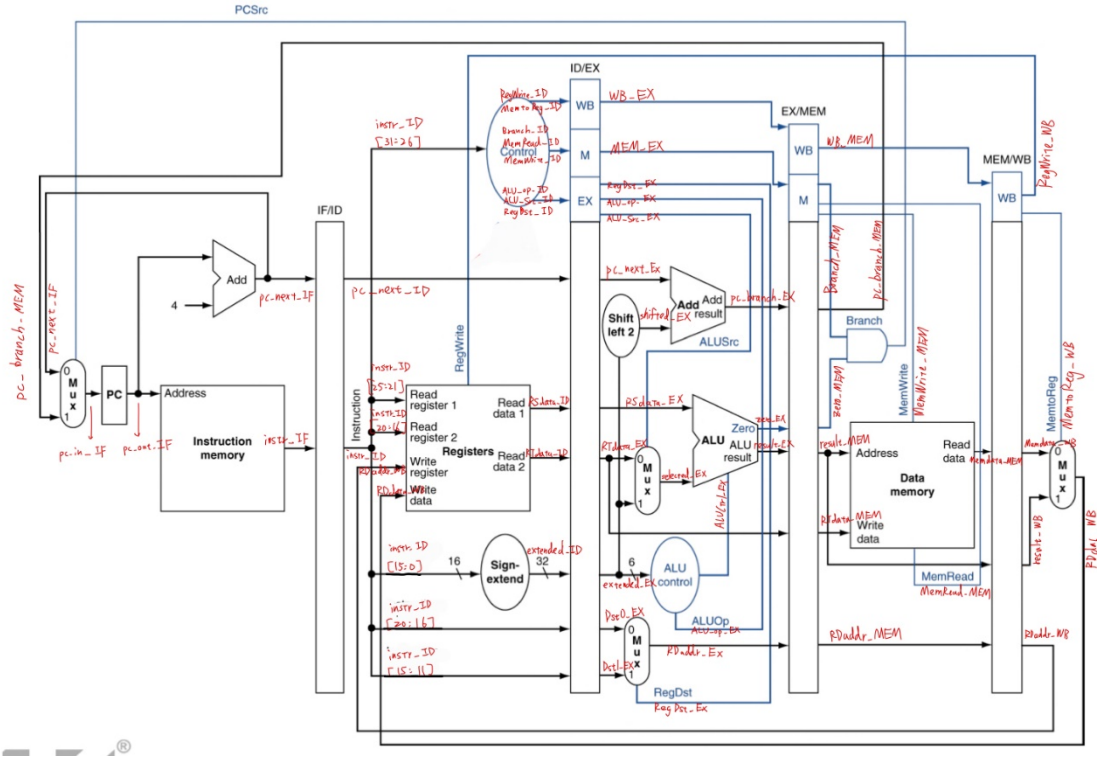
Computer Organization Lab4

Name: 陳子祈

ID: 0819823

Architecture diagrams:



R type						
Instruction set	**Op code**	**rs**	**rt**	**rd**	**shamt**	**funct**
Instr location	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
add $rd,$rs,$rt	000000 (0)				00000	100000 (32)
sub $rd,$rs,$rt	000000 (0)				00000	100010 (34)
and $rd,$rs,$rt	000000 (0)				00000	100100 (36)
or $rd,$rs,$rt	000000 (0)				00000	100101 (37)
slt $rd,$rs,$rt	000000 (0)				00000	101010 (42)
mult $rd,$rs,$rt	000000 (0)				00000	011000 (24)
I type						
Instruction set	**Op code**	**rs**	**rt**	**immediate**		
Instr location	[31:26]	[25:21]	[20:16]	[15:0]		
addi $rt,$rs,imm	001000 (8)					

slti $rt,$rs,imm	001010(10)			
beq $rt,$rs,imm	000100(4)			

Hardware module analysis:

(explain how the design work and its pros and cons)

Pipeline MIPS CPU 每過一段 pipeline clock period 就做 PC 與暫存器的運算，pipeline clock period 取大於所有 stage 執行指令 cycle time 的時間，因為總共有 5 個 stage，所以會有 5 個指令同時在 CPU 不同 stage 被執行，每個 stage 的間隔會有 buffer 存前一個 stage 的 output 與下一個 stage 的 input，如此設計可以讓每個 stage 之間的訊號不會直接互通，而是等一段 pipeline clock period 才將前一個 stage 的 output 與下一個 stage 的 input。PC 的運算只有分為 Sequential 的運算與 beq 跳行的運算。
五個 stage 分別如下:

1. IF: Instruction fetch from memory 從 memory 請求指令
2. ID: Instruction decode & register read 解碼指令、產生控制訊號，並把暫存器的資料讀出來
3. EX: Execution operation or calculate address 執行指令，這次實驗使用到 lw、sw 存取記憶體的指令，因此需計算地址
4. MEM: Access memory operand 存取記憶體資料，若執行 sw 指令，記憶體的資料就要被更改，若執行 lw 指令，則讀取記憶體的資料
5. WB: Write result back to register 依據不同指令，將 ALU 計算出來的結果、記憶體的資料、常數或 PC+4 寫回去 Write register，PC 也會依據 branch 更新 PC

