第三章　Pipelined MIPS CPU 規劃設計

在本研究中實作的記憶體相關指令、算術、邏輯運算指令、分支指令，已包含 MIPS 指令集中的主要指令。在指令集編碼方面，遵循 MIPS CPU 指令集編碼未做任何修改，唯獨浮點數指令更改與 R-Type 指令格式一致。本研究所實作指令集編碼如表 3-1 所示，而實作指令語法表如表 3-2 所示。有 21 道固定點指令，4 道浮點指令，共 25 道指令。

表 3-1　實作指令集編碼一覽表

Encoding	Encoding						Inst. Type
Mnemonic	31 26	25 21	20 16	15 11	10 6	5 0	
LW	100011	sssss	Ttttt	iiiii	iiiii	iiiiii	M
SW	101011	sssss	ttttt	iiiii	iiiii	iiiiii	M
ADD	000000	sssss	ttttt	ddddd	00000	100000	R
ADDU	000000	sssss	ttttt	ddddd	00000	100001	R
SUB	000000	sssss	ttttt	ddddd	00000	100010	R
SUBU	000000	sssss	ttttt	ddddd	00000	100011	R
AND	000000	sssss	ttttt	ddddd	00000	100100	R
OR	000000	sssss	ttttt	ddddd	00000	100101	R
XOR	000000	sssss	ttttt	ddddd	00000	100110	R
NOR	000000	sssss	ttttt	ddddd	00000	100111	R
SLL	000000	sssss	ttttt	ddddd	hhhhh	000000	R
SRL	000000	-----	ttttt	ddddd	hhhhh	000010	R
SRA	000000	-----	ttttt	ddddd	hhhhh	000011	R
ADDI	001000	sssss	ttttt	iiiii	iiiii	iiiiii	I
ADDIU	001001	sssss	ttttt	iiiii	iiiii	iiiiii	I
ANDI	001100	sssss	ttttt	iiiii	iiiii	iiiiii	I
ORI	001101	sssss	ttttt	iiiii	iiiii	iiiiii	I
XORI	001110	sssss	ttttt	iiiii	iiiii	iiiiii	I
BEQ	000100	sssss	ttttt	iiiii	iiiii	iiiiii	B
BNE	000101	sssss	ttttt	iiiii	iiiii	iiiiii	B
J	000010	iiiii	iiiii	iiiii	iiiii	iiiiii	J
F.ADD	010001	sssss	ttttt	ddddd	00000	000001	R
F.SUB	010001	sssss	ttttt	ddddd	00000	000010	R
F.MUL	010001	sssss	ttttt	ddddd	00000	000011	R
F.DIV	010001	sssss	ttttt	ddddd	00000	000100	R

表 3-2　實作指令語法一覽表

NO.	Mnemonic	OP Code	Syntax		Operation
1	LW	100011	lw	$t, offset($s)	$t=MEM[$s+offset]
2	SW	101011	sw	$t, offset($s)	MEM[$s+offset] =$t
3	ADD	000000	add	$d, $s, $t	$d =$s + $t
4	ADDU	000000	addu	$d, $s, $t	$d =$s + $t
5	SUB	000000	sub	$d, $s, $t	$d =$s - $t
6	SUBU	000000	subu	$d, $s, $t	$d =$s - $t
7	AND	000000	and	$d, $s, $t	$d =$s & $t
8	OR	000000	or	$d, $s, $t	$d =$s | $t
9	XOR	000000	xor	$d, $s, $t	$d =$s ^ $t
10	NOR	000000	nor	$d, $s, $t	$d = ~($s | $t)
11	SLL	000000	sll	$d, $t, shamt	$d =$s << shamt
12	SRL	000000	srl	$d, $t, shamt	$d =$s >> shamt
13	SRA	000000	sra	$d, $t, shamt	$d =$s >> shamt
14	ADDI	001000	addi	$t, $s, immed	$t =$s + immed
15	ADDIU	001001	addiu	$t, $s, immed	$t =$s + immed
16	ANDI	001100	andi	$t, $s, immed	$t =$s & immed
17	ORI	001101	ori	$t, $s, immed	$t =$s | immed
18	XORI	01110	xori	$t, $s, immed	$t =$s ^ immed
19	BEQ	000100	beq	$s, $t, offset	if $s==$t advance_pc
20	BNE	000101	bne	$s, $t, offset	if $s!=$t advance_pc
21	J	000010	j	Target	pc=pc&(000000| target)
22	F.ADD	010001	f.add	$d, $s, $t	$d =$s + $t
23	F.SUB	010001	f.sub	$d, $s, $t	$d =$s - $t
24	F.MUL	010001	f.mul	$d, $s, $t	$d =$s × $t
25	F.DIV	010001	f.div	$d, $s, $t	$d =$s ÷ $t

第一節 指令擷取階段（IF Stage）

指令擷取（Instruction Fetch）簡稱 IF 。IF 階段主要負責指令擷取的動作。圖 3-1 展示了 Pipelined MIPS CPU 在 IF 階段的資料路徑。

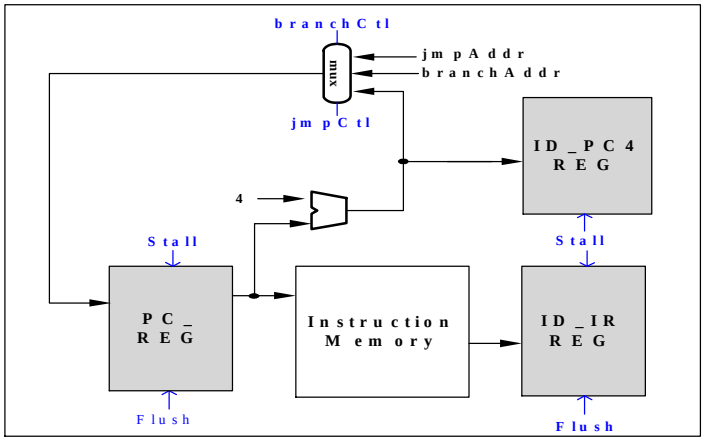

圖 3-1　IF Stage 資料路徑

指令擷取步驟如下：（1）將 PC_REG 的輸出值送到指令記憶體 IM（Instruction Memory），進行指令提取；（2）根據 Stall 和 Flush 控制信號決定是否載入新的 PC 值與指令至管線暫存器、不做任何動作亦或者將管線暫存器清為零。

IF Stage 除了擷取指令外，還要決定下一個 Cycle 的 PC 值。這個動作可由多工器來完成。利用 Beq、Bne、Jmp 等旗號對多工器做控制，選擇 PC 值是正常執行的 PC+4 或 Beq、Bne、Jmp 等分叉相對位址。

此階段所包含的電路有：程式計數器（PC）、指令記憶體（Instruction Memory）、PC 加法器與多工器。其多工器是用來選擇下一個 PC 值，最後還有一個管線暫存器，用來保存時脈之間的臨時資料與控制訊號。

本研究所設計的 Instruction Memory 是使用一塊 256x32 Bits 大小之 Block RAM。Block RAM 係利用 Xilinx ISE 中之 Core Generator 產生，其資料可參考[9]。

第二節 指令解碼階段（ID Stage）

指令解碼（Instruction Decode）簡稱 ID。ID 階段主要是負責指令的解碼動作和暫存器陣列的讀取動作。圖 3-2 展示了 Pipelined MIPS CPU 在 ID 階段的資料路徑。以下將說明圖中各個 Block 所負責的動作。

