第2章　電腦的硬體結構

在本章中，我們將學習電腦的硬體架構與指令集，以便理解指令的執行方式。我們將以簡易的 CPU0 處理器作為核心，學習其架構、包含暫存器的功能與用法，以及各個指令的用途與意義。最後，我們會在實務案例當中，以 Intel Pentium 的 IA32 處理器為範例，說明該處理器的架構與指令集，以便進一步理解商用 CPU 的設計方式。

2.1 CPU0 處理器

CPU0 是筆者所設計的一個簡易的 32 位元處理器，主要用來說明系統程式的運作原理。CPU0 的設計主要是為了教學考量，設計重點在於簡單、容易理解。因此 CPU0 幾乎不考慮成本與速度的問題。商業上的處理器通常很複雜，除了考慮成本與速度之外，有時還會考慮相容性的問題，因此並不容易理解。

CPU0 的架構如圖 2.1 所示，包含 R0..R15, IR, MAR, MDR 等 19 個暫存器，其中 IR 是指令暫存器 (Instruction Register)，用來儲存指令的機器碼。MAR 與 MDR 是輸出入暫存器，可用來與匯流排溝通。MAR 稱為記憶體位址暫存器 (Memory Address Register)，可用來暫存匯流排上的位址資訊。MDR 稱為記憶體資料暫存器 (Memory Data Register)，可用來暫存匯流排上的資料訊息。控制單元會自動操控 IR, MAR, MDR 等三個暫存器，我們無法透過組合語言存取這些暫存器。

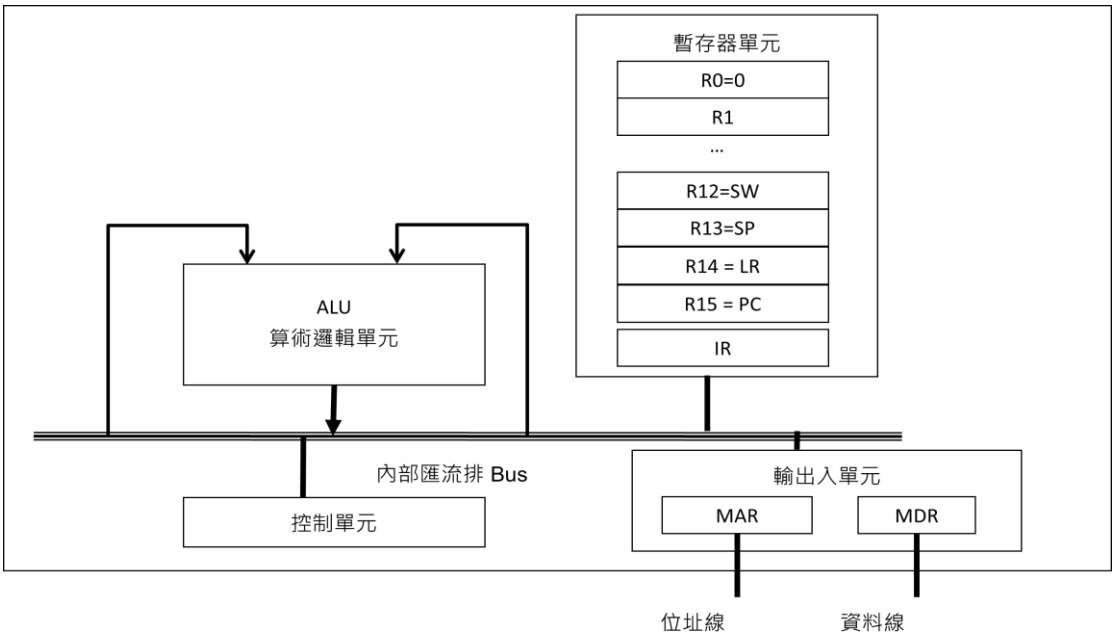

圖 2.1 CPU0 的架構圖

CPU0 包含 15 個可存取暫存器 R1-R15，以及一個唯讀的常數暫存器 R0。R0 的值永遠都是常數零，由於常數零在電腦上被使用的頻率極高，因此 CPU0 特別為常數零分配一個暫存器。

在可存取的暫存器 R1-R15 當中，R12、R13、R14 與 R15 具有特殊用途。R12 是狀態暫存器 (Status Word, SW)，R13 則是堆疊暫存器 (Stack Pointer Register, SP)，R14 代表連結暫存器 (Link Register, LR)，而 R15 則是程式計數器 (Program Counter, PC)。

除了暫存器之外，CPU0 還包含了算術邏輯單元 (Arithmatic Logic Unit, ALU)、控制單元 (Control Unit)、與輸出入單元 (Input Output Unit) 等。暫存器之間可透過內部匯流排傳遞資料，其速度較快。存取記憶體時，必須透過輸出入單元的 MAR 與 MDR 暫存器，以傳送或接收資料，記憶體的存取速度通常比暫存器慢上數倍。

馮紐曼架構

雖然，CPU 是電腦硬體的核心，但是本身無法獨立運作。我們必須為 CPU 加上記憶體 (Memory)、匯流排[1] (Bus)、輸出 (Output) 與輸入 (Input) 裝置，才能形成一台完整的電腦，這樣的電腦結構被稱為馮紐曼架構 (Von Neumann Architecture)。因此，我們可以將目前的電腦架構簡化成 Computer = CPU + BUS +

[1] 在目前的電腦中，匯流排是以主機板的形式存在的，雖然主機板的電路很複雜，但實際上只是由位址匯流排、資料匯流排與控制匯流排所形成的線路而已。

Memory + Input + Output 這個公式，馮紐曼電腦的架構如 **Error! Reference source not found.**圖 2.2 所示。

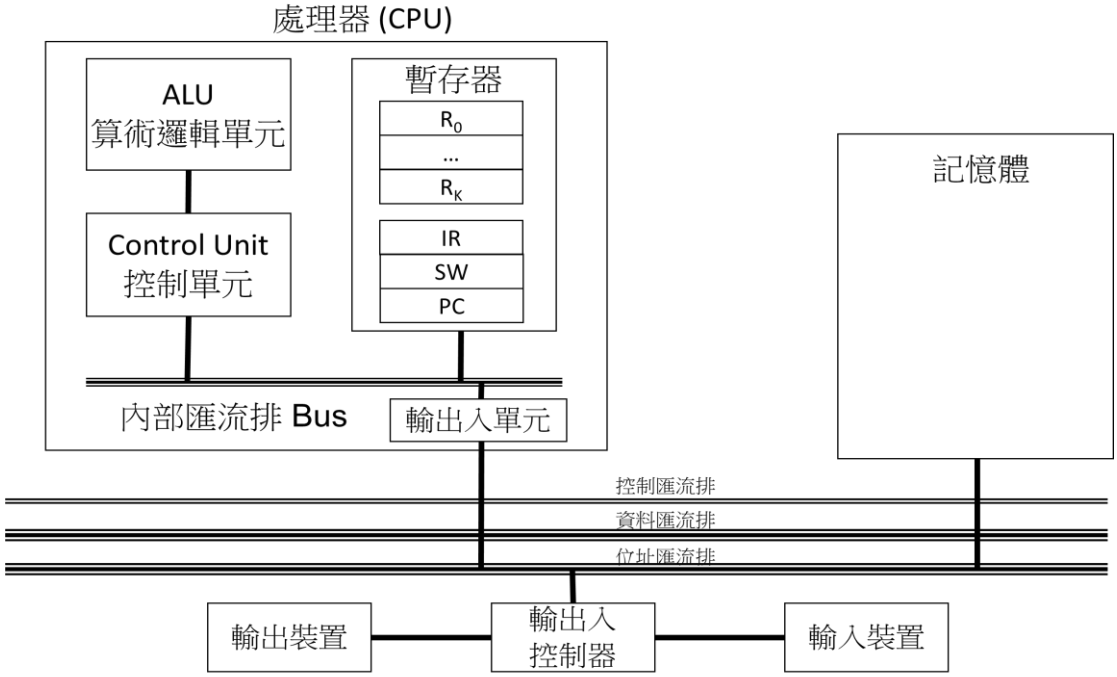

圖 2.2 馮紐曼電腦的結構

對於一個系統程式設計師而言，不需要鉅細靡遺的理解電腦的硬體結構。只要能理解馮紐曼架構，以及指令的運作過程，就能寫出很好的系統程式，設計出好的系統軟體。

Error! Reference source not found.處理器、記憶體與輸出入裝置之間，是透過匯流排進行溝通的。CPU 可以透過匯流排傳送資料給記憶體，也可以接收從記憶體傳來的資料。同樣的，當 CPU 想要進行輸出動作，例如將資料傳送給螢幕或寫到硬碟時，也是透過匯流排將資料傳送到該輸出裝置。在資料進入到輸入裝置之後，CPU 也必須透過匯流排才能取得這些資料。

為了控制輸出入的進行，輸出入裝置上通常有一個控制器存在。這個控制器會負責將匯流排上的資料，適時的傳送給輸出裝置。一旦有資料進入時，該控制器也會負責將輸入資料保存在某個暫存器當中，以方便 CPU 取得該資料。

細心的讀者可能會發現，CPU 內部的結構與外部的結構非常相似，兩者都會透過匯流排進行資料傳遞。讀者可以將 CPU 想像為隱藏在電腦中的微小電腦，只是其記憶體改成了暫存器，但暫存器的存取速度比記憶體更加快速。

在 CPU0 當中，暫存器 R1 到 R15 屬於通用型暫存器，其作用是儲存運算資料。舉例而言，像 ADD R1, R2, R3 這樣一個指令，其用途是將暫存器 R2 與 R3 相加的結果，儲存到暫存器 R1 當中。這個指令的來源參數 (R2, R3) 與目標參數 (R1) 都是暫存器。

算術單元 ALU 是 CPU0 的核心，具有加、減、乘、除、邏輯運算與旋轉移位等功能。資料從暫存器流出到 ALU 之後，運算結果會再度存回到暫存器當中。因此，ALU 可被想像成是 CPU 當中的小型處理器。

控制單元會根據 IR 當中的運算碼決定 ALU 的運算類型，並控制資料的傳遞方向，有時還必須根據狀態暫存器 SW 的內容，決定是否要進行跳躍動作。因此，控制單元可以被想像為 CPU 的指揮者。

CPU0 的暫存器中所儲存的二進位資料，通常只能被當成整數進行運算。CPU0 並沒有浮點數的硬體功能。因此，若需要進行浮點運算，必須使用外加的浮點運算器，或者採用軟體副程式的方式實作。

CPU0 當中的整數，是以二補數的方式表示的，因此 32 位元的暫存器 R1-R15，均可用來表達 $-2^{31} \sim 2^{31}-1$ 之間的整數。

CPU0 的 ALU 採用二補數的電路實作加減乘除運算，這種作法是目前最常被使用的。如果讀者想進一步瞭解二補數運算之電路設計方式，請參考數位邏輯與計算機結構的相關教科書。

雖然我們已經看完了 CPU0 的架構圖，但是這個資訊仍然無法充分揭露 CPU0 的設計理念。我們還需要瞭解指令集、指令格式、暫存器的位元資訊等，才能理解 CPU0 之運作原理。以下，就讓我們從指令表開始說明 CPU0 的指令架構。

2.2 CPU0 的指令集

表格 2.1 是 CPU0 的指令編碼表，其中第一欄是指令類型 (Type)、第二欄是指令格式 (Format)，第三欄的則是指令的名稱 (助憶符號, Mnemonic Operation)，第四欄是指令代碼 (Operation Code, OP)，第五欄則是指令的說明，第六欄是其語法格式 (Syntax)，第七欄則是指令的語義 (Semantics)。