Single cycle MIPS CPU 優點就是不會發生 hazards; 缺點就是以 Longest delay 的指令當作 clock period，導致執行大部分指令都有很多 CPU idle 的時間。不過 Pipeline MIPS CPU 則剛好相反，優點是以 Longest delay of a stage 的指令當作 clock period，減少很多 CPU idle 的時間; 不過這次 Lab 沒有處理 hazards，所以缺點是萬一發生 data hazards 或 load use hazards 的話最後結果就會出錯。

各module的description:

1) Decoder

功能：透過6bit的instruction operation code 決定各種控制訊號。

Port description：

instr_op_i ：6bit input instruction operation code

RegWrite_o ：1bit output RegFile Write or not

ALU_op_o ：3bit output for ALU_Ctrl to determine operation type

ALUSrc_o ：1bit output determine ALU source

RegDst_o ：1 bit output determine Read reg2 is rt or rd

Branch_o ：1bit output the instruction is branch type or not

MemRead_o : 1bit output for Data memory to determine read memory data or not

MemWrite_o : 1bit output for Data memory to determine write memory data or not

MemtoReg_o : 2bit output to determine where Register write data is from

Instr_op [31:26]		Instruction	RegDst	ALUSrc	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	ALU Op [2:0]
[31:29]	[28:26]									
000	000	R-type	1	0	0	1	0	0	0	000
	100	beq	X	0	X	0	0	0	1	011
I-type										
001	000	addi	0	1	0	1	0	0	0	010
	010	slti								111
100		lw	0	1	1	1	1	0	0	010
101		sw	0	1	X	0	0	1	0	010

2) ALU_Ctrl

功能：將ALU_op及function code轉成ALU所需的ALUCtrl，決定ALU的動作及控制其他MUX、Shifter。

Port description：

funct_i ：6bit input function code

ALUOp_i ：3bit input for ALU_Ctrl to determine operation type

ALUCtrl_o ：4bit output to ALU control

ALUOp_i	funct_i	operation	ALUCtrl
R-type			
000	100000(32)	add	0010
	100010(34)	sub	0110
	100100(36)	and	0000
	100101(37)	or	0001
	101010(42)	slt	0111
	011000(24)	mult	1111
I-type			
010		addi lw sw	0010
011		beq	0110
111		slti	0111

3) ALU

功能：32bit運算邏輯單位，參考課本附錄程式，可做add、sub、or、and、slt、mult。

Port description：

src1_i ：32bit input data

src2_i ：32bit input data

ctrl_i ：4bit ALU_Control

result_o ：32bit result for ALU

zero_o ：1 bit when the output is 0, zero must be set

4) Adder

功能：輸入兩個data輸出其相加結果。

Port description：

src1_i ：32bit input data

src2_i ：32bit input data

sum_o ：32bit output sum

5) Sign_Extend

功能：將輸入data做Sign Extend，data_i複製到data_o低16位，data_i最高位bit複製到data_o高16位。

Port description：

data_i ：16bit input data

data_o ：32bit output data

6) Shift_Left_Two_32

功能：將input data左移兩個bit。

Port description：

data_i ：32bit input data

data_o ：32bit output data

7) MUX_2to1

功能：如果 select_i = 0 則輸出 data0_i；select_i = 1 則輸出data1_i。

Port description：

data0_i ：32bit input data

data1_i ：32bit input data

select_i ：1bit select for MUX

data_o ：32bit output data

8) Pipe_CPU_1

功能：將上述所提到之 Module 依照 Architecture diagram 的附圖做連接，完成 Pipeline CPU。

Finished part:

(show the screenshot of the simulation result and waveform, and explain it)

使用“CO_P4_test_1.txt”作為指令輸入，最後各 register 與 data memory 結果如下:

```
Register==========================================================

r0=        0, r1=        3, r2=        4, r3=        1, r4=        6, r5=        2, r6=        7, r7=        1

r8=        1, r9=        0, r10=       3, r11=       0, r12=       0, r13=       0, r14=       0, r15=       0

r16=       0, r17=       0, r18=       0, r19=       0, r20=       0, r21=       0, r22=       0, r23=       0

r24=       0, r25=       0, r26=       0, r27=       0, r28=       0, r29=       0, r30=       0, r31=       0


Memory============================================================

m0=        0, m1=        3, m2=        0, m3=        0, m4=        0, m5=        0, m6=        0, m7=        0

m8=        0, m9=        0, m10=       0, m11=       0, m12=       0, m13=       0, m14=       0, m15=       0

r16=       0, m17=       0, m18=       0, m19=       0, m20=       0, m21=       0, m22=       0, m23=       0

m24=       0, m25=       0, m26=       0, m27=       0, m28=       0, m29=       0, m30=       0, m31=       0
```