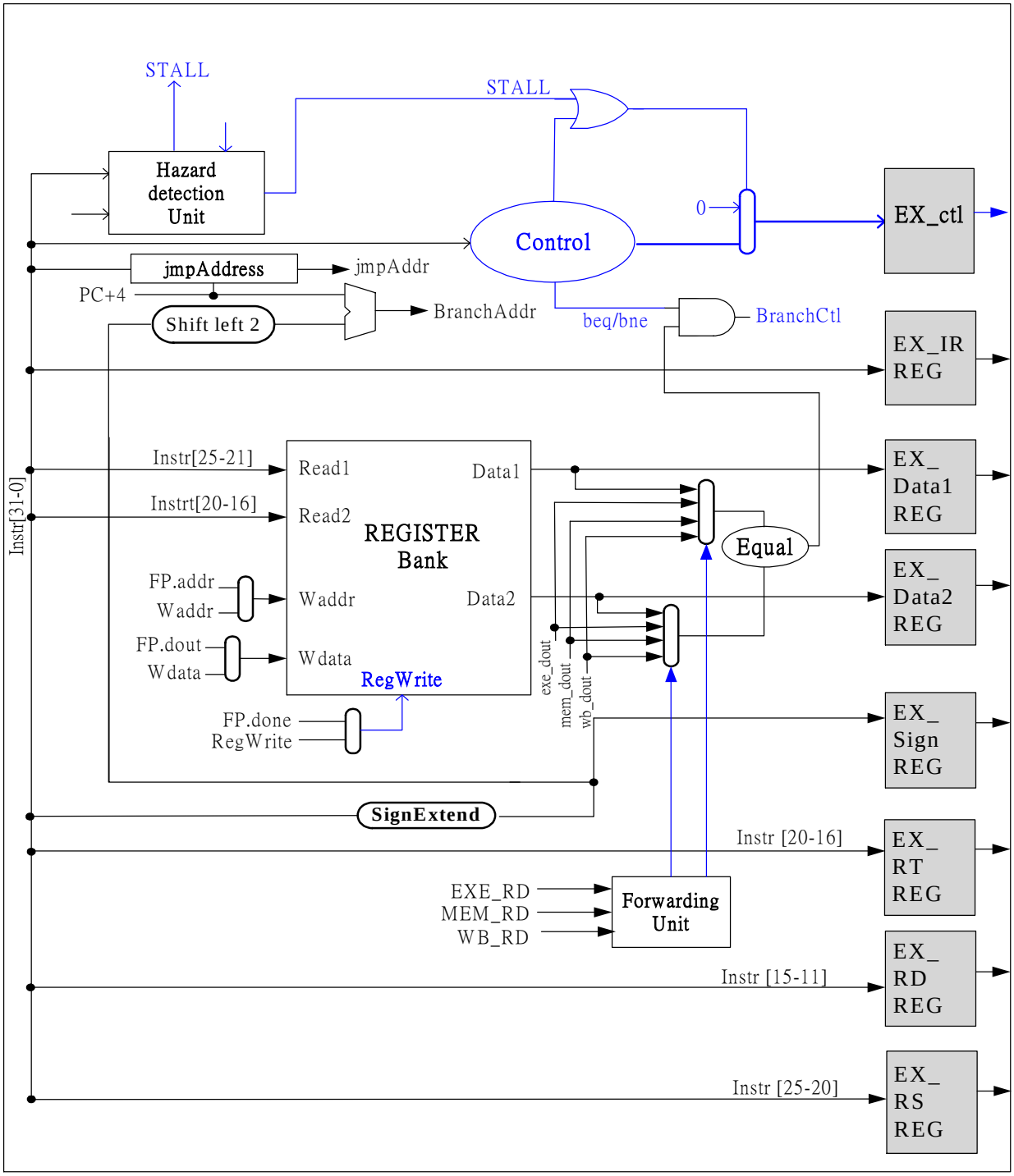

圖 3-2　ID Stage 資料路徑

- Control Unit ：負責指令的解碼（Decode）動作，並輸出管線中每個階段所需的控制信號。其細節參考第六節。

- Register Bank：包含 32 個 GPR，每個暫存器資料長度皆為 32-bits。內部電路方塊圖如圖 3-3 所示。Register Bank 的資料輸出到實際電路中，32 個暫存器是同時將所存資料送出的，因此在內部規劃三個多工器，由暫存器位址選擇，將所指定的資料送至外部連接埠。Data1、Data2 主要是 MIPS CPU 內的 ALU 模組的兩個運算資料來源。RegDout 則是接至外部 I/O 裝置顯示用，由硬體顯示電路可觀察 MIPS CPU 內部 Register Bank 的資料內容。5x32 Decoder 是根據 RegWrite 和 Waddr 來產生 32 個暫存器寫入致能信號，完成指定暫存器資料寫入功能[23]。

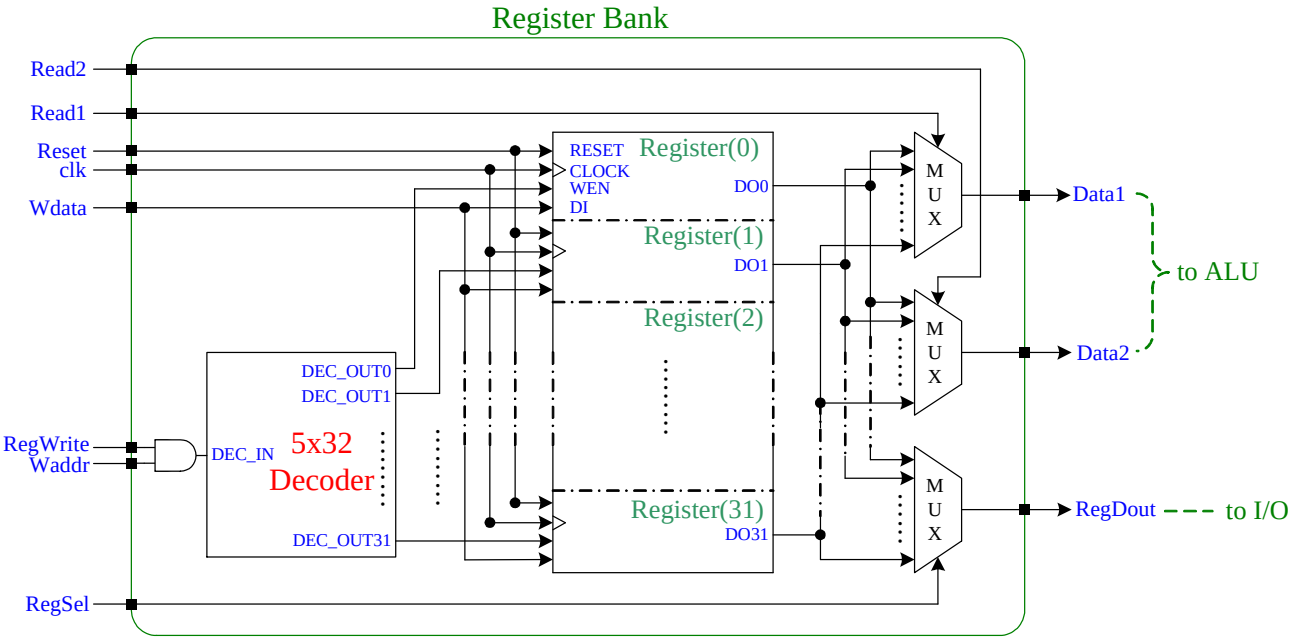

圖 3-3　Register Bank 內部電路方塊圖

- Sign Extend：將指令中的 0~15bits 擴充為 32bits，以便提供 EXE Stage 的 Immediate 與 Branch Address 所需的值。

- Jmp Address：提供 Jamp 指令所需的位址。

- Shift Left 2：提供 Branch 指令所需的位址。

- Equal：判斷 Branch 指令是否成立。

- Forward Unit：判斷是否將 EXE 階段執行的結果、MEM 階段執行的結果與 WB 階段執行的結果前饋（forwarding）到 ID 階段。其細節參考第七節。
- Hazard Detection Unit：用來偵測 Load-use Data Hazard 的發生。其細節參考第七節。

第三節 指令執行階段（EXE Stage）

指令執行（Instruction Execute）簡稱 EXE 。EXE Stage 主要負責一般算術邏輯運算和記憶體存取位址的計算，而本研究加入浮點運算功能，所以在 EXE Stage 包含浮點單元，如圖 3-4 所示。以下將說明圖中各個 Block 所負責的動作。

