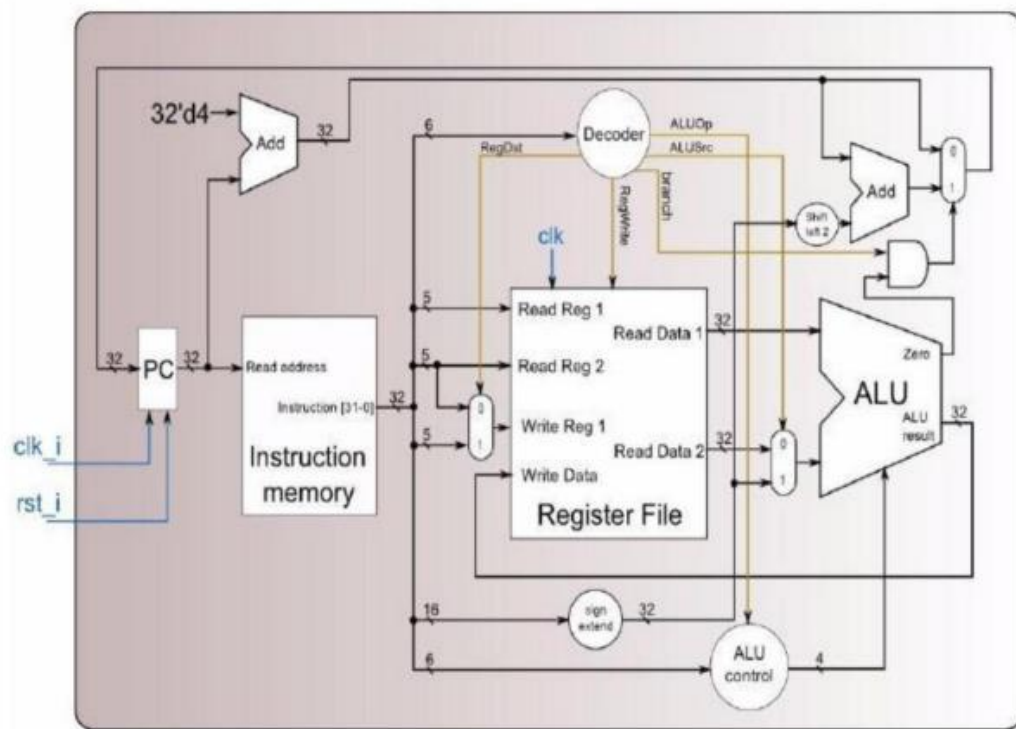


# Computer Organization Lab2

Name: 吳文心

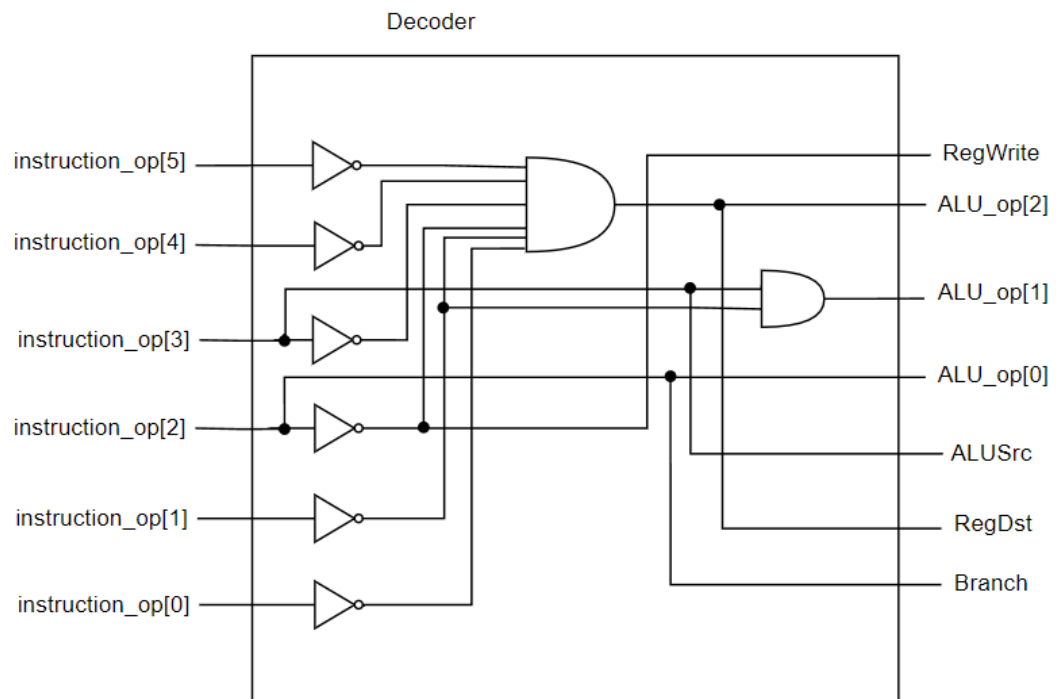
ID: 109550022

Architecture diagrams:

