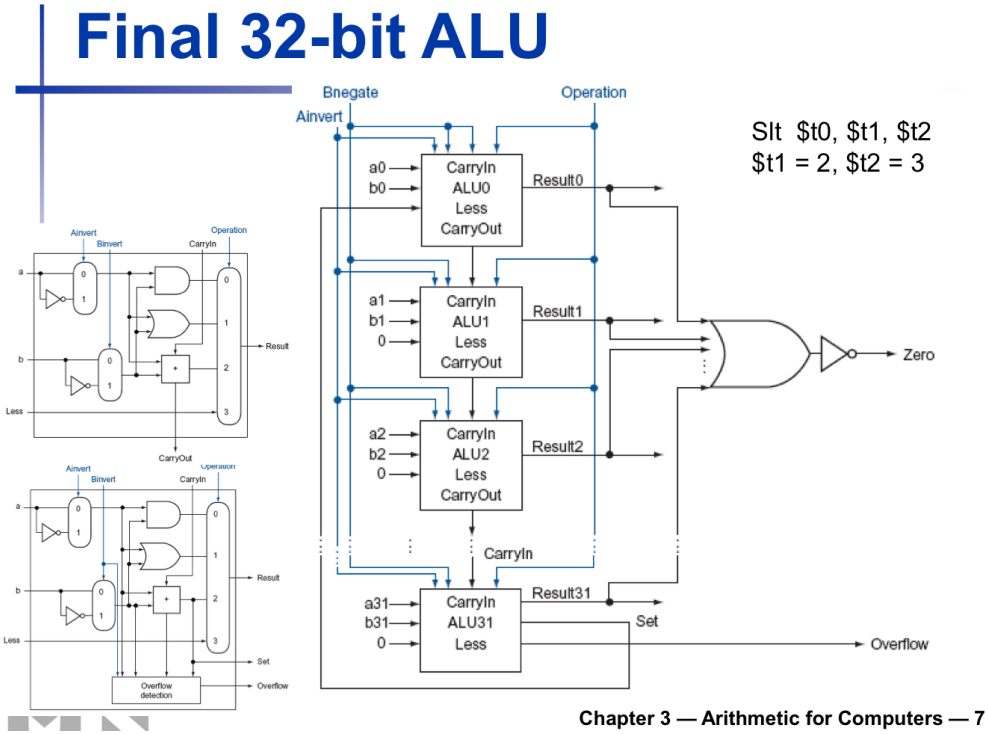
Computer Organization

Architecture diagrams:



主要參照上圖架構，左為alu_top.v，右為alu.v。

Hardware module analysis:

我使用兩個module實作，alu_top.v和alu.v，輸入的部分只有第0個不太一樣，如圖：

```
57  alu_top alu0(
58                  src1[0],       //1 bit source 1 (input)
59                  src2[0],       //1 bit source 2 (input)
60                  slt[31],       //1 bit less     (input)
61                  ALU_control[3],   //1 bit A_invert (input)
62                  ALU_control[2],   //1 bit B_invert (input)
63                  ALU_control[2],        //1 bit carry in (input)
64                  ALU_control[1:0],  //operation      (input)
65                  temp_result[0],     //1 bit result   (output)
66                  carry[0],       //1 bit carry out(output)
67  );
```

第0個alu的less輸入是將第31個set less than，也就是將src1 - src2接回來，以及carry_in輸入為Binvert。

```
69  alu_top alu1(
70                  src1[1],       //1 bit source 1 (input)
71                  src2[1],       //1 bit source 2 (input)
72                  1'b0,       //1 bit less     (input)
73                  ALU_control[3],    //1 bit A_invert (input)
74                  ALU_control[2],    //1 bit B_invert (input)
75                  carry[0],         //1 bit carry in (input)
76                  ALU_control[1:0],  //operation      (input)
77                  temp_result[1],     //1 bit result   (output)
78                  carry[1],       //1 bit carry out(output)
79  );
```

而除了第0不同，後面30個alu輸入輸出皆相同，less輸入為0，carry_in為上個bit的carry_out。

接線的部分完成後，討論到ALU_control的部分：

g. instruction set: basic operation instruction (60%)

ALU Action	Name	ALU Control Input
And	And	0000
OR	Or	0001
Add	Addition	0010
Sub	Subtraction	0110
Nor	Nor	1100
Slt	Set less than	0111