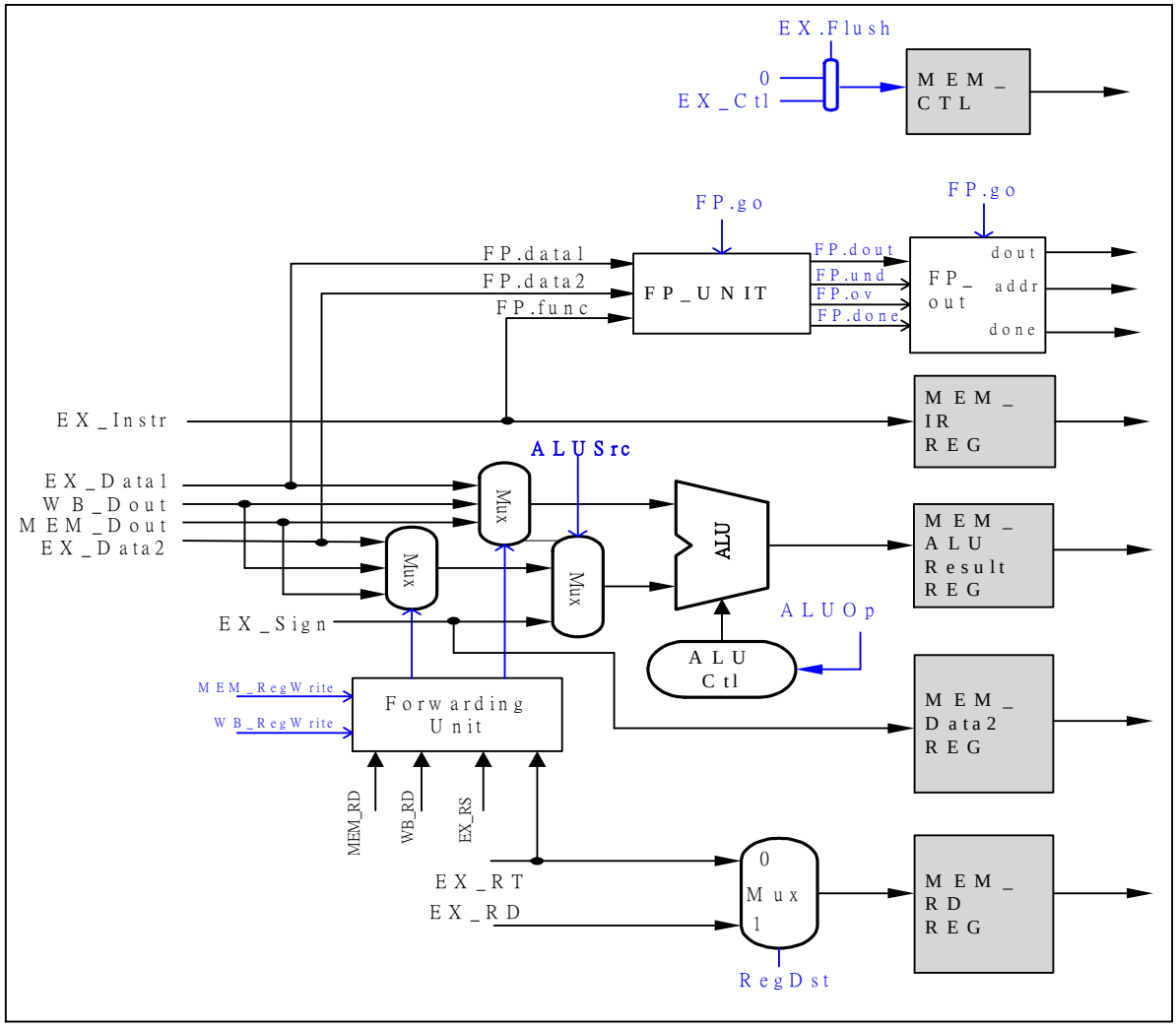

圖 3-4　EXE Stage 資料路徑

● ALU：算術邏輯運算單元是整個 MIPS CPU 運算執行核心，功能為一般算術邏輯運算和產生各種運算旗標。ALU 模組執行的運算類別有：算術運算、邏輯運算、移位運算。ALU 內部方塊圖如圖 3-5 所示，其中 Barrel Shifter 是一個執行快速移位的模組，使用 **Mano**[20]中提到的階層式的移位設計。基本的移位轉換為左移（Shift Left）與右移（Shift Right），資料載入暫存器中，下一步進行移位轉換，再將結果傳回目的暫存器中。Barrel Shifter 內部電路方塊圖如圖 3-6 所示。而 SHIFT_FUNC 的編碼如表 3-3。

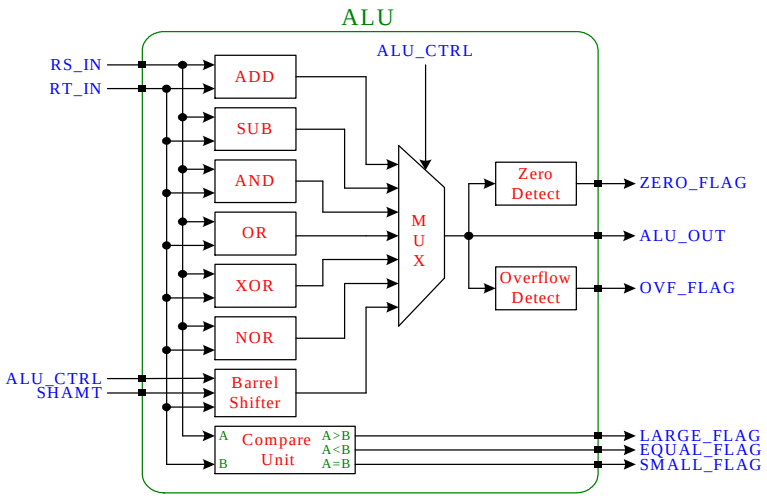

圖 3-5　ALU 內部電路方塊圖

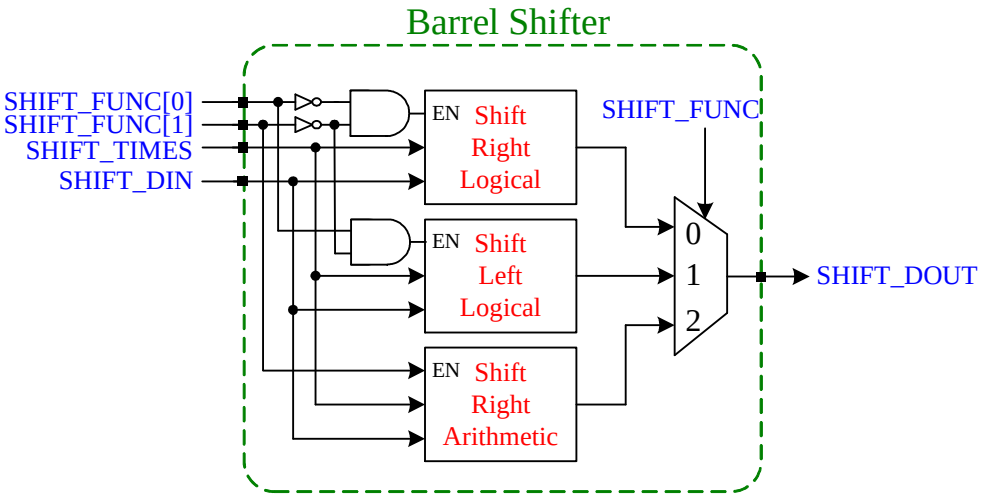

圖 3-6　Barrel Shifter 內部電路方塊圖

表 3-3　SHIFT_FUNC[1:0] Encoding

Shift Mode \ Encoding		SHIFT_ FUNC[1:0]	Function
SRL	Shift Right Logical	00	SHIFT_OUT[31:0] = SHIFT_IN srl (SHIFT_TIMES)
SLL	Shift Left Logical	01	SHIFT_OUT[31:0] = SHIFT_IN sll (SHIFT_TIMES)
SRA	Shift Right Arithmetic	10、11	SHIFT_OUT[31:0] = SHIFT_IN sra (SHIFT_TIMES)

●ALU CTL：主要在產生選擇 ALU 模組的運算子。由於 ALU 必須執行 R-Type、I-Type 運算，因此需要一個 ALU 控制單元，綜合判斷 R-Type 指令中的 Function Code 和 OP Code，而輸出 ALU 運算子選擇。此模組須與 ALU 模組整合，組成完整 ALU 運算單元。ALU_OP 及 ALU_FUNC 編碼如表 3-4 及表 3-5 所示。ALU 控制單元依據表 3-4 及表 3-5 進行編碼設計，產生對應的編碼輸出，正確啟動 ALU 模組進行算術邏輯運算[23]。

表 3-4　ALU_OP Encoding

Mnemonic or Function \ Encoding		ALU_OP[3:0]
R-TYPE		0010
I-TYPE	ADDI	0011
	ADDIU	0100
	ANDI	0101
	ORI	0110
	XORI	0111
B-TYPE		0001
Program Counter ADD		0000
Program Counter SUB		0001

表 3-5　ALU_FUNC[5:0] Encoding

Encoding / Mnemonic	ALU_FUNC[5:0]
ADD、ADDI、ADDIU	100000
ADDU	100001
SUB	100010
SUBU	100011
AND、ANDI	100100
OR、ORI	100101
XOR、XORI	100110
NOR	100111
SLL	000000
SRL	000010
SRA	000011

- ● Forward Unit：判斷是否將 MEM 階段的結果與 WB 階段的結果前饋（forwarding）到 EXE 階段。提供 ALU 所使用。詳細說明請參考第七節。
- ● Floating Unit：執行浮點之加減乘除運算。詳細說明請參考第八節。

第四節 記憶體存取階段（MEM Stage）

記憶體存取（Data Memory）簡稱 MEM。MEM Stage 主要在處理 Data Memory（DM）的存取動作。如圖 3-7 所示。DM 設計之方式是使用一塊 256x32 Bits 大小之 Block RAM。Block RAM 係利用 Xilinx ISE 中之 Core Generator 產生。由 Mem Write 控制是否決定寫入資料至 DM 中