表格 2.1 CPU0 的指令編碼表

類型	格式	指令	OP	說明	語法	語義
載入儲存	L	LD[2]	00	載入 word	LD Ra, [Rb+Cx]	Ra←[Rb+ Cx]
	L	ST	01	儲存 word	ST Ra, [Rb+ Cx]	Ra→[Rb+ Cx]
	L	LDB	02	載入 byte	LDB Ra, [Rb+ Cx]	Ra←(byte)[Rb+ Cx]
	L	STB	03	儲存 byte	STB Ra, [Rb+ Cx]	Ra→(byte)[Rb+ Cx]
	A	LDR	04	LD (暫存器版)	LDR Ra, [Rb+Rc]	Ra→(byte)[Rb+ Rc]
	A	STR	05	ST (暫存器版)	STR Ra, [Rb+Rc]	Ra→[Rb+ Rc]
	A	LBR	06	LDB (暫存器版)	LBR Ra, [Rb+Rc]	Ra←(byte)[Rb+ Rc]
	A	SBR	07	SB (暫存器版)	SBR Ra, [Rb+Rc]	Ra→(byte)[Rb+ Rc]
	L	LDI	08	立即載入	LDI Ra, Rb+Cx	Ra← Rb + Cx
運算指令	A	CMP[3]	10	比較	CMP Ra, Rb	SW ← Ra >=< Rb
	A	MOV	12	移動	MOV Ra, Rb	Ra ← Rb
	A	ADD	13	加法	ADD Ra, Rb, Rc	Ra ← Rb+Rc
	A	SUB	14	減法	SUB Ra, Rb, Rc	Ra ← Rb-Rc
	A	MUL	15	乘法	MUL Ra, Rb, Rc	Ra ← Rb*Rc
	A	DIV	16	除法	DIV Ra, Rb, Rc	Ra ← Rb/Rc
	A	AND	18	邏輯 AND	AND Ra, Rb, Rc	Ra ← Rb and Rc
	A	OR	19	邏輯 OR	OR Ra, Rb, Rc	Ra ← Rb or Rc
	A	XOR	1A	邏輯 XOR	XOR Ra, Rb, Rc	Ra ← Rb xor Rc
	A	ROL[4]	1C	向左旋轉	ROL Ra, Rb, Cx	Ra ← Rb rol Cx
	A	ROR	1D	向右旋轉	ROR Ra, Rb, Cx	Ra ← Rb ror Cx
	A	SHL	1E	向左移位	SHL Ra, Rb, Cx	Ra ← Rb << Cx
	A	SHR	1F	向右移位	SHR Ra, Rb, Cx	Ra ← Rb >> Cx
跳躍指令	J	JEQ[5]	20	跳躍 (相等)	JEQ Cx	if SW(=) PC ← PC+Cx
	J	JNE	21	跳躍 (不相等)	JNE Cx	if SW(!=) PC ← PC+Cx
	J	JLT	22	跳躍 (<)	JLT Cx	if SW(<) PC ← PC+Cx
	J	JGT	23	跳躍 (>)	JGT Cx	If SW(>) PC ← PC+Cx
	J	JLE	24	跳躍 (<=)	JLE Cx	if SW(<=) PC ← PC+Cx
	J	JGE	25	跳躍 (>=)	JGE Cx	If SW(>=) PC ← PC+Cx

[2] LD 為 LOAD 的縮寫，ST 代表 STORE，因此 LDB 就是 Load byte，STB 就是 Store byte，而 LDI 則是 Load immediate value，LDR 則是 Load by Register。

[3] CMP 為 Compare 的縮寫，MOV 代表 Move，SUB 代表 Substract，MUL 代表 Multiply，DIV 代表 Divide。

[4] ROL 為 Rotate Left 的縮寫，ROR 的全名為 Rotate Left，SHL 代表 Shift Left，SHR 則是 Shift Right。

[5] JEQ 為 Jump if Equal 的縮寫，JNE 的全名為 Jump if Not Equal，JLT 則是 Jump if Less Than，JGT 則是 Jump if Greater Than，JLE 則是 Jump if Less or Equal，JGE 代表 Jump if Greater or Equal，JMP 則代表 Jump。

	J	JMP	26	跳躍 (無條件)	JMP Cx	PC ← PC+Cx
	J	SWI[6]	2A	軟體中斷	SWI Cx	LR ← PC; PC ← Cx
	J	CALL	2B	跳到副程式	CALL Cx	LR← PC; PC ← PC+Cx
	J	RET	2C	返回	RET	PC ← LR
堆	A	PUSH	30	推入 word	PUSH Ra	SP-=4; [SP] = Ra;
疊	A	POP	31	彈出 word	POP Ra	Ra = [SP]; SP+=4;
指	A	PUSHB	32	推入 byte	PUSHB Ra	SP--; [SP] = Ra; (byte)
令	A	POPB	33	彈出 byte	POPB Ra	Ra = [SP]; SP++; (byte)

在表格 2.1 中的 CPU0 指令類型中，包含了『載入儲存』、『運算指令』、『跳躍指令』、『堆疊指令』等四大類型的指令。每個指令都包含了指令名稱與代碼 (OP)，語法、語義與說明等欄位。由於這個編碼表在本書中經常會使用到，因此也被收錄在本書的附錄 A 當中。當您需要查詢指令的編碼或意義時，可以翻閱本書最後的附錄，以便查詢這些指令的意義。

CPU0 的指令格式

在表格 2.1 的第二個欄位中，記載了指令的格式。CPU0 的指令可分為三種格式，分別是 A 型、J 型與 L 型。雖然 CPU0 的所有指令都佔用 32 位元，但是每一種格式的指令都有不同的內部結構。

圖 2.3 顯示了 CPU0 的三種指令格式，其中的 A 型通常是運算指令 (Arithmatic)，J 型通常是跳躍指令 (Jumping)，L 型通常是載入儲存 (Load&Store) 指令。

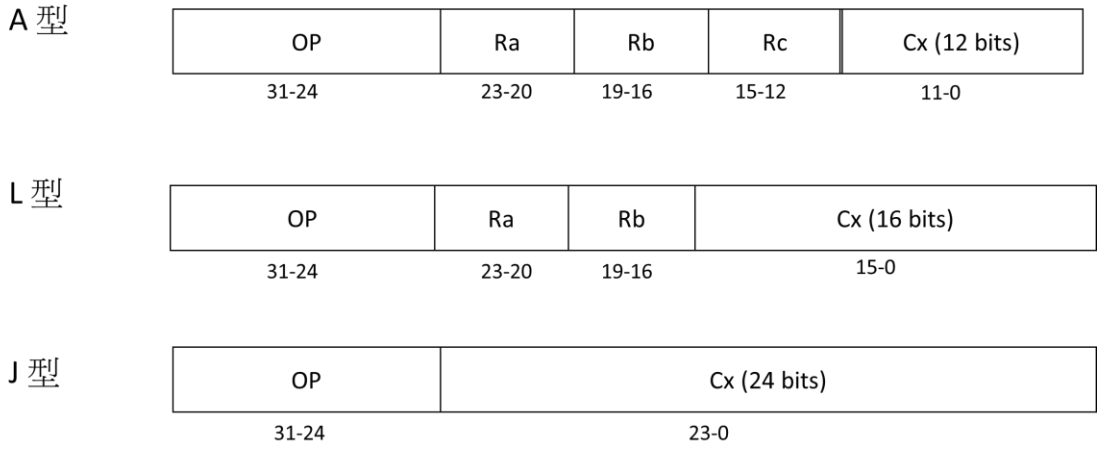

圖 2.3 CPU0 的指令格式

[6] SWI 為 SoftWare Interrupt 的縮寫，CALL 則是 Call Subroutine，RET 則是 return 的縮寫。

雖然 L 型格式代表載入儲存指令 (Load&Store)，但這只是個記憶原則，並非必然如此。舉例而言，像 LDR、STR、LBR、SBR 這樣的指令，雖然是載入儲存指令，但是卻是 A 型格式的指令，因為這種載入儲存指令包含三個暫存器參數，無法編為只有兩個暫存器的 L 型格式。

在圖 2.3 的 CPU0 指令格式中，有三種不同的欄位，其中 OP 是指令代碼，Ra, Rb, Rc 則是暫存器欄位，Cx 則代表常數欄位。

OP 欄是指令代碼，座落在 31-24 位元，例如 0x00[7] 代表 LD、0x01 代表 ST、0x08 代表 LDI、0x13 代表 ADD 等。這些代碼可以用來提供 CPU 判斷指令類型，以便執行對應的動作。

OP 的長度固定為 8 位元，總共可以容納 256 種不同的指令。但是 CPU0 的指令只有 36 種，其實只要用 6 個位元就足以容納。筆者採用 8 個位元的目的，除了考慮擴充性之外，主要還是為了簡化。因為 8 個位元的 OP 代碼可以被寫成兩個 16 進位字元，這會讓機器指令更容易閱讀。

圖 2.3 當中的 Ra, Rb, Rc 是暫存器代碼，大小都是 4 位元，恰好可以寫成一個 16 進位字元。若字元為 0 則代表暫存器 R0，字元為 1 則代表 R1，以此類推。因此，字元為 A 則代表 R10，字元為 F 則代表 R15。

圖 2.3 當中還有三個 Cx 常數區，分別佔據 12、16 與 24 位元。佔據位元數愈多的常數，其常數範圍就愈大。由於採用 2 補數的表達法，所以 12 位元的常數只能表達 $-2^{11} \sim 2^{11}-1$ (也就是 -2048~2047) 之間的數字，但是 24 位元的常數就能表達 $-2^{23} \sim 2^{23}-1$ (也就是 -8388608~8388607) 之間的數字，其表達範圍增加了 4096 倍。

座落在 23-0 位元的部分為參數區，參數區的編碼方式根據指令類型 A、L、J 而有所不同。參數區通常以 4 個位元為一單位，以便對應到 16 進位的字元，這也是為了方便讀者理解而設計的。

A 型指令的格式為 OP Ra, Rb, Rc, Cx，像是加法指令 ADD R1, R2, R3 就是一個 A 型指令。但是並不是每一個指令都會使用全部的參數，像是 ADD 就沒有使用到 Cx 的常數部分。

L 型指令的格式為 OP Ra, Rb, Cx，像是載入指令 LD R1, [R2+100] 就是一個 L 型

[7] 註：在本書中提到 16 進位表示法時，會使用 C 語言的語法，像是 0x0F 代表十進位中的 15。

指令。載入儲存指令通常是 L 型指令，但是有幾個例外，分別是 LDR、STR、LBR、SBR 等四個指令。由於這四個指令都具有三個暫存器參數，因此只能編碼為 A 型格式。

J 型指令的格式為 OP Cx，像是 JMP 100 則代表向前跳躍 100 個記憶體位址，也就是讓程式計數器 PC = PC+100。在 J 型指令當中，由於 Cx 常數佔 24 個位元，因此跳躍範圍限制在 $-2^{23} \sim 2^{23}-1$ 之間，大約是前後 8MB 的距離範圍，如果要跳到更遠的地方，必須採用載入儲存指令，直接將位址載入到程式計數器 PC 當中，以達成遠距跳躍的目的。

利用 CPU0 的指令表與指令格式，我們就可以對每一個指令進行編碼。舉例而言，我們可以從指令表中，看出 XOR 指令的代碼為 1A，而且具有 A 型格式。於是我們就可以根據 A 型格式，可以得知三個暫存器參數 Ra, Rb, Rc 的編碼位置。然後根據這些資訊將 XOR R1, R2, R3 指令編為 1A 12 30 00。

