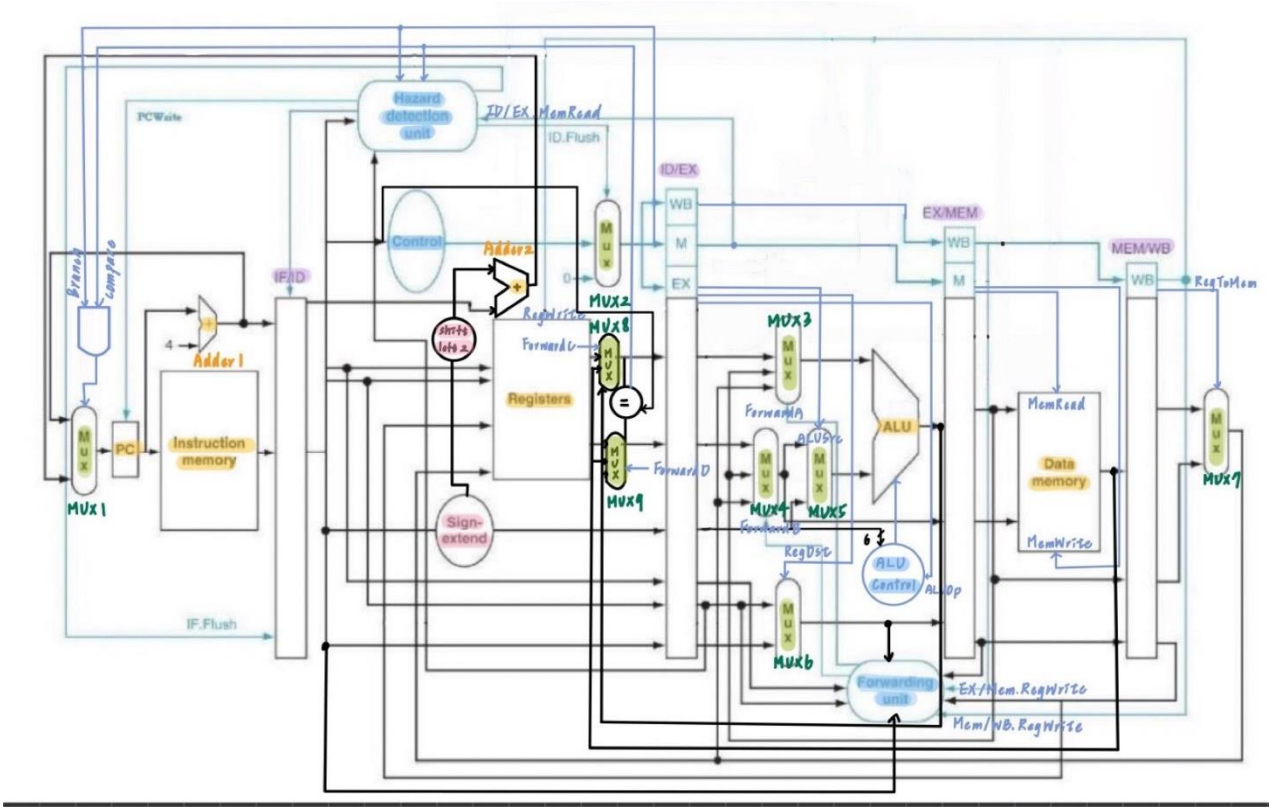
Computer Organization Lab5

Name:龔祐萱

ID: 0713216

Architecture diagrams:



針對所給的電路架構圖進行了以下的修改，使得 pipeline CPU 架構可以符合各項指令要求:

1. beq 指令

為了增加 pipeline CPU 的執行效率，增加了 comparator，並將原本在 EX stage 的 adder 和 shifter 移至 ID stage，使 pipeline CPU 在 ID stage 就可以知道下一個指令是否要 branch。