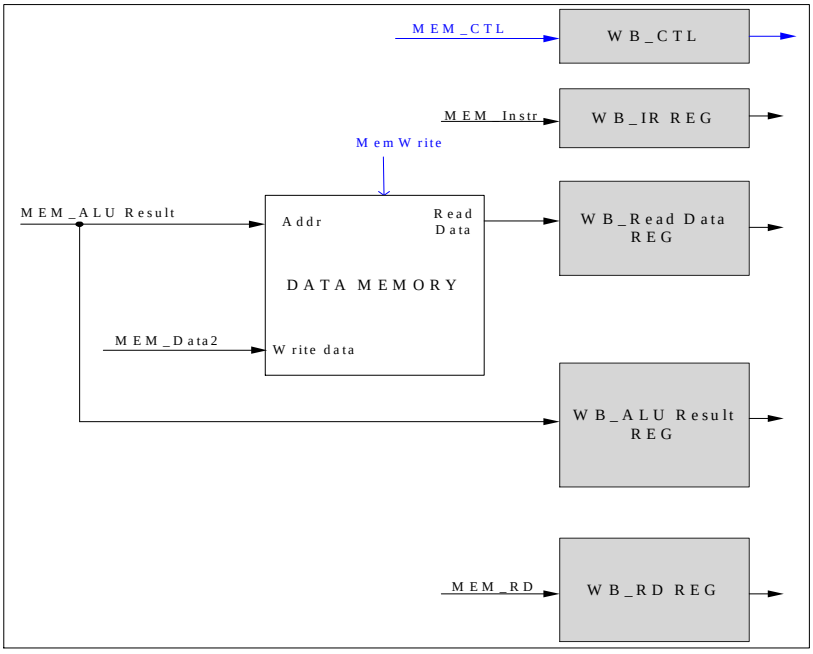

圖 3-7　MEM Stage 資料路徑

第五節 資料寫回階段（WB Stage）

資料寫回（Write Back）簡稱 WB。WB Stage 主要是負責將結果寫回 Register Bank 中。如圖 3-8 所示。其中 MemtoReg 用來選擇寫回的資料是由 Data Memory 提供或者 ALU 計算的結果。而多工器之輸出、WB_RegWrite、WB_RD 都將傳回 ID Stage，以供 Register Bank 寫入。

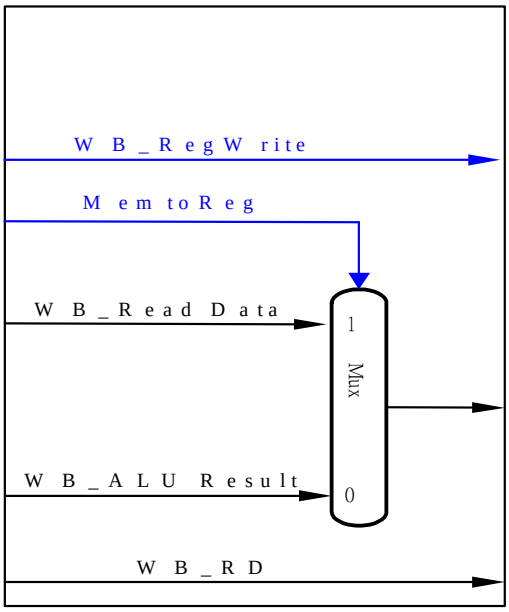

圖 3-8　WB Stage 資料路徑

第六節 控制單元（Control Unit）的設計

控制單元(CU)是跟 ID Stage 同步運作的。主要根據不同的指令模式，作各種控制訊號的輸出，並存放於 ID/EXE 管線暫存器裡，參考圖 3-9。

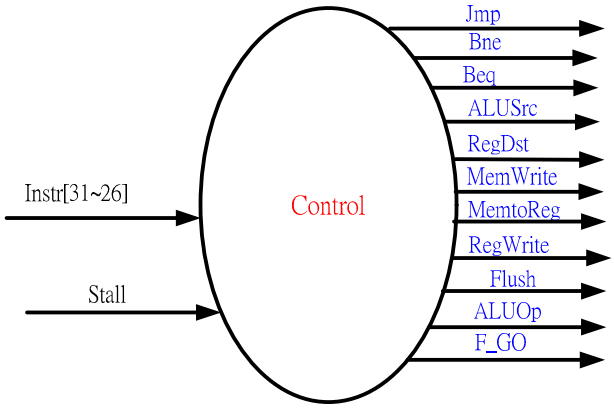

圖 3-9　Control Unit

主要輸出控訊號如下：

1、ALUOp 旗號為 ALU 操作碼。

2、 ALUSrc 旗號為選擇 ALU 輸入是由 Register Bank 讀出或者立即值。

3、RegDst 旗號選擇回寫之暫存器為 Rt 或者 Rd。

4、是否為 LW 指令，要將資料記憶體內含寫入暫存器 MemtoReg。

5、是否為 R-type 或 Lw 或 I-type 的資料等指令的暫存器回寫 RegWrite 旗號。

6、是否為 Sw 指令，作記憶體資料寫入控制 MemWrite 旗號。

7、是否為 Beq 指令，Branch 旗號控制。

8、是否為 Bne 指令，nBranch 旗號控制。

9、是否為 Jmp 指令，作無條作程式分岔控制。

10、F_GO 判斷是否為浮點數指令。

當為分支指令時，設定 Flush 為 1，將適當之管線暫存器資料清為零。當輸入 Stall 為 1 時，會將 ALUOp、ALUSrc、RegDst、MemtoReg、RegWrite、MemWrite 等訊號設定為零，功能為暫停（Stall）也相當於做 nop 指令（即空運算）。

第七節 管線的控制（Pipelined Control）

本節主要在說明管線的三大危障（Hazard）在 CPU 中的解決方案。

壹、結構危障（Structural Hazard）的處理

本研究使用 Harvard Architecture 設計記憶體，讓對外的指令存取介面和資料存取介面是分開的，也就是使用了二顆記憶體，可同時讀取指令及存取資料。所以在 IF Stage 與 MEM Stage 同時存取外部記憶體時並不會發生 Structural Hazard。

而在 ID Stage 和 WB Stage 同時對 Register Bank 執行讀取與寫回的動作時，我們解決的方式就是：在時脈正緣進行寫入動作，在時脈負緣進行讀取的動作，如此設計就不會產生 Structural Hazard 的問題了。

還有一個會發生 Structural Hazard 的情況：有一指令的運算無法在下一個 Clock 內完成，例如浮點指令在執行單元需花費多個 Clock，因此當浮點指令完成並寫回 Register Bank 時，另一道指令也要寫回 Register Bank，此時就造成了 Structural Hazard 的發生。圖 3-10 說明解決的方式。

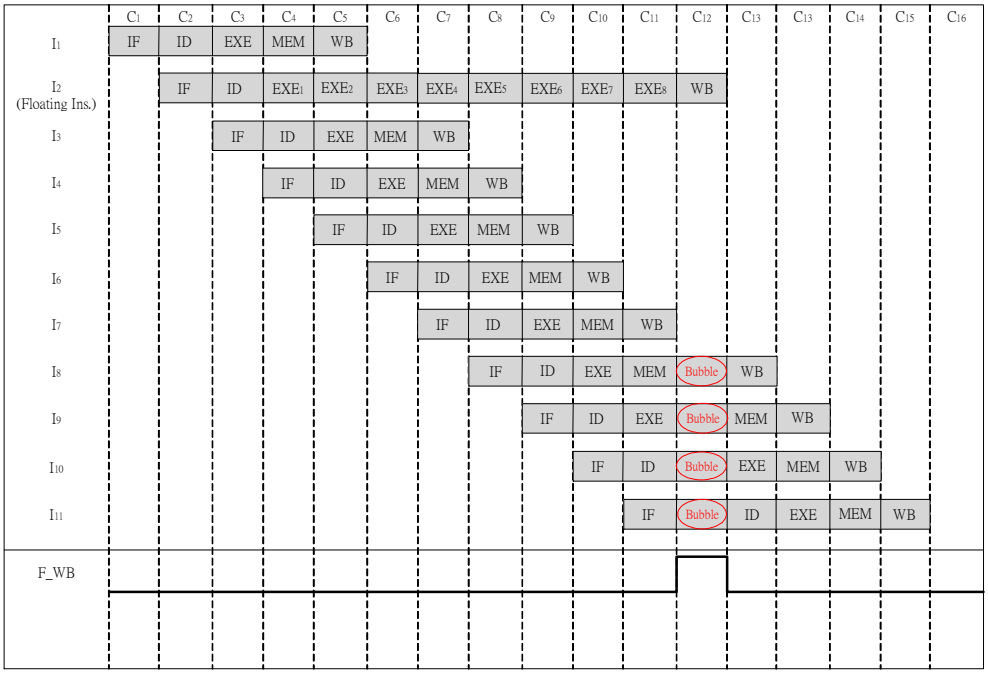

圖 3-10　指令需要執行多重週期時的管線控制圖

圖中 I_1~I_{11} 為連續的指令。假設 I_2 為一道浮點運算指令，需佔用 8 個 Clock 的執行單元電路才能完成，而後寫回至 Register Bank 中，由於所設計之指令集中，浮點運算只包含加、減、乘和除，其運算結果不用至 MEM Stage 進行讀取的動作，直接對 Register Bank 寫回，方可減少一個 Clock 的浪費。

