B$, `slt` is equal to MSB of A.

- If A and B have same sign, consider $A < B$ as $A - B < 0$, so just check the sign of result of subtraction. I have implement subtract in previous part, so we can use this idea directly.

- `assign slt = eq[31] ? src1[31] : sign;`

  In 1-bit ALU I will maintain `eq[i] = ~(in1[i] ^ in2[i])` to record if each bit of `src1` and `src2` is equal and add `eq` as a return output by myself, so in `alu.v` just use `eq[31]` to check if `src1` and `src2` have same sign.

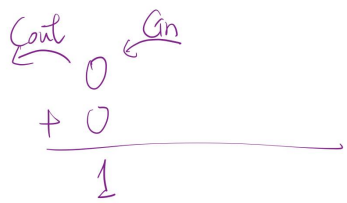
  As for `sign` variable, calculated in `alu32` module : `assign Sign = ~eq[31] ^ over[6];` to represent the result sign of ALU calculation, where `over[6]` is Cin of MSB. This assignment is equal to `assign Sign = ~( src1[31] ^ src2[31]) ^ over[6]` , just like `sum` defined in full adder.

  According to the table below, `sign = 1` only when `src1` shares same sign with `src2` and `Cin = over[6] = 1`.

~eq[31] \ over[6]	0	1
0	0	1
1	1	0

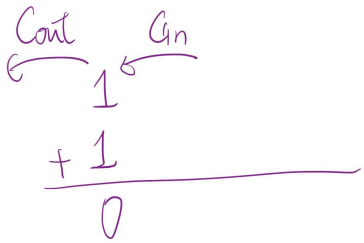
- overflow detection
  - There are two kinds of overflow
    - Two positive numbers add up and get a negative answer.

      You can see from the picture below, two zeros add up, `Cout` always equal to 0, if we don't want overflow to appear, `Cin` of MSB must equal to 0 too.



    - Two negative numbers add up and get a positive answer

      You can see from the picture below, two ones add up, `Cout` always equal to 1, if we don't want overflow to appear, `Cin` of MSB must equal to 1 too.



So combine this two condition, we know that overflow occurs only when `Cout` and `Cin` of MSB are different.

→ `assign overflow = per_cout[3] ^ over[6];` ,that is `V = Cout ^ Cin` .

module1 - alu_1bit

alu_1bit.v

```
`timescale 1ns/1ps
module alu_1bit(        //7 input 、5 output
    input               src1,
    input               src2,
    input               less,       //如果以做slt ，這邊就是1，那result就直接是1
    input               Ainvert,
    input               Binvert,
    input               cin,
    input       [2-1:0] operation,

    output reg          result,
    output              cout,
    output              p,
    output              g,
    output              eq
    );


    wire _and,_or_add;
    wire in1,in2;

    assign in1 = Ainvert ^ src1;
    assign in2 = Binvert ^ src2;

    assign _and = in1 & in2; // op=0
    assign _or = in1 | in2;  // op=1
    assign _add = in1 ^ in2 ^ cin; //op=2，就是sum拉，做 XOR。

    assign p = _or;
    assign g = _and;
    assign eq = ~(in1 ^ in2);//判斷AB是否相同，如果一樣的畫 eq = 1
    assign cout = (in1 & in2) | (in1 & cin) | (in2 & cin);

    always @ (*)
        begin
            case(operation)
                2'b00: result = _and;
                2'b01: result = _or;
                2'b10: result = _add;
                2'b11: result = less;
            endcase
        end

endmodule   //1 bit alu
```

module2 - cla4 (4-bit carry lookahead generator)

```
module cla4(        //3 input 、 1 output
    input [4-1:0]   G,
    input [4-1:0]   P,
    input           cin,
    output [4-1:0]  cout
    );

    assign cout[0] = G[0] | (P[0] & cin);
    assign cout[1] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & cin);
    assign cout[2] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] & P[0] & cin);
    assign cout[3] = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] & P[1] & G[0]) | (P[3] & P[2] & P[1] & P[0] & cin);
endmodule   //cla4

