

Z80 微電腦

軟體，硬體

(上 冊) 陳金追 編著

儒林圖書公司 印行

前 言

微電腦教育目前正於國內各學校如火如荼地展開。這是一種十分令人鼓舞之現象，亦是一種極為正確之作法。微電腦對社會所帶來之衝擊是無可抗拒的。瞬即，此一小精靈將遍及社會之各行各業，進而與生活之每件事物相結合。但據筆者了解，除一些較具規模之大專院校外，其它許多專科或高工學校之微電腦教學，若非遲未見實現，即實行之效果不彰。究其原因，若非師資、設備與教材不足，即為教法運用不當。有鑑於此，作者特編著此書，期能略盡微薄之力，以收拋磚引玉之效。

微電腦之學習，必須由認識微處理器之內部結構、了解其動作原理開始，學習如何對之作程式設計（亦即如何使用之）。進而再熟悉輸入／輸出之技巧，輸入／輸出界面晶片之構造、功能、與程式規劃。最後將微處理器、ROM、RAM、界面晶片、以及週邊設備等諸系統組件連接在一起，加上必要之軟體，構成一完整之微電腦系統。

本書之內容涵蓋上述範圍。第一章介紹程式設計之基本概念。第二章說明一般微處理器與Z80微處理器之內部結構與動作原理。第三章至第十二章主要談Z80微處理器之組合語言程式設計，其教讀者如何使用Z80 CPU。十三章至十五章則討論輸入／輸出。十三章讀輸入／輸出程式設計，十四章介紹輸入／輸出技巧，十五章描述輸入／輸出界面晶片之內部構造、功能、與如何使用。最後一章則簡短介紹Z80微電腦系統之硬體組成。全書假設讀者以前毫無程式設計經驗，內容敘述淺顯且詳細，並採邊介紹、邊舉例說明，且邊作練習之敘述方式。課文中穿插許多圖解，幫助讀者了解，並備有許多習題，提供讀者最佳練習機會。

本書適用於以Z80微處理器構成之任何微電腦系統，其中包括國

致讀者

人自製之 Edu-80，PA-800，以及國外進口之 ZDS，ZDOS，以及 TRS-80 等系統。由於作者所知有限，加上時間瑣碎不整，錯誤與瑕疵之處在所難免。誠盼專家先進不吝指正，不勝感激！

編著者

陳金迫 謹識

1981.8.1

於台北

一般初學者都覺電腦很難學，尤其是組合語言程式設計，學了好幾遍以後還是不能甚解。其實，這也無所值得驚訝的。因為，學習組合語言程式設計必須具備符號思考能力，以及計算機系統組織與硬體方面較專門之基礎，而不如學 BASIC，FORTRAN，或 COBOL 等高階語言，學者祇需具備高中數學、英文程度，與簡單之邏輯思考能力即可。雖非必要，但具有某一高階語言（如 BASIC 或 FORTRAN）之程式設計經驗，對學組合語言程式設計會有助益。因為，這樣可先使您了解計算機到底是怎麼回事——其到底能“做什麼”，以及其究竟“如何”在做事（如何一步一步地執行程式）。

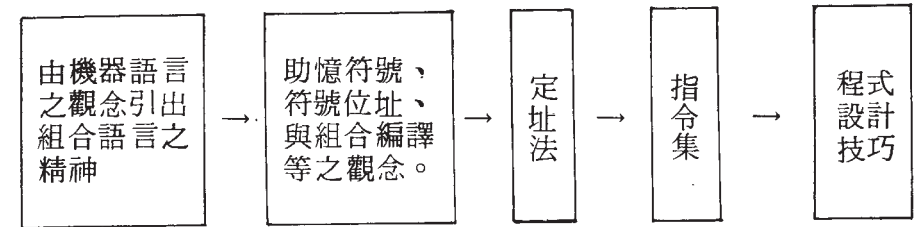
根據作者之經驗，一般初學者最覺困惑之處有：記憶位址與記憶內含之觀念、符號位址之觀念、定址法之意義與使用、以及虛指令之使用等等。除記憶位址與記憶內含之觀念可參考余所著之“微電腦基礎”一書外，其餘的在本書都有詳細介紹。此外，指令集之龐大複雜亦常令初學者有捉不著重心之感。因此，如何抓住重心與學到組合語言之精髓乃最重要之課題。初學程式設計者應將重點置於隱含、立即、直接（擴展）、與索引等基本定址法之意義與使用的了解，資料傳送、算術、邏輯、移位旋轉、與跳越等指令之運算功能了解，以及各種特定程式之設計技巧的了解（例如，多段算術如何製作，乘除算程式如何設計等等）三方面。

「熟能生巧」，乃學習計算機程式設計之最佳座右銘。多找題目寫程式上機，是唯一的最佳學習途徑。此外，多讀別人之程式（必須是好的程式），以他人之經驗為自己之經驗，是增進自己程式設計技巧之最快方法。初學電腦時滿腦子疑問是正常的，而且是好事，切莫因此而灰心。遇有疑難就要多查資料多發問，來一個解決一個。如此，進步就會很神速。問題問多了，一一得到解答，有一天您就會撥雲

見天，豁然開朗。屆時，順水推舟之日即已來到！

致教師

講授計算機程式設計的確是一種高超之“藝術”，尤以組合語言程式設計為然。一般而言，若學生已具備計算機系統組成與動作原理方面之基礎，則介紹此一門課之大致程序為：



先具基礎：
計算機系統組成與動作原理。
機器指令格式。
記憶位址與記憶內含觀念。

邊介紹指令
邊講解程式。
順便說明虛
指令使用。

副程式，
MACRO，
結構化程
式設計。

指令集介紹時，必須邊介紹指令，邊舉程式例題，並作講解，然後讓學生作練習。若時間不足，教師可視狀況介紹每一類指令中某幾個較常用者即可，而不必所有指令均一一介紹完畢。甚至，某一指令亦不必其所有的定址法均介紹完盡。指令集之介紹可依下列之順序：

- 1 資料搬運指令（取入、存出）。寫一最簡單之程式。
- 2 算術指令（加、減）。程式舉例可舉兩數相加，兩數相減，多段算術，與 BCD 算術等例題。
- 3 邏輯運算指令。介紹 AND, OR, XOR 等指令之功用與常見應用，如資料合併、分離、保留、與測試等。
- 4 跳越指令（形成迴路）。例舉求一系列數目之和的例子，順便說明索引定址之功效。
- 5 比較指令。舉求最大（小）值例題。介紹排序（**sorting**）與

搜尋 (searching) 技巧。

6. 移位與旋轉指令。介紹軟體乘 / 除算程式。

7. 輸入 / 輸出指令。介紹輸入 / 輸出作業之概念。

每階段一舉例講解後，立即指定習題，讓學生上機練習。先指定類似習題，再指定具創造性習題。

程式設計涉及了解與使用許多之規則，工作精細而瑣碎。教師必須懷以最大之耐心，不厭其煩地一一為學生解惑、偵錯，澄清其觀念，（此一個別指導機會甚為重要）方能苦盡甘來，漸顯效果。眼見自己灌溉之幼苗開花結果那份喜悅是無法言喻的，讓您我共勉！

目 錄

前 言

上 冊

第〇章 引 言

第一章 基 本 概 念

1-1 何謂程式設計.....	5
1-2 畫流程圖.....	6
1-3 資訊表示.....	7
1-4 計算機內部資訊表示.....	8
1-4-1 程式.....	9
1-4-2 數值資料.....	10
1-4-3 文數字資料.....	32
1-5 外部資訊表示.....	34
1-5-1 組合語言.....	34
1-5-2 八進制與十六進制.....	35
習題.....	39
習題解答.....	40

第二章 Z80硬體結構

2-1 Z80 微電腦系統結構.....	43
2-2 一般微處理器之結構與動作原理.....	46
2-2-1 暫存器.....	47

2-2-2	堆疊器	50
2-2-3	指令執行週期	52
2-2-4	拿取下一指令	54
2-2-5	指令之執行	54
2-2-6	臨界競賽問題	57
2-3	Z 80 微處理器之結構	59
2-3-1	一般用途暫存器	62
2-3-2	旗號暫存器	66
2-3-3	特殊用途暫存器	67
2-4	Z 80 之指令格式	73
2-5	Z 80 之指令執行	77
2-5-1	拿取週期	80
2-5-2	解碼與執行	82
2-5-3	重要練習	85

第三章 定 址 法

3-1	隱含定址	98
3-2	立即定址	99
3-3	擴展立即定址	100
3-4	暫存器定址	101
3-5	暫存器間接定址	102
3-6	擴展定址	104
3-7	修正零頁定址	105
3-8	相對定址	106
3-9	索引定址	110
3-10	位元定址	114

第四章 Z80 指令集

4-1	計算機指令之種類	119
4-2	Z 80 指令集	125
4-3	資料傳送指令	126
4-3-1	八位元傳送	126
4-3-2	十六位元傳送	129
4-3-3	交換指令	132
4-3-4	區段(整批)傳輸指令	133
4-3-5	區段搜尋指令	136
4-4	資料處理指令	137
4-4-1	算術與邏輯指令	137
4-4-2	移位與旋轉指令	146
4-4-3	位元運作指令	151
4-5	測試與控制轉移指令	151
4-5-1	旗號	151
4-5-2	控制轉移指令	158
4-6	輸入/輸出指令	164
4-7	各種 CPU 控制指令	168
	摘 要	169
	習題解答	170
	Z 80 指令集摘要	172

第五章 Z80 組譯程式

5-1	機器語言	421
5-2	組合語言	421
5-3	組譯程式	423
5-4	組譯程式之特色	424

5-4-1	符號位址	424
5-4-2	組合語言格式	427
5-4-3	數底表示	430
5-4-4	算式求值	430
5-4-5	虛指令	431
5-5	上機	437

第六章 資料傳送

6-1	八位元傳送	440
6-2	十六位元傳送	449
6-3	區段(整批)傳送	455
6-4	資料互換	459
6-5	摘要	461
6-6	副程式	466

第七章 算術程式

7-1	加 算	471
7-1-1	八位元加算	471
7-1-2	十六位元加算	474
7-2	減 算	477
7-3	BCD算術	479
7-3-1	八位元BCD加算	479
7-3-2	十六位元BCD加算	481
7-3-3	濃縮BCD減算	482
7-4	乘 算	484
7-4-1	8×8乘算	486
7-4-2	改良之8×8乘算程式	492
7-4-3	16×16乘算	496
7-5	除 算	498
7-6	比較運算	504

第八章 移位、旋轉、與位元運作

8-1	邏輯移位	511
8-2	旋轉型移位	516
8-3	算術移位	520
8-4	BCD數字移位	522

Z80 微電腦軟體硬體

8—5 位元運作.....	526
---------------	-----

第九章 表列與表格處理

9—1 資料串.....	535
9—2 表格作業.....	544
9—2—1 表格索引.....	544
9—2—2 剔除一已知元素.....	547
9—2—3 加入一新元素.....	550
9—2—4 二分搜尋.....	553
9—3 表列作業.....	557
9—3—1 泡浮排序.....	559
9—3—2 單端連鎖表列.....	565

第十章 程式設計技巧

10—1 表格跳越	572
10—2 副程式	578
10—2—1 何謂副程式.....	578
10—2—2 副程式之叫用與回返	578
10—2—3 副程式之使用及優缺點	533
10—2—4 參數傳遞	587
10—2—5 副程式巢串	588
10—2—6 副程式文書	590
10—3 Z80 副程式之特色.....	593
10—3—1 條件叫用與回返	593
10—3—2 重始指令	596
10—4 再進入.....	599

第十一章 常用副程式

11—1 比較副程式	604
11—2 計時副程式	608
11—3 乘除副程式	609
11—4 多段算術常式	612
11—5 ASC II 至基底X之轉換.....	614
11—6 基底X至ASC II 之轉換	620
11—7 資料填補常式	625
11—8 資料串比較常式	625
11—9 找最大值副程式	626
11—10 泡浮排序副程式.....	629
11—11 表格搜尋常式	632

第十二章 Z80 CPU之界面信號與時序

12—1 位址與資料巴士	634
12—2 巴士控制信號	636
12—3 記憶體控制信號	636
12—4 輸入／輸出信號.....	637
12—5 其他CPU 控制信號	638
12—6 與插斷有關之信號	639
12—7 Z80 CPU之電特性	640
12—8 M1 週期	643
12—9 記憶體資料讀取與寫入週期.....	645
12—10 輸入與輸出週期	648
12—11 巴士請求／認可週期	650
12—12 插斷請求／認可週期	651
12—13 不可罩蓋插斷週期	653
12—14 脫離暫停	654

第十三章 輸入／輸出程式設計

13-1 輸入／輸出指令	656
13-1-1 累加器輸入／輸出指令	657
13-1-2 使用暫存器C之輸入／輸出指令	662
13-1-3 整批輸入／輸出指令	663
13-2 產生脈衝信號與延遲	668
13-3 資訊傳遞之方式	676
13-3-1 並行資訊傳輸	676
13-3-2 串行資訊傳輸	680
13-4 與輸入／輸出設備溝通	685
13-4-1 握手連絡	686
13-4-2 七段LED顯示	686
13-4-3 電傳打字機輸入／輸出	692
13-4-4 印出一串文數字	698

第十四章 輸入／輸出技巧

14-1 取 樣	701
14-2 插 斷	705
14-2-1 何謂插斷	705
14-2-2 插斷之用途	706
14-2-3 插斷處理	708
14-2-4 可罩蓋與不可罩蓋插斷	711
14-2-5 Z80插斷之致／禁能	712
14-3 Z80之插斷系列	714
14-4 巴士請求	716
14-5 不可罩蓋插斷	716
14-6 可罩蓋插斷	718

14-6-1 插斷型態0	719
14-6-2 插斷型態1	722
14-6-3 插斷型態2 (向量式插斷)	723
14-6-4 多個設備共用同一插斷線	727
14-6-5 巢串插斷	732
14-6-6 與插斷有關之Z80指令	734
14-7 直接記憶體存取 (DMA)	736

第十五章 輸入／輸出界面電路

15-1 Z80-PIO	740
15-1-1 特 性	740
15-1-2 接腳說明	741
15-1-3 內部結構	745
15-1-4 作業型態	748
15-1-5 插斷與重置	751
15-1-6 程式規劃	754
15-1-7 插斷服務	757
15-1-8 應 用	761
15-2 Z80-CTC	763
15-2-1 接腳功能	764
15-2-2 內部結構	767
15-2-3 作業型態	774
15-2-4 程式規劃	776
15-2-5 時 序	781
15-2-6 插斷服務	784