在浮點指令在 EXE Stage 運算完成後，將 F_done 訊號線拉起（assert），此訊號線將會暫停動作，等待一個 Clock 後再繼續執行，其等待的一個 Clock 即為寫入浮點運算完成之資料。而 I_8 寫回的動作會被暫停一個 Clock 後再寫回；I_9 記憶體存取、I_{10} 指令執行、I_{11} 指令解碼均會延遲一個 Clock 後再運作。

貳、資料危障（Data Hazard）的處理

資料危障發生的情況（如圖 3-11 所示）可分為以下幾種：

Case1: 當 I_2 指令在 EXE 階段需要用到 I_1 指令在 EXE 階段的結果。

Case2: 當 I_3 指令在 EXE 階段需要用到 I_1 指令在 EXE 階段的結果或在 MEM 階段從 Data Memory 讀出的結果。

Case3: 當 I_1 指令為 Load 指令時，I_2 指令的 EXE 階段需要用到 I_1 指令 MEM 階段從 Data Memory 讀出的結果。

Case4: 當 I_2 指令為 Branch 指令且在 ID 階段時，需要用到 I_1 指令在 EXE 階段的結果。

Case5: 當 I_3 指令為 Branch 指令且在 ID 階段時，需要用到 I_1 指令在 EXE 階段的結果。

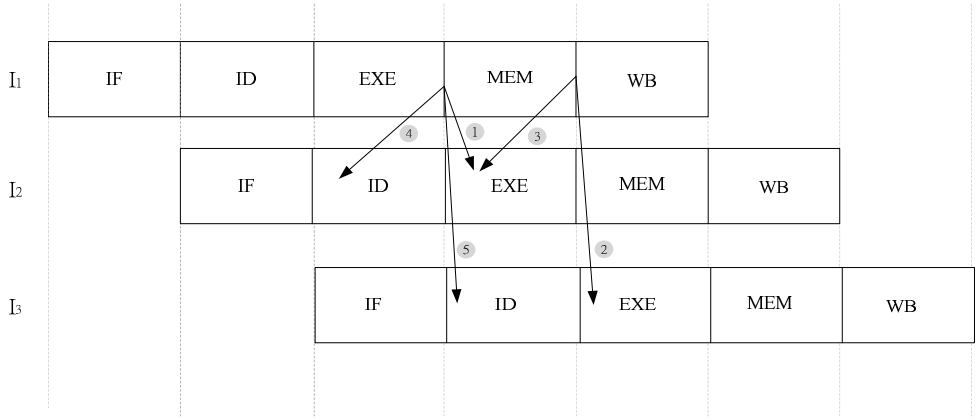

圖 3-11　產生 Data Hazard 的情況

在上述五種可能產生 Data Hazard 的情況中，Case1、Case2、Case4 和 Case5 所造成的 Data Hazard，可用 Forwarding Unit 來解決，如圖 3-12 所示。

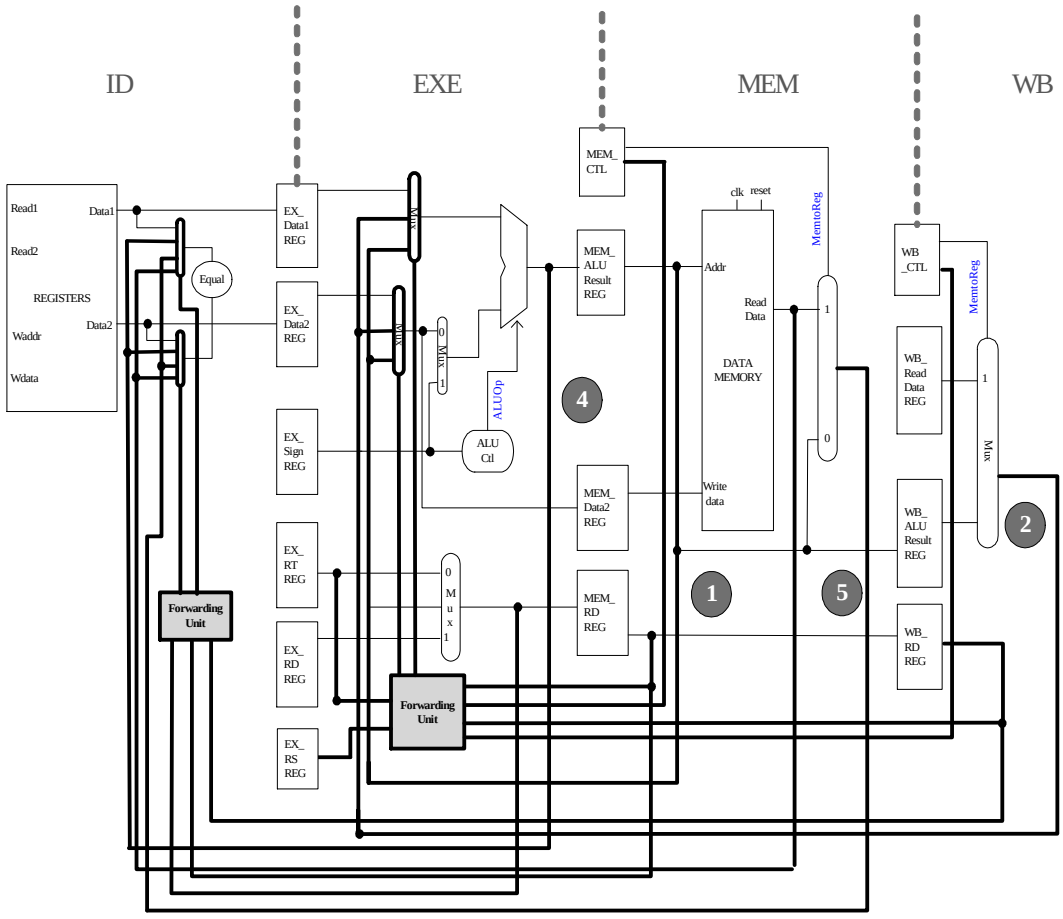

圖 3-12　Forwarding 電路圖

Case 3 所造成的 Data Hazard，稱為 Load-use Data Hazard。這種情況無法用 Forwarding 來解決，必須透過 Hazard Detection Unit 來偵測此情形的發生，並且產生暫停信號。Hazard Detection Unit 是在 ID 階段進行，它可以在 Load 指令和緊跟著之後的指令之間加入暫停。圖 3-13 說明了針對 Load-use Data Hazard 的處理方法。當 I_2 指令進入 ID 階段（C_3）時，Hazard Detection Unit 會偵測 I_1 指令是否為 Load 指令，並且將 I_2 指令所讀取的暫存器編號和 I_1 指令的目的暫存器編號比較是否相同，如果以上兩條件都成立，則 Hazard Detection Unit 的 Flush 會等於 1，將 I_2 指令解碼的結果清為零，此時也會產生 Interlock 信號。在下一個 Clock 中（C_4），重新對 I_2 指令解碼，I_2 指令進入 EXE 階段時（C_5），I_1 指令也已經完成 Data Memory 的讀取動作，I_2 指令透過 Forwarding Unit 來獲得正確的資料。