由教授給的pdf可發現，ALU_control前兩bit為A,Binvert，後兩bit為alu_top.v中的operation，再來講到alu_top.v，src_temp為接到AND，OR，以及ADD時src的值，而ALU_control後兩個bit則表示該進行哪個指令(AND, OR, ADD, SLT)，而因為SUB只是當Binvert為1時進行ADD，NOR是當A,Binvert都為1時進行AND（Demorgan's law），故實際上只需判斷4種狀況。（如下圖）

```
47  wire          src1_temp, src2_temp;
48
49  assign src1_temp = A_invert ^ src1;
50  assign src2_temp = B_invert ^ src2;
51
52  always@(*)
53  begin
54      case(operation)
55          2'b00:
56              result = src1_temp & src2_temp;
57          2'b01:
58              result = src1_temp | src2_temp;
59          2'b10:
60              result = src1_temp ^ src2_temp ^ cin;
61          2'b11:
62              result = less;
63      endcase
64  end
```

判斷是否進位(Carry_out)則是利用以下算法

```
66  assign cout = (src1_temp & src2_temp) | (cin & (src1_temp ^ src2_temp));
```

接著是Zero, Carry_out, Overflow的判斷：

```
449     zero <= !(temp_result[0] | temp_result[1] | temp_result[2] | temp_result[3] | temp_result[4] |
450                   temp_result[5] | temp_result[6] | temp_result[7] | temp_result[8] | temp_result[9] |
451                   temp_result[10] | temp_result[11] | temp_result[12] | temp_result[13] |
452                   temp_result[14] | temp_result[15] | temp_result[16] | temp_result[17] |
453                   temp_result[18] | temp_result[19] | temp_result[20] | temp_result[21] |
454                   temp_result[22] | temp_result[23] | temp_result[24] | temp_result[25] |
455                   temp_result[26] | temp_result[27] | temp_result[28] | temp_result[29] |
456                   temp_result[30] | temp_result[31]);
457
458     cout = ((ALU_control==4'b0010) | (ALU_control==4'b0110)) ? carry[31] : 0;
459     overflow = ( (ALU_control==4'b0010) & src1[31] & src2[31] & ~result[31]) ? 1
460           :( (ALU_control==4'b0010) & ~src1[31] & ~src2[31] & result[31]) ? 1
461           :( (ALU_control==4'b0110) & src1[31] & ~src2[31] & ~(result[31])) ? 1
462           :( (ALU_control==4'b0110) & ~src1[31] & src2[31] & result[31]) ? 1
463           : 0;
```