第十六章 組成一微電腦系統

16-1 最小之Z80系統	789
---------------------	-----

Z 80 微電腦軟體硬體

16—2	ROM與RAM之界面·····	793
16—3	動態記憶體之界面·····	795
16—4	記憶體速度控制·····	797
16—5	Z80 PIO 之界面·····	798
附錄A	：十六進制轉換表·····	802
附錄B	：ASCII 轉換表·····	803
附錄C	：相對跳越位移表·····	804
附錄D	：十進數與BCD轉換·····	805
附錄E	：Z80 指令碼·····	806
附錄F	：Z80 與 Z80 指令對等·····	813
附錄G	：8080 與 Z80 指令對等·····	814

第 〇 章

引 言

西元 1971 年，美國 Intel 公司推出世界上第一部字長為四位元之微處理器——Intel 4004。雖然 4004 並非真正的單晶微處理器（single-chip microprocessor），但其確已包含一計算機中央處理單元之大部份電路。Intel 4004 的來臨，使得一塊大型積體電路（LSI）便能取代傳統迷你電腦之數以百計電路。Intel 4004 之 46 個指令的指令集雖說不大，但其却已能滿足可規劃邏輯陣列（PLA）不易達成，需要決策能力但不需廣泛數學計算能力之控制應用。4004 每次能處理四位元之資料（因此稱為四位元微處理器），並且每秒鐘能做十萬次兩個四位元數目之相加。

Intel 推出之第二代微處理器，保留了第一代微處理器之 PMOS 製造技術，但資料巴士（字長）却增為八位元，且指令集亦增為 48 個指令。此一微處理器稱為 Intel 8008。因為不論指令之解碼、執行，或運算元，每次均能八位元一併處理，因此，8008 之指令週期時間比 4004 快速。除此之外，8008 亦具有下列優點：其能選取 16384 個八位元之記憶位置，包括七個八位元之暫存器，具有記憶堆疊能力，以及一單層之插斷能力。8008 每秒鐘能做 80000 次兩八位元數目的相加。但其指令集並不吻合（Compatible）於 4004。

於推出後，8008 與 4004 很快就受到電子工業界廣泛的採用，主要因為當時幾乎沒有其它類似之產品。就像於 4004 後曾推出其改良型 4040 一樣，繼 8008 後，Intel 於 1973 又推出在軟體上與 8008 相吻合之 Intel 8080。此一微處理器除了具有 8008 之所有指令外，又額外增加了 30 個指令。8008 的用者現在可以換上一速度

更快，用途更廣之微處理器，而不必捨棄其原來 8008 之程式。因為 8008 之軟體在 8080 上大概皆能執行。8080 由 NMOS 技術製成，具有比 8008 更高之時序頻率，每秒鐘能做五十萬次八位元數的相加。此外，由於 8080 做於僅有 40 支接腳之晶片，因此，於資訊傳輸與指令執行時，CPU 必須共用資料巴士之時間也隨之縮短。

在硬體方面，8080 亦強於 8008。8080 所能選取之記憶位址高達 65536 (64K) 個，而非 16384 (16K)。8080 之堆疊器亦非有限七層之 CPU 堆疊器，而是設於外部記憶器內，幾乎可“無限”延伸之記憶器堆疊器。此外，8080 亦具有 BCD 算術運算能力，能做 BCD 加法，以及較廣泛之定址法，能做記憶器之直接定址。對許多程式設計而言，8080 之 78 個指令似乎有點不可思議，但此一指令集確無疑地將 8080 之應用範圍，自單獨之控制領域，轉移至更一般化之領域。

1976 年，Intel 又推出了 8080 之多種變樣。新改良的 Intel 8085 不僅本身具有串行輸入／輸出能力，同時亦僅需一單相時序（8008 與 8080 均為雙相時序）與單一正 5 伏特電源（8008 與 8080 分別需要 2 個及 3 個電源）。在支援晶片之數量逐漸增加之際（如可程式化週邊界面、插斷控制器、與 CRT 控制器），Intel 為用者提供了愈來愈迅速之高度計算能力（8085 每秒能作七十七萬次兩八位元數目之相加）。而却保存了與 8008 及 8080 軟體之吻合性。

雖然在許多方面 8085 是 8080 之改良，但其指令集却極類似於 8080。8085 總共才僅增加兩個指令，一個用以讀取串行與插斷（interrupt）資料，另一個則用以寫入串行與插斷資料。8008 與 8080 先天上的許多缺點却仍存在。

1976 年，原來在 Intel 設計 8080 的一批工程師，離開了 Intel 加入 Zilog，設計了一新的八位元微處理器——Zilog Z 80。此一微處理器之目標是在軟體與硬體上皆比 8080 強，但在軟體上又能與 8080 吻合。因此，其象徵了 8008/8080 之另一成熟層次。記得，今日的產品總比昨日為佳，Z 80 之指令集與能力勝於 8080 的道理，

就如同 8080 優於 8008 一般。Zilog 公司同時也為 Z 80 設計一整套支援電路。此外，Z 80 在軟體上亦與 8080 相吻合，致使現有 8008 與 8080 之軟體，皆能在 Z 80 上執行。雖然 8008 與 8080 之指令與結構上的某些限制，Z 80 仍無法克服，但 Z 80 却提供了新的指令，新的定址法，以及比以前具有更高能力與更廣泛用途之硬體特色。

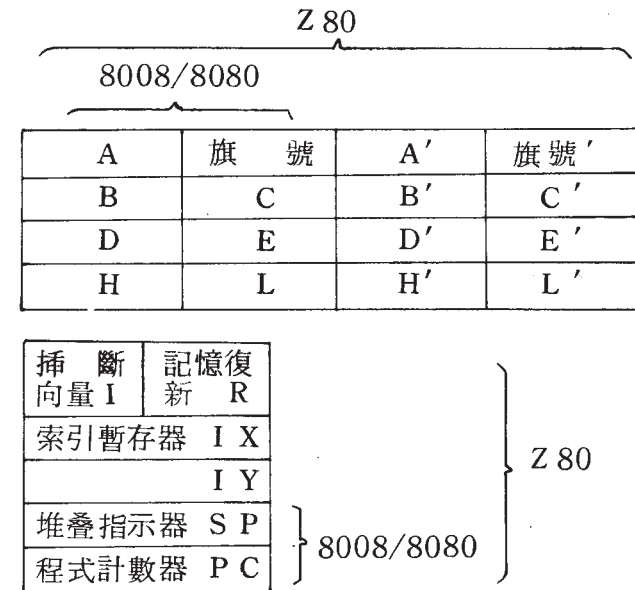


圖 0-1 8008，8080 與 Z 80 之暫存器比較

除了具有 8080 之八個八位元 CPU 暫存器外，Z 80 又再設置了八個同樣之暫存器，使其一般用途之暫存器達十六個。Z 80 還具有兩個 8080 所沒有的十六位元索引暫存器 IX 與 IY。另外，插斷向量暫存器與記憶復新暫存器，分別提供了特殊之插斷功能與動態記憶復新能力。圖 0-1 所示即為 8008，8080，與 Z 80 之基本暫存器結構圖。

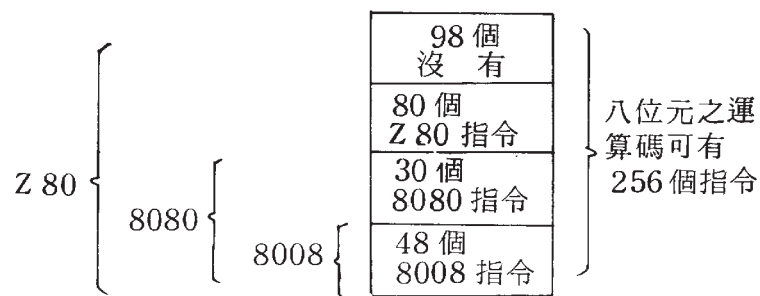


圖 0-2 8008，8080 與 Z 80 之指令集比較

Z 80 總共具有 158 個指令，涵蓋了 8080 之 78 個指令。所增加之指令有許多是 8080 指令之邏輯延伸，但亦有許多是十分能幹而為 8080 所沒有的。圖 0-2 所示即為 8008，8080，與 Z 80 三者之指令集的邏輯差異。

Z 80 保留了 8080 之所有輸入／輸出與插斷能力。但却擴充成能對任一 CPU 暫存器作業，以及“整批”(block)之作業方式。換言之，每一程式化(非 DMA)之輸入／輸出通道(I/O Channel)，每次能傳輸數個位元組。Z 80 之插斷則除了具有 8080 之標準插斷請求外，尚包括一類似於 Motorola MC6800 與 MOS Technology MCS 6502 所具備之“不可罩蓋”(nonmaskable)插斷。其它之插斷能力則使插斷向量能指至記憶器之任一位置，而非僅第 0 頁之八個記憶位置。同時，插斷之層次可高達 128 層，而非僅僅 8 層。

第 1 章

基本概念

本章介紹一些與計算機程式設計有關之基本概念與定義。熟悉此些概念之讀者或許想僅大略地翻一下，就跳至第 2 章。可是，作者還是建議您從頭至尾再讀一遍，因為，這裡面含有許多非常重要之概念，如 2 補數、BCD、以及其它之數目表示法等。其中有些或許是您以前沒學過的；其它的或許會增進您之常識與技巧也說不定！

1-1 何謂程式設計

已知一個問題，最重要的就是要找出其解決方案。此一解決方案，表示成一步一步之程序，即稱為**演算法**(algorithm)。因此，演算法即為一已知問題之解決方案的逐步說明。但其必須在有限步驟之內結束。演算法可以任意語言或符號表示。下面即為演算法之一簡單例子：

1. 鑰匙插入鑰匙孔。
2. 鑰匙向左轉一圈。
3. 握住門之把柄。
4. 將門把柄左旋，且推門。

此時，對門鎖之種類而言，若演算法正確，則門就會開。此一四步程序，夠資格算是一開門之演算法。

在以計算機解題時，問題之解決方案一旦表示成演算法，即可以計算機加以執行。很遺憾的，當前的事實是計算機根本聽不懂，亦無法執行一般之口語英文(或甚至其它任何人類所使用之語言)。這原因主要在於所有一般人類語言之語法均有**模擬兩可性**(syntactic

ambiguity)。計算機僅能了解一部份經過非常嚴格定義之自然語言，此即所謂的**程式語言** (programming language)。

將演算法轉換成某種程式語言之一系列指令的工作，即稱為**程式設計** (programming)。嚴格說，將演算法轉換成程式語言之階段，僅稱為**寫碼** (coding)。實際上，程式設計除了指寫碼外，尚包括實現演算法之整個程式與資料結構的設計。

有效之程式設計不僅需對標準演算法的可能製作技巧有所了解，同時亦需能善用諸如內部暫存器、記憶體、與週邊設備等計算機硬體資源。此些技巧將在下面幾章中討論。

程式設計同時亦需有嚴格之文書 (documentation) 紀律，以使程式能為自己以及他人所了解。程式文書不論在程式內或程式外，同時都需要。內部程式文書指的是置於程式本體內，解釋其作業之註解 (comment)。而外部程式文書則指程式以外之文書設計，其中包括書寫之解釋，手冊，與流程圖 (flowchart) 等。

1-2 畫流程圖

演算法與程式之間通常有一中間步驟，那就是**流程圖**。流程圖即為將演算法之步驟以一系列矩形與菱形表示所獲致之圖形。其中，矩形代表**命令** (command)，或可執行之述句 (executable statement)。菱形代表諸如“若資訊X為真，則作動作A；否則，作動作B”之**測試** (test)。此刻，我們並不打算對流程圖作正式定義。其細節將留待爾後與程式一起討論。

畫流程圖 (flowcharting) 乃程式設計過程中，由演算法設計進展至解決方案之寫碼時，非常有價值的中間步驟。據了解，大約僅有十分之一的人，能不必畫流程圖而寫出正確的程式。不幸的是，其餘百分之九十的人亦相信其亦能不畫流程圖而寫出正確之程式。結果，平均有百分之八十的程式，第一次上機時都跑 (run) 不出結果來。(當然，此一百分比僅為一約略數值)。簡言之，絕大多數程式設計之

生手並不了解畫流程圖之必要性。這通常導致程式錯誤或不完全之後果。然後，必須再費一段很長的時間，重新測試與校正式 (該階段稱為程式設計之**偵錯**，debugging，階段)。因此，不論何時，必須盡量養成畫流程圖之習慣。雖然這在寫碼之前會增加一些工夫，但却可獲致能正確與迅速執行之清徹 (clear) 程式。對腦筋較靈光之程式設計者而言，雖然有時流程圖能不必畫於紙上，但為了使其他人能輕易了解自己所設計之程式，流程圖還是不可免。

圖 1-1 所示即為前面開門之演算法繪成流程圖的情形。

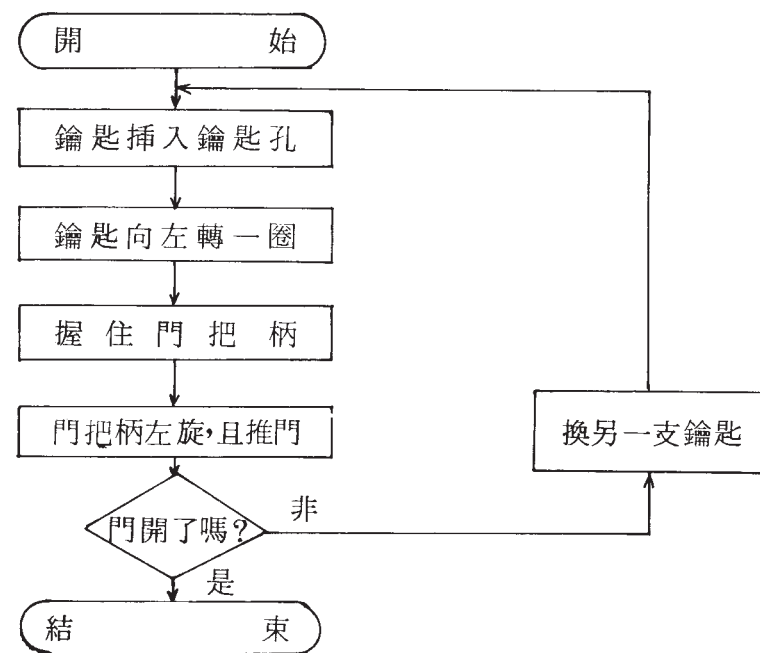


圖 1-1 開門演算法之流程圖

1-3 資訊表示

計算機是一種資訊處理機器，其所處理之資訊皆必須事先儲存於計算機之主記憶體內。計算機內部所儲存之資訊包括**程式** (program) 與**數據** (data) 兩種。程式即為一系列指令。每一指令“命令”計算