並新增一個控制訊號 compare 來得知 comparator 的比較結果，如果符合 branch 指令的條件時，compare=1；反之，如果不符合 branch 的指令時，compare=0。將控制訊號 compare 和 branch 送至 MUX1 控制下一個進入 PC 的指令為 sequential 指令還是 target address，如果 compare=1 且 branch=1，則 MUX1 會輸出 target address；反之，MUX1 會輸出 sequential address (PC+4)。

2. Hazard detection unit

Hazard detection unit 可以分成 load-use data hazard 以及 control hazard 兩個部分進行說明:

(1) Load-use data hazard

Load-use data hazard 會發生的條件為(ID/EX.MemRead and (ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt)。如果以上條件符合，則控制訊號 IF_Keep=1, PC_Keep=1, ID_Flush=1，IF_Keep 和 PC_Keep 會分別控制 IF/ID pipelind register 和 PC 中所儲存的資料不會被變更，ID_Flush 則會清除原本在 ID stage 的指令，使其不會進入 EX stage，讓有 load-use data hazard 的指令之間會 stall 一個 clock cycle，並使用 forwarding unit 來取得 use 指令的 input 資料。

(2) Control hazard

Control hazard 會發生的條件為控制訊號 compare=1 且 Branch=1。如果符合以上條件，則控制訊號 IF_Flush=1，會輸入 IF/ID pipeline register 並清除裡面所存的資料，使得 sequential 指令無法進入 ID stage，並透過在 ID stage 的 adder2 來送回正確的 target address，使其進入 PC。

3. Forwarding unit

Forwarding unit 可以分成一般指令的 data hazard 以及 branch 指令的 data hazard 兩個部分進行說明:

(1) 一般指令的 data hazard

針對一般指令的 data hazard 新增兩個控制訊號 Forward_A 以及 Forward_B，分別控制進入 ALU 的 Src1 和 Src2 的資料，其中 00 表示資料從 ID stage 而來，01 表示資料從 WB stage 而來，10 表示資料從 MEM stage 而來。

A. EX hazard (前後兩個指令的 data hazard)

其所發生的條件以及輸出的控制訊號為以下兩種:

a. If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
   Forward_A = 10
b. If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
   Forwar_B = 10

B. MEM hazard (相隔一個指令的 data hazard)

其所發生的條件以及輸出的控制訊號為以下兩種:

a. If ((MEM/WB.RegWrite) and (MEM/WB.RegisterRd ≠ 0)
   and ~((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0) and
   (EX/MEM.RegisterRd = ID/EX.RegisterRs))

and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
Forward_A = 01

b. If ((MEM/WB.RegWrite) and (MEM/WB.RegisterRd ≠ 0)
and ~((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0) and
(EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
Forwar_B = 01

(2) branch 指令的 data hazard

針對 branch 指令的 data hazard 新增兩個控制訊號 Forward_C 以及 Forward_D，分別控制從 register 輸出以及進入 comparator 的資料，其中 00 表示資料從 register 而來，01 表示資料從 data memory 而來，10 表示資料從 ALU 而來。

A. ID hazard (前後兩個指令的 data hazard)

其所發生的條件以及輸出的控制訊號為以下兩種:

a. If ((ID/EX.RegWrite) and (ID/EX.RegisterRd ≠ 0) and (ID/EX.RegisterRd = IF/ID.RegisterRs))
Forward_C = 10

b. If ((ID/EX.RegWrite) and (ID/EX.RegisterRd ≠ 0) and (ID/EX.RegisterRd = IF/ID.RegisterRt))
Forwar_D = 10

B. EX hazard (相隔一個指令的 data hazard)

a. If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0)
and ~((ID/EX.RegWrite) and (ID/EX.RegisterRd ≠ 0) and (ID/EX.RegisterRd = IF/ID.RegisterRs))
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
Forward_C= 01

b. If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0)
and ~((ID/EX.RegWrite) and (ID/EX.RegisterRd ≠ 0) and (ID/EX.RegisterRd = IF/ID.RegisterRt))
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
Forward_D= 01

Hardware module analysis:

1. R-type instruction (add, sub, and, or, slt, mult) 執行過程

(1) Stage1 (IF)

PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出。

(2) Stage2 (ID)