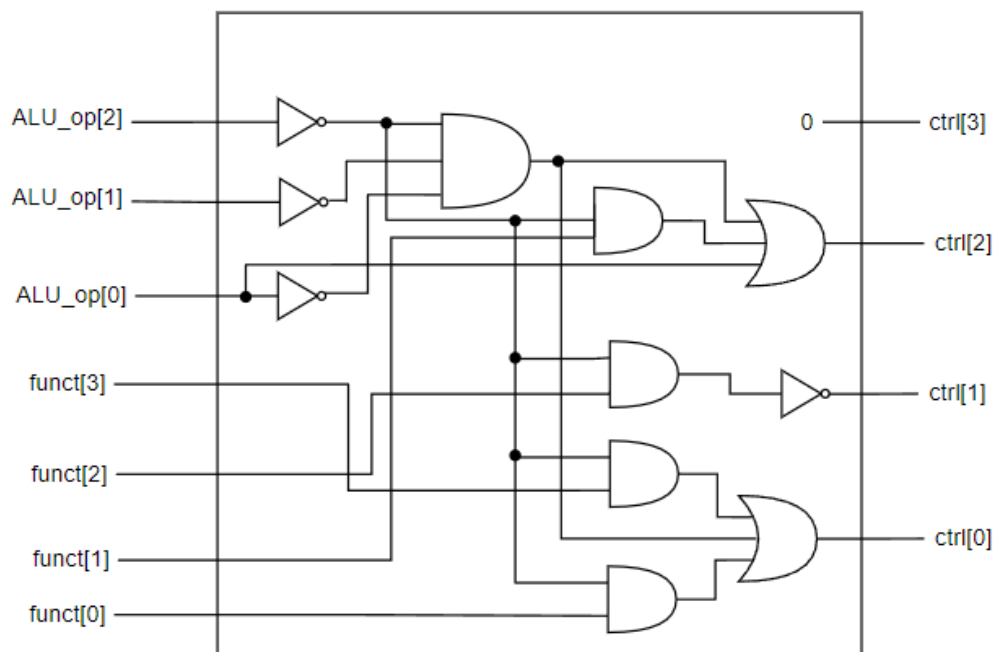


Top module: Simple\_Single\_CPU

Decoder:



ALU\_Ctrl:



## Hardware module analysis:

ALU\_op 的定義單純依下圖 operation 順序，分別定義為 100, 010, 001, 000，每個 ALU\_op 中最多只有一個 1，方便電路簡化。

依據 decoder 的輸入輸出把電路做了簡化:

ALU_op	operation	instr_op[5]	instr_op[4]	instr_op[3]	instr_op[2]	instr_op[1]	instr_op[0]
100	R-type (add, sub, and, or, slt)	0	0	0	0	0	0
010	addi	0	0	1	0	0	0
001	beq	0	0	0	1	0	0
000	slti	0	0	1	0	1	0

	instr_op[5]	instr_op[4]	instr_op[3]	instr_op[2]	instr_op[1]	instr_op[0]
RegWrite	x	x	x	0	x	x
ALU_op[2]	0	0	0	0	0	0
ALU_op[1]	x	x	1	x	0	x
ALU_op[0]	x	x	x	1	x	x
ALUSrc	x	x	1	x	x	x
Branch	x	x	x	1	x	x

簡化電路可以節省成本，可以不用依據沒個 case 給特定的輸出，而是直接用 instruction 的 operation 部分組出 output 的電路。

依據 ALU\_Ctrl 的輸入和輸出簡化電路: (第一章圖是簡化之前的電路，在此僅用於表示輸入輸出的關係)

```
// case (ALUOp_i)
//   // R-type
//   3'b100: case (funct_i)
//     6'b100000: ALUCtrl_o <= 2; // add
//     6'b100010: ALUCtrl_o <= 6; // sub
//     6'b100100: ALUCtrl_o <= 0; // and
//     6'b100101: ALUCtrl_o <= 1; // or
//     6'b100110: ALUCtrl_o <= 7; // slt
//     default: ALUCtrl_o <= 4'bxxxx;
//   endcase
//   // addi
//   3'b010: ALUCtrl_o <= 2;
//   // beq
//   3'b001: ALUCtrl_o <= 6;
//   // slt
//   3'b000: ALUCtrl_o <= 7;
//   default: ALUCtrl_o <= 0;
// endcase
```