2.3 CPU0 的運作原理

雖然我們已經看過了 CPU0 的架構與指令集，但是，要能理解這些指令的運作方式，還需要一些更動態的概念。在本節中，我們將以圖解的方式，利用資料流向圖，說明各類指令的資料流向與運算方式，讓讀者得以理解這些指令的運作原理。

移動指令

移動指令的目的是將某暫存器的內容移動到另一個暫存器當中。例如，MOV R1, R2 可以將 R2 的內容傳給 R1。請注意，這個指令只能將暫存器的內容移動到另一個暫存器當中，但是不能將資料從暫存器移動到記憶體當中[8]，也不能從記憶體移到暫存器當中。

移動指令的執行過程很簡單，在圖 2.4 中，正在被執行的指令是 MOV R1, R2。假如該指令執行前 R2=2，則執行完之後，R1 的值將變成 2，請讀者對照圖 2.4 以便理解其執行過程。

[8] 在 CPU0 當中，記憶體的存取必須透過載入儲存指令，而非 MOV 指令。

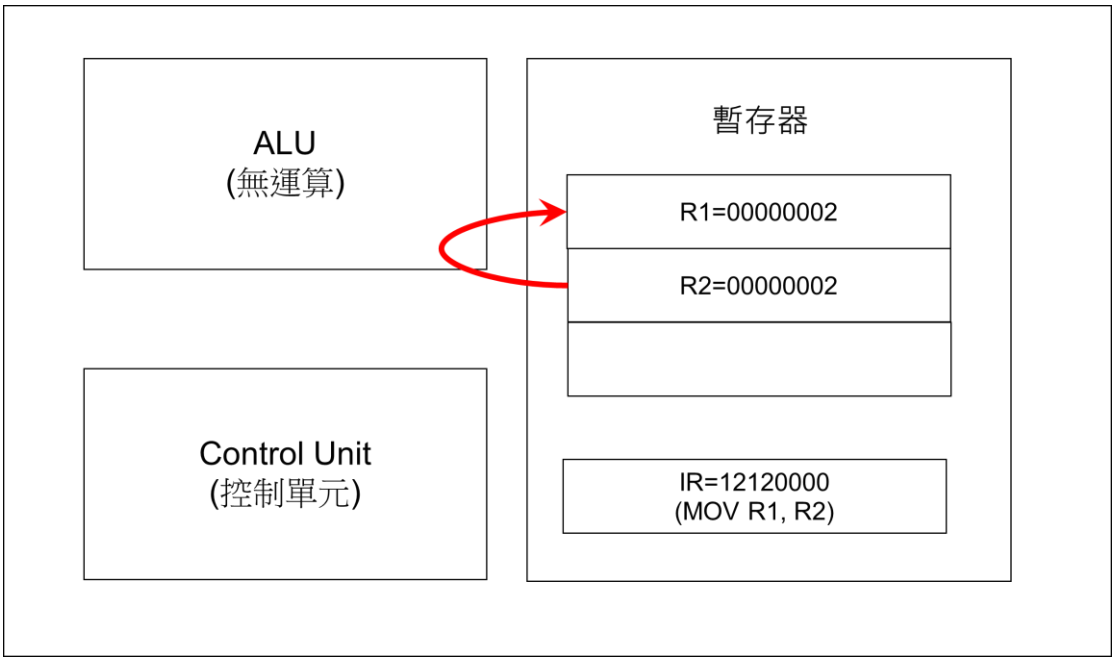

圖 2.4 移動指令 MOV R1, R2 的執行過程

算術指令

算術指令基本上可再細分為數學運算與邏輯運算兩類。但是，由於這兩類指令的格式都相同。因此，在 CPU0 當中被合併為同一類指令，統稱為算術指令。在 CPU0 當中，數學運算類的指令包含了加法 (ADD) 、減法 (SUB)、乘法(MUL) 與除法 (DIV)。讓我們以加法為範例，看看算術指令的執行過程。

加法指令的執行過程

假如指令暫存器 IR 當中的值是 13 12 30 00，由於該指令是 ADD R1, R2, R3 的指令碼，所以控制單元會決定讓暫存器 R2 與 R3 的資料傳到 ALU 作為輸入，並且設定 ALU 執行加法運算，然後再將 ALU 的輸出送回目標暫存器 R1 當中，其資料的流向關係如圖 2.5 所示。因此，假如該指令執行前 R2=2 , R3 = 3，則執行完之後，R1 的值將變成 5。

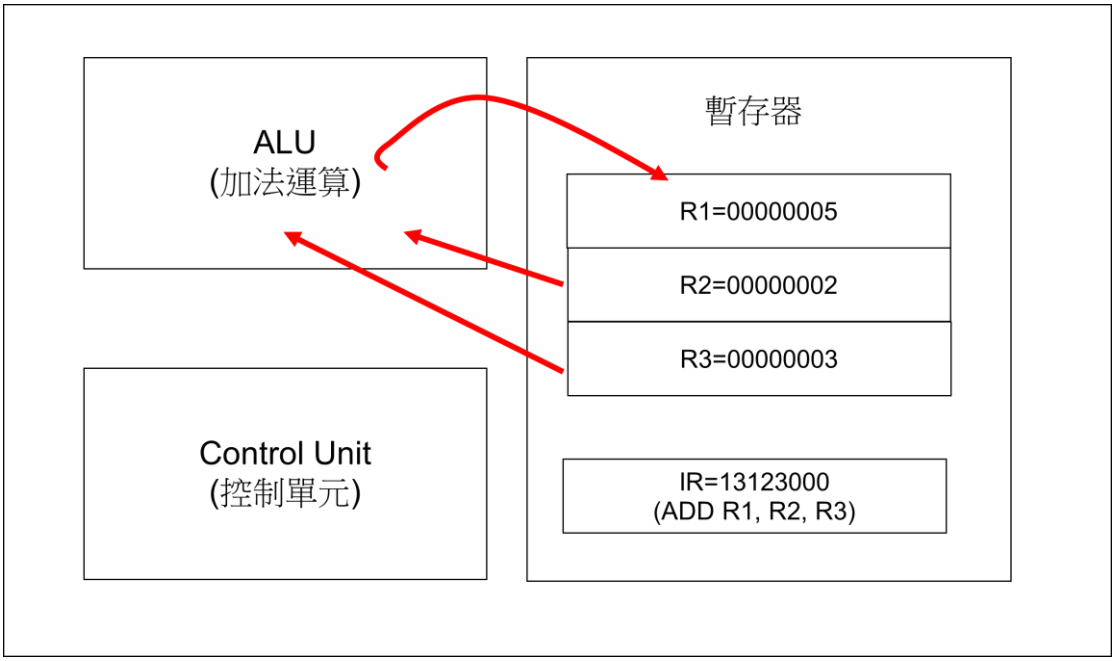

圖 2.5 加法指令 ADD R1, R2, R3 的執行過程

減法指令的執行過程

同樣的，假如執行的指令是減法，則指令 SUB R1, R2, R3 會被編碼為 14 12 30 00。因此，假如該指令執行前 R2=3 , R3 = 2，則執行完之後，R1 的值將變成 1，其執行過程如圖 2.6 所示。

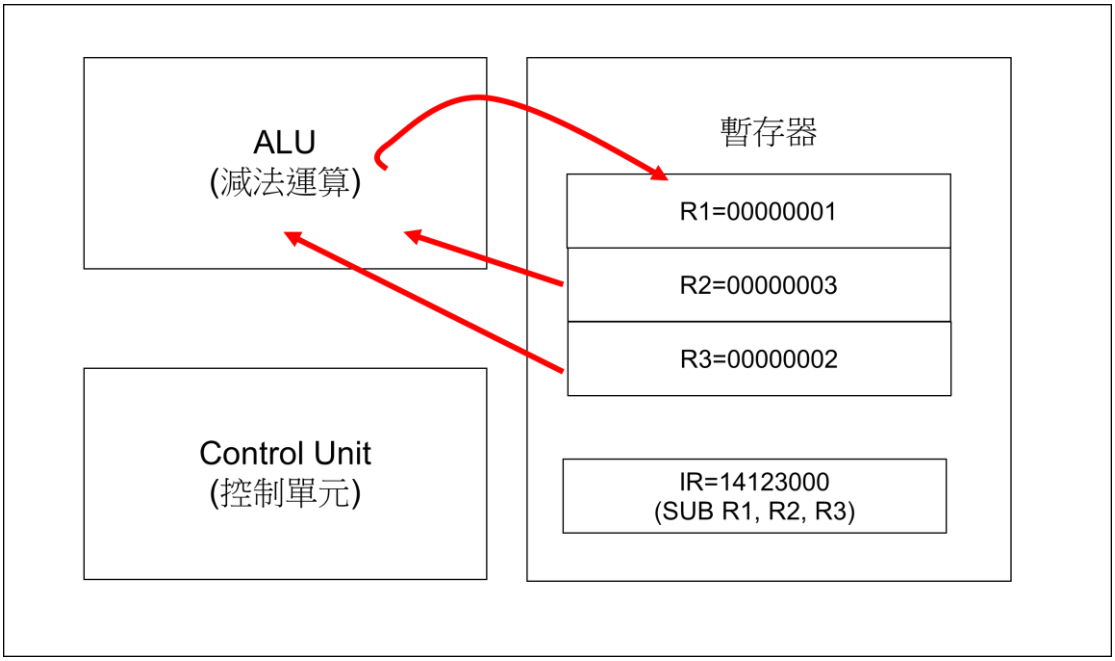

圖 2.6 減法指令 SUB R1, R2, R3 的執行過程

邏輯運算指令

CPU0 的邏輯運算指令包含 AND、OR 與 XOR。這些指令都是以位元對位元 (bit to bit) 的方式進行的。對於邏輯運算指令而言，其執行過程與算術指令雷同，只是 ALU 改以邏輯運算的方式執行。

舉例而言，我們可以根據 CPU0 的指令表，找到 XOR 指令的代碼為 1A，然後根據其指令格式，將 XOR R1, R2, R3 指令編碼為 1A 12 30 00。

如圖 2.7 中的 IR 暫存器所示。假如該指令執行前 R2=0xF4 且 R3=0xB5，則執行完之後，R1 的值將變成 0x41。一般人通常很難直接看出十六進位值的邏輯運算結果，讀者可以先將 0xB5 與 0xF4 還原成二進位的 10110101 與 11110100 之後，再對每個位元進行 XOR 運算，其結果為 01000001，也就是十六進位的 0x41，其過程如圖 2.7 所示 (請注意，圖 2.7 當中的 * 符號，代表暫存器前面的 24 個位元被省略了)。