指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal，指令的 25-16 bit 會輸入 Register 作為 Read Register，並輸出該 register 的資料，指令的 15-11 bit 會作為 Write Register，並和上述資料輸入 ID/EX pipeline register，以及將下一個指令輸入 PC (PC+4)。

(3) Stage3 (EX)

將從 ID/EX pipeline register 輸出的資料分別輸入 ALU、ALU control 以及 MUX 進行運算，並將結果輸至 EX/MEM pipeline register 儲存。

(4) Stage4 (MEM)

將 EX/MEM pipeline register 輸出的資料輸入 MEM/WB pipeline register 儲存。

(5) Stage5 (WB)

將從 MEM/WB pipeline register 輸出的資料分別輸至 Register 的 WriteReg 和 WriteData 中，更新指定 register 的數值。

(6) 以下是 R-type instruction 的 control signal 整理表格

Compare	Branch
依 data 決定	0

ALUSrc	ALUOp (3 bits)	RegDst	ALU_control_output (4 bits)
0	010	1	add : 0010 sub : 0110 and : 0000 or : 0001 slt : 0111 mult : 1100

MemWrite	MemRead
0	0

MemtoReg	RegWrite
0	1

2. I-type instruction (addi, slti) 執行過程

(1) Stage1 (IF)

PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出。

(2) Stage2 (ID)

指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal，指令的 25-21 bit 會輸入 Register 作為 Read Register，並輸出該 register 的資料，指令的 15-0 bit 會進入 sign-extend 進行 sign extension，指令的 15-11 bit 會作為 Write Register，並和上述資料輸入 ID/EX pipeline register，以及將下一個指令輸入 PC (PC+4)。

(3) Stage3 (EX)

將從 ID/EX pipeline register 輸出的資料分別輸入 ALU、ALU control 以及 MUX 進行運算，並將結果輸至 EX/MEM pipeline register 儲存。

(4) Stage4 (MEM)

將 EX/MEM pipeline register 輸出的資料輸入 MEM/WB pipeline register 儲存。

(5) Stage5 (WB)

將從 MEM/WB pipeline register 輸出的資料分別輸至 Register 的 WriteReg 和 WriteData 中，更新指定 register 的數值。

(6) 以下是 I-type instruction 的 control signal 整理表格

Compare	Branch
依 data 決定	0

ALUSrc	ALUOp (3 bits)	RegDst	ALU_control_output (4 bits)
1	addi : 000 slti : 011	0	addi : 1000 slti : 0101

MemWrite	MemRead
0	0

MemtoReg	RegWrite
0	1

3. lw instruction 執行過程

(1) Stage1 (IF)

PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出。

(2) Stage2 (ID)

指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal，指令的 25-21 bit 會輸入 Register 作為 Read Register，並輸出該 register 的資料，指令的 15-0 bit 會進入 sign-extend 進行 sign extension，指令的 20-16 bit 會作為 Write Register，並和上述資料輸入 ID/EX pipeline register，以及將下一個指令輸入 PC (PC+4)。

(3) Stage3 (EX)

將從 ID/EX pipeline register 輸出的資料分別輸入 ALU、ALU control 以及 MUX 進行運算，並將結果輸至 EX/MEM pipeline register 儲存。

(4) Stage4 (MEM)

將 EX/MEM pipeline register 輸出的資料輸入 Data Memory 的 address 讀取資料並輸出至 MEM/WB pipeline register 儲存。

(5) Stage5 (WB)

將從 MEM/WB pipeline register 輸出的資料分別輸至 Register 的 WriteReg 和 WriteData 中，更新指定 register 的數值。

(6) 以下是 lw instruction 的 control signal 整理表格

Compare	Branch
依 data 決定	0

ALUSrc	ALUOp (3 bits)	RegDst	ALU_control_output (4 bits)
1	100	0	0010

MemWrite	MemRead
0	1

MemtoReg	RegWrite
0	1

4. sw instruction 執行過程

(1) Stage1 (IF)

PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出。

(2) Stage2 (ID)