機做一種運算，如加、減、移位、旋轉等。數據即為計算機作運算時所需之資料 (data)。此些資料可分為**數值資料** (numeric data) 與**非數值資料** (nonnumeric data) 兩種。數值資料即為可作算術運算之數目，解科學題目之程式所處理者，通常屬於此一類。一般微電腦所能直接表示與處理之數值資料有整數、有號整數、與BCD 數等。大型的計算機尚能表示與處理小數。非數值資料則指諸如文章或人名等之文數字資料 (alphanumeric data)。商業應用之程式所處理者，多屬此一類型之資料。下面各節將分別就上述各種資訊加以介紹。由於資訊在計算機內部與外部之表示有所不同。因此，我們將分兩方面加以探討。

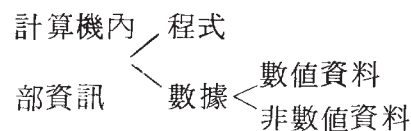


圖 1-2 計算機內部之資訊

1-4 計算機內部資訊表示

計算機內部之所有資訊皆表示成位元串的形式。一個**位元** (bit) 即為一位二進數字：0 或 1。由於構成計算機之電子電路皆設計成僅動作於兩種狀態（完全導電與完全不導電）。因之，計算機內部之資訊皆表示成二進制；不是 0，就是 1。每一項資訊就由一系列不同之 0 與 1 代表。譬如，十進數 12 就以“1100”代表，英文字母 A 就以“01000001”代表等。

於計算機內部，所有的資訊位元通常都幾個位元一組，幾個位元一組地儲存。於計算機術語上，八個位元稱為一個**位元組** (byte)，而四位元稱為半個位元組 (a nibble)。

現在，我們看看於計算機內部，資訊如何表示成二進形式。

1-4-1 程式

目前之計算機均屬於**儲存程式型** (Stored-program) 之計算機。此意即，任何程式在能被執行之前，必須先行存入計算機之主記憶器。然後，計算機之中央處理器方能從記憶器中，將程式之指令一一取入中央處理器，並且加以執行。

於計算機內部，程式之指令亦表示成一串的 0 與 1。於 Z 80，每一指令可能長 1, 2, 3, 或 4 個位元組（因此，有所謂的一位元組指令，二位元組指令，……之稱）。不論那一種，其皆具有下列機器指令之一般格式：

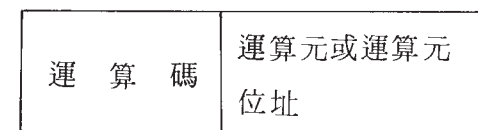


圖 1-3 計算機指令之格式

任何計算機之指令，儲存於記憶器內時，均包括兩部份。第一部份為**運算碼** (operation code，簡記為 **OP code**)。第二部份為**運算元位址** (operand address) 或運算元本身。運算碼代表計算機所必須執行之運算，如加、減、左移、右移等。於 Z 80，每一指令之運算確有長一位元組者，亦有兩位元組者。運算元位址部份則為計算機執行指令之運算時，所需之數據所在之記憶位置的位址。該部份有時亦可直接為運算元。於 Z 80，指令之運算元位址部份可為零，一，或兩個位元組。

舉個例子來說，於 Z 80，將數目 5 加至累加器 (accumulator) 之加法指令，存於記憶器的情形即如

```

11000110
00000101
  
```

，該指令長兩個位元組（因此，必須佔兩個記憶位置）。第一位元組為加法運算之運算碼（表示成十六進數即為C 6），而第二位元組即為運算元（十六進數05）。

圖 1-4 所示即為一已翻譯成機器碼，儲存於記憶器內之 Z80 程式的典型例子（右邊所附者則為註解）。

00111010	}	第一指令。
00100101		前一位元組為運算碼，後兩
00000000		位元組為運算元位址。
11000110	}	第二指令。前八位元為運算
00000101		碼，後八位元為運算元。
00110010	}	第三指令。
00100110		前一位元組為運算碼。
00000000		後兩位元組為運算元位址。
01110110	}	第四指令。（只有運算碼）

圖 1-4 Z 80 之程式例子

至此，有人或許要問，計算機怎能知道那一位元組是運算碼，那一位元組是運算元（或運算元位址）呢？這沒問題。只要計算機能正確拿取第一個指令之運算碼，經解碼後，其即可立即知曉次一緊接位元組究竟還是運算碼，或該為運算元位址（只要程式不寫錯）。

1-4-2 數值資料

數目之表示並不十分直截了當，而且必須分清數種狀況。若計算機欲能適合廣泛之應用，其必須能表示與處理整數，有號數——正數與負數，與十進數（包括小數）。下面，我們分別討論此各種情況。

一、整數

整數可以直接二進數(direct binary)表示。直接二進數就是將

數目之十進值，表示成二進制。於二進制，最右邊之位元代表 2^0 ，次一左邊位元代表 2^1 ，再次左代表 2^2 ，……等等。就一八位元之二進數

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

而言，其真正所代表之數值為

$$b_7 \times 2^7 + b_6 \times 2^6 + b_5 \times 2^5 + b_4 \times 2^4 + b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

。其中，

$$2^7 = 128, 2^6 = 64, 2^5 = 32, 2^4 = 16$$

$$2^3 = 8, 2^2 = 4, 2^1 = 2, 2^0 = 1$$

。二進數之表示類似於十進數，譬如，123 即表

$$1 \times 100 + 2 \times 10 + 3 \times 1 \\ = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

。於此“位置符號”(positional notation)，每一數位表示一10乘冪。於二進制，每一數位或“位元”則代表一2乘冪。

例：二進制之“00001110”代表

0 ×	1 = 0	2^0
1 ×	2 = 2	2^1
1 ×	4 = 4	2^2
1 ×	8 = 8	2^3
0 ×	16 = 0	2^4
0 ×	32 = 0	2^5
0 ×	64 = 0	2^6
0 ×	128 = 0	2^7
十進數		14

例：二進數“10000000”代表

$$\begin{array}{rcl}
 0 \times 1 & = & 0 \\
 0 \times 2 & = & 0 \\
 0 \times 4 & = & 0 \\
 0 \times 8 & = & 0 \\
 0 \times 16 & = & 0 \\
 0 \times 32 & = & 0 \\
 0 \times 64 & = & 0 \\
 1 \times 128 & = & 128
 \end{array}$$

十進數 128

因此，二進數 10000000 即等於十進數 128。

由上面之敘述讀者可看出，為何八位元二進數之各位元，欲稱為

表 1.1 八位元二進數與其對等十進數

十進數	二進數	十進數	二進數
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	.	.
3	00000011	.	.
4	00000100	.	.
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	.	.
9	00001001	.	.
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101	.	.
14	00001110	.	.
15	00001111	.	.
16	00010000	.	.
17	00010001	.	.
.	.	254	11111110
.	.	255	11111111
31	00011111		

b_0, b_1, \dots, b_7 。表 1-1 所示即為十進數 0 至 255 之等效二進數情形。由該表可看出，八位元直接二進數所能表示之最大數目為十進數 255。

習題 1-1：直接二進數 “11111100” 之等效十進值為何？

十進數換成二進數

上面討論的是二進數表示與如何求得其對等十進數。緊接，我們討論如何將一十進數換成其對等直接二進數。將一十進數轉換成其對等二進數之最簡便方法，即以 2 連除之，每次記下餘數，直至商等於零時為止。之後，再將每次餘數依獲得次序排列，愈後得者當高次位，所得即為對等二進數。例如，將十進數 26 轉換成其對等二進數之程序為

$$26 \div 2 = 13 \quad \text{餘 } 0$$

$$13 \div 2 = 6 \quad \text{餘 } 1$$

$$6 \div 2 = 3 \quad \text{餘 } 0$$

$$3 \div 2 = 1 \quad \text{餘 } 1$$

$$1 \div 2 = 0 \quad \text{餘 } 1$$

因此，26 之對等二進數為 11010。

習題 1-2：十進數 245 之等效二進數為何？

習題 1-3：將十進數 39 轉換成二進數，再將所得二進數換回十進數。

二進制算術

二進數之算術規則甚為直截了當。其加法規則為：

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 10$$

留意，1 + 1 之和為 0，進位 1。多位元二進數之相加，如同十進數：由最右一行起，逐次向左相加。除了最右一行外，各行必須計入右一行所產生之進位。例如：

$$\begin{array}{r} 10110 \\ + 01011 \\ \hline 100001 \end{array} = \begin{array}{r} 22 \\ + 11 \\ \hline 33 \end{array}$$

習題 1—4：以二進制作 5 + 16，並驗證其和為 21。

二進數之減法規則為

$$0 - 0 = 0$$

$$0 - 1 = 1 \quad \text{借位 1}$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

例如：

$$\begin{array}{r} 1101 \\ - 1010 \\ \hline 0011 \end{array} \quad \begin{array}{r} 13 \\ - 10 \\ \hline 3 \end{array}$$

習題 1—5：以二進制演算 24 - 8，並證明結果為 16。

二進數之乘法規則亦相當簡短，只有四項。不像十進制有 100 項。

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

下面為一二進制乘算之例子。

$$\begin{array}{r} 1101 \\ \times 1010 \\ \hline 0000 \\ 1101 \\ 0000 \\ 1101 \\ \hline 10000010 \end{array}$$

注意：如同十進制乘算一般，每次部份積必須向左移一位。此外，若乘數位元為 0，則部份積必為 0；否則，若乘數為 1，部份積祇需將被乘數照抄一遍即可。

習題 1—6：以二進制演算 25 × 12，並驗算結果。

二進制的除法同樣相當簡易。就如同十進制一般，除數為 0 時無意義。因此，二進制乘法規則為

$$0 \div 1 = 0$$

$$1 \div 1 = 1$$

例如：

$$\begin{array}{r} 011 \\ 110 \overline{) 10010} \\ \underline{110} \\ 110 \\ \underline{110} \\ 0 \end{array}$$

習題 1—7：以二進制演算 125 ÷ 13，並驗算所得結果。

二、有號整數

由以上之敘述我們了解，八位元之直接二進數能表示“

“00000000”至“11111111”之間的數目，亦即，十進數“0”至“255”。顯然，這有兩個缺點：第一，我們僅能表示正數，而不能表示負數。第二，能表示數目之最大值限於255。底下，我們分別探討這兩個問題。

於二進制，負數有三種表示法：帶號絕對值法(signed magnitude)，1補數法(one's complement)，與2補數法(two's complement)。茲將其分別介紹如下：

1. 帶號絕對值法

於帶號絕對值法，數目之最高次位元為**符號位元**(sign bit)，用以代表正負號。傳統的作法是，“0”表正數，“1”表負數。然後，其餘之位元再用以表示數目之大小——絕對值。譬如，127之對等直接二進數為1111111，於帶號絕對值表示法，

+127 即表示成 01111111

-127 即表示成 11111111

。換言之，帶號絕對值表示即於直接二進數之最左邊再加添一符號位元；若正數，則符號位元添0，若負數，則符號位元添1。

習題1—8：“-5”於帶號絕對值法如何表示？

由以上之敘述顯然可看出，以帶號絕對值法，八位元二進數所能表示之數值，最大為+127，最小為-127。倘若欲表示之數目超出此範圍，則即必須增加數目之位元數。譬如，若用十六位元（兩個位元組），帶號絕對值法所能表示之數目，即由-16384至+16384。當然，欲表示更大的數，就須使用更多之位元。

緊接，我們探討速度效率之問題。試看帶號絕對值表示之加算：

+ 7	表示成	00000111	
- 5	表示成	+ 10000101	
		10001100	= -12

顯然，所得結果並不正確。(+ 7) + (- 5) 之正確結果應為 + 2，而非 - 12。至此，我們了解，表示成帶號絕對值法之數目，無法直接作算術運算。由於計算機除了儲存資訊外，尚必須能對其作運算，因此，帶號絕對值法顯然不適合於計算機應用。這導致了補數法的產生。

2. 1 補數法

於1補數表示法，所有正數均表示成其直接二進形式。譬如，+6即表示成00000110。不過，負數則表示成正數之補數形式——將每位元逐次反相，1變0，0變1。譬如，-6即表示成11111001，此一結果即為將+6之表示，00000110，各位元逐次反相而得。

再看一例子：

+ 2 為 00000010

- 2 為 11111101

注意1補數之求法。將各位元逐次反相之結果，同於以1去減各位元所得之結果。

習題1—9：於1補數法，+127與-127如何表示？

習題1—10：1補數法中，零有幾種表示？

習題1—11：於1補數法，八位元所能表示之最大值與最小值各為何？

下面，我們看看1補數之算術。先看-4加+6，

- 4	11111011	
+ + 6	+ 00000110	
+ 2	(1) 00000001	(1) 表進位

正確結果應為“+2”或“00000010”。再看一例子，

$$\begin{array}{r}
 -3 \qquad 11111100 \\
 + \quad -2 \quad + \quad 11111101 \\
 \hline
 -5 \qquad (1) \quad 11111001
 \end{array}$$

正確結果應為“-5”或“11111010”。

顯然，1補數法雖然能同時表示正數與負數。但其直接加算的結果並不正確。因此，我們勢必再另謀它途。（事實上，只要將最高次位元之進位迴加至運算結果之最低次位元，1補數之結果仍屬正確。）

3. 2補數法

2補數法是真正一般計算機所採用的正負數表示法。於2補數法，正數亦如帶號絕對值法般地表示。但負數則表示成正數之2補數形式。任何二進數之2補數即為其1補數再加1。譬如，

$$\begin{array}{ll}
 +3\text{之帶號絕對值表示為} & 00000011 \\
 \text{其1補數為} & 11111100 \\
 \text{2補數為} & 11111101
 \end{array}$$

所以，在2補數法，+3表示成00000011，而-3表示成11111101。

注意，於三種表示法，數目之最高次位元均為符號位元。若該位元之值為0，則數目為正數；若該位元為1，則數目必為負數。

習題1—12：“+127”之2補數表示為何？

習題1—13：“-128”之2補數表示為何？

習題1—14：零於2補數法表示成什麼？有幾種形式？

接著，我們試看2補數之算術，先看加法，

$$\begin{array}{r}
 3 \qquad 00000011 \\
 + 5 \quad + 00000101 \\
 \hline
 8 \qquad 00001000 \quad = +8
 \end{array}$$

結果正確。再看一減法，

$$\begin{array}{r}
 3 \qquad 00000011 \\
 - 5 \quad + 11111011 \\
 \hline
 -2 \qquad 11111110
 \end{array}$$

注意，減法是以加上減數之2補數達成。讓我們看看運算所得之結果是否正確。

結果11111110之1補數為 00000001

$$\begin{array}{r}
 \text{加1} \quad + \quad 1 \\
 \hline
 00000010 \quad = +2
 \end{array}$$