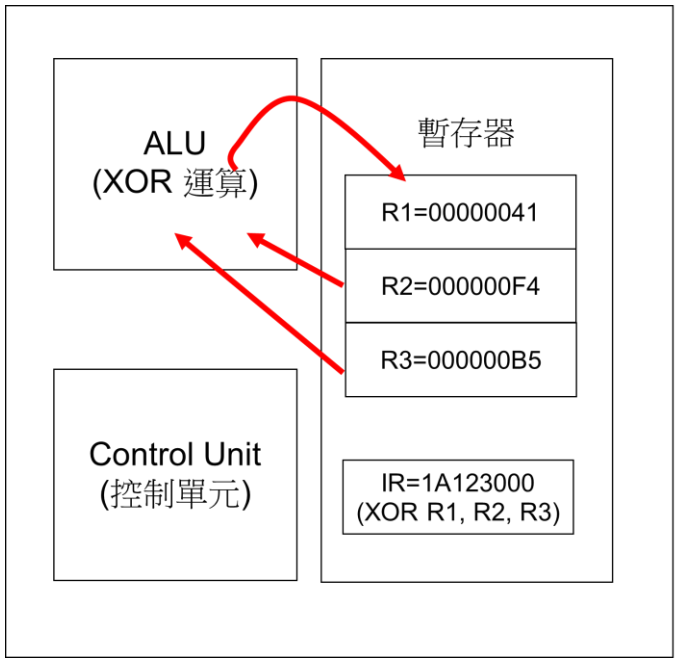

圖 2.7 邏輯運算 XOR R1, R2, R3 的執行過程

移位指令

ROL (左旋), ROR (右旋) 屬於旋轉指令，而 SHL (左移),SHR (右移) 則屬於移位指令，旋轉與移位指令可用來對於製作軟體的乘法模擬功能很有用，而在模擬浮點運算

時，也常被用來作為小數點對齊之用。

假如指令暫存器 IR 當中的值是 1E120004 ，由於該指令是 SHL R1, R2, 4 的指令碼，因此，控制單元會決定讓暫存器 R2 中的資料傳入 ALU 後，向左移 4 位後，再傳入目標暫存器 R1 當中，其資料流向圖 2.8 如所示。因此，假如該指令執行前 R2=0x41，則執行完之後，R1 的值將變成 0x410。

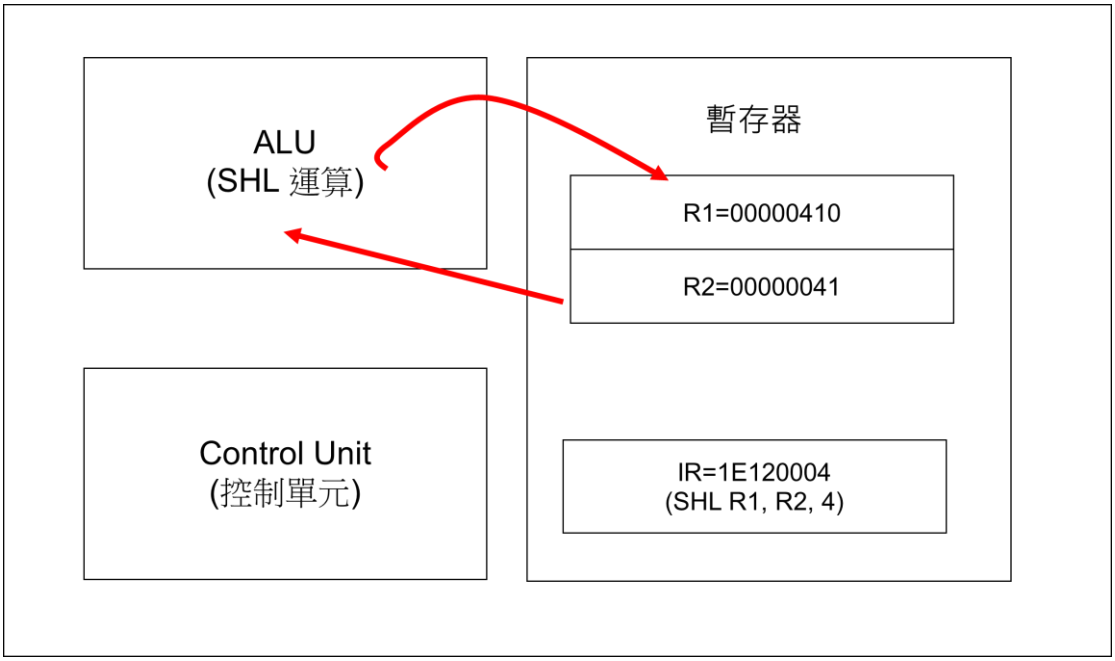

圖 2.8 移位運算 SHL R1, R2, 4 的執行過程

比較指令

比較指令 CMP 是相當特殊的一個指令，這個指令雖然不像算術指令一樣，會算出結果。但是，其設計原理與減法指令 SUB 相當類似。例如， CMP R1, R2 會對 R1 與 R2 進行比較的動作，其方法同樣是將 ALU 設定為減法模式，讓 R1 減去 R2。但不同的是，相減完之後的結果並不會被存入任何暫存器。當 ALU 中的減法執行完畢後，控制單元會根據減法的結果，設定狀態暫存器當中的條件旗標 (Condition Code)。

狀態暫存器 (Status Word : SW) 乃是 CPU0 用來儲存狀態的場所，該暫存器的位元配置如圖 2.9 所示。SW 中包含負號 (N)、零 (Z)、進位 (C) 與 溢位 (V) 等旗標，這四個旗標合稱『條件旗標』 (Condition Code : CC)。

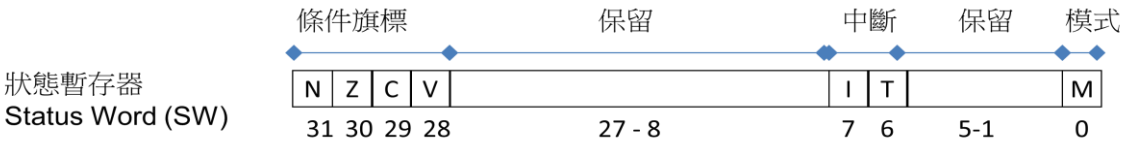

圖 2.9 CPU0 當中的狀態暫存器 SW (R12)

比較指令的結果通常會造成 N、Z 兩個旗標的改變，其中的 N 代表減法的結果為負數 (Negative)，Z 代表減法的結果為 0 (Zero)。

條件旗標的 N、Z 旗標值，可以用來代表比較結果，當執行 CMP Ra, Rb 動作後，可能會有下列三種情形。

1. 若 Ra > Rb，則 N=0, Z=0。
2. 若 Ra < Rb，則 N=1, Z=0。
3. 若 Ra = Rb，則 N=0, Z=1。

如此，用來進行條件跳躍的 JGT、JGE、JLT、JLE、JEQ、JNE 等指令，就可以根據 N、Z 旗標決定是否進行跳躍。

在圖 2.9 的狀態暫存器 SW 中，還包含了中斷控制旗標 I (Interrupt) 與 T (Trap)。這兩個旗標可以控制中斷行為，假如將 I 旗標設定為 0，則 CPU0 將禁止所有種類的中斷，也就是對任何中斷都不會起反應。如果只是將 T 旗標設定為 0，則只會禁止軟體中斷 SWI (Software Interrupt)，但是不會禁止由硬體觸發的中斷。

SW 中還儲存有『處理器模式』的欄位，M=0 時為『使用者模式』 (user mode)，M=1 時為『特權模式』(super mode)等。在作業系統的設計上，處理器模式經常被用來製作安全保護功能。在使用者模式當中，修改狀態暫存器 R12 的動作會被視為是非法的，這是為了進行保護功能的緣故。但是在特權模式中，有權進行任何動作，包含設定中斷旗標與處理器模式等位元。通常作業系統會使用特權模式，而一般程式只能處於使用者模式。

現在，讓我們以範例說明比較指令的運作方式，如圖 2.10 所示。假如在 CMP R1, R2 指令執行前， R1 = 1、R2 = 2、而且 SW 的值是 00000000，則在執行完之後，狀態暫存器將會變成 80000000。這是因為 R1 < R2，所以相減的結果為負數，因此 N 會被設定為 1，而 Z、C、V 旗標都會變成 0。

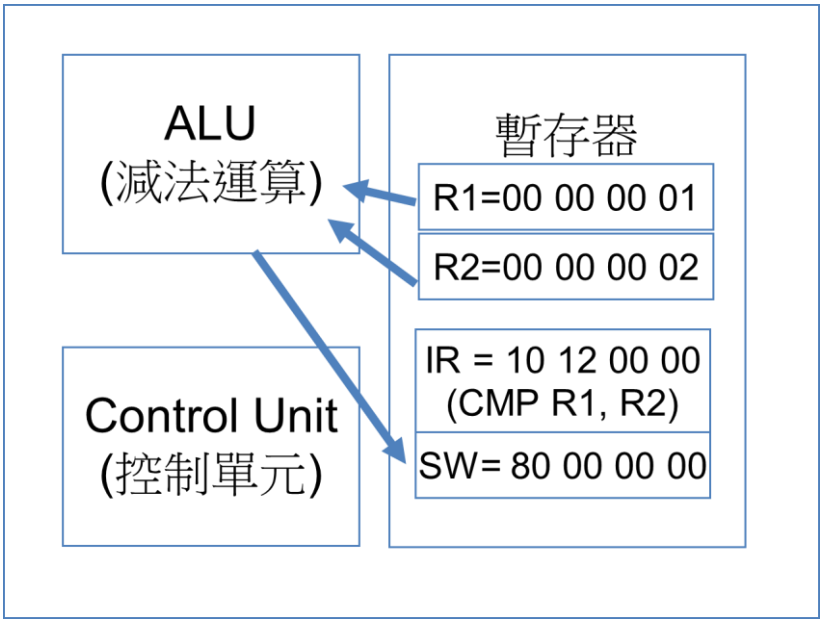

圖 2.10 比較指令 CMP R1, R2 的執行過程

跳躍指令

在 CPU0 中，跳躍指令包含 JMP、JEQ、JNE、JLT、JGT、JLE、JGE 等 7 個，其中的 JMP 是無條件跳躍指令，其餘的則是條件跳躍指令。條件跳躍指令中的 E 代表相等 (Equal)，G 代表大於 (Greater than)，L 代表 (Less than)。因此，JGE 代表的是 (Jump if Greater or Equal) 的縮寫，JLT 則是 (Jump if Less Than) 的縮寫，JEQ 則是 (Jump if EQual) 的簡稱。

無條件跳躍指令

跳躍指令主要作用在程式計數器 PC 上，利用改變 PC 當中的位址，以達成跳躍的動作。其設計方式可分為絕對跳躍與相對跳躍兩類，目前以相對跳躍的方式較為常見。

絕對跳躍的方式，程式會直接跳到參數指定的位址中，在此種方式下，JMP [0x30] 會跳到記憶體位址 0x30 的地方，於是下一個被執行的指令將是位於 0x30 位址當中的指令。

相對跳躍的方式，通常是指相對於 PC 而言。因此，假如指令 JMP [0x30] 的記憶體位址為 0x28，則在指令提取階段完成之後，程式計數器 PC 的值將會是 0x28+4 = 0x2C，在指令 JMP [0x30] 的執行階段完成之後，將會使 PC 值更新為 0x5C，於是下一個被執行的指令將是位於 0x5C 位址當中的指令。

圖 2.11 顯示了相對跳躍指令的原理。跳躍指令 JMP [0x30] 的功能，乃是將 PC 加上十六進位的 0x30。但是，在指令提取階段完成後，PC 已經從 0x28 變為 0x2C。因此，圖中 PC 值為 00 00 00 2C，在跳躍指令執行完畢後，其 PC 值將變為 00 00 00 5C。於是在下一次指令提取時，CPU 會對位址匯流排送出 00 00 00 5C 的位址，因而取得位於該處的指令（若未執行跳躍指令，則下一次應該會提取位址 00 2C 而非 00 5C 的指令）。換句話說，跳躍指令藉由改變 PC 的內容，達成跳躍的目的。