指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal，指令的 25-16 bit 會輸入 Register 作為 Read Register，並輸出該 register 的資料，指令的 15-0 bit 會進入 sign-extend 進行 sign extension，並和上述資料輸入 ID/EX pipeline register，以及將下一個指令輸入 PC (PC+4)。

(3) Stage3 (EX)

將從 ID/EX pipeline register 輸出的資料分別輸入 ALU、ALU control 以及 MUX 進行運算，並將結果輸至 EX/MEM pipeline register 儲存。

(4) Stage4 (MEM)

將 EX/MEM pipeline register 輸出的資料輸入 Data Memory 的 address 儲存，更新指定 memory address 的資料。

(5) Stage5 (WB)

sw 指令不會執行 write back 的程序。

(6) 以下是 sw instruction 的 control signal 整理表格

Compare	Branch
依 data 決定	0

ALUSrc	ALUOp (3 bits)	RegDst	ALU_control_output (4 bits)
1	101	0	0010

MemWrite	MemRead
1	0

MemtoReg	RegWrite
0	0

5. Branch instruction (beq, bne, bge, bgt)執行過程

(1) Stage1 (IF)

PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出。

(2) Stage2 (ID)

指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal，指令的 25-21 bit 會輸入 Register 作為 Read Register，並輸出該 register 的資料，指令的 15-0 bit 會進入 sign-extend 進行 sign extension，指令的 20-16 bit 會作為 Write Register，並和上述資料輸入 ID/EX pipeline register，以及依照是否會做 branch 的結果將下一個指令輸入 PC (PC+4/target address)。

(3) Stage3 (EX)

將從 ID/EX pipeline register 輸出的資料分別輸入 ALU、ALU control 以及 MUX 進行運算，並將結果輸至 EX/MEM pipeline register 儲存。

(4) Stage4 (MEM)

將 EX/MEM pipeline register 輸出的資料輸入 Data Memory 的 address 讀取資料並輸出至 MEM/WB pipeline register 儲存。

(5) Stage5 (WB)

將從 MEM/WB pipeline register 輸出的資料分別輸至 Register 的 WriteReg 和 WriteData 中，更新指定 register 的數值。

(6) 以下是 branch instruction 的 control signal 整理表格

Compare	Branch
依 data 決定	1

ALUSrc	ALUOp (3 bits)	RegDst	ALU_control_output (4 bits)
0	x	0	x

MemWrite	MemRead
0	0

MemtoReg	RegWrite
0	0

6. Hazard detection unit 的執行過程

(1) Load-use hazard

Load-use data hazard 會發生的條件為(ID/EX.MemRead and (ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt)。

如果以上條件符合，則控制訊號 IF_Keep=1, PC_Keep=1, ID_Flush=1，IF_Keep 和 PC_Keep 會分別控制 IF/ID pipelind register 和 PC 中所儲存的資料不會被變更，ID_Flush 則會清除原本在 ID stage 的指令，使其不會進入 EX stage，讓有 load-use data hazard 的指令之間會 stall 一個 clock cycle，並使用 forwarding unit 來取得 use 指令的 input 資料。

以下為發生 load-use data hazard 時相關控制訊號的整理表格:

ID/EX.MemRead	PC_Keep	IF_Keep	ID_Flush
1	1	1	1

(2) Control hazard

Control hazard 會發生的條件為控制訊號 compare=1 且 Branch=1。

如果符合以上條件，則控制訊號 IF_Flush=1，會輸入 IF/ID pipeline register 並清除裡面所存的資料，使得 sequential 指令無法進入 ID stage，並透過在 ID stage 的 adder2 來送回正確的 target address，使其進入 PC。

以下為發生 control hazard 時相關控制訊號的整理表格:

Branch	Compare	IF_Flush
1	1	1

7. Forwarding unit 的執行過程

(1) 一般指令的 forwarding 機制

針對一般指令的 data hazard 新增兩個控制訊號 Forward_A 以及 Forward_B，分別控制進入 MUX3 和 MUX4 的資料，其中 00 表示資料從 ID stage 而來，01 表示資料從 WB stage 而來，10 表示資料從 MEM stage 而來。