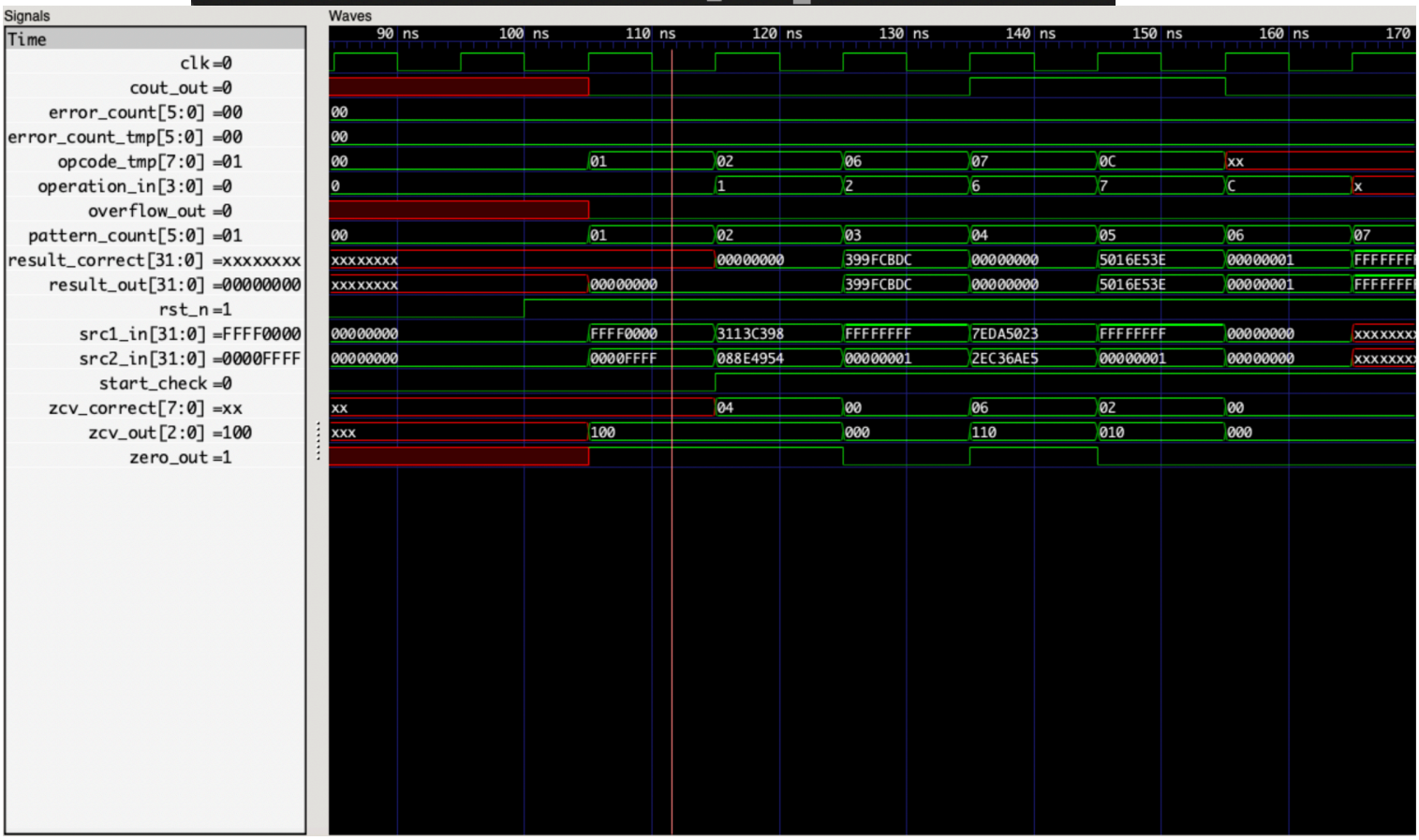
Zero的部分很直接，就只是把全部32個bit 做OR運算再invert；Carry_out的部分先判斷是否為ADD, SUB，否則就輸出0，若為加減運算則輸出第31bit的Carry_out；再來是Overflow的判斷，會產生overflow的狀況有4種：

(1) 正+正 = !正
(2) 負+負 = !負
(3) 正 - 負 = !正
(4) 負 - 正 = !負

如有上述狀況輸出1，無則輸出0。

Experiment result:

```
exfruit@Jeffs-MacBook-Pro LAB_1 % vvp testbench
VCD info: dumpfile mytest.vcd opened for output.
************************************************
 Congratulation! All data are correct!
************************************************
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 175000 ticks.
> finish
** Continue **
exfruit@Jeffs-MacBook-Pro LAB_1 %
```



Problems you met and solutions:

其實在LAB_0就有的問題，因為MacOS不支援Xilinx Vivado，一開始試著灌了ubuntu(因為室友也是Mac，處理方式就是使用虛擬機)，但在安裝vivado時說不支援32位元處理，可能是ARM的結構讓他覺得不是64位元(？
問了許多人也得不出答案，因為沒人用M1的Macbook，所以就用了icarus verilog編譯和執行，並用GTKwave跑波形圖。

再來應該就是verilog語法的問題，由於數電修的非常的不確實，一開始在寫這次作業時甚至會在always裡面接線，還有alu_top.v的input，我看其他人的code，在less的輸入直接數入0就可以，但我的編譯器顯示接收到32 bit資料，應該要收到1 bit，於是就改成1’b0，還有module的調用語法，一開始寫.src1(src[0])跑不過，全部改上面的寫法就可以，讓我理解到verilog是個很玄的東西；再來就是邏輯的問題吧，本來是把output 的result直接接到各個alu的輸出，可是為了在reset後運作，就換成接到wire temp_result, carry，到要輸出實在把線接到輸出，還有在第一次執行時的zcv跟答案不一樣，在想問題時就到FB看一下，才知道cout只需要在ADD, SUB考慮就好，然後就成功啦，超感動。

Summary:

對verilog的恐懼降低了一點點，了解到ALU是如何運作。