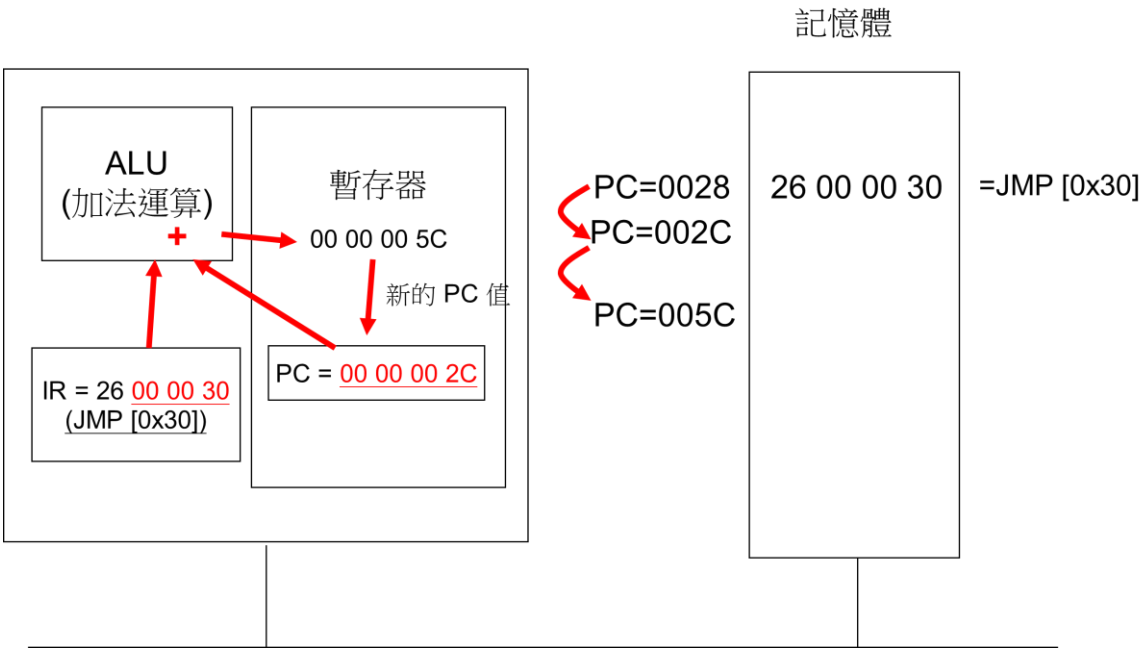

圖 2.11 跳躍指令 JMP [0x30] 的執行過程

由於 JMP 指令的位址常數大小為 24 位元，因此定址範圍只有 2^{24} = 16MB。在 CPU0 當中，由於我們使用了有正負號的二補數表示法，因此合法的跳躍範圍被限制在 PC - 2^{23} 到 PC + 2^{23}-1 之間。

如果我們想要跳躍的位址不在這個範圍之內，就必須使用像 LD 或 LDR 這樣載入指令，將目標位址直接先載入到 PC 暫存器 (R15) 當中。這種方法就能充分存取整台電腦的記憶體，讓程式跳到任何的位址上。

舉例而言，假如我們想要跳到記憶體位址 0xFF000000 的地方，可以用下列方法。首先將常數 0xFF000000 存放到記憶體當中，然後再載入到 PC (R15) 中，就可以完成此種跳躍動作。範例 2.1 顯示了我們利用 LD 指令進行遠距離跳躍的組合語言程式。

範例 2.1 使用 LD 指令進行跳躍的動作

組合語言程式	說明
ADDR: WORD 0xFF000000 ... LD R15, ADDR ...	宣告一個整數變數以存放 0xFF000000 將 0xFF000000 放入到 PC，也就是跳躍到該處

條件跳躍指令

同樣的，CPU0 的條件式跳躍指令也是採用相對跳躍的方式，條件跳躍指令會根據狀態暫存器 SW 當中的值，決定是否要進行跳躍的動作。在一般的組合語言程式當中，通常在條件跳躍指令之前，會先以比較指令 CMP 進行比較，然後再根據比較的結果，利用條件跳躍指令 (例如 JLE) ，決定是否要進行跳躍動作。

圖 2.12 顯示了條件跳躍指令 JLE [0x30] 的執行過程。在該圖中，位於 0x28 中的條件跳躍指令 JLE [0x30] 被執行時，狀態旗標 SW 為 80 00 00 00，這個狀態代表 N 旗標為 1，其餘皆為 0 的小於 (<) 狀態。此時，控制單元會判斷該狀態符合 JLE 的條件，於是進行跳躍動作。除了必須先根據狀態進行判斷之外，條件跳躍指令的運作原理與 JMP 完全相同。

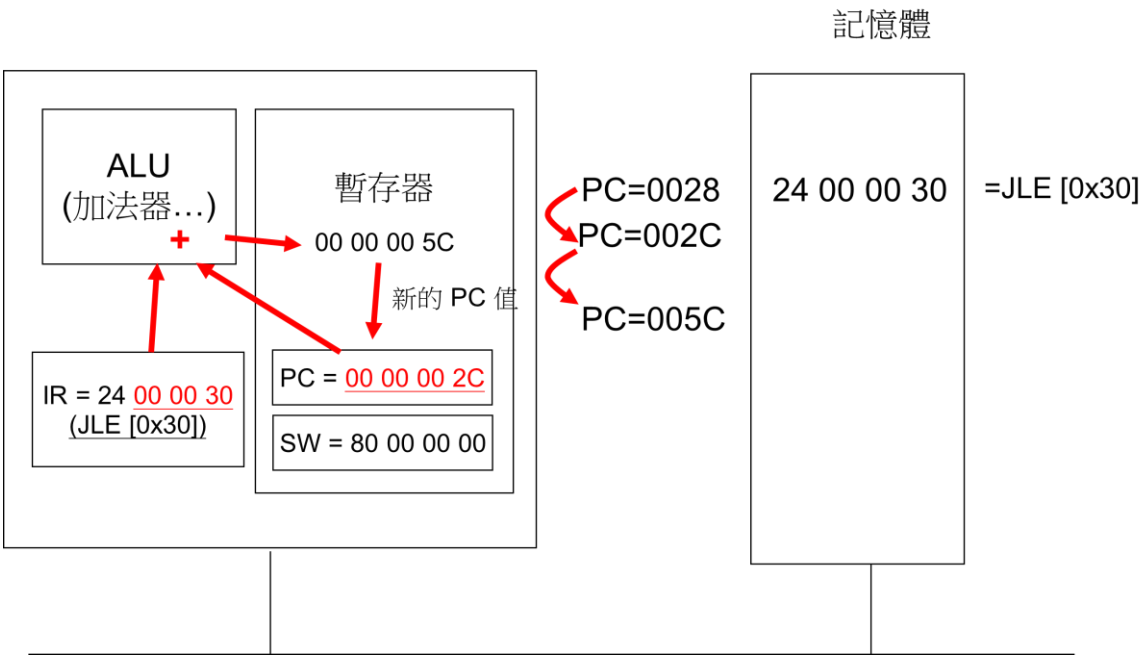

圖 2.12 條件式跳躍指令 JLE [0x30] 的執行過程

我們可以利用比較指令與條件跳躍指令，模擬高階語言當中的迴圈 (例如 for、while 等) 與條件判斷 (例如 if、switch 等) 指令。這些迴圈與條件指令是電腦可以具有如此強大能力的重要原因。換句話說，電腦之所以變化多端卻又循環運行不已，就是利用比較指令與條件跳躍指令的搭配所造成的。

載入儲存指令

傳統上，電腦的位元以 8 個為一組，稱為位元組 (Byte)。在 32 位元的電腦當中，一個字組 (Word) 即為 32 個位元，也就是 4 byte。通常，32 位元 CPU 的內部匯流排也是 32 位元的，也就是有 32 條內部匯流排線，資料的傳遞也是以 32 位元為一單位。

由於 CPU0 是精簡指令集電腦，因此，只有少數指令可以存取記憶體，這些指令稱為『記憶體存取指令』。CPU0 的記憶體存取指令包含了 LD, ST, LDB, STB, LDR, STR, LBR, SBR 等 8 個指令，其中的 LD 是載入一個字組 (Word) 的指令，而 ST 則是儲存一個字組的指令，LDB 則與 LD 類似，但是存取的大小則只有一個位元組 (Byte)。

以 R 字母結尾的四個載入儲存指令，LDR, STR, LBR, SBR 與 LD, ST, LDB, STB 功能相似，只是最後一個參數由常數改為暫存器而已。舉例而言，LD R1, [R2+300]與 LDR R1, [R2+R3] 兩者之間，只是將最後一個參數中的 300 改為暫存器 R3 而已，

這種作法通常被稱為索引定址法。

載入指令的執行過程

載入指令的功能，是將記憶體的內容載入到暫存器的指令。像是 LD R1, [0x28] 就可以將記憶體位址 0x28 當中的內容載入到暫存器 R1 當中，這個載入過程是由 CPU 啟動的，當 CPU 執行載入指令時，會將記憶體位址傳送到位址匯流排當中，然後將控制匯流排當中的記憶體訊號設定為讀取 (Read)。接著當記憶體『看到』該讀取訊號時，就會根據位址匯流排上的位址，提取出對應的資料，放到資料匯流排上。最後，CPU 再從匯流排當中取回此一資料，放入目標暫存器當中。其執行過程如圖 2.13 所示。

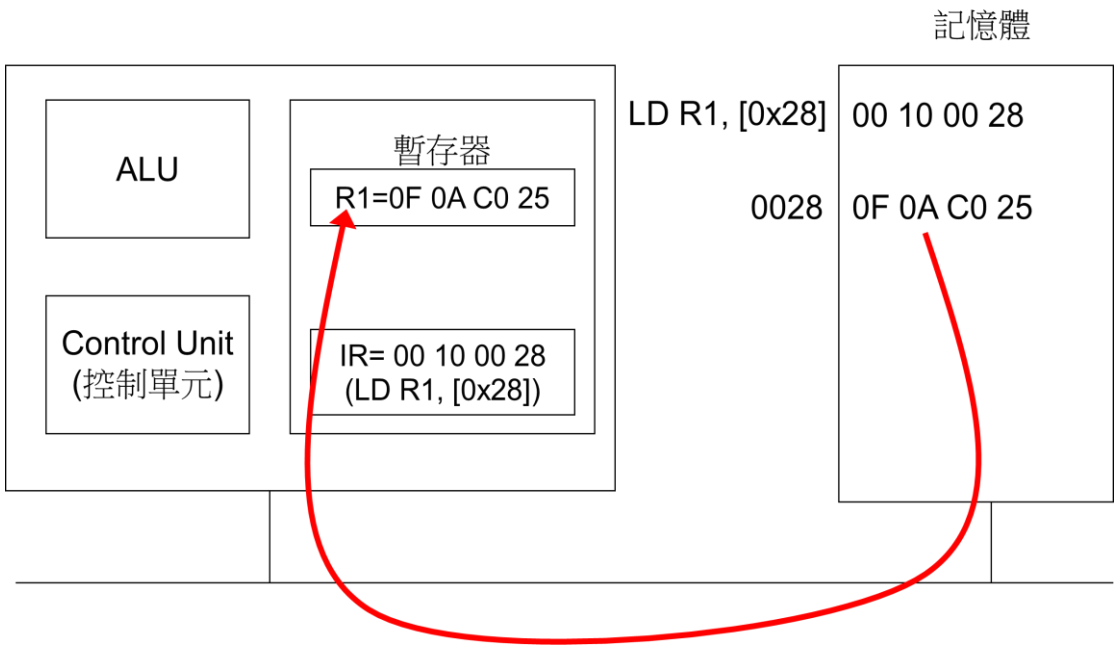

圖 2.13 載入指令 LD R1, [0x28] 的執行過程

儲存指令的執行過程

儲存指令與載入指令的動作恰好相反，是將 CPU 當中的暫存器寫入到記憶體的指令。像是 ST R1, [0x32] 就可以將暫存器 R1 當中的內容寫入到記憶體位址 0x32 當中。這個寫回過程同樣是由 CPU 啟動的，當 CPU 執行寫回指令時，同樣會將記憶體位址傳送到位址匯流排當中，並將欲寫入的資料 (例如暫存器 R1)，傳送到資料匯流排當中。然後設定控制匯流排當中的記憶體訊號為寫入 (Write)。接著，當記憶體『看到』該寫入訊號時，就會根據位址匯流排上的位址，將資料匯流排上的資料寫入到記憶體當中。圖 2.14 顯示了載入指令 ST R1, [0x32] 的執行過程。