A. EX hazard (前後兩個指令的 data hazard)

其所發生的條件以及輸出的控制訊號為以下兩種:

a. If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
   Forward_A = 10
b. If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
   Forwar_B = 10

B. MEM hazard (相隔一個指令的 data hazard)

其所發生的條件以及輸出的控制訊號為以下兩種:

a. If ((MEM/WB.RegWrite) and (MEM/WB.RegisterRd ≠ 0)
   and ~((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0) and
   (EX/MEM.RegisterRd = ID/EX.RegisterRs))
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
   Forward_A = 01
b. If ((MEM/WB.RegWrite) and (MEM/WB.RegisterRd ≠ 0)
   and ~((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0) and
   (EX/MEM.RegisterRd = ID/EX.RegisterRt))
   and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
   Forwar_B = 01

C. 以下為一般指令 forwarding 機制相關控制訊號的整理表格:

	EX/MEM.RegWrite	MEM/WB.RegWrite	Forward_A	Forward_B
EX hazard	1	x	10	10
MEM hazard	x	1	01	01
沒有 hazard	-	-	00	00

(2) branch 指令的 data hazard

針對 branch 指令的 data hazard 新增兩個控制訊號 Forward_C 以及 Forward_D，分別控制從 MUX8 和 MUX9 輸出的資料，其中 00 表示資料從 register 而來，01 表示資料從 data memory 而來，10 表示資料從 ALU 而來。

A. ID hazard (前後兩個指令的 data hazard)

其所發生的條件以及輸出的控制訊號為以下兩種:

a. If ((ID/EX.RegWrite) and (ID/EX.RegisterRd ≠ 0) and (ID/EX.RegisterRd = IF/ID.RegisterRs))
   Forward_C = 10
b. If ((ID/EX.RegWrite) and (ID/EX.RegisterRd ≠ 0) and (ID/EX.RegisterRd = IF/ID.RegisterRt))
   Forwar_D = 10

B. EX hazard (相隔一個指令的 data hazard)

a. If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0)
   and ~((ID/EX.RegWrite) and (ID/EX.RegisterRd ≠ 0) and (ID/EX.RegisterRd = IF/ID.RegisterRs))
   and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
   Forward_C= 01
b. If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd ≠ 0)
   and ~((ID/EX.RegWrite) and (ID/EX.RegisterRd ≠ 0) and (ID/EX.RegisterRd = IF/ID.RegisterRt))
   and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
   Forward_D= 01

C. 以下為 branch 指令 forwarding 機制相關控制訊號的整理表格:

	ID/EX.RegWrite	EX/MEM.RegWrite	Forward_C	Forward_D
ID hazard	1	x	10	10
EX hazard	x	1	01	01
沒有 hazard	-	-	00	00

8. 優缺點

因為 pipeline CPU 加入了 pipeline register，所以使得 pipeline CPU 可以被分成多個部分，可以分別執行 instruction fetch、decoding、execution、memory store and load、write back 等五個 stage，而每個 stage 都可以分別執行一個指令，所以在每個 clock cycle，都可以丟入一個指令進入 pipeline CPU 執行，因此每個 clock cycle 可以同時執行多個指令，使得 CPU 的整體執行效率提升。

也因為 pipeline CPU 的特殊架構，所以一個指令從丟入到執行完成需要五個 clock cycle 才會完成，也表示需要五個 clock cycle pipeline CPU 中的 register 或 data memory 中的資料才會做更新。因此如果前後兩個指令具有 data dependency 的關係時，兩個指令如果一前一後丟入 pipeline CPU 執行，會因為前一個指令對於 register 或 memory 的資料未做更新，而導致後一個指令的執行結果錯誤。針對以上的問題可以透過加入 forwarding unit 以及 hazard detection unit 來改進，forwarding unit 可以透過是否符合做 forwarding 的條件來輸出相對應的控制訊號到 MUX 中，控制 MUX 所輸出的資料，並提前將還未寫入 register 或未從 data memory 中讀出的資料給下一個指令做使用，可以增加 pipeline CPU 的效能。Hazard detection unit 則可以針對 load-use data hazard 以及 control hazard 進行修正，使得指令可以正確運作。