```

module3 - cla8 (8-bit carry lookahead generator)

```
//-----------------------------------------------------------------------------------------------
module cla8 (   //3 input 3output
    input [8-1:0] G,
    input [8-1:0] P,
    input   cin,
    output [8-1:0] cout,
    output  Pout,
    output  Gout
    ) ;

   wire   pw, gw;

   assign cout[0] = G[0] | (P[0] & cin);
   assign cout[1] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & cin);
   assign cout[2] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] & P[0] & cin);
   assign cout[3] = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] & P[1] & G[0]) | (P[3] & P[2] & P[1] & P[0] & cin);
   assign cout[4] = G[4] | (P[4] & G[3]) | (P[4] & P[3] & G[2]) | (P[4] & P[3] & P[2] & G[1]) | (P[4] & P[3] & P[2] & P[1] & G[0]) | (P[4] & P[3] & P[2] & P[1] & P[0] & cin);
   assign cout[5] = G[5] | (P[5] & G[4]) | (P[5] & P[4] & G[3]) | (P[5] & P[4] & P[3] & G[2]) | (P[5] & P[4] & P[3] & P[2] & G[1]) | (P[5] & P[4] & P[3] & P[2] & P[1] & G[0]) | (P[5] & P[4] & P[3] & P[2] & P
   assign cout[6] = G[6] | (P[6] & G[5]) | (P[6] & P[5] & G[4]) | (P[6] & P[5] & P[4] & G[3]) | (P[6] & P[5] & P[4] & P[3] & G[2]) | (P[6] & P[5] & P[4] & P[3] & P[2] & G[1]) | (P[6] & P[5] & P[4] & P[3] & P

   assign gw = G[7] | (P[7] & G[6]) | (P[7] & P[6] & G[5]) | (P[7] & P[6] & P[5] & G[4]) | (P[7] & P[6] & P[5] & P[4] & G[3]) | (P[7] & P[6] & P[5] & P[4] & P[3] & G[2]) | (P[7] & P[6] & P[5] & P[4] & P[3] &
   assign pw = (P[7] & P[6] & P[5] & P[4] & P[3] & P[2] & P[1] & P[0]);

   assign cout[7] = gw | (pw & cin);

   assign Pout = pw;
   assign Gout = gw;

endmodule // 8 bit cla generator cla8

```

module4 - alu_8bit

```
module alu_8bit ( //7 input 5 outpur ，跟 1bit alu 一樣
    input [8-1:0]   src1,
    input [8-1:0]   src2,
    input           less,
    input           Ainvert,
    input           Binvert,
    input           cin,
    input [2-1:0]   operation,

    output [8-1:0]  result,
    output [8-1:0]  cout,
    output          P,
    output          G,
    output [8-1:0]  eq          //存 8 bit 是否bitwise相同的結果，注意 eq = 0 才代表兩bit相同
    );

    wire [8-1:0]    per_cout;
    wire [8-1:0]    p, g;

    cla8 cl(.G(g),
            .P(p),
            .cin(cin),
            .cout(per_cout),
            .Pout(P),
            .Gout(G)
            );


    alu_1bit a1(…
    alu_1bit a2(…
    alu_1bit a3(…
    alu_1bit a4(…
    alu_1bit a5(…
    alu_1bit a6(…
    alu_1bit a7(…
    alu_1bit a8(…
   assign cout = per_cout;

endmodule // alu8
```

module5 - alu32

```
module alu32 ( //7 input、5 output
    input [32-1:0]  src1,
    input [32-1:0]  src2,
    input           less,
    input           Ainvert,
    input           Binvert,
    input           cin,
    input [2-1:0]   operation,

    output [32-1:0] result,
    output          cout,
    output [32-1:0] eq,         //回傳AB的每一個 bit 是否相同，注意相同的話 eq 值是0
    output          overflow,   //也就是ZCV中的V
    output          Sign
    );

   wire [4-1:0]     per_cout;
   wire [4-1:0]     p, g;
   wire [8-1:0]     over;       //存放最高位的那個 8bit alu 中的 MSB 的每一個 bit 的 cin，但其實我們只關心 32 bit中的


    //top level carry look ahead
    cla4 cl(
        .G(g),
        .P(p),
        .cin(cin),
        .cout(per_cout)
        );

    alu_8bit e1(…
    alu_8bit e2(…
    alu_8bit e3(…
    alu_8bit e4(…
    assign cout = per_cout[3];
    assign overflow = per_cout[3] ^ over[6]; //V = MSB的cout xor MSB的cin，原因請看數電筆記
    assign Sign = ~eq[31] ^ over[6];          //代表算術結果的正負號，也就是result[31]是0還是1
    //eq[31] == 1 代表 AB 的 MSB 相同、 over[6]則是 MSB 的Cin。
    //劃出2*2表格就知道了，只有當 AB 同號且Cin = 1時，結果才會是1。就跟full adder裡面定義的 sum = A^B^Cin 一樣
    //  ~eq\over  |    0    |    1
    //  0         |    0    |    1
    //  1         |    1    |    0

endmodule // alu32
```