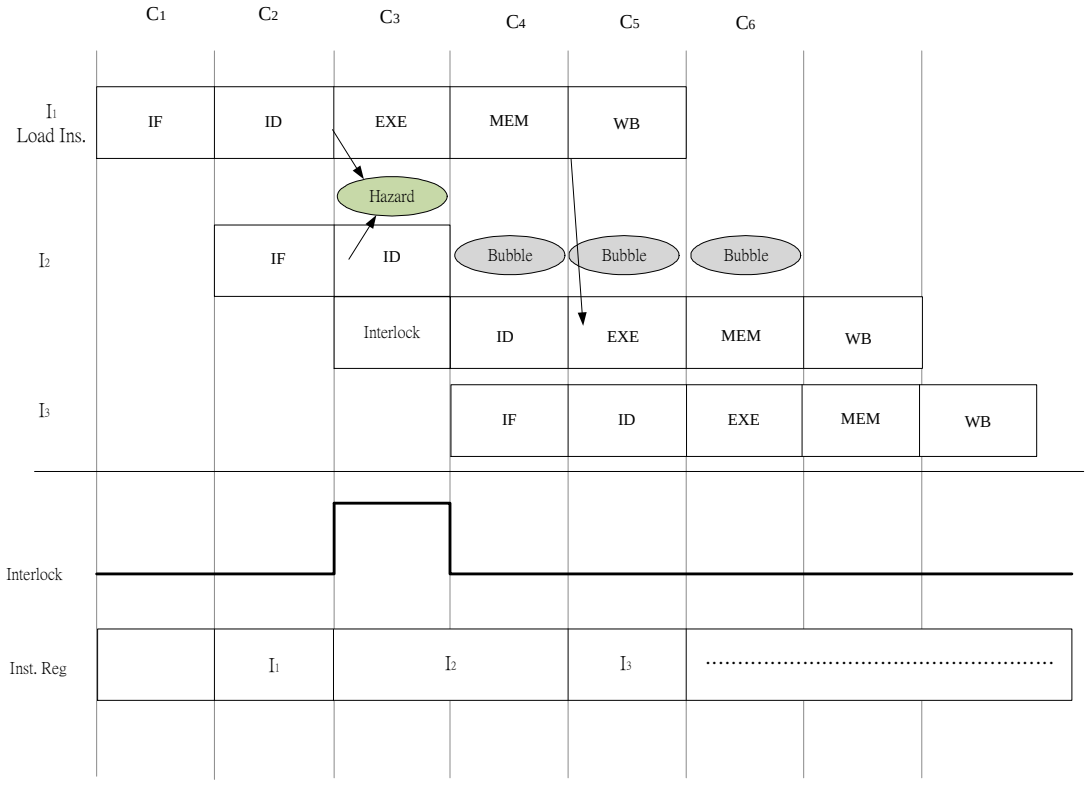

圖 3-13　Load-use Data Hazard 的處理方法

參、控制危障（Control Hazard）的處理

控制危障的產生在於出現 Branch 指令時，也就是擷取到的指令並非需要的指令，使得原本需要的指令不能在適當的時脈週期進行。其解決方法就是暫停（Stall）。因此將 Branch 指令的條件判斷和執行移至 ID 階段中進行。根據是否發生 Data Hazard，由圖 3-14 來說明解決方法。圖中 I_1~I_5 為連續 5 道指令，當 I_4 是分支指令，並且沒有和 I_3 指令有資料相依（Data Dependence）的情況，或許跟 I_2 有資料相依，但 I_2 不是 Load 指令。此情況下，I_4 進入 ID 階段（C_5），開始做指令解碼，同時也會計算分支位址，並且在控制單元判斷是否為分支指令，以及決定分支是否發生；此時此刻 IF 階段會提取下一個 PC 值。若分支條件成立，在 C_6 會將分支位址送至 IF 階段，並且提取分支位址，在 ID 階段的指令和在 EXE 階段的分支指令都會被沖刷（Flush）。

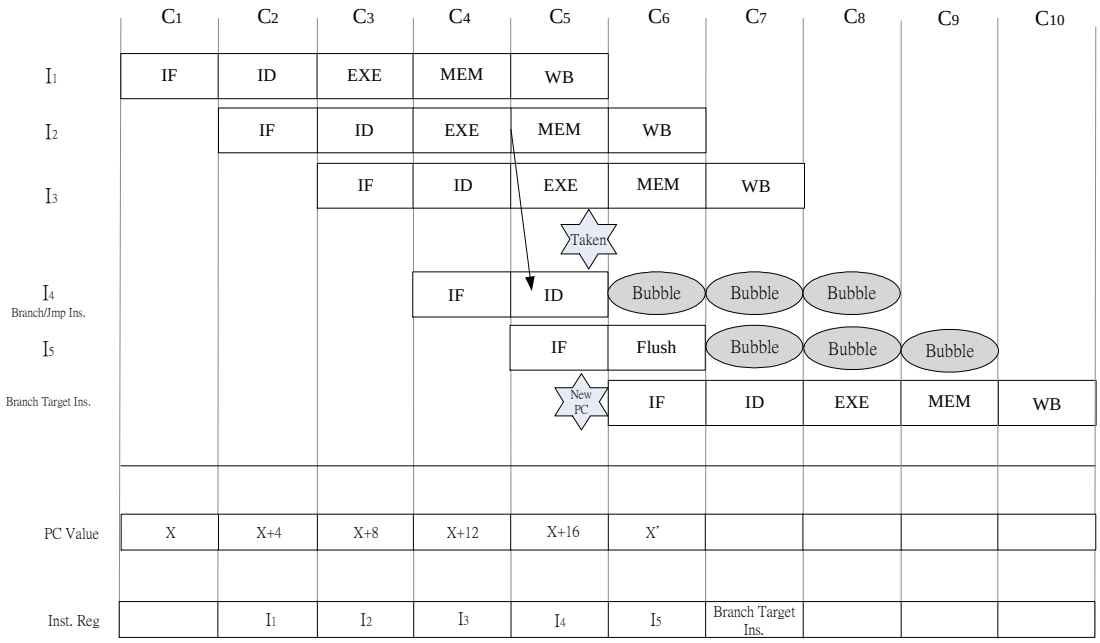

圖 3-14　針對 Branch 指令的管線控制時脈圖

第八節 浮點運算單元（Floating-point Unit）的設計

以上 Pipelined MIPS CPU 皆設計完成並且測試無誤後，再加入浮點運算指令，首先設計一個 32bits 的浮點運算處理器，後續再進行與 Pipelined MIPS CPU 電路之連接。以下敘述浮點的相關知識與設計架構。

壹、IEEE754 浮點數標準格式

本研究參考其格式[17]，IEEE754 是由美國電機電子工程協會（Institute of Electrical Electronic Engineers，IEEE）針對浮點數的格式所制訂的標準，具有 3 種儲存格式:

1、 float precision 格式（亦稱 Single Precision；單倍精度)：利用 32 位元來儲存浮點數。
2、 double precision 格式 (雙倍精度)：利用 64 位元來儲存浮點數。
3、 double-extended precision 格式 (延伸雙倍精度)：利用至少 80 位元來儲存浮點數。

本研究之浮點數是使用 32 位元的「單倍精度」，格式如圖 3-15 所示。

$$(-1)^{s} \times (1 + significand) \times 2^{(exponent - bias)} \quad \text{.....式 3-1}$$

s：正負符號（Sign）由一個位元（bit）來表示

exponent：為指數部分，共 8its，指數範圍為 $2^{-126} \sim 2^{127}$ 。

singificand：為有效數字，共 23bits，放小數用的。

bias：偏差值，IEEE754 單倍精度的偏差值為 127。

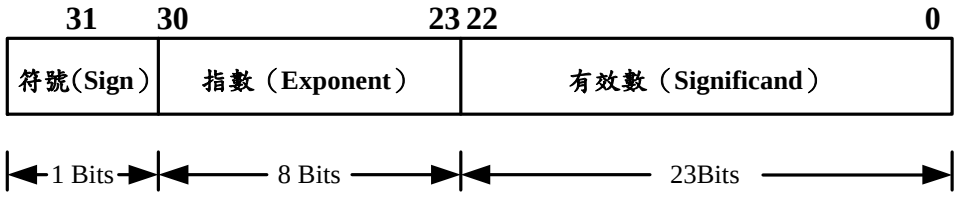

圖 3-15　IEEE754 單倍精度浮點數格式

貳、浮點運算指令格式與定義

浮點運算指令是將 Rn 暫存器與 Rm 暫存器中浮點數值進行運算將運算結果存入 Rd 所指定的暫存器。指令格式如圖 3-16 所示。

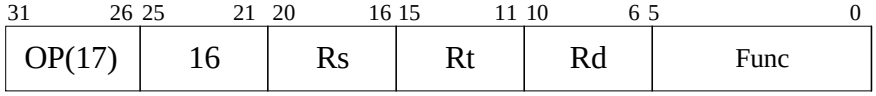

圖 3-16　浮點運算指令格式

[19]有提及，當使用浮點運算處理器時，MIPS 有規劃浮點運算暫存器堆（Register Bank）供輔助使用，以避免跟一般運算資料混淆。但本研究僅實作單倍精度浮點運算處理器，考量並無迫切需要且控制資源消耗之下，並無另外規劃浮點運算暫存器堆，僅與 ALU 共用原來的一般暫存器堆。由於浮點指令與一般定點指令的指令格式不相同，因此參考[25]之方式，修改浮點指令之格式，方可共用相同的硬體架構。如圖 3-17 所示：

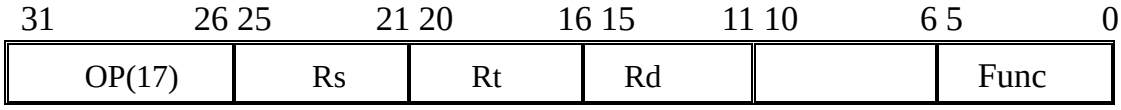

圖 3-17　修改後的浮點指令格式

由圖 3-17 可知，修改後的浮點指令格式，主要是 25~21Bits 的欄位取消，並且將後面 Rs、Rt、Rd 欄位往前移，如此便可符合 R-Type 的格式。之所以取消 25~21Bits 的欄位是原因在於，此欄位原本作為選擇單倍精度或雙倍精度浮點處理器，而本研究僅實作單倍精度浮點處理器，所以此選擇欄位事實上並沒有使用到，故可省略[25]。

參、浮點數之乘法設計

浮點乘法的設計比浮點加法的設計簡單許多。因此，先討論乘法設計，由簡至深，再延伸至加減法設計。浮點數的乘法的設計中，首先檢查任一運算元是否為零，則結果直接輸出零；接下來進行指數的相加，因為指數是以偏移值的形式儲存，所以相加之後會使偏移值加倍，因此必須將指數相加完後的結果再減去偏移值；若有發生指數欠位、溢位等情形，則離開整個浮點乘法運算流程；若無，則將有效數相乘，並將結果做正規化及捨入的動作。浮點乘法演算法流程圖如圖 3-18 所示。