因此，結果11111110代表-2，答案正確無誤。

再看一例子：+4與-3相加。

$$\begin{array}{r}
 +4 \qquad 00000100 \\
 + (-3) \quad + 11111101 \\
 \hline
 +1 \quad (1) \quad 00000001
 \end{array}$$

注意，若最高次位元之進位捨棄，則結果亦正確。

至此，我們已經看過了2補數之加與減運算，並且證實結果均為正確（若進位捨棄的話）。為了節省篇幅，在此我們不作證明而直接下結論：**於2補數，若捨棄進位，則任何兩數（不論正負）直接相加或相減之結果絕對正確。**

習題1—15：於2補數表示，八位元所能代表之最大數值與最小數值各為若干？

習題1—16：求出20之2補數。將所得結果又取2補數，得到20嗎？

不過，試看下面之例子：

$$\begin{array}{r}
 108 \qquad 01101100 \\
 + 29 \quad + 00011101 \\
 \hline
 137 \qquad 10001001
 \end{array}$$

表 1—2 八位元 2 補數表

+	2 補數	-	2 補數
+127	01111111	-128	10000000
+126	01111110	-127	10000001
+125	01111101	-126	10000010
...		-125	10000011
...		...	
+65	01000001	-65	10111111
+64	01000000	-64	11000000
+63	00111111	-63	11000001
...		...	
+33	00100001	-33	11011111
+32	00100000	-32	11100000
+31	00011111	-31	11100001
...		...	
+17	00010001	-17	11101111
+16	00010000	-16	11110000
+15	00001111	-15	11110001
+14	00001110	-14	11110010
+13	00001101	-13	11110011
+12	00001100	-12	11110100
+11	00001011	-11	11110101
+10	00001010	-10	11110110
+9	00001001	-9	11110111
+8	00001000	-8	11111000
+7	00000111	-7	11111001
+6	00000110	-6	11111010
+5	00000101	-5	11111011
+4	00000100	-4	11111100
+3	00000011	-3	11111101
+2	00000010	-2	11111110
+1	00000001	-1	11111111
+0	00000000		

此時結果便不正確了——兩正數之和却為負！此乃因為運算所得結果超過八位元 2 補數所能表示之最大數值範圍（-128 至 +127）的原故。這種情況稱為**溢位**（overflow）（結果滿了，並且溢出來），是一種錯誤情況。因此，為確保運算結果恒為正確，計算機必須想辦法測得所有的溢位情況，以便加以校正。

表 1.2 所示即為八位元 2 補數所能表示之所有數目的情形。

溢 位

溢位對於計算機之作業極為重要，幾乎每一微處理器均設有一溢位偵測電路，並以一正反器或旗號（能記錄一位元資訊之電路）記錄溢位曾否發生。為了使讀者對之有更深一層之認識，緊接，我們對溢位作更進一步的探討。

根據前述對溢位之定義，我們知曉，2 補數算術於四種情況下可能發生溢位：

1. 兩極大正數相加。
2. 兩極大負數相加。
3. 大負數減大正數。
4. 大正數減大負數。

至於，溢位情況如何測知呢？我們先看一個例子。

$$\begin{array}{r}
 \begin{array}{l} \text{第 6 位元} \\ \text{第 7 位元} \end{array} \begin{array}{l} \longrightarrow \\ \longrightarrow \end{array} \begin{array}{l} \downarrow \\ \downarrow \end{array} \begin{array}{l} 0 \\ 1 \end{array} \begin{array}{l} 1000000 \\ 0000001 \end{array} \\
 \begin{array}{r}
 01000000 \\
 + 01000001 \\
 \hline
 10000001
 \end{array}
 \begin{array}{l}
 64 \\
 + 65 \\
 \hline
 -127
 \end{array}
 \end{array}$$

上面的例子告訴我們，兩個正數（64 與 65）相加，結果却變為負數（-127）。很明顯，問題出在加算過程中，第 6 位元（b₆）之相加產生進位至第 7 位元，改變了符號位元之值（由 0 變 1），致使結果變為負數。但，第 6 位元產生進位至第 7 位元永遠是錯誤的嗎？亦

不盡然。

看下面的例子便知：

$$\begin{array}{r} 11111111 \\ + 11111111 \\ \hline (1) 11111110 \end{array} \quad \begin{array}{r} -1 \\ + -1 \\ \hline -2 \end{array}$$

此時，雖然第 6 位元與第 7 位元同時產生進位，但結果却是正確的。
由此可見，溢位並不等於第 6 位元至第 7 位元之進位。

再看一例子：

$$\begin{array}{r} 11000000 \\ + 10111111 \\ \hline (1) 01111111 \end{array} \quad \begin{array}{r} -64 \\ + -65 \\ \hline -129 \end{array}$$

此時，雖然第 7 位元有進位，但第 6 位元並無進位至第 7 位元，而結果仍是錯誤的（兩負數相加得到正數）。

至此，我們恍然發覺，若第 6 位元與第 7 位元皆無進位，或同時有進位，則加算結果正確。否則，若兩個位元位置中僅有其中一者有進位，則溢位便發生。換言之，溢位即等於第 7 位元（符號位元）之進位輸入與進位輸出，兩者之 *XOR*（互斥）函數。

進位與溢位旗號

上面提過，一般微處理器均設有一溢位旗號（overflow flag），以記錄 2 補數之加減運算是否產生溢位。事實上，除了溢位外，微處理器通常亦具有一進位旗號，以記錄最高次位元是否產生進位輸出。下面，我們列舉幾個實際例子，說明這兩個旗號的動作情形。假設溢位旗號以 *V* 代表，當溢位發生時，該旗號之值為 1（ $V=1$ ），無溢位發生時，溢位旗號之值為 0（ $V=0$ ）。同時假設進位旗號以 *C* 代表，當最高次位元有進位輸出時，進位旗號之值為 1（ $C=1$ ），否則，該旗號之值為 0（ $C=0$ ）。

①正數與正數

$$\begin{array}{r} 00000110 \\ + 00001000 \\ \hline 00001110 \end{array} \quad \begin{array}{r} (+6) \\ (+8) \\ = (+14) \end{array}$$

$V=0, C=0$ ，結果正確。

②正數與正數，有溢位。

$$\begin{array}{r} 01111111 \\ + 00000001 \\ \hline 10000000 \end{array} \quad \begin{array}{r} (+127) \\ (+1) \\ = (-128) \end{array}$$

$V=1, C=0$ ，結果錯誤。

③正數與負數（結果為正）

$$\begin{array}{r} 00000100 \\ + 11111110 \\ \hline (1) 00000010 \end{array} \quad \begin{array}{r} (+4) \\ (-2) \\ = (+2) \end{array}$$

$V=0, C=1$ （捨棄），結果正確。

④正數與負數（結果為負）

$$\begin{array}{r} 00000010 \\ + 11111100 \\ \hline 11111110 \end{array} \quad \begin{array}{r} (+2) \\ (-4) \\ = (-2) \end{array}$$

$V=0, C=0$ ，結果正確。

⑤負數與負數

$$\begin{array}{r} 11111110 \\ + 11111100 \\ \hline (1) 11111010 \end{array} \quad \begin{array}{r} (-2) \\ (-4) \\ = (-6) \end{array}$$

$V=0, C=1$ （捨棄），結果正確。

⑥負數與負數，有溢位

$$\begin{array}{r} 10000001 \\ + 11000010 \\ \hline (1) 01000011 \end{array} \quad \begin{array}{r} (-127) \\ (-62) \\ = +67 \end{array}$$

$V = 1$ ， $C = 1$ ，結果錯誤。

兩負數相加，所得結果超出八位元 2 補數所能表示之下限的情形，有時稱為“**潛位**”(underflow)。

習題 1—17：完成下列加算。記下結果，進位、溢位，以及結果是否正確。

$$\begin{array}{r} 10111111 \\ + 11000001 \\ \hline \end{array} \quad \begin{array}{l} (\quad) \\ (\quad) \\ = (\quad) \end{array}$$

$V = \quad$ ， $C = \quad$ ，() 正確，() 錯誤。

$$\begin{array}{r} 11111010 \\ + 11111001 \\ \hline \end{array} \quad \begin{array}{l} (\quad) \\ (\quad) \\ = (\quad) \end{array}$$

$V = \quad$ ， $C = \quad$ ，() 正確，() 錯誤。

$$\begin{array}{r} 00010000 \\ + 01000000 \\ \hline \end{array} \quad \begin{array}{l} (\quad) \\ (\quad) \\ = (\quad) \end{array}$$

$V = \quad$ ， $C = \quad$ ，() 正確，() 錯誤。

$$\begin{array}{r} 01111110 \\ + 00101010 \\ \hline \end{array} \quad \begin{array}{l} (\quad) \\ (\quad) \\ = (\quad) \end{array}$$

$V = \quad$ ， $C = \quad$ ，() 正確，() 錯誤。

習題 1—18：您能找出一正數與一負數相加而發生溢位的例子嗎？何故？

至此，我們已經解決了有號整數表示之問題，但仍留下最大值問

題。假若我們欲表示更大的數目，則唯一的方法還是增加數目所佔之位元組數。換言之，即以多個位元組代表一個數目。然而，為了有效率地履行算術運算，數目之長度（位元組數）仍以固定者為佳。如此，一旦數目之位元組數選定了，所能表示之最大數值亦固定。

習題 1—19：兩位元組長之 2 補數，所能表示之最大與最小值各為若干？

大小問題

上面曾經說過，八位元 2 補數所能表示之整數，最大為 +127，最小為 -128。顯然，對許多應用而言，這是不夠的。

欲擴大能表示數目之範圍，最簡易之方法就是增加數目之位元組數，使之變成二，三，……或 N 個位元組等之格式。例如，若採取“**雙倍長**”(double-precision)（即十六位元）格式，則所能表示之數目範圍就增至 -32768 ~ +32767。

01111111	11111111	+32767
⋮	⋮	
00000000	00000001	+1
00000000	00000000	0
11111111	11111111	-1
11111111	11111110	-2
⋮	⋮	
10000000	00000000	-32768

然而，這種方法亦有缺點。由於八位元微處理器每次僅能處理八個位元之資料，因此，多位元組之資料必須分批處理，一次一個位元組。這自然遲緩了資料處理之速度。此外，對於小值數目而言，即使能以八位元表示，我們還得須使用十六位元，這亦造成記憶空間之浪費。

緊接我們考慮一點：不管用幾位元作 2 補數表示，其皆為固定的

。若任意最後或中間結果超出已知位元數，則有些位元勢將必須捨棄。
。程式通常僅保留最高次之N位元，而捨棄多餘之低次位元。此一作業稱為**裁截**(truncate)結果。

總之，固定格式之表示可能造成精確度之損失，但這對一般日常之計算與數學運算而言，是可容忍的。不幸的是，在諸如會計等之若干應用，精確度之受損是不容許的。在此些應用，就必須尋求其它方式之表示法，如BCD等。

三、BCD 數

所謂**BCD**(Binary-Coded Decimal)，即**二進寫碼之十進數**。換言之，就是將十進數表示成二進電碼的形式。其主要原理是，將十進數之每一十進數字，分別以一四位元之二進碼代表。如表 1.3 所示，該二進碼即為十進數字之對等直接二進數。由於十進數字僅有10個(0到9)，而四位元二進碼有16種不同組合情況，因此，有6種組合情況空白不用。此六種組合情況在BCD加與減運算形成了有待解決之困擾。

表 1.3 BCD碼

BCD碼	十進數字	BCD碼	十進數字
0000	0	1000	8
0001	1	1001	9
0010	2	1010	不用
0011	3	1011	不用
0100	4	1100	不用
0101	5	1101	不用
0110	6	1110	不用
0111	7	1111	不用

由於每一BCD數字僅需以四位元加以寫碼，因此，一八位元之字組內可同時容納兩位BCD數字。每八位元寫碼兩位BCD數字之BCD碼，稱為**濃縮BCD**(packed BCD)。顯然，濃縮BCD碼能寫碼“00”至“99”等一百個十進數。

0000	0000	0
0000	0001	1
⋮	⋮	
0001	1001	19
0010	0000	20
⋮	⋮	
1001	1001	99

例：將十進數78寫碼成BCD。

解：

7	8
↓	↓
0111	1000

故十進數78之濃縮BCD碼為01111000。

例：濃縮BCD碼“00100110”所代表之十進數為何？

解：

0010	0110
↓	↓
2	6

故BCD碼“00100110”代表十進數26。

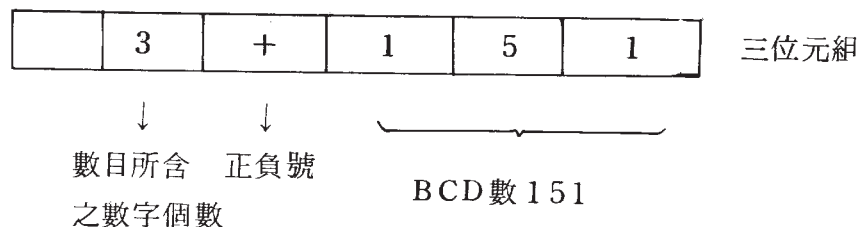
習題1—20：十進數29與94之BCD表示各為何？

習題1—21：“10100000”為有效之BCD碼嗎？何故？

大於99之BCD數如何表示呢？答：以多位元組表示。於多位元組BCD數，除了BCD數之各位數字必須記錄外，數目之符號與數目所含之數字個數，亦均需分別加以記憶。當有小數時，則亦須記憶小數點

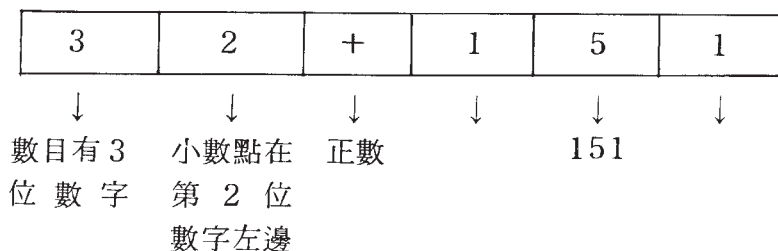
所在之位置。

下面為一多位元組 B C D 整數表示之例子：



該例子以三個位元組表示十進數 +151。其中，最低次之十二位元寫碼數目之三位數字，次左之四位元寫碼數目之正負號（可以 0000 代表正號，0001 代表負號），再次左四位元則記錄數目所含之十進數字個數。最高次四位元空白不用。

下面則為一多位元組 B C D 小數表示之例子：



此例子與上一例子之差別為，增加一表示小數點之位置的欄。此例子所表示之 B C D 數為 +1.51。

B C D 之優點為其能產生人類所慣用之十進結果。而缺點為佔用大量之記憶空間，且其運算速度。

習題 1—22：將“9999”寫碼成 B C D 需要幾個位元？

BCD 算術

知道 B C D 數如何表示後，現在，我們開始介紹 B C D 之加與減

運算。由於 B C D 只是一種電碼，而非數目系統，致其算術運算較為複雜。儘管如此，但是，由於 B C D 為計算機與十進制外界之溝通，提供了甚為便捷之方式。因此，B C D 算術仍經常被採用。

