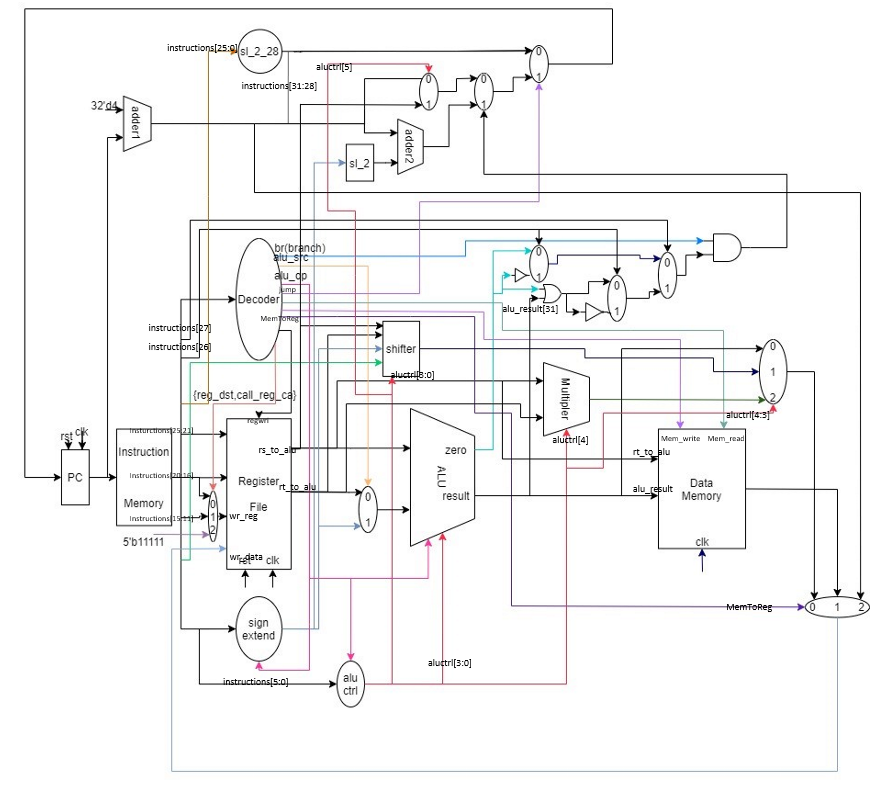
CO\_Lab3 Report

0710028 陳敬諺

1. **Architecture diagram**



1. **Implementation**

延續lab2的cpu架構，新增了一些module或修改:

Mux\_Write\_Reg:

因為要實現jal的功能，必須叫出r31($ra)，所以將原本的Mux\_2to1改成Mux\_3to1，當select==2則以5’b11111作為write\_reg的address。

Decoder:

增加Mem\_Read、Mem\_Write、MemToReg、jump、call\_reg\_ra等wire

ALU\_Ctrl:

輸出的aluctrl從原本的4bits變6bits，方便控制multipler、ALU、shifter、alu\_confirm、Mux\_PC\_Source\_Pre(jr)

Multipler:

執行mul的功能

Shifter:

增加sll的功能

blez\_vs\_bgtz:

決定要輸出blez或bgtz的結果

branch\_type:

blez、bgtz與lab2的beq、bne作區別，決定輸出哪種類型的結果。

Shifter\_jump\_addr:

將{2'b00,instructions[25:0]}向左位移兩位

Mux\_PC\_Source\_pre:

確認是否執行jr，輸出取代lab2的Mux\_PC\_Source的pc\_temp1

Mux\_Jump\_confirm:

確認是否執行j、jal

alu\_confirm:

決定輸出wr\_data\_pre的型態是alu\_result、mult\_result、shift\_result

result\_confirm

改為Mux\_3to1，決定寫入register的data為wr\_data\_pre、mem\_result、pc\_temp1

1. **Question**

* Please list the machine code of each branch/jump (**j**, **bne**, **beq** and **bgtz**) instructions in "\_CO\_Lab3\_test\_data\_bubble\_sort.txt" on the report.

// j Jump //16

00001000000000000000000000011001

// bgtz $at, no\_swap //56

00011100001000000000000000000011

// bgtz $at, next\_turn //84

00011100001000000000000000000001

// j inner //88

00001000000000000000000000001011

//next\_turn: bne $t1, $0, outer//92

00010100000010011111111111110000

// j End //96

00001000000000000000000000011101

// beq $t1, $t2, Loop //108

00010001010010011111111111111110

// j bubble //112

00001000000000000000000000000101

* What is the main purpose of "**jal**" and "**jr**" instructions? Please write down a simple program which explains the scenario.

//jal 1

00001100000000000000000000000001

…

…

//jr $ra

00000011111000000000000000001000

如上述的程式碼，令”//jal 1”的address為0(0x00000000)，執行完 後，ra會存入(0(pc)+4)，當執行到”jr ra”則會跳回到address 1(0x00000100)的地方

從上述示範可知，jal與jr會配合用在需要重複呼叫執行函式的地方

* Assume there is an instruction "**beqi $rt, imm, offset**" (branch if equal to immediate). Although it is not supported by our CPU, it can be achieved by the combination of two instructions "**ori $ra, $zero, imm**" and "**beq $rt, $ra, offset**". Can the instruction "**bge $rs, $rt, offset**" (branch if greater than or equal, branch if $rs >= $rt) be replaced with instructions implemented in lab2/3? (Hint: slt)

“slt $rd, $rs, $rt”

“beq $rd, $zero, offset”

* Following the previous question, can we reduce some of the instructions in lab2/3 without reducing the capability of CPU? If possible, what is the minimum instruction set, and what are the advantages and disadvantages of this reduction?

可被取代的：

sra $rd, $rt, shamt:

“addi $r1, $zero, shamt”

“srav $rd, $rt, $r1”

blez $rs, imm:

“addi $r1, $zero, 1”

“slt $r2, $rs, $r1”

“bne $r2, $zero, imm”

bgtz $rs, imm:

“addi $r1, $zero, 1”

“slt $r2, $rs, $r1”

“beq $r2, $zero, imm”

lui $r1, imm:

“addi $r2, $zero, imm”

“sll $r1, $r2, 16”

ori $r1, $r2, imm:

“addi $r3, $zero, imm”

“or $r1, $r2, $r3”

sltiu $r1, $r2, imm:

“addi $r3, $zero, imm”

“slt $r1, $r2, $r3”

mul:

用and addu sra實現

因此minimum instruction set為

R-type:

addu, subu, and, or, slt, sra, jr

I-type:

addi, beq, bne, lw, sw

J-typr:

j, jal

Advantages:

需要的machine code少，電路上的體積跟邏輯閘數也會變少

Disadvantage:

要實現一項運算的敘述會變得複雜，且須要消耗較多的時間

1. **Contribution**

Most of cpu’s architecture, bug fixing

1. **Discussion**

有了上次lab的經驗，這次實作起來順手很多，不過確實如我自己上次所說的，decoder沒有用變數命名或註解，導致自己在tracing code上遇到不少問題，就在這次一並把這問題給改善了，另外需要下比較多功夫的大概就是問題的最後一題，在構思上需要花點時間。