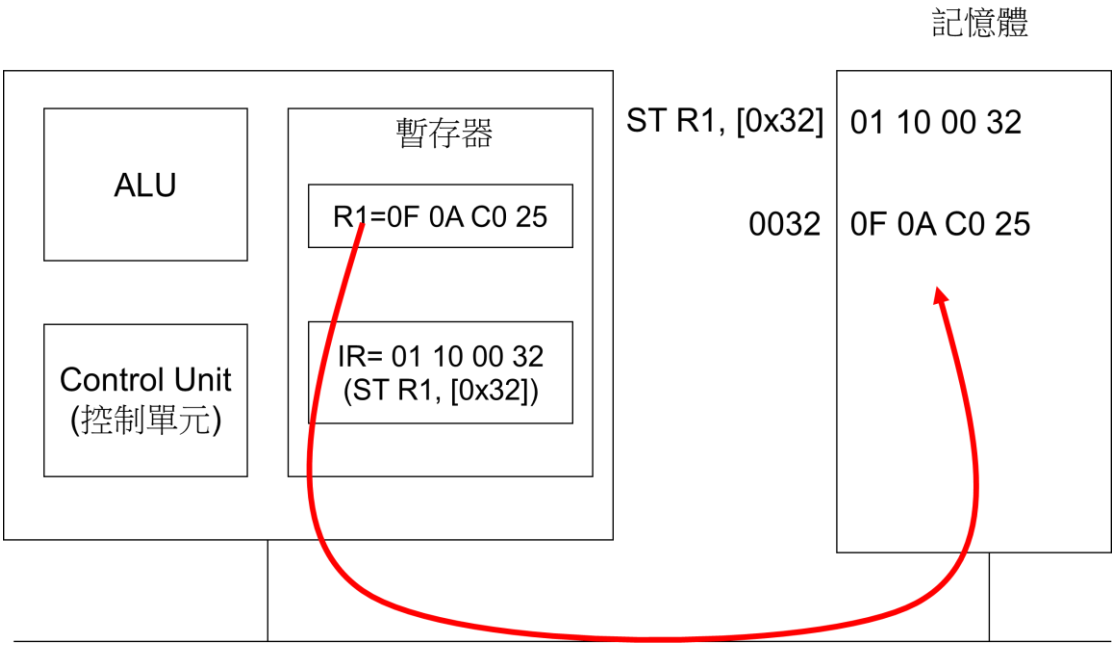

圖 2.14 儲存指令 ST R1, [0x32] 的執行過程

定址模式

在載入儲存指令當中，可以採用不同的定址模式，像是立即載入法、相對定址法、索引定址法與絕對定址法等。以下讓我們介紹載入儲存指令所採用的定址方法。

CPU0 的定址模式可分為三種，包含『立即載入』、『相對定址』與『索引定址』等。在 CPU0 當中，採用立即載入的指令只有 LDI 一個，相對定址的指令有 LD、ST、LDB、STB 等，索引定址的指令有 LDR、STR、LBR、SBR 等。

立即載入

所謂的立即載入，是直接將指令中的常數值載入到暫存器中。例如， LDI R1, 100 就會將 100 載入到 R1 當中。但是，CPU0 的 LDI 指令，功能要比單純的立即載入要更強大一些，因為像 LDI R1, R2+100 這樣的指令，在載入時順便做了加法，會將 R2+100 的值存入到 R1 當中。這樣，就能讓 CPU0 處理器變得更快，而且不需要為 LDI 設計特別的指令格式，直接延用 L 型格式即可。

LDI 指令的寫法，可寫為 LDI Ra, Rb+Cx，例如在 LDI R1, R2+100 這個指令中，參數的指定方式為 Ra=R1, Rb=R2, Cx=100。但是，如果將 Rb 設定為 R0，例如 LDI R1, R0+100，此時，由於 R0 代表常數零，因此，可以被省略，於是，該指令可以簡化為 LDI R1, 100。這只是一種簡寫方式，可以讓組合語言的語法更為簡潔。

利用 LDI 指令，我們可以直接將常數放在 Cx 欄位當中，如此，就不需要額外分配一個記憶體給該常數，因此可以節省記憶體。

相對定址

除了 LDI 之外的載入儲存指令，包含 LD、ST、LDB、STB 等，採用的是相對定址的方式。例如，LD R1, [R2+100] 這樣的指令，就會把 R2+100 所指向的記憶體的字組 (Word) 載入。而 ST R1, [R2+100] 則會將 R1 的內容存入 R2+100 的記憶體位址中。

在上述兩個指令中，參數 [R2+100] 被加上了括號 []，這代表指令是存取 R2+100 的記憶體的內容，而非僅僅將 R2+100 算出而已。由於參數 [R2+100] 是將暫存器 R2 加上一個位移 100 所形成的記憶體位址，因此，此種定址法被稱為相對定址，也就是相對於基準位址 R2 的定址法，此時我們稱呼這個 R2 為基底暫存器。

LD 指令的寫法，可寫為 LD Ra, [Rb+Cx]，例如在 LD R1, [R2+100] 這個指令中，參數的指定方式為 Ra=R1, Rb=R2, Cx=100。但是，如果將 Rb 設定為 R0，由於 R0 代表常數零，因此可以被省略。於是，我們可以將 LD R1, [R0+100] 這個指令簡寫為 LD R1, [100]，這種簡寫方式可以簡化組合語言的語法。

索引定址

LDR, STR, LBR, SBR 等指令，所使用的定址方式被稱為索引定址法，其中的最後一個暫存器 Rc 被稱為索引值暫存器。

舉例而言，像是 LDR R1, [R2+R3] 這樣一個指令，其中的 R2 通常被稱為基底暫存器，而 R3 則被稱為索引暫存器。於是，我們可以利用這樣的語法進行陣列的存取。

為了說明索引定址的用法，我們假設有一個陣列 int a[100]; 那麼，我們要存取 a 陣列的第 i 個元素 a[i] 時應該如何處理呢？此時採用索引定址就是個好辦法。我們可以將 a 存入 R2、i*4 存入 R3 中 (因為一個整數佔 4 bytes)，然後利用 LD R1, [R2+R3] 這樣的指令，將 a[i] 存入 R1 中。

索引定址對於陣列的存取非常方便，而陣列的存取在程式中又經常被使用到，因此，支援索引定址，可以增加 CPU 的執行效率。

絕對定址

所謂的絕對定址，就是直接存取記憶體絕對位址內容的方法。舉例而言，像 LD R1, [100] 這樣的指令就採用了絕對定址法。該指令可以將記憶體位址 100 的內容搬到 R1 當中。

CPU0 不需要絕對定址的指令，而是改用 R0 暫存器進行絕對定址。舉例而言，LD R1, [R0+100] 就相當於 LD R1, [100]，因為 R0 暫存器永遠都是常數 0。這使得 CPU0 可以省略掉絕對定址指令，也不會造成困擾。

抽象來說，載入指令 LD Ra, [R0+Cx] 的意義就是 LD Ra, [Cx]，這就相當於一般 CPU 當中的絕對定址。但必須注意的是，使用此種絕對定址的方式，只能將 Cx 設定為 -2^{15}~2^{15}-1 的範圍，因為 Cx 採用的是 2 補數的方式。但是，負數的絕對定址是沒有用的，因為記憶體位址通常沒有負值。所以只有 0-32767 的位址可以被絕對定址，其餘的記憶體空間，必須採用相對定址的方式。

在設計 CPU 與指令集時，必然會碰到定址範圍的問題，由於指令的長度是有限的，當 CPU 的設計者想盡辦法縮短指令時，往往只能在定址範圍上進行妥協。在 CPU0 的設計上，就因為捨棄絕對定址專用指令，而使定址範圍縮小了一些，但卻讓處理器的架構變得更為精簡。這符合 CPU0 的設計原則 – Keep It as Simple as Possible。

2.4 CPU0 的程式執行

雖然我們在前兩節中已經看過了 CPU0 的指令集與運作原理，但是仍然不足以理解程式是如何被執行的。在本節中，我們將說明 CPU 如何執行整個程式 (也就是一連串的指令)，以便讓讀者建立一個整體的流程概念。

在整個程式的執行過程中，指令會一個接著一個被取出後執行，直到出現跳躍指令為止。舉例而言，在圖 2.15 當中，程式首先會從第一個指令 LD R1, SUM 開始執行，其指令的機器碼為 001F0028。

接著程式會執行位址 0x0004 的第二個指令，然後是位址 0x0008 的第三個指令，如此循序的執行下去。直到位址 0x0020 的 JMP FOR 指令時，程式會跳回位址 0x0010 的 FOR: 標記，再次執行 CMP R2, R3 這個指令，這就是跳躍指令的功能。

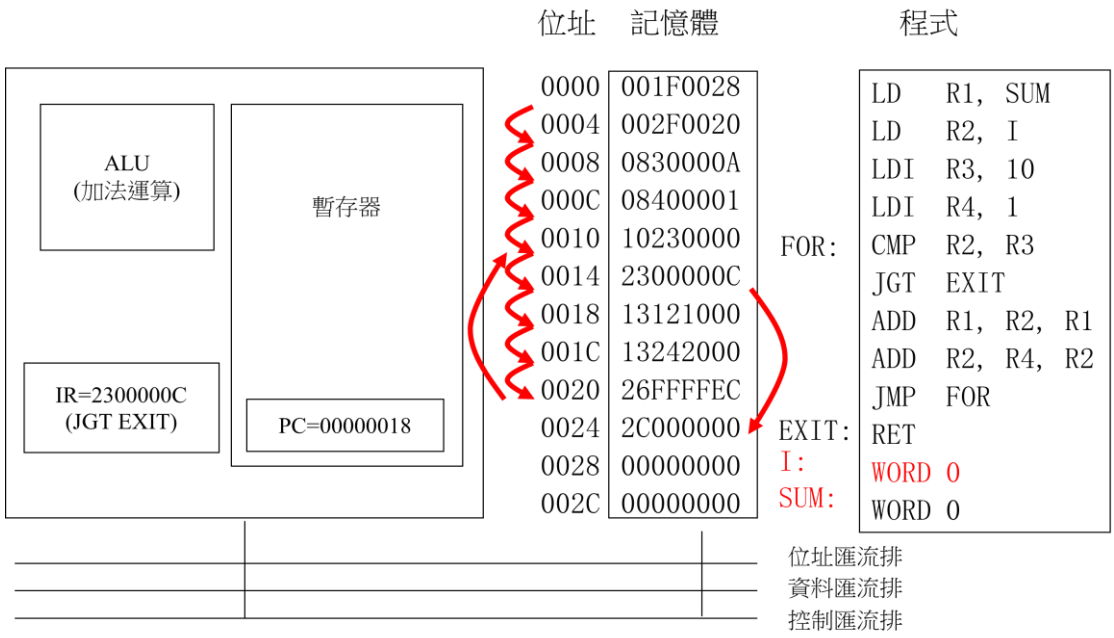

圖 2.15 程式在 CPU0 中的執行過程

一個指令的執行過程

CPU0 在執行一個指令時，必須經過提取、解碼與執行等三大階段。圖 2.16 顯示了這三個階段的詳細步驟。

階段 (a)：提取階段
　　動作 1、提取指令　　：IR = [PC]
　　動作 2、更新計數器　：PC = PC + 4
階段 (b)：解碼階段
　　動作 3、解碼　　：控制單元對 IR 進行解碼後
　　，設定資料流向開關與 ALU 的運算模式
階段 (c)：執行階段
　　動作 4、執行　　：資料流入 ALU，經過運算後，流回指定的暫存器