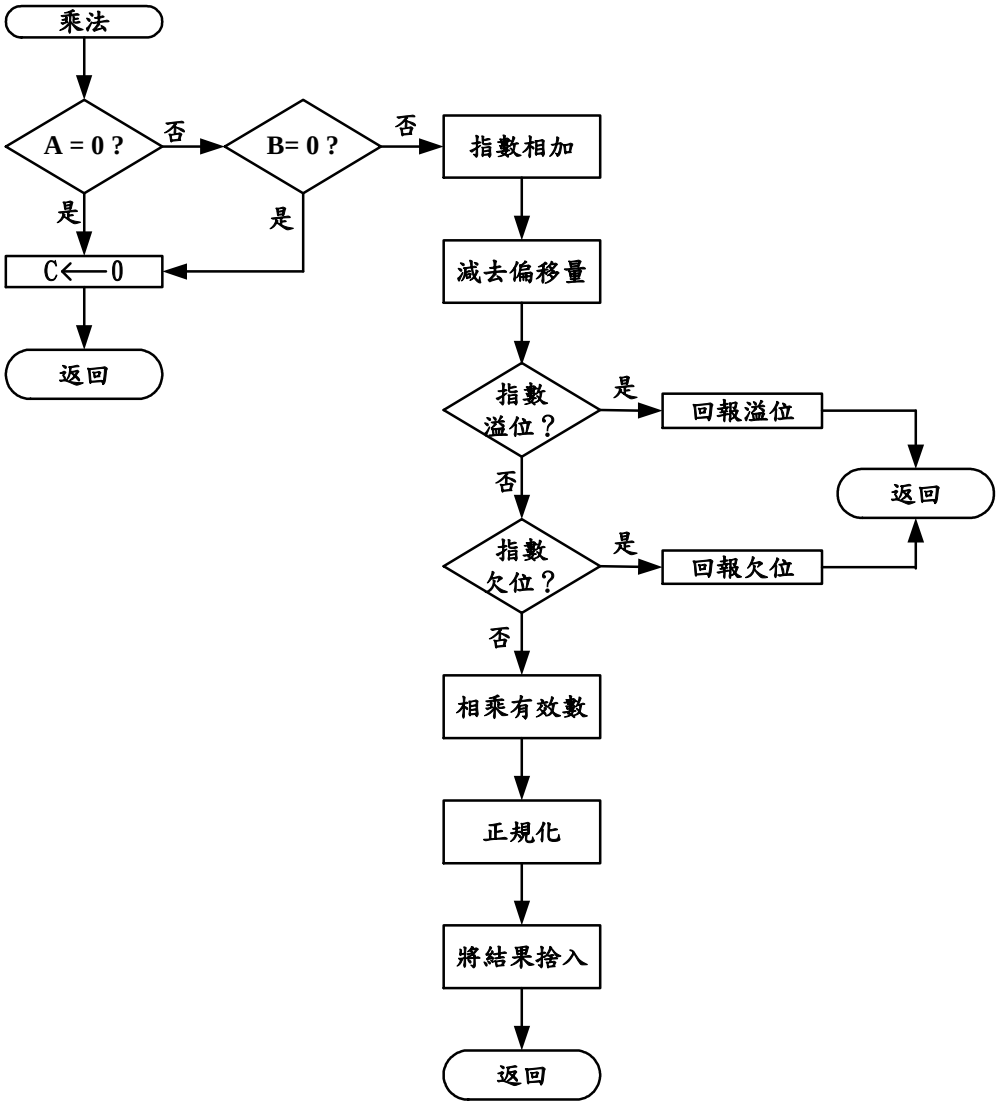

圖 3-18　浮點乘法演算法流程圖

肆、浮點數之除法設計

如圖 3-19 所示，先檢查除數是否為零，若成立，則回報錯誤，以及被除數是否為零，成立則結果輸出為零；下一步將指數相減，此動作會移走偏移值所以必須將偏移值加回去，然後測試指數是否發生欠位或溢位。最後將有效數相除，並做正規化及捨入的動作。

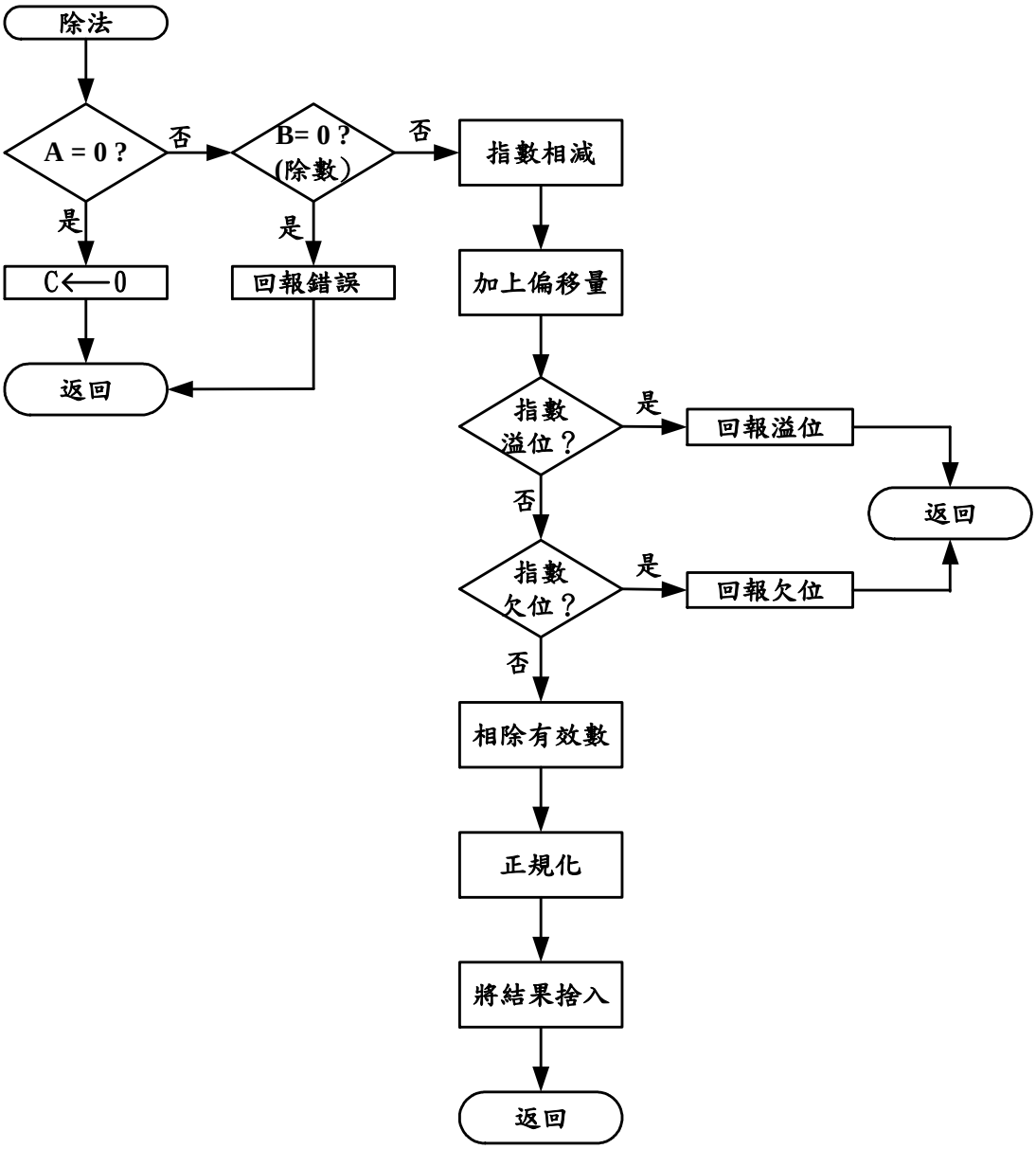

圖 3-19　浮點除法演算法流程圖

伍、浮點數之加減法設計

在浮點數的算術中，加減法的複雜度高於乘除法，是由於要先將兩運算元的指數部分調整為一樣，才可以進行浮點數加減法運算。浮點數加減法演算法流程圖如圖 3-21 所示，浮點數加減法演算法有四個基本步驟：