前面說過，所謂 B C D，就是將一十進數之各個十進數字，分別以其對等之四位元直接二進數取代所得。因此，若將 B C D 數目直接相加，應可達十進制之算術。例如，

十進數	B C D 數	
3	0011	→ 3 之 B C D 碼
+	4	→ 4 之 B C D 碼
7	0111	→ 7 之 B C D 碼

又如，

十進數	B C D 數	
20	0010 0000	→ 20 之 B C D 碼
+	45	→ 45 之 B C D 碼
65	0110 0101	→ 65 之 B C D 碼

上述兩例子顯示，若將 B C D 碼直接以二進制之加法規則相加，則所得結果恰為十進數和之 B C D 碼；亦即，我們已達成十進制之運算。但是，事實並不盡然。下面之例子就有問題。

十進數	B C D 數	
5	0101	→ 5 之 B C D 碼
+	9	→ 9 之 B C D 碼
14	1110	→ 無定義之 B C D 碼

5 加 9 在十進數得到 14。因此，B C D 數之相加照理應得到 0001 0100。但事實却不然，和仍為四位元（相當於一十進數字），而且是未經定義的 B C D 碼。

經仔細分析，我們發覺，問題就出在未使用之六個電碼上。雖然我們以四位元之二進碼表示每一十進數字，但却沒將四位元之所有可能組合情形用完，而在後面留下六個未使用之電碼，致和會出現未經

定義之 B C D 碼。對十進算術而言，每數位在達到十後就應進位，但是，於 B C D 碼，每四位元必須滿十六後才會進位，其間有 6 之差距。換言之，在 B C D 碼，1010 至 1111 等六個電碼是沒定義的。因此，為達正確結果，當和落於此範圍內時，我們必須再將之加 6，使其越過此一未經定義範圍，並能達到進位。亦即，B C D 之算術，在當每數位之和大於 9 時，其和必須再加 6，才算正確。

例： 十進數	B C D	
5	0 1 0 1	→ 5 之 B C D 碼
+ 9	+ 1 0 0 1	→ 9 之 B C D 碼
<hr/> 14	1 1 1 0	→ 無效之和
	+ 0 1 1 0	→ 加 6 校正
	<hr/> 0 0 0 1 0 1 0 0	→ 正確結果 14

本能上，由於計算機僅能作直接二進數之算術運算（加法電路之特性）。因此，欲以微電腦達成十進算術，必須先以普通之加或減指令運算，然後再以**十進制調整**（D A A）指令，校正運算所得結果，使其合乎十進數算術之結果。

以上，我們已解決了整數，有號整數，與大整數之表示問題，同時亦介紹了一種十進數之表示法——BCD，最後，我們再看看如何以**定長格式**（fixed-length format）儲存小數。

四、浮點表示

於計算機應用，欲將所處理之數目限於一律整數或一律小數，通常是辦不到的。在許多問題上，數目之大小可能自 10^{-10} 至 10^{+10} 皆有。為了作業方便起見，對於極大與極小的數，最好亦能以定長格式，將其表示成**浮點**（floating-point）形式。

所謂浮點形式，亦即乘幕形式。例如，

10540000	即表示成	0.1054×10^8
0.0000536	即表示成	0.536×10^{-4}

10540000 之浮點表示的第一部份 0.1054，稱為**假數**（mantissa）或小數（fraction）。而第二部份，8，則稱為**指數**（exponent），此處基底為 10。於二進制表示，基底為 2。

為了免於浪費記憶空間，浮點數通常須經**正規化**（normalized）。經正規化的浮點數，其假數的絕對值必大於 0.1 而小於 1。以數學符號表示即

$$0.1 \leq M \leq 1 \text{ 或 } 10^{-1} \leq M < 10^0$$

其中，M 代表假數之絕對值。

同樣地，於二進制

$$2^{-1} \leq M < 2^0 \text{ (或 } 0.5 \leq M < 1 \text{)}$$

例如，十進數 0.00173 與二進數 111.01 分別各有下列幾種表示法：

0.000173×10^1	11.101×2^1
0.00173×10^0	111.01×2^0
0.0173×10^{-1}	1110.1×2^{-1}
0.173×10^{-2}	$11101. \times 2^{-2}$

但經正規化後，它們皆僅有一種表示，那就是

$$0.173 \times 10^{-2}, \quad 0.11101 \times 2^3$$

換言之，經正規化之浮點數，假數部份之小數點右側無前導零。

了解浮點數之意義與格式後，我們緊接看其在記憶器內如何表示。浮點數於記憶器內通常表示成下面之格式。



圖 1-5 典型浮點數表示

此一例子總共使用四個位元組。圖形左邊之位元組用以代表指數，右邊三個位元組表示假數。指數與假數兩者皆表示成 2 補數形式。因此，其最高位元均代表符號。顯然，此例中，指數之範圍乃由 -128 至

+ 127。而假數之範圍，絕對值必小於 2^{23} 。

以上所舉僅是浮點表示之其中一例。於其它情況，長度可能更長或更短。不過，就精確度、數值大小、記憶空間利用率，與算術運算之效率各方面言，四位元組之長度算是最尋常的。

至此，我們已探討過程式與數值數據之計算機表示，最後，我們看看非數值數據於計算機內部如何表示。

1-4-3 文數字資料

文數字資料，亦即英文字母、標點符號、與特殊符號等，之表示最直截了當：**所有文數字皆寫碼成一八位元之電碼**。計算機界常用的僅有兩種寫碼方式：**ASCII 碼與EBCDIC 碼**。ASCII 乃 American Standard Code for Information Interchange（美國資訊交流標準碼）之縮寫。時下之微處理器均使用此種電碼。EBCDIC 則為 Extended BCD Interchange Code（擴展BCD交換碼）之縮寫，此種電碼主要為美國IBM公司所採用。由於微電腦界均使用ASCII碼，因此，本小節主要將介紹此一電碼。

表 1.4 ASCII 轉換表

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	-	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	,	7	G	W	g	w
8	1000	BS	CAN	(8	H	X	h	x
9	1001	HT	EM)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	=	K	[k	{
C	1100	FF	FS	,	<	L	\	l	
D	1101	CR	GS	-	=	M]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

於日常資訊交流上，常用之文數字有：26 個大寫英文字母，26 個小寫英文字母，10 個十進數字，加上 20 個左右之特殊符號。此文數字可輕易地以 7 位元加以寫碼（因為，七位元有 128 種不同組合情況，每一組合情況代表一文數字，足敷使用）。因此，所有文數字均寫碼成七位元。不過，為了提高資訊傳遞之可靠性，文數字碼通常又外加一第八位元，該位元稱為**極性位元**（parity bit）。

極性是一種證明資訊於傳輸過程中是否發生錯誤之技巧。於送出文數字碼前，資料之傳輸端可計數每一七位元文數字碼所含“1”之個數。若總數為奇數，則傳輸端可將極性位元置定為 1，並將之與文數字碼一起送出；若總數為偶數，則極性位元送出 0。藉此，資訊接收端於收到資訊時，即可藉著計數每一項文數字碼所含“1”之總個數，並核對極性位元值是否吻合，而測知資訊於傳遞過程中是否發生 0 變 1，1 變 0 之錯誤。當然，此種極性技巧僅能測知單數個錯誤。

以上所使用之極性技巧稱為**偶極性**（even parity）。您亦可使用奇極性（odd parity）。所謂**奇極性**，即文數字碼所含之“1”位元的總個數（包括極性位元本身）恒為奇數。於偶極性，每一文數字碼（八位元）所含之“1”位元的總個數，恒為偶數。

例如，若採用偶極性，而某一文數字之七位元電碼為 0010011，則此時極性位元必須為 1。因為，這樣才能使八位元文數字碼之“1”位元的總個數為偶數。若極性位元置於最前（左），則完整之文數字碼即為 10010011。

表 1.4 所示即為七位元之 ASCII 文數字碼。表中央為所有經寫碼之文數字，最上端為七位元電碼之最高次三位元（第一列為八進制，第二列為二進制），最左邊則為文數字電碼之低次四位元（最左一行為十六進制，左第二行為二進制）。該表之使用法甚為簡易：祇要找到欲查取之文數字在表中的位置（列數與行數），即可立即寫出該文數字之七位元 ASCII 碼。例如，大寫英文字母 A 位於第五行第二列，其上端為 100，其左邊 0001，因之，A 之七位元 ASCII 碼為

1000001。

在實際應用上，設計者可選擇採用極性位元而將之加於最左邊，或可選擇不採極性位元而在七位元電碼之最左邊加一“0”位元。

習題 1—23：寫出十進數字 0 至 9 之八位元 ASCII 碼，採用偶極性。

習題 1—24：以英文字母 A 至 F，重作上題。

習題 1—25：以無極性之 ASCII 碼（最左位元加 0），寫出文字串“JOHN, MALE, 27”儲存在記憶器中之二進內含。

1-5 外部資訊表示

由於對象不同，資訊於計算機內部與外部之表示亦有所差異。由於計算機之內部電路僅能記憶與處理二進資訊，因此，於計算機內部，所有資訊（包括程式與數據）皆表示成二進制形式——一串的 0 與 1。但是，由於人類慣於使用十進制與符號表示，因此，此種二進資訊對處於計算機外界之人而言，並不適合。為了便於駕馭起見，計算機之用者經常採取不同之資訊表示法。此節，我們所要探討的即為此些外部資訊表示法。

1-5-1 組合語言

計算機可用以解題。人類欲令計算機做一件事，必須先設計一程式，告訴計算機如何逐步地完成此件工作。換言之，程式即包括一系列之指令（instruction），每一指令即相當於計算機能立即執行之一運算（動作）。計算機只要執行完程式之指令，即可完成人類所期許之工作。

前節曾經說過，程式之每一指令在計算機內部必須以機器碼（二進制）表示。因為，計算機僅能看懂與執行機器碼。然而，就人而言

，此種機器碼並不自然。人類慣於使用十進制與抽象符號（如 X，Y 等）代表一切。

為了便利人類對計算機之應用，於計算機外部，程式通常可以較接近人類所使用之語言的符號表示，此些語言即稱為**程式語言**（programming language）。依其接近機器語言之程度，程式語言可分為數種層次。其中“最低層次”（最接近或類似機器語言）的一種，即稱為**組合語言**（assembly language）。

於組合語言，每一指令皆以一**助憶符號**（mnemonic）代表。譬如，

加法運算就以	ADD 代表
減法運算就以	SUB（SUBtract）代表
資料移轉運算就以	LD（LoaD）代表
跳越運算就以	JP（JumP）代表

等等。因此，加 5 的運算指令在組合語言就可寫成

ADD A, 5

其中，ADD 代表加法運算之命令。5 代表欲加之數目。A 則代表加算另一運算元之所在。於計算機，一般算術運算所需之另一運算元，均來自一稱為**累加器**（Accumulator，通常以 A 代表）之固定地方。同時，運算後之結果亦存回累加器。上式中之 A 即代表累加器。

雖然組合語言便於人類駕馭，可惜機器却“看不懂”。因此，以組合語言寫成之程式，在能被執行前必須先經翻譯。此點留待第五章再詳加探討。

1-5-2 八進制與十六進制

1. 二進制

前面說過，所有資料在計算機內部皆表示成二進形式——一串的 0 與 1，並且皆八位元一組一組地存在一起。有時，此種資訊會透過

某種輸入／輸出媒體，直接呈現於外界。譬如，某些微電腦之操作面板上，即裝有整排之發光二極體（LED），以顯示計算機內部某一暫存器或某一記憶位置之內含。此時，每一發光二極體負責顯示一位元，若發光二極體亮，則表計算機內部所對應之位元為1。若發光二極體不亮，則表計算機內部之對應位元為0。雖然有些輸入／輸出情況需要直接使用二進制，但對人類而言，這總不比八進制、十進制、或十六進制來得方便。

2. 八進制與十六進制

為了易讀與易寫起見，二進資訊在計算機外部經常縮寫成八進制（octal）或十六進制（hexadecimal）。八進制與十六進制分別將三位元與四位元寫碼成一個符號。如表 1.5 所示，於八進制，三位元之每一種組合分別以 0 至 7 的一個數目代表。於十六進制，四位元之每一種組合情況則分別以十進數字 0 至 9，以及英文大寫字母 A 至 F 表示。十六進寫碼之情形如表 1.6 之第 2 行與第 3 行所示。

表 1.5 八進制寫碼

二 進 碼	八 進 碼
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

八進數與二進數之互換相當容易。二進數轉換成八進數，只要由右至左每三位元一組，將之分別代換成其對等八進符號即可。例如，二進數“00100100”換成八進數

表 1.6 二進，八進，與十六進寫碼

十 進 碼	二 進 碼	十 六 進 碼	八 進 碼
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

00 100 100
 ↓ ↓ ↓
 0 4 4

即為“044”。又如，“11011110”寫成八進制

11 011 110
 ↓ ↓ ↓
 3 3 6

即為“336”。

反之，八進數欲換成二進數，亦只要將每位數字分別代換成其對等之三位元二進數即可。例如，八進數“615”換成二進數

6	1	5
↓	↓	↓
110	001	101

即為“110001101”。

十六進數與二進數之互換原理完全相同，唯一的差別是，此時是四個位元一組，而非三個位元一組。例如，二進數“10011100”換成十六進制

1001	1100
↓	↓
9	C

即為“9C”。又“1010100110”換成十六進制

10	1010	0110
↓	↓	↓
2	A	6

即為“2A6”。

反之，十六進數“45F”換成二進數

4	5	F
↓	↓	↓
0100	0101	1111

即為“10001011111”。

習題 1—26：二進數“10101010”之十六進表示為何？

習題 1—27：反之，十六進數“FA”之二進等效為何？

習題 1—28：二進數“01000001”之八進寫法為何？

十六進制之優點為，每八位元之資料正巧可寫碼成兩位十六進數

字，因此，極易於閱讀、記憶、或輸入計算機。目前有許多微電腦系統均採用此一系統與外界取得溝通，此種系統具有一十六進之鍵盤，容許用者將程式與數據以十六進形式輸入計算機內部。同時亦具有一十六進數字顯示器，能將微電腦內部之資訊，以十六進形式顯示於用者之眼前。

3. 符號表示

符號表示指將資訊實際表示成符號形式之外部表示。譬如，十進制，或甚至是數學之抽象符號 X 或 Y 等。對人而言，此種表示是最理想的了。不過，除了一些具有像電視螢幕顯示或印字機等較老練之輸入／輸出設備的大型計算機系統外，在一般的小型計算機系統上，此種表示法均不可得。