圖 2.16 指令執行的三大階段 – 提取、解碼、執行

提取階段的動作有兩個，除了提取記憶體中的指令到 IR 暫存器，還必須讓程式計數器 PC 前進到下一個指令 (在 CPU0 當中就是讓 PC=PC+4)。

動作 1 的指令提取過程又可進一步細分為 4 個步驟，如圖 2.17 所示。首先 CPU 要將程式計數器 PC 的內容傳上位址匯流排。然後透過設定控制匯流排的內容，以便向記憶體請求讀取指令。當記憶體收到該請求後，就會將位於 PC 的指令傳到資料匯流排上。然後 CPU 會將資料匯流排上的指令儲存到 IR 當中。如此就完

成了指令的提取動作。

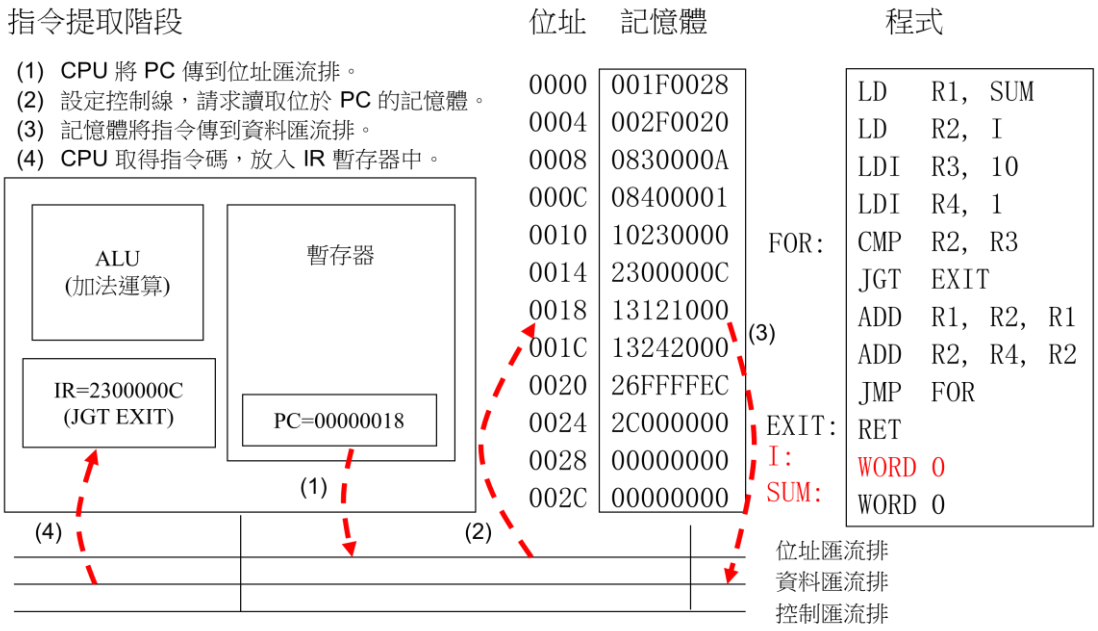

圖 2.17 指令提取的細部動作

在指令提取完畢之後，指令機器碼已經被提取到 IR 當中了，此時就可以進行解碼動作。控制單元會根據 IR 的指令碼，設定許多線路，像是資料流向開關，ALU 的動作線路等，一但這些線路設定完畢後，CPU 就進入執行階段了。

執行階段的動作根據指令的類型而有所不同，基本上就是我們在前一節中用資料流向圖所表達的那些動作。舉例而言，假如執行的指令是 ADD R1, R2, R1，那麼再解碼階段時，控制單元就會先設定好 ALU 的動作為加法，並且打開 R2, R1 暫存器流向 ALU 的開關，讓 R1, R2 的資料流入 ALU。接著會將 ALU 的輸出設定為 R1，於是 R2+R1 的結果就會流回到 R1 當中。

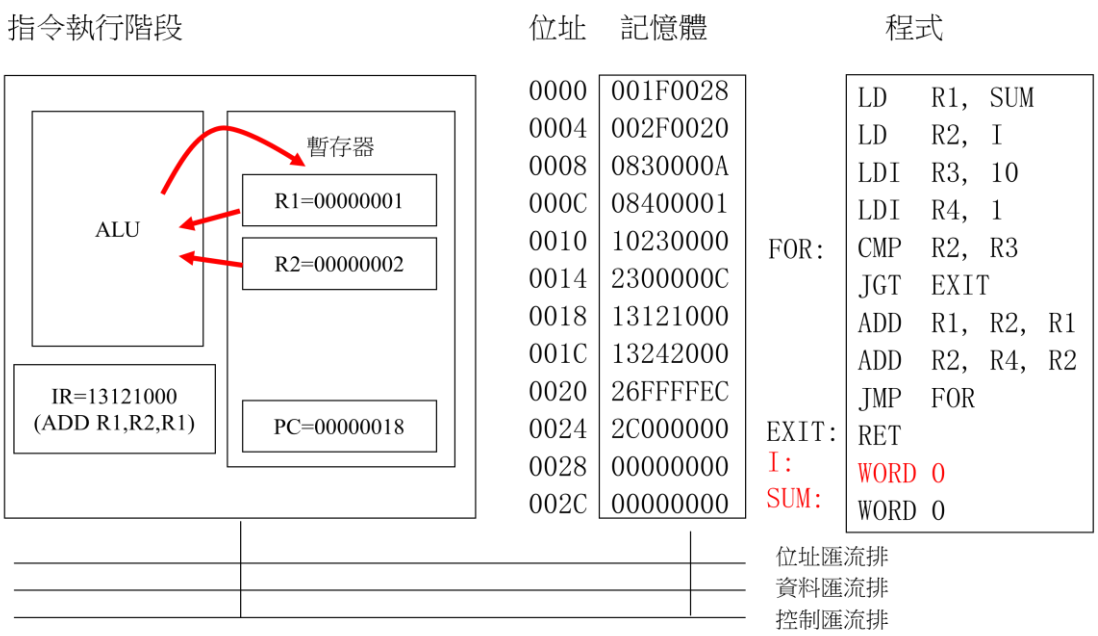

圖 2.18 加法指令 ADD R1, R2, R1 的執行階段動作

到目前為止，我們已經介紹完 CPU0 的處理器、暫存器、指令集、指令格式、定址模式等基本元素，並且說明了 CPU0 程式的執行過程。這對我們學習組合語言、連結器與載入器等主題已經足夠了。在後續章節中，我們將利用本章所學到的 CPU0 架構，進一步說明系統軟體的相關主題，我們對 CPU0 架構的探討將到此暫時畫下句點。

2.5 實務案例

2.5.1 IA32 處理器

IA32 是 Intel 公司所設計的處理器，屬於 x86 系列處理器的成員。由於 x86 系列的歷史很長，因此，IA32 的設計上可以看到很多歷史的遺跡。這也是 IA32 較為複雜的原因之一。

目前我們所使用的 IBM PC 個人電腦，是以 IA32 CPU 為核心的一個複雜架構，包含有 CPU、匯流排、平行埠、序列埠、南北橋晶片、記憶體、磁碟機、顯示裝置等，如圖 2.19 所示。其中的中央處理器通常就是 IA32 處理器。

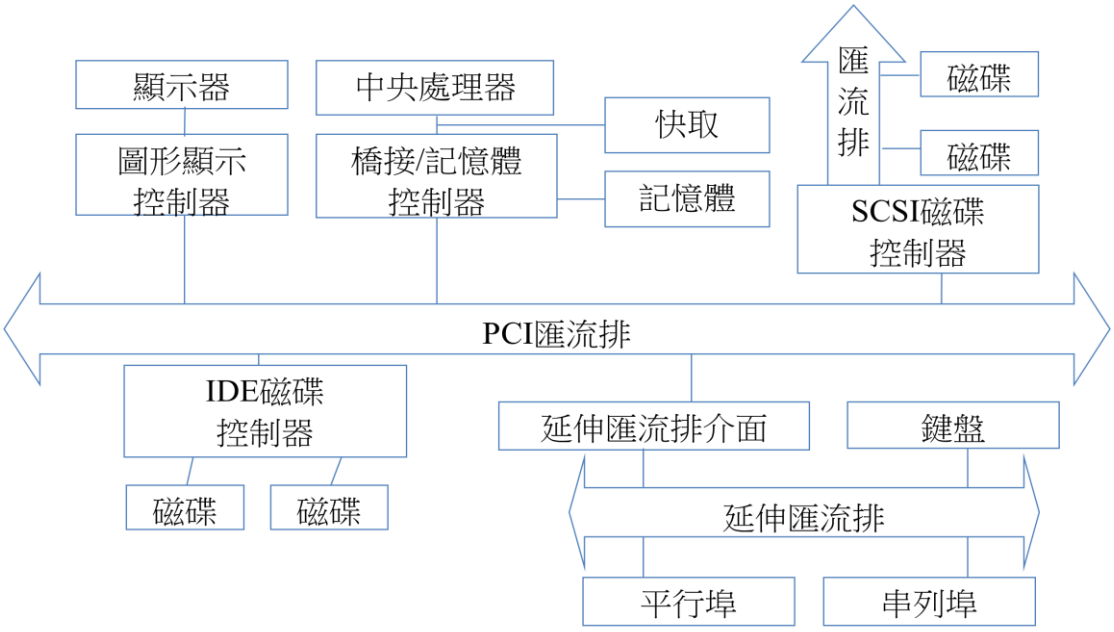

圖 2.19 個人電腦的結構圖

圖 2.20 顯示了 IA32 的常用暫存器，包含四個通用暫存器 EAX, EBX, ECX, EDX。這些通用暫存器常用來載入變數與儲存運算結果。暫存器 EBP (Extended Base Pointer) 可是框架指標 (Frame Pointer, FP)。ESP(Extended Stack Pointer) 則是堆疊指標，相當於 CPU0 中的 Stack Ponter (SP)。ESI (Extended Source Index) 與 EDI (Extended Destination Index) 則是索引暫存器，ESI 通常指向來源位址，EDI 則指向目標位址。