(1) 運算元零檢查

兩個運算元中，若有任一運算元為零時，則運算結果即直接輸出另一個非零運算元的值；若兩運算元皆為零，則運算結果即輸出為零，不需做任何運算。

(2) 對齊有效數（校正指數）

浮點數要進行加減法運算前，要先將兩方指數部分調整為一樣才可進行運算。

(3) 加減法有效數

指數調整完畢後，接下來進行有效數的加法或減法，運算完畢後，需判斷結果（有效數）是否為零，以及有效數、指數是否有發生溢位，一旦發生則離開整個浮點加減法運算的流程。

(4) 將運算結果正規化

運算結束後，得到的結果可能不符合 IEEE754 的浮點數標準，因此必須將運算結果轉換為正規化的格式。

完成浮點運算器與指令格式定義設計後，所設計之浮點運算單元如圖 3-20 所示。

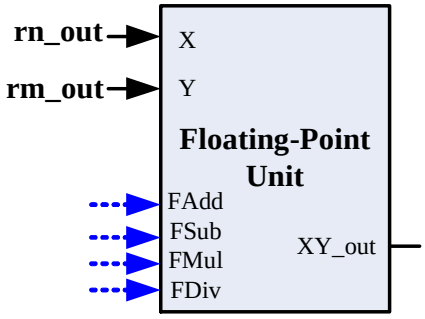

圖 3-20　浮點數運算單元之示意圖

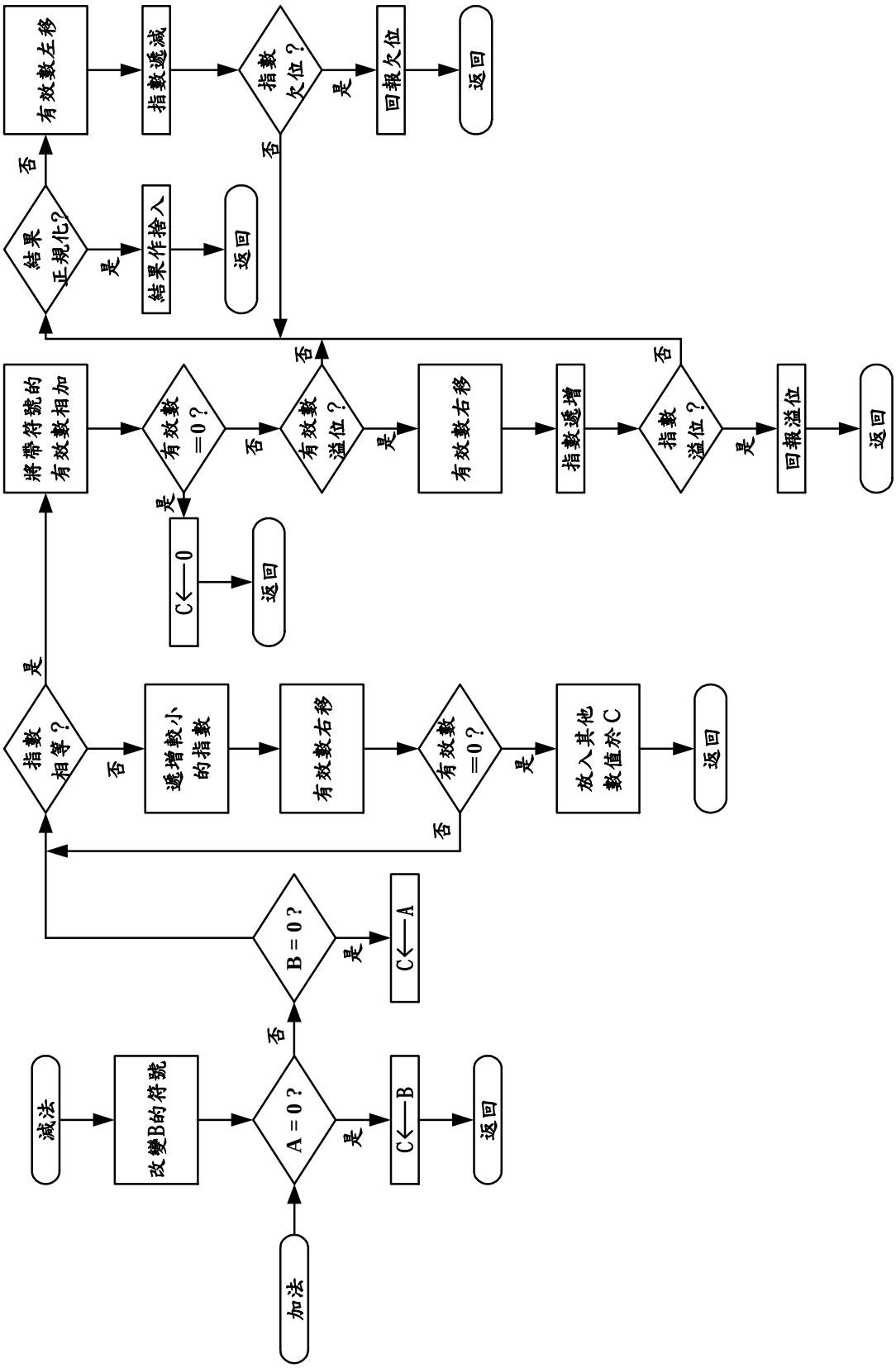

圖 3-21　浮點數加減法演算法流程圖

陸、與 Pipelined MIPS CPU 整合

浮點數運算處理器設計完成後，即加入所設計之 Pipelined MIPS CPU 中，進行整合。圖 3-22 為實作之 Pipelined MIPS CPU 架構圖。

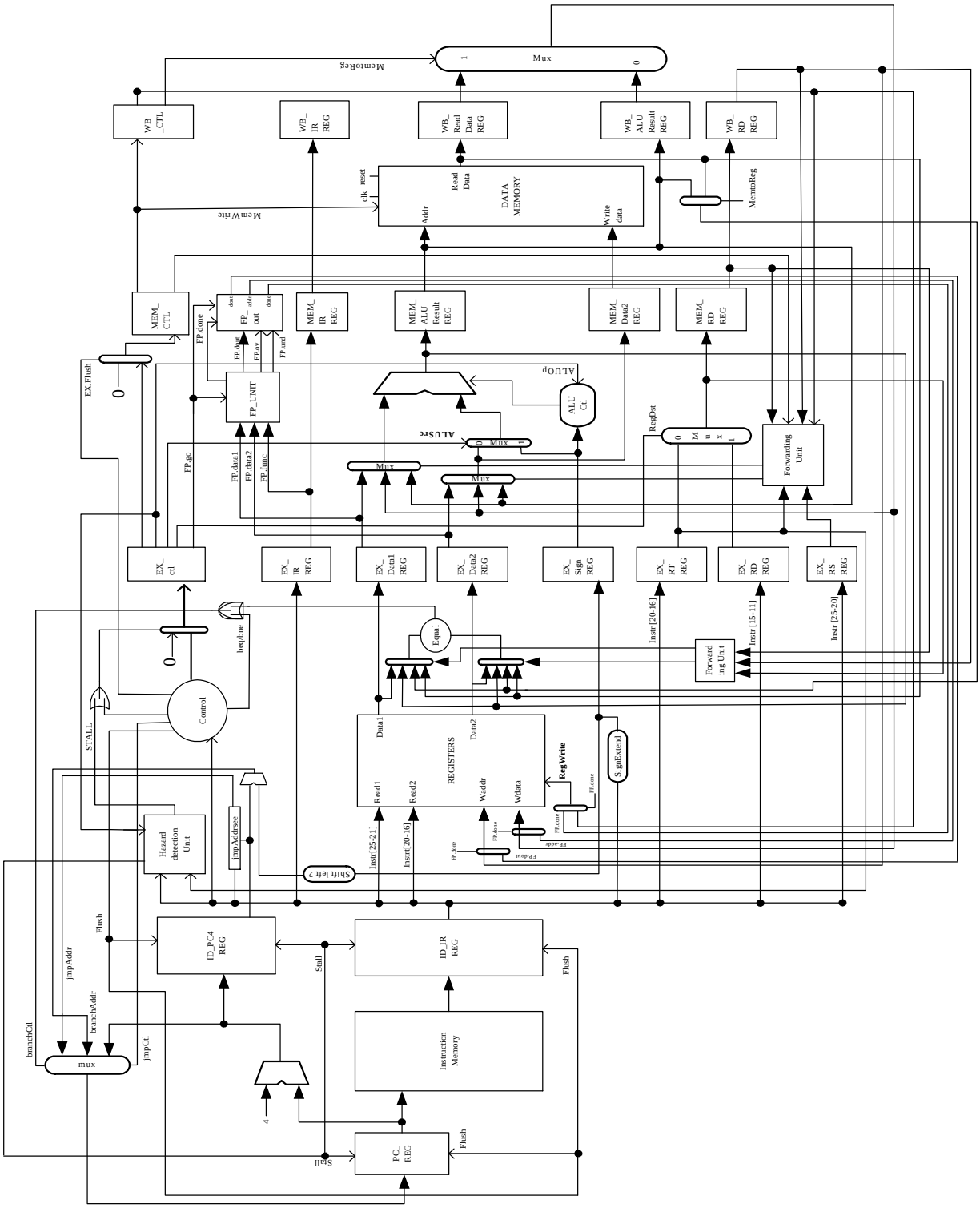

圖 3-22　實作 Pipelined MIPS CPU 架構圖