4. 摘要

對身為人類之計算機用者而言，符號表示是最自然與習慣的了。不過，此種表示法經常不可得，尤其在微電腦系統，因為，使用此種表示法，系統必須具有諸如文數字鍵盤、印字機、與 CRT 顯示器等昂貴之輸入／輸出設備。退而求其次的方法，是使用八進制或十六進制表示。目前多數微電腦所採用者，即為十六進表示。而使用於計算機內部之二進制表示，除了極少數硬體或軟體偵錯 (debugging) 時必用外，一般人類是甚少使用的。

知道資訊於計算機內部與外部如何表示後，緊接，我們即可開始研究計算機之處理器本身，以及其如何處理資訊。

額 外 習 題

習題 1—29：比起其它方法，以 2 補數法表示有號數有何優點？

習題 1—30：十進數“1024”於直接二進數，帶號絕對值二進數，與 2 補數各如何表示？

習題 1—31：溢位旗號是什麼？程式設計者於每次加或減運算後必須加以測試嗎？

習題 1—32：求出下列各數之 2 補數：“+16”，“+17”，“+18”，“-16”，“-17”，“-18”。

習題 1—33：假設文字串“MESSAGE”以無極性之 ASCII 碼儲存於記憶器內，試寫出此些記憶內含之十六進形式。

習 題 解 答

第 一 章

1. 252。
2. 11110101。
3. $39 = 100111$ ， $100111 = 39$ 。
4. $101 + 10000 = 10101$ 。 $10101_2 = 21_{10}$ 。
5. $11000 - 01000 = 10000$ 。 $10000_2 = 16_{10}$ 。
6. 100101100。
7. 商 = 1001，餘數 1000。
8. 10000101。
9. $+127 = 01111111$ 。 $-127 = 10000000$ 。
10. 兩種。 $+0 = 00000000$ 。 $-0 = 11111111$ 。
11. 最大： $+127$ ，最小： -127 。
12. $+127 = 01111111$ 。
13. $-128 = 10000000$ 。

14. 00000000。只有一種。

15. 最大： $+127$ 。最小： -128 。

16. 11101100。

17. ① 10000000。V = 0，C = 1，正確。

② 11110011。V = 0，C = 1，正確。

③ 01010000。V = 0，C = 0，正確。

④ 10101000。V = 1，C = 0，錯誤。

18. 不能。因兩數和之絕對值必小於原來任一數之絕對值，故不可能“越界”而產生溢位。

19. 最大： $+32767$ 。最小： -32768 。

20. 29：00101001。94：10010100。

21. 不，因高次四位元無定義。

22. 十八。

23. 0：00110000，1：10110001，……………
8：10111000，9：00111001。

24. A → 01000001，B → 01000010

C → 11000011，D → 01000100

E → 11000101，F → 11000110

25. 01001010

01001000

01001111

01001110

00100111

01001101

01000001

01001100

01000101

00100111

00110010

00110111

26. AA

27. 11111010

28. 101

29. 數目可直接作算術運算，不理進位，結果不需再調整。

30. 直接二進數： 10000000000

帶號絕對值：010000000000

2 補數： 110000000000

31. 溢位旗號即為一記錄前一加減運算是否產生溢位情況之位元。一般微處理器會自動測試。

32. 00010000 (+16)，11110000 (-16)。

33. 4D 45 53 53 41 47 45

第 2 章

Z80 硬體結構

就寫最簡單之程式而言，程式設計者並不須了解太多有關中央處理器之細節。但是，若欲講求程式設計之效率，則此種了解乃是必須的。本章之目的乃在提示了解 Z 80 系統作業所需之硬體概念。一完整之微電腦系統除了應有微處理器（此地為 Z 80）外，尚須包括其它組件。本章將先介紹 Z 80 微處理器之內部構造與動作原理，其它的（主要為輸入／輸出）則留待後面幾章再加詳述。

本章之內容包括 Z 80 微電腦系統之基本結構，Z 80 微處理器之內部構造（特別是各種暫存器），程式之執行，以及循序技巧（sequencing mechanism）等。

2-1 Z80微電腦系統結構

圖 2-1 所示即為一標準微電腦系統之結構。圖中之左邊為**微處理器單元**（microprocessor unit，簡記為**MPU**）。微處理器即將計算機之中央處理單元做在一塊矽晶片上，其內部包括一算術／邏輯單元（ALU），幾些暫存器，以及一控制整個系統作業之控制單元（CU）。微處理器乃整部微電腦之心臟，其主宰與控制了微電腦系統內之一切作業。本章將詳細探討其動作情形。

微處理器產生三種**巴士**（bus）：圖之上方為一八位元之雙向**資料巴士**（data bus），下方為一十六位元之單向**位址巴士**（address bus），以及一**控制巴士**（control bus）。所謂巴士，即一束用以傳遞電信號之電線。下面，我們分別介紹此三種巴士之功能。

資料巴士負責輸送來回傳遞於系統各組件間之資料。典型上，其

負責微處理器與記憶器間或微處理器與輸入／輸出晶片間之資訊傳輸（輸入／輸出晶片乃負責微處理器與外部設備間溝通的組件）。

位址巴士負責傳送微處理器所產生之位址資訊。該位址選取記憶器之某一記憶位置，或輸入／輸出晶片內之某一暫存器。其指明了資料巴士上所傳送之資料的來源地或目的地。

控制巴士則傳送系統作業所需之各種同步信號，其包括數個控制信號。這些控制信號在後面會有詳細介紹。

了解各種巴士之用途後，我們即可將所有組件連接在一起，以獲致一完整系統。

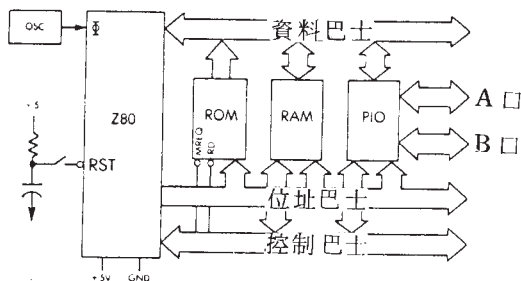


圖 2-1 標準 Z 80 微電腦系統

每一微處理器都需要一明確之基準時序，該時序由一**晶體與時序電路**產生。在較早期之微處理器，時序產生電路都做在微處理器之外，並且自成一塊晶片。目前之微處理器，時序產生電路經常做於微處理器晶片內。不過，由於體積龐大，因此，晶體本身仍置於晶片之外。於圖 2-1，晶體與時序示於最左邊。

現在，我們依圖上由左至右之次序，開始介紹系統之其它組件。

唯讀記憶器（Read-Only Memory，以 **ROM** 表示）乃是一種資料僅能讀出而不能寫入之半導體記憶器。其主要用以儲存不變之程式或數據。唯讀記憶器之優點為，其記憶內含並不隨電源之消失而消失

，因此，每次電源一打開時，程式或數據不必再重新存入（load）。在一般微電腦系統，唯讀記憶器內經常儲存一控制整個系統作業的**監督程式**（monitor program）。該程式使整個系統能起始（initialization），並且使用者能輸入程式，執行程式，以及檢驗結果等。於諸如程序控制等特殊用途之微電腦系統，由於微電腦被指定專作某種用途，履行固定之任務，因此，其程式幾乎都存於 ROM 內。

讀寫記憶器（Random-Access Memory，簡記為 **RAM**）乃是一種資料既可讀出又可寫入之半導體記憶器。由於電源關掉後，此種記憶器之記憶內含會隨之消失，因之，每次電源一打開後，其內部之程式或數據必須重新存入，方才堪用。讀寫記憶器通常用以儲存運算過程之輸入資料或中間結果。於程式設計過程，其亦可用以儲存正在測試之程式以及有關數據。

最後，系統至少包括一個或一個以上之**界面晶片**（interface chip），以使其能與外界溝通。界面晶片使得微處理器能與諸如鍵盤或顯示器等之輸入／輸出設備（又稱週邊設備，peripherals，或外部設備）互相溝通。如圖 2-1 所示，最常用之界面晶片（或稱界面電路）為**並行輸入／輸出**（parallel input/output，簡記為 **PIO**）晶片。如系統之其它晶片一樣，PIO 亦連接至系統之三種巴士。PIO 通常提供了兩個八位元之輸入／輸出口，以便微處理器能與外部設備溝通。有關輸入／輸出，本書後面幾章會再詳細說明。

系統之所有晶片皆必須連接至三種巴士，包括控制巴士。不過，為了簡化圖形起見，於 2-1 圖中，控制巴士與各晶片間之詳細連接情形並沒畫出。

以上所介紹之各種功能單元，並不一定都要做在一塊晶片上。事實上，我們亦可使用諸如含有 PIO 以及有限 RAM 或 ROM 之**組合晶片**（combination chips）。

就構成一實際系統而言，事實上還必須增加一些其它組件。譬如，巴士通常需要**緩衝**（buffer）。同時，RAM 晶片亦需使用**解碼電**

路(decoding logic)。此外，有些信號亦需以**推動器(driver)**加以放大。此些輔助電路雖非主角，但卻也不能沒有。

2-2 一般微處理器之結構與動作原理

目前市面上大多數微處理器幾乎都具有相同之結構。本節將描述此一“標準”結構。如圖 2-2 所示，此一標準結構包括控制單元，算術／邏輯單元 (ALU)，累加器，移位器，旗號暫存器，資料暫存器，與位址暫存器等幾大部份。

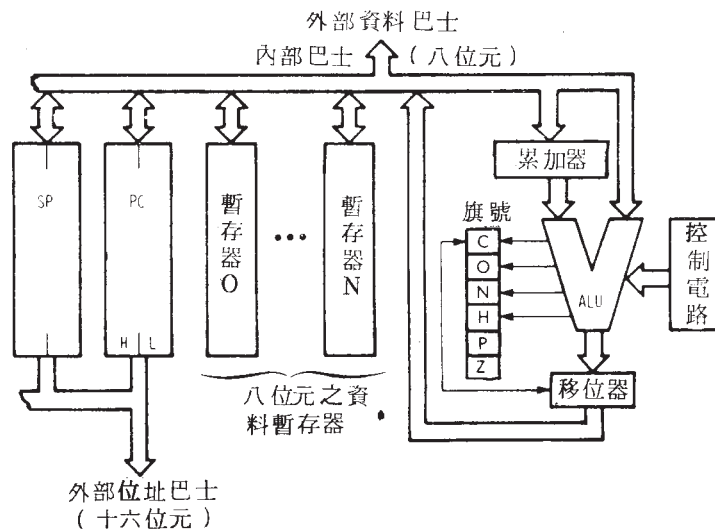


圖 2-2 八位元微處理器之標準結構

控制單元調諧了整個系統之作業，其角色在隨後之討論中會逐漸顯明。

算術／邏輯單元履行算術與邏輯運算。該單元之其中一輸入設置了一累加器 (accumulator)。**累加器**經常兼具了指令之其中一運算元的來源，以及最終結果之目的地址兩種角色。除了算術與邏輯運算外

，算術／邏輯單元亦需具備移位 (shift) 與旋轉 (rotate) 之能力。移位即將某一位元組之內含向左或右移動一個位元位置。這在指令集一章會詳加介紹。

如圖 2-2 所示，移位器 (shifter) 可置於算術 / 邏輯單元之輸出，亦可置於累加器之輸入。

算術／邏輯單元之左邊為一**旗號或狀態暫存器**（flags or status register）。該暫存器之每一位元即為一反映微處理器運算後之狀態的旗號。此些旗號之內含受指令執行之影響而改變，並可以某些特殊指令加以測試。程式然後能根據測試之不同結果，採取不同之對策。Z 80 微處理器之旗號位元的角色，將於次一節中討論。

2-2-1 暫存器

下面，我們看微處理器內兩種比較重要之暫存器。

一、一般用途暫存器

一般之微處理器均設有一個或一個以上之一般用途暫存器（general-purpose registers）。此些暫存器等於是一種快速之少量記憶器，使微處理器能以更快速度處理資料。一般用途暫存器經常用以儲存指令運算之**中間結果**，或被設為**迴路計數器**（loop counter）。由於晶片空間與指令運算碼內用以選取一般用途暫存器之位元數有限，因此，一般用途暫存器的數量通常不致太多。

於八位元微處理器，一般用途暫存器之長度均為八位元。如圖 2-2 所示，這些暫存器皆掛至八位元之資料巴士上，並且各有一獨特名稱代表（如 R0, R1, ……等等）。

二、位址暫存器

除了一般用途暫存器外，微處理器內通常還會含一些位址暫存器（address register）。顧名思義，位址暫存器就是用以儲存記憶位

址的暫存器。此些暫存器有時又稱**資料計數器** (data counter) 或**指示器** (pointer)。

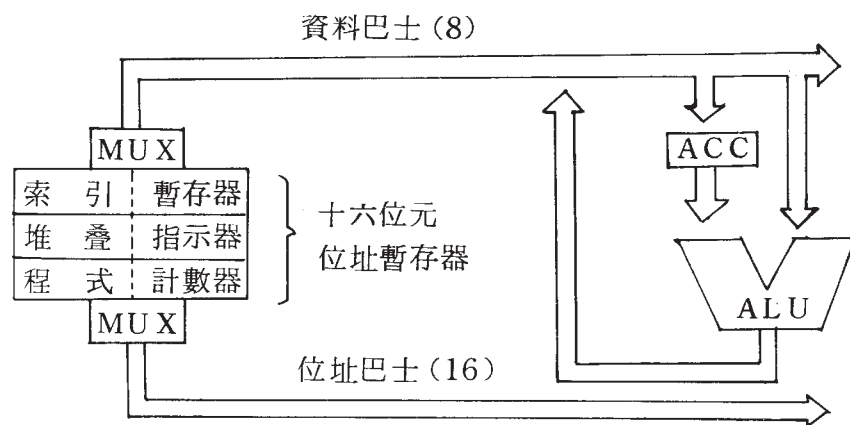


圖 2-3 位址暫存器產生位址巴士

如圖 2 - 3 所示，於當前八位元之微處理器，位址暫存器均為十六位元（因記憶位址長十六位元）。此些暫存器產生位址巴士輸出，由微處理器送至記憶器或輸入／輸出界面電路之位址，即來自於此。

資訊欲存入位址暫存器僅有一種方式，那就是透過資料巴士。由於資料巴士僅有八位元寬，因此，存入此些暫存器須分兩次進行。職是之故，一般之位址暫存器都會平分成高次位元組（H）與低次位元組（L）兩部份。

微處理器內之位址暫存器通常可分三種，茲將其分別介紹如下：

1. 程式計數器 (PC)