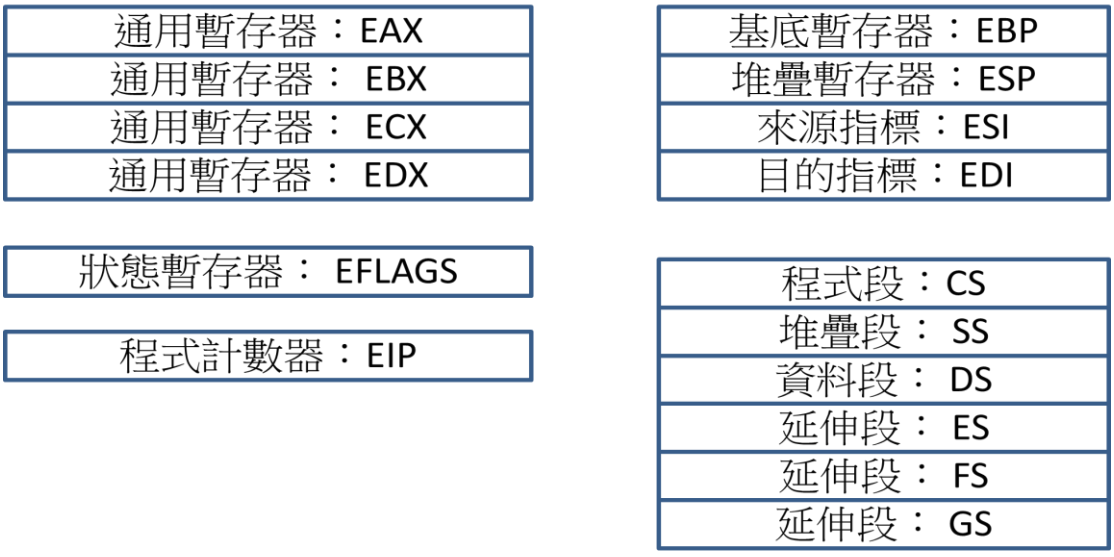

圖 2.20 IA32 的常用暫存器

在乘法與除法上，EAX 會自動作為目標暫存器，因此 EAX 又稱為延伸累加器 (Extended Accumulator)。另一個通用暫存器 ECX 則經常被當成迴圈計數器使用。

狀態暫存器 EFLAG (Extended Flags Register)用來儲存旗標值，相當於 CPU0 中的 SW 暫存器 (R12)。EFLAG 包含進位 (Carry)、溢位 (Overflow)、符號 (Sign)、零值 (Zero) 等條件旗標，另外還包含輔助進位旗標 (Auxiliary Carry:AC) 與同位旗標 (Parity Flag:PF) 等。

EIP (Extended Instruction Pointer) 是程式計數器，相當於 CPU0 中的 Program Counter (PC, R15)。另有六個 16 位元的區段暫存器，CS、SS、DS、ES、FS、GS。暫存器 CS (Code Segment) 通常指向程式段開頭，DS (Data Segment) 通常指向資料段開頭，SS (Stack Segment) 通常指向堆疊段開頭。ES (Extra Segment) 則指向不特定段落。FS (Flag Segment) 指向旗標段開頭。而 GS (Global Segment) 則指向全域段的開頭。

在 80286 的時代，x86 系列處理器是 16 位元的，因此 AX, BX, CX, DX, CS, SS, DS, ES, FS, GS 等暫存器都只有 16 位元。由於 16 位元只能定址 64 K 的記憶體，不符合當時的需求，因此 80286 採用了『區段+位移』的組合方式，讓 80286 可以定址到 1MB 的記憶體空間。

80286 的『區段+位移』定址法，是利用區段暫存器 (CS, DS, SS, ES) 與位移暫存器 (IP, SI, DI, BP) 組合出來的，其計算方式如下列公式所示。

實際位址 = 區段位址 × 16 +位移

在 80386 的時代，x86 系列處理器變成 32 位元的。為了與 80286 相容，Intel 選擇將原先的 AX, BX, CX, DX, IP, BP, SP, SI, DI, FLAGS 等 16 位元暫存器的名稱保留，然後以 EAX, EBX, ECX, EDX, EIP, EBP, ESP, ESI, EDI, EFLAGS 等名稱代表整個 32 位元版的暫存器，因此，這些 32 位元暫存器的較低 16 位元有專有的名稱，而該 16 位元又可以被分為高位元組 (High Byte) 與低位元組 (Low Byte) 兩部分。以 EAX 為例，其各名稱之間的關係如圖 2.21 所示。

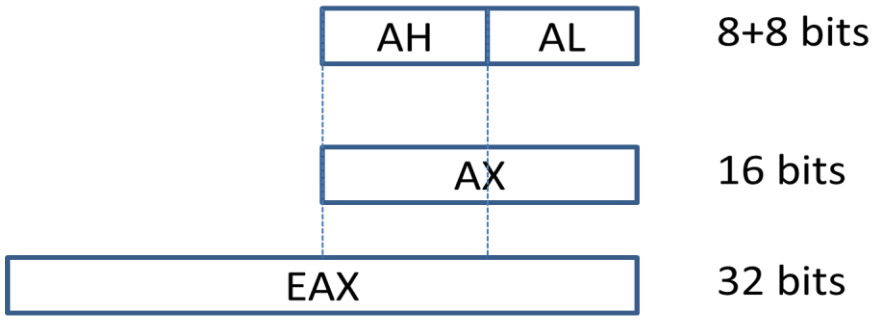

圖 2.21 IA32 的 EAX 暫存器

在 80386 的時代，浮點運算器是外加的晶片，但是在 80486 之後，就被整合進 CPU 當中。在浮點運算器當中，共有 ST(0)..ST(7) 等八個浮點資料暫存器。另外，在 Pentium II 的時代，為了多媒體的應用，特別又加入了八個 64 位元的 MMX 暫存器。然後，在 Pentium III 的時代，又加入了八個 XMM 暫存器，以便支援 SIMD (Single-Instruction Multiple Data) 的平行計算方式，以加快速度。

IA 32 的指令集

x86 乃是複雜指令集的典型，除了指令的數目眾多之外，指令的格式也特別複雜。

在精簡指令集當中，通常只有載入與儲存指令可以存取記憶體，其他指令是不能存取記憶體的。但是在 IA32 當中，像是 ADD、MOV 等指令都可以存取記憶體，所以這些指令的參數有很多種組合形式。舉例而言，像是 ADD 指令就有下列六種參數組合形式。

表格 2.2：ADD 指令的六種參數組合形式

ADD reg, reg	ADD reg, mem	ADD mem, imm
ADD mem, reg	ADD reg, imm	ADD accum, imm

MOV 指令的參數組合形式更多，共有表格 2.3 所示的九種組合形式。

表格 2.3：MOV 指令的九種參數組合形式

MOV reg, reg	MOV mem, reg	MOV reg, mem
MOV reg16, segreg	MOV segreg, reg16	MOV reg, imm
MOV mem, imm	MOV mem16, segreg	MOV segreg, mem16

這樣的設計導致 IA32 具有一個相當大的指令表，而且每個指令都有對應的參數

組合形式。表格 2.4 顯示了 IA32 的各個指令的簡表 [9] [10] [11]，進一步的資訊可以參考 Irvine 的組合語言一書[12]。

表格 2.4：IA32 的指令分類表

類型	代表指令	屬於本類型的指令
運算	ADD	ADD, SUB, ADC, MUL, DIV, IDIV, IMUL, SBB
邏輯	AND	AND, OR, NOT, XOR, NEG
位元	BT	清除：CLC, CLD, CLI, CMC 設定：SET, STC, STD, STI 測試：BT, BTC, BTR, BTS, TEST
副程式	CALL	CALL, RET, RETN, RETF
轉換	CBW	CBW, CDQ, CWD
比較	CMP	CMP, CMPS, CMPSB, CMPSW, CMPSD, CMPXCHG
字元調整	AAA	AAA, AAD, AAM, AAS, DAA, DAS
交換	SWAP	BSWAP, XCHG, XADD, XLAT,XLATB
增減	INC	INC, DEC
框架	ENTER	ENTER, LEAVE
暫停	HLT	HLT, NOP
輸出入	IN	輸入：IN, INS, INSB, INSW, INSD, 輸出：OUT, OUTS, OUTSB, OUTSW, OUTSD
中斷	INT	INTO, IRET
跳躍	JMP	JA, JNA, JAE, JNAE, JB, JNB, JBE, JNBE, JG, JNG, JGE, JNGE, JL, JNL, JLE, JCXZ, JECXZ
載入儲存	LEA	LEA, LDS, LES, LFS, LGS, LSS, LAHR, LAHF, SAHF
迴圈	LOOP	LOOP, LOOPW, LOOPD, LOOPE, LOOPZ, LOOPNE, LOOPNZ
重複	REP	REP, REPZ, REPE, REPNE, PEPNZ
移動	MOV	MOV, MOVSX, MOVZX
字串	LODS	(載入) LODS, LODSB, LODSW, LODSD, (移動) MOVS, MOVSB, MOVSW, MOVSD (掃描) SCAS, SCASB, SCASW, SCASD (儲存) STOS, STOSB, STOSW, STOSD
堆疊	PUSH	PUSH, PUSHA, PUSHAD, PUSHF, PUSHFD, PUSHW,

[9] x86 Instruction Set Reference, http://siyobik.info/index.php?module=x86
[10] The Intel 8086 / 8088/ 80186 / 80286 / 80386 / 80486 Instruction Set, http://home.comcast.net/~fbui/intel.html
[11] Intel Architecture Software Developer's Manual - Volume 2, http://developer.intel.com/design/pentium/manuals/24319101.pdf
[12] 該書的網址位於 http://kipirvine.com/asm/。

		PUSHD, POP, POPA, POPAD, POPF, POPFD
移位	SHL	SHL, SHLD, SHR, SHRD, ROR, RCL, RCR, SAL, SAR
浮點	FADD	FADD, F2XMI, FABS, FADDP, FIADD, FBLD, FBSTP, FCHS, FCLEX, FCMOVcc, FCOM, FCOMI, FCOS, FDECSTP, FDIV, FDIVP, FIDIV, FDIVR, FDIVRP, FIDIVR, FFREE, FICOM, FILD, FINCSTP, FINIT, FIST, FISTTP, FLD, FLD1, FLDL2T, FLDL2E, FLDPI, FLDLG2, FLDLN2, FLDZ, FLDCW, FLDENV, FMUL, FMULP, FIMUL, FNOP, FPATAN, FPREM, FPTAN, FRNDINT, FRSTOR, FSAVE, FSCALE, FSIN, FSINCOS, FSQRT, FST, FSTCW, FSTENV, FSTSW, FSUB, FSUBP, FISUB, FSUBR, FSUBRP, FISUBR, FTST, FWAIT, FXAM, FXCH, FXRSTOR, FXSAVE, FXTRACT, FYL2X, FYL2XPI
其他	WAIT	WAIT, BOUND

x86 系列的處理器已經從 8088, 80286, 80386 不斷改良，經過 80486、IA32 系列的 Pentium I, II, III, IV 等，目前仍不斷進步中。甚至已有 64 位元的 IA64 處理器出現了，未來，應該會持續改良進步，其架構也會不斷變化。

習題

2.1 請畫出馮紐曼電腦的基本架構圖。
2.2 請說明暫存器在電腦中的用途？
2.3 請說明控制單元在電腦中的用途？
2.4 請說明 ALU 在電腦中的用途？
2.5 請問 CPU0 有哪些暫存器？並說明這些暫存器的功能？
2.6 請問 CPU0 的指令可分為哪幾類？並且以範例說明每一類指令的功能？
2.7 請問 CPU0 當中有哪些定址方式，並以範例加以說明？
2.8 請說明 CPU0 程式的執行原理，並說明指令暫存器與程式計數器在程式執行時的作用？
2.9 請畫出 ADD R5, R6, R1 指令的資料流向圖，並說明該指令的運作方法。
2.10 請畫出 LD R5, [480] 指令的資料流向圖，並說明該指令的運作方法。
2.11 請簡要說明 IA32 處理器的特性？