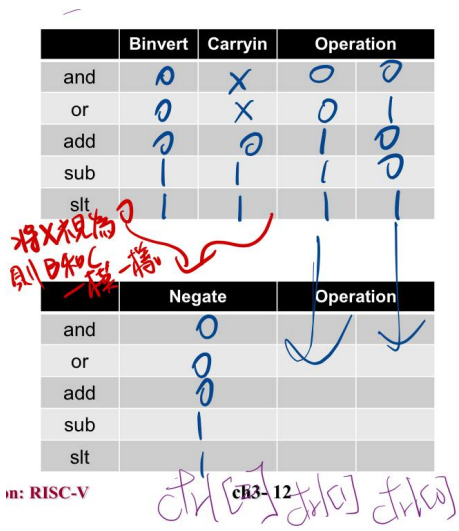
module6 - alu (in `alu.v`)

alu.v

```
`timescale 1ns/1ps
module alu(
    input           rst_n,
    input [32-1:0]  src1,
    input [32-1:0]  src2,
    input [4-1:0]   ALU_control,
    output reg [32-1:0] result,
    output reg      zero,
    output reg      cout,
    output reg      overflow
    );
    wire [32-1:0]   res;
    wire            per_cout;
    wire            slt;        //就是傳回去的set，但其實set不只有slt，我簡化成只做slt
    wire [32-1:0]   eq;
    wire            sign_check, ad, sign;
    wire            over;

    alu32 a1(.src1(src1),
            .src2(src2),
            .less(slt),
            .Ainvert(ALU_control[3]),
            .Binvert(ALU_control[2]),   //ALU_ctrl同時代表 Binvert 跟 cin 的原因在老師PPT有教
            .cin(ALU_control[2]),
            .operation(ALU_control[1:0]),
            .result(res),
            .cout(per_cout),
            .eq(eq),
            .overflow(over),
            .Sign(sign)     //sign回傳的是計算結果的正負號，也就是MSB是0還是1。Sign = eq[31] ^ over[6];
            );
    assign slt = eq[31] ? src1[31] : sign; //eq[31] = src1[31] ^ ~src2[31];  、Sign = eq[31] ^ over[6];
        //in1、in2同異號? 異號:同號。本來他前面是slt，但我不會用到其他Set，所以直接簡化成只做slt
        //slt : 當 A<B 時設為 1      //AB同號時，考慮減法出來的結果, MSB = 1 ，也就是 A-B<0 ，則 slt = 1。
                                     //AB異號時，只要 A 的 MSB = 1 ，也就是 A 為負且 B 為正，則 slt = 1
    assign ad = ALU_control[1] & ~ALU_control[0];       //確認是否正在做加減法
    always @ (*) begin
        if (rst_n) begin
            result <= res;
            zero <= ~|res;                      //將 result 的所有位數拿出來做 nor
            cout <= per_cout & ad;              //由 cla adder 拿出 carry out 之後, 再確認是否是作加減法
            overflow <= over & ad;              //加了& ad，因為做加剪髮才會需要判斷overflow
        end
    end
endmodule
```

- input of alu32 module
    - According to course slide, we know that `Binvert` and `Cin` can use same control input, so `Negate` and `Operation` only take three bits. The one bit left is `Ainvert`, this is for NOR and NAND calculation, which I have mentioned in previous part.



- ZCV :
  - check zero : just take every bits of result out and NOR together.
  - check carryout : take out from CLA adder and if operation is doing ADD or SUB, then set.
  - check overflow : write in previous part.

PART 3. Experience

The first problem I encountered is unfamiliarity with Ubuntu, setting up environment took me lots of time.

In implementation, I think overflow and hierarchical structure are most difficult. I took out my note of Digital Circuit Design (數位電路設計) and studied one more time. Lots of pictures in this report are notes at that time, the things about how to do overflow detection, what is the circuit of CLA and hierarchical CLA, are both coming from this previous course.

Also, I googled about ALU implementation, none of them take `Ainvert` into account, it took me some time to find out what is `Ainvert` for and how to implement NOR, NAND.

I think Lab2 help me to review almost everything taught in Digital Circuit Design, so it will let me understand better about our Chapter 3.