任何計算機之中央處理器皆必須具備一**程式計數器** (program counter，簡記為 PC)。程式計數器用以儲存計算機**次一欲執行之指令的位址**。就程式執行而言，程式計數器之設置乃是必需而且基本的。平常，程式之指令皆依其欲被執行之次序，一一循序儲存於記憶器之緊接位置。在能執行之前，指令必須先被取入微處理器。為了正

確拿取 (fetch) 所欲執行之指令，微處理器必須隨時記住所欲拿取之指令的位址，方不致於拿錯了指令。程式計數器即應此而生。其永遠存放微處理器正欲拿取之指令所在之記憶位置的位址。在拿取指令時，微處理器將程式計數器之內含置於位址巴士上，送給記憶器。記憶器然後讀取該位址所選取之記憶位置的內含，置於資料巴士，送給微處理器。微處理器然後將此讀取之指令置於**指令暫存器** (Instruction Register，簡記為 IR)，並且開始解碼與執行。

於少數微處理器，諸如雙晶片之 F 8，微處理器晶片內並無程式計數器。這並不說此種系統並不使用程式計數器，而是為效率原故，程式計數器直接做於記憶器內。

2. 堆疊指示器 (SP)

堆疊器 (stack) 至此尚未介紹過，次一小節將立即加以介紹。於大多數能力較強之一般用途微處理器，堆疊器皆以“軟體” (software) 形式製作於記憶器內。為了追蹤 (keep track of) 此種記憶堆疊器之頂端位置，微處理器內必須設置一隨時指至該頂端位置之**堆疊指示器** (stack pointer，簡記為 SP)。堆疊指示器長十六位元，用以儲存記憶堆疊器之頂端位置的記憶位址。堆疊器是插斷 (interrupt) 與副程式 (subroutine) 處理所不可或缺的。

3. 索引暫存器 (IX)

索引 (indexing) 是一種並非任何微處理器都有之記憶位置選取技巧。透過索引技巧，程式可以單一指令存取一系列之資料項目。於索引選取，記憶位置之真正位址得自一基底位址與一位移兩者之和。索引暫存器 (index register) 即用以儲存欲加至基底位址之位移（如 6502），或反之，儲存欲加至位移之基底位址（如 Z 80）。微處理器之各種位置選取技巧，將於下一章中討論。

2-2-2 堆疊器

堆疊 (stack) 是一種後進先出 (Last-In First-Out, 簡記為 LIFO) 之資料結構。而**堆疊器** (stack) 則為一組用以儲存此種資料結構之暫存器或記憶位置。堆疊結構之主要特性為, 其是一種與時間之先後次序有關的 (chronological) 結構。如圖 2-4 所示, 堆疊器就像僅有一個向上開口的容器, 進出堆疊器的資料皆必須經由此同一開口。因此, 最先被存入堆疊器之資料, 一定位於堆疊之最底端, 而最後被存入者, 一定位於**堆疊之最頂端** (Top Of Stack, 以 **TOS** 代表)。由於資料僅能由同一端出入, 因此, 最先被取出之資料, 一定為最後被存入者 (此即為後進先出之名的由來), 因為, 其位於最上端。

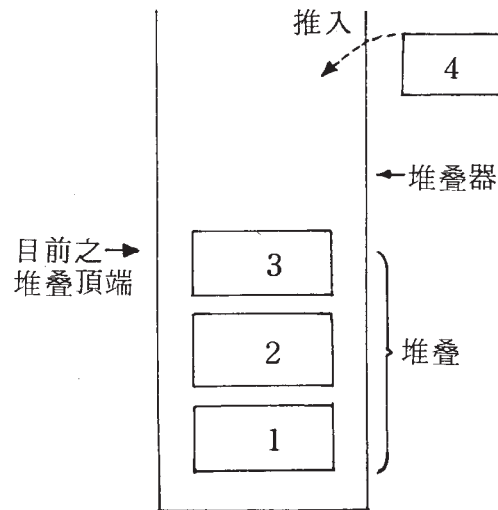


圖 2-4 堆疊器與堆疊

將資料存入堆疊器之動作稱為**推入** (push), 而將資料取出堆疊器之動作則稱為**拉取**或**取出** (pull or pop)。推入堆疊器之資料來自微處理器內之某一暫存器, 而取出堆疊器之資料亦存至微處理器內之一暫存器。此兩種作業所涉及之暫存器通常為累加器。圖 2-5 所示即

為兩種堆疊作業之情形。

堆疊器主要用於**副程式**與**插斷**之處理, 以及**暫時資料**之儲存。此些用法, 將在爾後討論此些主題時, 再分別予以介紹。

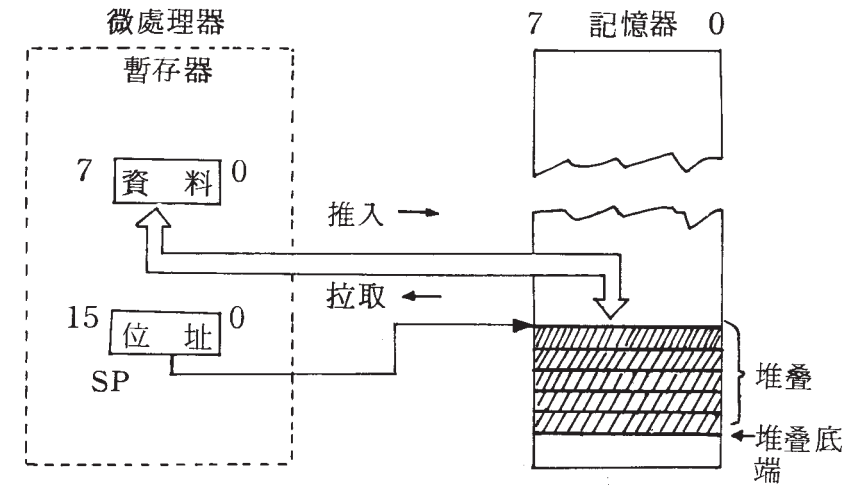


圖 2-5 兩種堆疊作業

堆疊器之製作有兩種方式：

1. 堆疊器可為微處理器內, 某一固定數目之暫存器。此種堆疊器稱為**中央處理器堆疊器** (CPU stack), 或**硬體堆疊器** (hardware stack)。硬體堆疊器之優點為速度快, 缺點為暫存器之數目有限。
2. 大多數一般用途之微處理器皆採取另一種方式——將堆疊器設置於某一段記憶區域內。此種堆疊器稱為**記憶堆疊器** (memory stack) 或**軟體堆疊器** (software stack)。軟體堆疊器之優點為, 其容量不受限制, 幾乎可以“無限量”地增加。但缺點為, 微處理器內必須設置一儲存堆疊頂端位址之堆疊指示器。此外, 其存取速度也較慢。Z 80 即採用記憶堆疊器之方法。

2-2-3 指令執行週期

下面，我們討論微處理器之一很重要的作業——執行程式指令。看 2-6 圖，微處理單元位於左邊，記憶單元位於右邊。記憶單元可為 ROM 或 RAM，或其它任何包含記憶器之晶片。該單元用以儲存程式指令與數據。本小節，我們以微處理器自記憶器拿取一指令之作業，來說明程式計數器之角色。

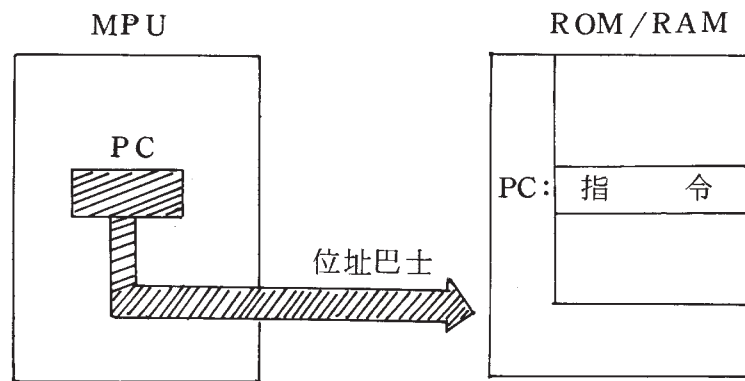


圖 2-6 由記憶器拿取一指令之第一步驟

假若程式計數器所含為一有效位址，且為微處理器次一欲拿取指令之位址。則任何微處理器都將以下列三個步驟（或稱週期或階段）執行每一指令：

1. 拿取次一指令。
2. 解碼指令。
3. 執行指令。

拿 取

現在，我們開始尋繹此一系統。首先，程式計數器之內含置於位址巴士，並送至記憶器。同時，必要時，系統之控制巴士亦產生一讀

取信號。一收到讀取信號，記憶器即假藉內部解碼器，解碼已收到之位址，並選取位址所指明之位置。幾百 ns (10^{-9} 秒) 後，記憶器將被選取位置所儲存之資料置於資料巴士。該八位元之字組即為我們所欲拿取之指令。

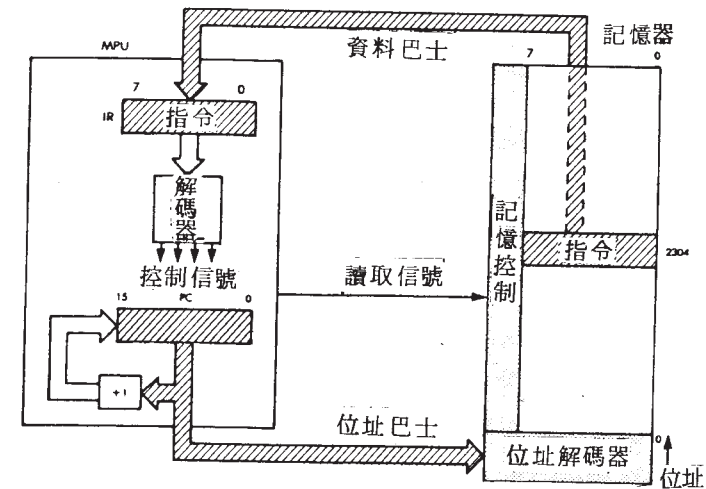


圖 2-7 指令之拿取作業（自動循序）

然後，微處理器自資料巴士讀取指令，並將之置於指令暫存器（Instruction Register，簡記為 IR）。程式計數器之內含然後自動加 1，以指令至次一緊接指令。至此，指令之拿取週期結束。圖 2-7 所示即為此一指令拿取之情形。

解碼與執行

指令一旦取入指令暫存器後，微處理器之控制單元即會將之解碼，並產生執行此一指令所需之一系列內部與外部信號。職是之故，執

行階段之前會有一小段因解碼所造成之延遲。此一時間的長短則視指令之不同而定。某些指令可以完全在微處理器內執行。其它指令則必需再由記憶器讀取數據，或將結果存回記憶器。此即為微處理器各指令之執行時間長短不一之原因所在。由於每一微處理器所使用之時序頻率有所不同，因此，每一指令所需之執行時間，通常以時序週期數表示，而不以秒數表示。

2-2-4 拿取下一指令

前一段，我們已說明了使用程式計數器，自記憶器拿取指令的情形。由於指令**依序**(in sequence)自記憶器被取出與執行，因此，必須建立一套自動循序拿取指令之方式。如圖 2-7 所示，此一任務以一連接至程式計數器之加一電路(incrementer)達成。每當程式計數器之內含被置於位址巴士，其內含就自動被加 1，並存回原處。例如，若程式計數器含 0，則於 0 被置於位址巴士後，程式計數器之值即自動被加 1，並存回程式計數器。如此，微處理器下一次再拿取指令時，即會拿到位址“1”之記憶位址上的指令，而非再為位址“0”位置上的指令。因此，我們就建立了**自動循序拿取指令之技巧**(automatic mechanism for sequencing instructions)。

必須強調的一點是，上述之說明是經過簡化的。實際上，有些指令可能長兩個位元組或三個位元組，以致微處理器必須以同樣方式連續自記憶器讀取幾個位元組。雖然如此，但方法是完全相同的。程式計數器不僅被用於拿取連續之指令，同時亦被用以拿取同一指令之連續位元組。程式計數器，與其加一電路，共同提供了自動指至緊接記憶位置之技巧。

2-2-5 指令之執行

緊接，我們以 2-8 圖之結構，舉例說明指令在微處理器內執行之情形。此處所舉之典型指令為

$$R0 = R0 + R1$$

。意即，“將 R0 與 R1 兩暫存器之內含相加，結果存回 R0 暫存器”。於此一運算，R0 暫存器之內含先被讀取，並經由單一巴士送至 ALU 之左輸入，存於緩衝器。然後，R1 被選取，其內含被讀入資料巴士，並送至 ALU 之右輸入。此一系列分別如圖 2-9 與圖 2-10 所示。至此，ALU 可開始作加。如圖 2-11 所示，加法運算之結果呈現於 ALU 之輸出端，並被送回 R0 暫存器存起。實際上，這表示 R0 暫存器之輸入鎖住致能，使資料能夠寫入。指令執行至此全部完成。加算之結果存於 R0 暫存器。必須注意的是，R1 暫存器之內含並未因運算而改變。這是一般的通律：讀取動作並不改變暫存器或讀寫記憶器原先之內含值。