但因為程式可能會有 control hazard 或 load-use data hazard，即使有 forwarding 機制的幫助，load-use data hazard 的兩個指令之間仍需要 stall 一個 clock cycle；如果 branch 指令需要做 branch 的話，就需要把 sequential 指令 flush 掉，因此 pipeline CPU 在執行的過程中，可能會有幾個 stage 會是 no operation 的狀況，導致 pipeline CPU 的效能無法被完全使用。

此外，需要提升 pipeline CPU 的效能就需要更複雜的電路結構，例如:forwarding unit、hazard detection unit…等，因此硬體的實作上也會更加複雜，成本也會增加。

Finished part:

以下是測資的執行指令、指令執行過程、執行結果以及程式執行結果截圖:

1. Case 1

執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) addi $1,$0,16 (2) mult $2,$1,$1 (3) addi $3,$0,8 (4) sw $1,4($0) (5) lw $4,4($0) (6) sub $5,$4,$3 (7) add $6,$3,$1 (8) addi $7,$1,10 (9) and $8,$7,$3 (10) slt $9,$8,$7	(1) r1 = r0+16 = 0+16 = 16 (2) r2 = r1*r1 = 16*16 = 256 (3) r3 = r0+8 = 8 (4) Store r1=16 to M1, M1 = 16 (5) Load M1=16 to r4, r4 = 16 (6) r5 = r4-r3 = 16-8 = 8 (7) r6 = r3+r1 = 8+16 = 24 (8) r7 = r1+10 = 16+10 = 26 (9) r8 = r7 AND r3 = 8 (10) (r8=8) < (r7=26), r9 = 1	r0 = 0 r1 = 16 r2 = 256 r3 = 8 r4 = 16 r5 = 8 r6 = 24 r7 = 26 r8 = 8 r9 = 1 r10~r31 = 0	M0 = 0 M1 = 16 M2~M31 = 0

程式執行結果截圖

MAX_COUNT = 17

```
################################ clk_count = 17####################################
=====================================Register=====================================
r0 =   0, r1 =  16, r2 = 256, r3 =   8, r4 =  16, r5 =   8, r6 =  24, r7 =  26

r8 =   8, r9 =   1, 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

m16=   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
```

2. Case 2

執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) addi $2, $0, 3 (2) sw $2, 0($0) (3) addi $2, $0, 1 (4) sw $2, 4($0) (5) sw $0, 8($0) (6) addi $2, $0, 5 (7) sw $2, 12($0) (8) addi $2, $0, 0 (9) addi $5, $0, 16 (10) addi $8, $0, 2 (11) beq $0, $0, 2 (12) addi $2, $2, 4 (13) bge $2, $5, 6 (14) lw $3, 0($2) (15) bgt $3, $8, 1 (16) beq $0, $0, -5 (17) addi $3, $3, 1 (18) sw $2, 0($3) (19) beq $0, $0, -8	(1) r2 = r0+3 = 0+3 = 3 (2) Store r2=3 to M0, M0 = 3 (3) r2 = r0+1 = 1 (4) Store r2=1 to M1, M1 = 1 (5) Store r0=0 to M2, M2 = 0 (6) r2 = r0+5 = 0+5 = 5 (7) Store r2=5 to M3, M3 = 5 (8) r2 = r0+0 = 0+0 = 0 (9) r5 = r0+16 = 0+16 = 16 (10) r8 = r0+2 = 0+2 = 2 (11) (r0=0) = (r0=0), go to (14) (12) Load M0=3 to r3, r3 = 3 (13) (r3=3) > (r8=2), go to (17) (14) r3 = r3+1 = 3+1 = 4 (15) Store r3=4 to M0, M0 = 4 (16) (r0=0) = (r0=0), go to (12) (17) r2 = r2+4 = 0+4 = 4 (18) (r2=4) ≧/ (r5=16), No branch (19) Load M1=1 to r3, r3 = 1 (20) (r3=1) ≯ (r8=2), No branch (21) (r0=0) = (r0=0), go to (12) (22) r2 = r2+4 = 4+4 = 8	r2 = 16 r3 = 6 r5 = 16 r8 = 2 others = 0	M0 = 4 M1 = 1 M3 = 6 others = 0