此結果與答案相符。

使用“CO_P4_test_2.txt”作為指令輸入，最後各 register 與 data memory 結果如下:

```
Register==========================================================

r0=        0, r1=       16, r2=        4, r3=        8, r4=       16, r5=       -8, r6=       24, r7=       26

r8=        0, r9=      100, r10=       0, r11=       0, r12=       0, r13=       0, r14=       0, r15=       0

r16=       0, r17=       0, r18=       0, r19=       0, r20=       0, r21=       0, r22=       0, r23=       0

r24=       0, r25=       0, r26=       0, r27=       0, r28=       0, r29=       0, r30=       0, r31=       0


Memory============================================================

m0=        0, m1=       16, m2=        0, m3=        0, m4=        0, m5=        0, m6=        0, m7=        0

m8=        0, m9=        0, m10=       0, m11=       0, m12=       0, m13=       0, m14=       0, m15=       0

r16=       0, m17=       0, m18=       0, m19=       0, m20=       0, m21=       0, m22=       0, m23=       0

m24=       0, m25=       0, m26=       0, m27=       0, m28=       0, m29=       0, m30=       0, m31=       0
```

因為這份指令輸入的不同指令之間有 data dependency，會產生 data hazards 或 load use hazards，所以此結果與答案不符。詳細來看，可以發現 Instruction 1、Instruction 2 發生 data hazard，Instruction 2 讀到的 r1 會是 0，r2 就會是 0 + 4 = 4; 同樣的，Instruction 5、Instruction 6 發生 load use hazard，Instruction 6 讀到的 r4 會

是 0，r5 就會是 0 – 8 = -8; 還有另外一個地方也是，Instruction 8、Instruction 9 發生 data hazard，Instruction 8 讀到的 r7 會是 0，r8 就會是 0 & 8 = 0。

Bonus (optional):

將 CO_P4_test_2.txt 指令重新編排如下:

```
00100000000000010000000000010000        I1:   addi $1,$0,16
00000000000000000000000000000000              nop
00100000000000110000000000001000        I3:   addi $3,$0,8
00100000001000100000000000000100        I2:   addi $2,$1,4
10101100000000010000000000000100        I4:   sw $1,4($0)
10001100000000100000000000000100        I5:   lw $4,4($0)
00000000000000000000000000000000              nop
00000000011000010011000000100000        I7:   add $6,$3,$1
00000000100000110010100000100010        I6:   sub $5,$4,$3
00100000001001110000000000001010        I8:   addi $7,$1,10
00000000000000000000000000000000              nop
00100000000010010000000001100100        I10:  addi $9,$0,100
00000000111000110100000000100100        I9:   and $8,$7,$3
```

將此編排存到 CO_P4_test_3.txt(再請助教用這個檔案測試)，使用“CO_P4_test_3.txt”作為指令輸入，最後各 register 與 data memory 結果如下:

```
Register=============================================

r0=   0, r1=  16, r2=  20, r3=   8, r4=  16, r5=   8, r6=  24, r7=  26
r8=   8, r9= 100, r10=   0, r11=   0, r12=   0, r13=   0, r14=   0, r15=   0
r16=   0, r17=   0, r18=   0, r19=   0, r20=   0, r21=   0, r22=   0, r23=   0
r24=   0, r25=   0, r26=   0, r27=   0, r28=   0, r29=   0, r30=   0, r31=   0

Memory=============================================

m0=   0, m1=  16, m2=   0, m3=   0, m4=   0, m5=   0, m6=   0, m7=   0
m8=   0, m9=   0, m10=   0, m11=   0, m12=   0, m13=   0, m14=   0, m15=   0
r16=   0, m17=   0, m18=   0, m19=   0, m20=   0, m21=   0, m22=   0, m23=   0
m24=   0, m25=   0, m26=   0, m27=   0, m28=   0, m29=   0, m30=   0, m31=   0
```

重新編排指令可以手動解決 hazards 的問題，可發現 r2、r5、r8 的結果都是正確的。

Problems you met and solutions:

1. 暫存器 bit 總數數錯

這次實作 Pipeline CPU 需要在每個 stage 之間架好 Pipeline register 的模組，因為要依據每個 stage 的輸入多少 bit 輸出就要多少 bit，所以要計算每個 stage 輸出的暫存器總 bit 數量，一不小心數錯讓我結果出錯，數對之後結果就與答案相符了。

Summary:

這次實作 Pipeline MIPS CPU 不會太困難，因為之前已經做過兩次 Single cycle MIPS CPU，這次只要將之前的模組加入 Pipeline register 的設計，就可以實作出來簡易的 Pipeline MIPS CPU。不過這次沒有處理 hazards 的部分，希望下次 Lab 可以實作有處理 hazards 的 Pipeline MIPS CPU，感覺滿有挑戰性。