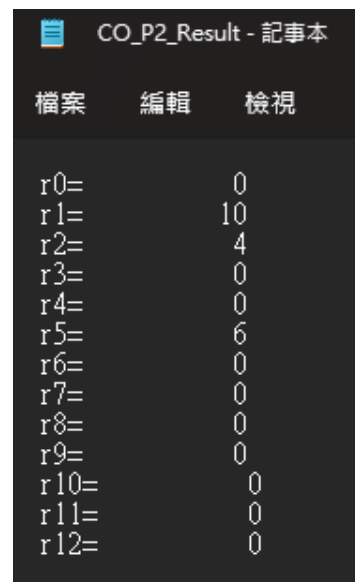
ALUCtrl[3]設為 0，因此並未在下圖中出現。

	ALU_op[2]	ALU_op[1]	ALU_op[0]	funct[3]	funct[2]	funct[1]	funct[0]
ALUCtrl[2]	1	x	x	x	x	1	x
	0	0	0	0	0	0	0
	x	x	1	x	x	x	x
ALUCtrl[1]	0	x	x	x	x	x	x
	x	x	x	x	0	x	x
ALUCtrl[0]	0	0	0	0	0	0	0
	1	x	x	x	x	x	1
	1	x	x	1	x	x	x

Shift\_Left\_Two\_32.v 實作時是直接把後 30 個位元的資料給前 30 個位元，並在最後兩個位元填 0。

Sign\_Extended.v 實作是把原本 16 位元的資料放到 Extend 之後的資料的後 16 位元，並將原先資料的 sign bit assign 給前 16 位元。

## Finished part:



檔案	編輯	檢視
r0=		0
r1=		10
r2=		4
r3=		0
r4=		0
r5=		6
r6=		0
r7=		0
r8=		0
r9=		0
r10=		0
r11=		0
r12=		0

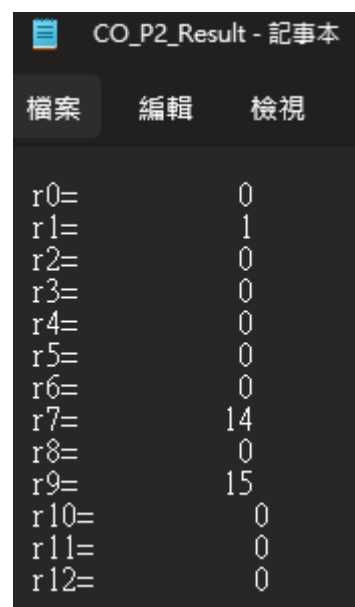
r1 是 r0 加上 10，而 r0 就是 \$zero，所以 r1 = 10。

r2 是 r0 加上 4，所以 r2 = 4。

r3 紀錄 r1 是否小於 r2，此例中為否，所以 r3 = 0。

又 beq r3, r0, r1，程式會跳過 add r4, r1, r2，所以 r4 仍然為預設值 0。

r5 是 r1 - r2 = 10 - 4 = 6。



檔案	編輯	檢視
r0=		0
r1=		1
r2=		0
r3=		0
r4=		0
r5=		0
r6=		0
r7=		14
r8=		0
r9=		15
r10=		0
r11=		0
r12=		0

beq 之前:

r6 是 r0 加上 2，所以 r6 = 2。

r7 是 r0 加上 14，所以 r7 = 14。

r8 是 r6 and r7，也就是 10 and 1110 (皆為二進位，省略第一個 1 之前的 0)，r8 = 2。

r9 是 r6 or r7，也就是 10 or 1110，r9 = 14。

addi r6, r6, -1 使得 r6 = 1。

r1 紀錄 r6 是否小於 1，此例為否，r1 = 0。

因為 beq r1, r0, -5，又 r1 == r0 == 0，程式跳回到 and r8, r6, r7。

beq 之後: (也就是 register 最終顯示的結果)

r8 是 r6 and r7，也就是 1 and 1110，r8 = 0。

r9 是 r6 or r7，也就是 1 or 1110，r9 = 15。

addi r6, r6, -1 使得 r6 = 0。

r1 紀錄 r6 是否小於 1，此例為是，r1 = 1。

因為 r1 != r0，beq r1, r0, -5 不會再 branch 一次，程式結束。

## Problems you met and solutions:

一開始 register 的結果都是 0，後來把資料一步一步印出來之後發現是 ALUCtrl 出了問題，我使用的 ALUCtrl 是 3 個 bits，但 Lab 中預設的是 4 個 bits，4-bit 應該才是正確的，只是因為此次 Lab 中沒有用到最高位(即 A\_inverse 在此 lab 中都是 0)，改成 4-bit 就可以跑出預期的答案了。

## Summary:

Debug 越來越上手了，好感動，終於找到了適合自己的、有效率的 debug 方式，不再是盲目撞牆了。