	(23) (r2=8) ≧/ (r5=16), No branch (24) Load M2=0 to r3, r3 = 0 (25) (r3=0) ≯ (r8=2), No branch (26) (r0=0) = (r0=0), go to (12) (27) r2 = r2+4 = 8+4 = 12 (28) (r2=12) ≧/ (r5=16), No branch (29) Load M3=5 to r3, r3 = 5 (30) (r3=5) > (r8=2), go to (17) (31) r3 = r3+1 = 5+1 = 6 (32) Store r3=6 to M3, M3 = 6 (33) (r0=0) = (r0=0), go to (12) (34) r2 = r2+4 = 12+4 = 16 (35) (r2=16) ≧ (r5=16), branch		

程式執行結果截圖

MAX_COUNT = 63

```
################################## clk_count = 63###################################
=====================================Register=====================================
r0 =    0, r1 =    0, r2 =   16, r3 =    6, r4 =    0, r5 =   16, r6 =    0, r7 =    0

r8 =    2, r9 =    0, 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 =    4, m1 =    1, m2 =    0, m3 =    6, m4 =    0, m5 =    0, m6 =    0, m7 =    0

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

m16=    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
```

3. Case 3

執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) begin: (2) addi $1, $0, 3 (3) addi $2, $0, 4 (4) addi $3, $0, 1 (5) sw $1, 4($0) (6) add $4, $1, $1 (7) or $6, $1, $2 (8) and $7, $1, $3 (9) sub $5, $4, $2 (10) slt $8, $1, $2 (11) beq $1, $2, begin (12) lw $10, 4($0)	(1) r1 = r0+3 = 0+3 = 3 (2) r2 = r0+4 = 0+4 =4 (3) r3 = r0+1 = 0+1 =1 (4) Store r1=3 to m1, M1 = 3 (5) r4 = r1+r1 = 3+3 =6 (6) r6 = r1 OR r2 =7 (7) r7 = r1 AND r3 = 1 (8) r5 = r4 – r2 = 2 (9) (r1=3) < (r2=4), r8 = 1 (10) (r1=3) ≠ (r2=4), no branch (11) Load m1=3 to r10, r10 = 3	r0 = 0 r1 = 3 r2 = 4 r3 = 1 r4 = 6 r5 = 2 r6 = 7 r7 = 1 r8 = 1 r9 = 0 r10 = 3 r11~r31 = 0	M0 = 0 M1 = 3 M2~M31 = 0

程式執行結果截圖

MAX_COUNT = 15

```
################################ clk_count = 15###################################
=====================================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

m16=    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
```

4. Case 4

執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) begin: (2) addi $1, $2, 4 (3) addi $2, $0, 4 (4) addi $3, $0, 1 (5) sw $1, 4($0) (6) add $4, $1, $1 (7) or $6, $1, $2 (8) and $7, $1, $3 (9) sub $5, $4, $2 (10) slt $8, $1, $2 (11) beq $1, $2, begin (12) lw $10, 4($0)	(1) r1 = r2+4 = 0+4 = 4 (2) r2 = r0+4 = 0+4 =4 (3) r3 = r0+1 = 0+1 =1 (4) Store r1=4 to m1, M1 = 4 (5) r4 = r1+r1 = 4+4 =8 (6) r6 = r1 OR r2 =4 (7) r7 = r1 AND r3 = 0 (8) r5 = r4 – r2 = 4 (9) (r1=4) ≮ (r2=4), r8 = 0 (10) (r1=4) = (r2=4), go to begin (11) r1 = r2+4 = 4+4 = 8 (12) r2 = r0+4 = 0+4 =4 (13) r3 = r0+1 = 0+1 =1 (14) Store r1=8 to m1, M1 = 8 (15) r4 = r1+r1 = 8+8 =16 (16) r6 = r1 OR r2 =12 (17) r7 = r1 AND r3 = 0 (18) r5 = r4 – r2 = 12 (19) (r1=8) ≮ (r2=4), r8 = 0 (20) (r1=8) ≠ (r2=4), no branch (21) Load m1=8 to r10, r10 = 8	r0 = 0 r1 = 8 r2 = 4 r3 = 1 r4 = 16 r5 = 12 r6 = 12 r7 = 0 r8 = 0 r9 = 0 r10 = 8 others = 0	M0 = 0 M1 = 8 M2~M31 = 0
程式執行結果截圖			
MAX_COUNT = 26			

```
################################ clk_count = 26###################################
=====================================Register=====================================
r0 =    0, r1 =    8, r2 =    4, r3 =    1, r4 =   16, r5 =   12, r6 =   12, r7 =    0

r8 =    0, r9 =    0, r10=    8, 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 =    8, 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

m16=    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
```

5. Case 5

執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) addi $1, $0, 16 (2) addi $2, $1, 4 (3) addi $3, $0, 8 (4) sw $1, 4($0) (5) lw $4, 4($0) (6) sub $5, $4, $3 (7) add $6, $3, $1 (8) addi, $7, $1, 10 (9) and $8, $7, $3 (10) addi $9, $0, 100	(1) r1 = r0+16 = 0+16 = 16 (2) r2 = r1+4 = 16+4 =20 (3) r3 = r0+8 = 0+8 =8 (4) Store r1=16 to M1, M1 = 16 (5) Load M1=16 to r4, r4 = 16 (6) r5 = r4-r3 = 16-8 =8 (7) r6 = r3+r1 = 8+16 =24 (8) r7 = r1+10 = 16+10 = 26 (9) r8 = r7 AND r3 = 8 (10) r9 = r0=100 = 0+100 = 100	r0 = 0 r1 = 16 r2 = 20 r3 = 8 r4 = 16 r5 = 8 r6 = 24 r7 = 26 r8 = 8 r9 = 100 Others = 0	M0 = 0 M1 = 16 Others = 0

程式執行結果截圖

MAX_COUNT = 30

```
################################ clk_count = 30###################################
=====================================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

m16=   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
```

Problems you met and solutions:

這次的實作式基於 lab4 進行修改，並加上 forwarding unit 和 hazard detection unit。因為在 lab4 的實作中已經將偵測是否會 branch 的 comparator、adder 以及 shifter 移動至 ID stage 中，所以只需要做針對新增的指令做先關的實作，以及重新計算 pipelind register 每個 bit 所儲存的資料就好。

比較麻煩的是 forwarding unit 的實作，因為一開始只有考慮到一般指令會需要做 forwarding 的條件，忽視在 ID stage 的 comparator 也會有 data hazard 的問題，導致跑出的結果錯誤，因此會需要將 forwarding unit 分成一般指令的 forwarding 以及 branch 指令的 forwarding 做處理。但找到錯誤後就比較好做處理了，基本上就是依照一般指令做 forwarding 的條件，把每個條件往前一個 stage 就會變成 branch 指令做 forwarding 的條件，在基於以上條件輸出控制訊號 Forward_C 和 Forward_D 到 MUX 中選擇會進入 comparator 的資料是從哪裡來的。

Summary:

透過這次 lab 可以更加認識 pipeline CPU 各個指令的運作方式，並且以 lab4 為基礎，進行電路結構上的調整，使得 CPU 可以同時執行多個指令，並且做 hazard 以及 forwrding 的處理，增加 CPU 的效能。這次的 pipeline CPU 需要考慮較多的情況，把 branch 指令搬到 ID stage 之後，對於其可能發生的 hazard 以及 forwarding 做處理，需要花一些時間去釐清要連接哪一條電路，以及找出其可能會發生的錯誤並做相關處理，才能做出更有效率的 pipeline CPU 實作方案。