算術／邏輯單元左輸入之緩衝暫存器（亦即累加器）乃是必須的，因為，其必須記憶 R0 暫存器之內含，使單一巴士能再用於 R1 暫存器內含之傳輸。

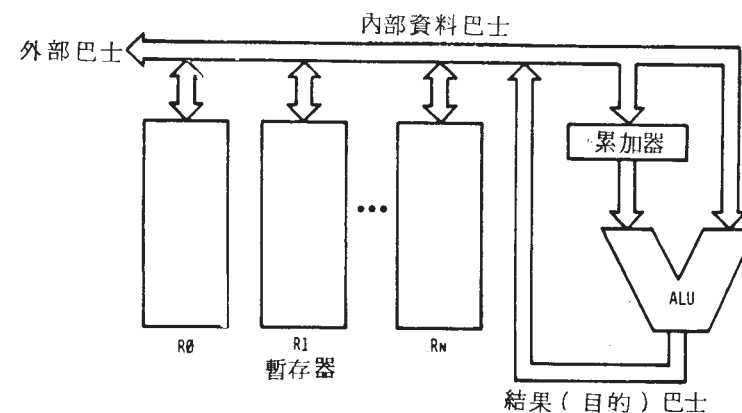


圖 2-8 單一巴士結構之微處理器

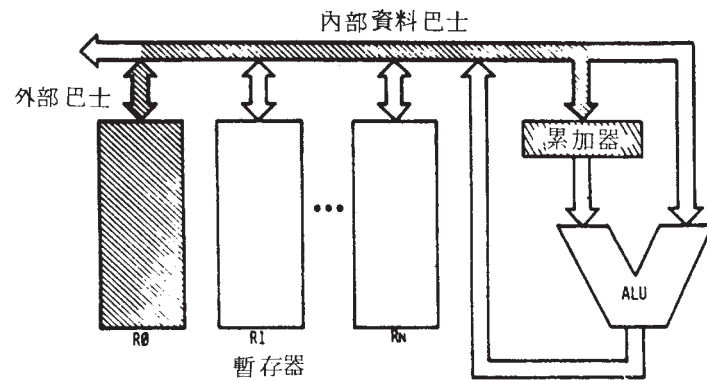


圖 2-9 第 1 暫存器 R0 之內含送入 ALU 之左輸入 ACC。

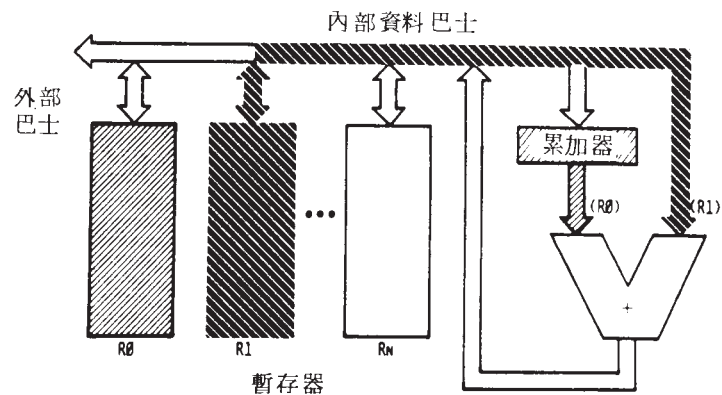


圖 2-10 第 2 暫存器 R1 之內含送至 ALU 之右輸入。

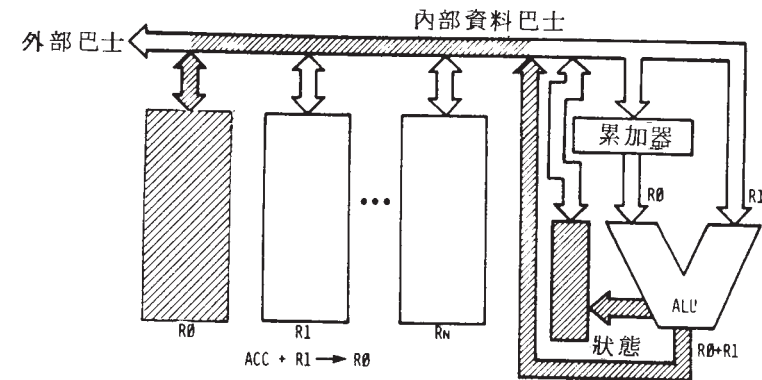


圖 2-11 產生結果，並存至 R0。

2-2-6 臨界競賽問題

事實上，圖 2-8 所示之簡單結構並不能正常動作。

問：時序問題為何？

答：問題為 ALU 輸出之結果必須再置回資料巴士，該結果不僅會沿 R0 之方向傳輸，而且會沿著整個巴士游動。如此，ALU 之右輸入又會受該結果之影響而改變，致使 ALU 輸出之結果又遭改變。此即為一**臨界競賽** (critical race)。為了避免發生錯誤，此時 ALU 之輸出必須與其輸入隔離。

有數種方法可將 ALU 之輸出與其輸入隔離。但都必須使用緩衝暫存器 (buffer register)。緩衝暫存器可置於 ALU 之輸出，亦可置於其輸入。經常的作法是將之置於輸入。就目前而言，緩衝器只需

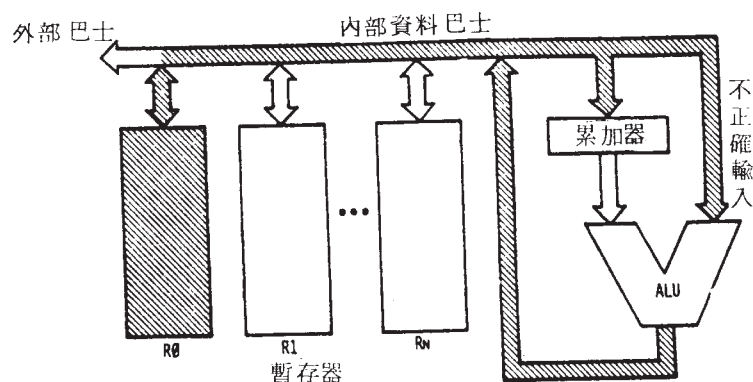


圖 2-12 臨界競賽問題

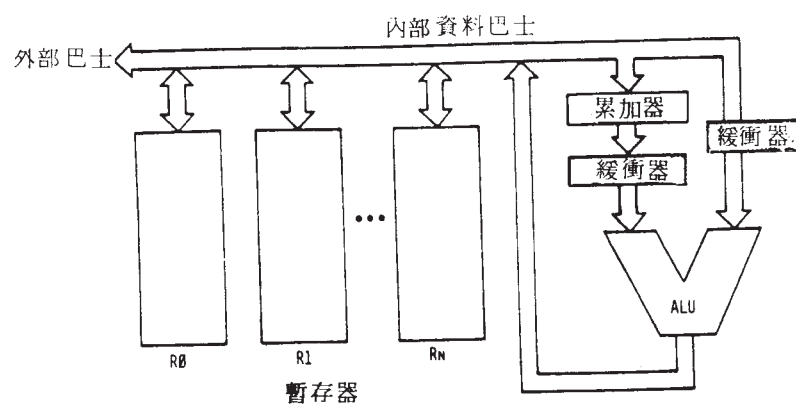


圖 2-13 一共需要兩個緩衝器

加於ALU 之右輸入即夠了。此章後面我們會談到，若ALU 左輸入之緩衝暫存器欲作為累加器（以使微處理器能有單位元組之指令），則累加器自己亦需要一緩衝器，此一結構即如圖 2-13 所示。

2-3 Z80微處理器之結構

如圖 2-14 所示，Z 80 為一具有 40 支接腳之大型積體電路。此

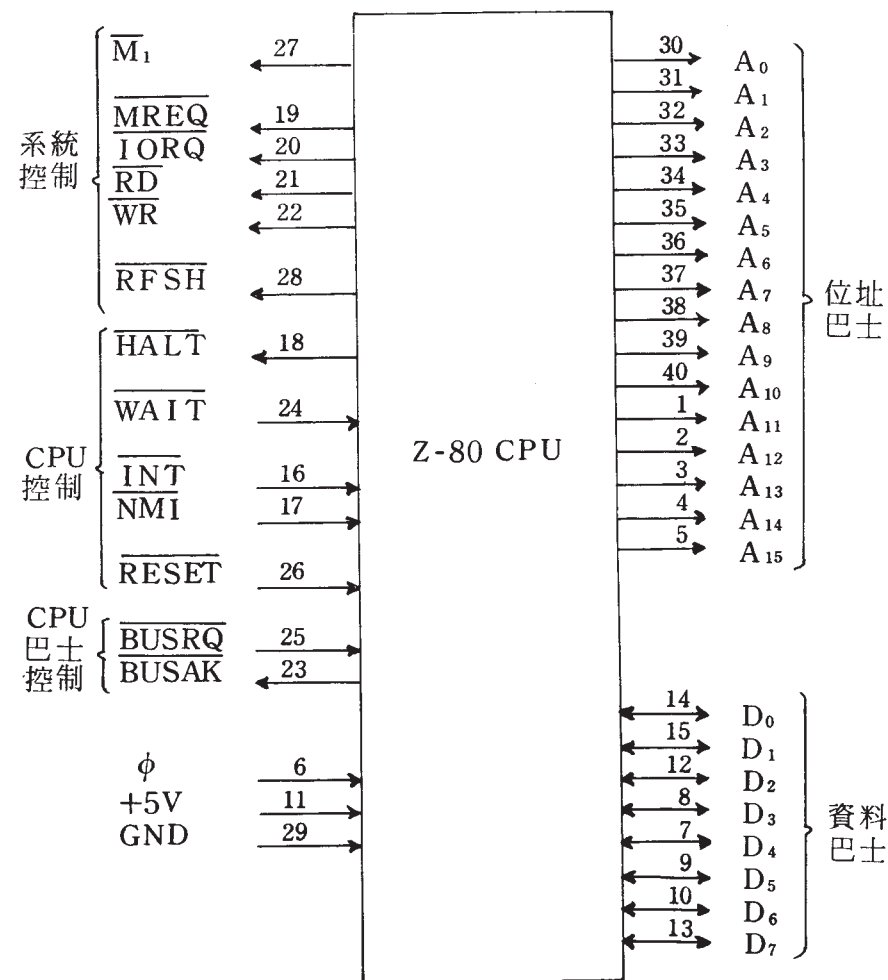


圖 2-14 Z 80 CPU之接腳外觀圖

40 支接腳中，有 16 支是 Z 80 微處理器輸出至外部記憶器之位址線（如圖之右側上方所示），此 16 條位址線構成了 Z 80 之位址巴士（address bus）。由於具有十六位元之位址，因此，Z 80 微處理器最高能選取 $65536 (2^{16})$ 個外部（記憶或輸入／輸出）位置。2-14 圖之右側下方所示則為 Z 80 微處理器與外部設備（如記憶器或輸入／

輸出界面電路）互遞資訊的孔道。此一孔道包括 8 條資料能雙向（bidirectional）傳遞之電線，構成所謂的資料巴士（data bus）。

2-14 圖之左側所示則為 Z 80 微處理器之電源線與控制巴士（control bus）。控制巴士包括 13 種控制信號。此些控制信號可分為三類：系統控制、CPU 控制、與 CPU 巴士控制。每一控制信號之功能將留待後面專章再作詳細介紹。

若將 2-14 圖再深入一層看，則所得的結果即如圖 2-15 所示。此一 Z 80 微處理器之內部構造方塊圖，總共包括七大部份：指令解碼與 CPU 控制、指令暫存器、算術／邏輯運算單元（ALU）、CPU 暫存器、位址巴士控制、資料巴士控制、與內部資料巴士。

13 種 CPU 與系統控制信號即由指令解碼與 CPU 控制部份產生，或送至該部份。八位元之資料巴士為 CPU 暫存器與外部記憶器及輸入／輸出設備間，資訊傳輸的路徑。位址巴士之大小為十六位元。平常，由於 Z 80 有完整之輸入／輸出指令，並且不需作“記憶映像式”（memory-mapped）之輸入／輸出，因此，其位址巴士能產生 65536 個記憶位址（0 至 65535）。（於記憶映像輸入／輸出，有一部份之記憶位址必須留作輸入／輸出設備之位址。）

CPU 內部資訊傳輸之主要路徑，即為連接 CPU 暫存器、算術／邏輯單元、資料巴士控制、與指令暫存器之內部資料巴士。Z 80 之算術／邏輯單元（ALU）能執行加、減等算術運算，AND、OR、與 XOR 等邏輯運算，以及移位之作業。此外，在十進調整指令（DAA）之控制下，其亦能履行 BCD 運算。指令暫存器（Instruction Register，簡記為 IR）則用以儲存方由記憶器取出，正欲解碼與執行之指令。

此節，我們主要將介紹 Z 80 微處理器之內部暫存器的結構。然後，2-5 節再介紹算術邏輯單元與控制單元之功能。至於各接腳信號之功能，以及 Z 80 CPU 之時序，則將留待第十二章再加以討論。

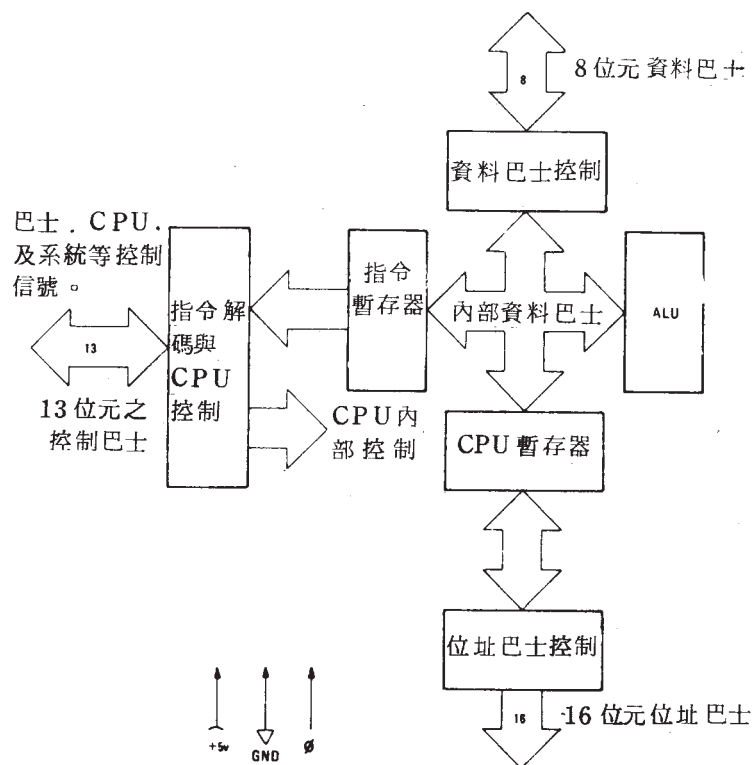


圖 2-15 Z 80 微處理器之內部結構方塊圖

CPU 暫存器

Z80 微處理器內部含有 208 位元記憶容量，此些暫存器皆做成靜態 RAM 形式，並可分為三類：一般用途暫存器，旗號暫存器，與特殊用途暫存器。茲將其分別介紹於后：

2-3-1 一般用途暫存器

如圖 2-16 所示，Z80 微處理器包括十四個八位元之一般用途暫存器。此些暫存器分別稱作 A, B, C, D, E, H, L, 以及 A', B', C'

A		A'	
B	C	B'	C'
D	E	D'	E'
H	L	H'	L'

圖 2-16 Z80 之一般用途暫存器

A	F	非補數	A	F	非補數
B	C	非補數	B'	C'	補數
D	E		D'	E'	
H	L		H'	L'	

A'	F'	補數	A'	F'	補數
B	C	非補數	B'	C'	補數
D	E		D'	E'	
H	L		H'	L'	

圖 2-17 Z80 一般用途暫存器之四種組合

,D',E',H',與L'。在任一時刻，只有其中一組七個暫存器以及其對應之旗號暫存器（F 或 F'）能動作。有一特殊之 Z80 指令可用以從 A 與 F 或 A' 與 F' 兩組中選取一組動作，而另一指令則可用以於 B, C, D, E, H, L 或 B', C', D', E', H', L' 兩組中選取一組。AF 與其餘六個一般用途暫存器之四種組合情形，如圖 2-17 所示。

具有兩組一般用途暫存器之優點為，程式設計者能迅速地由其中一組轉換至另一組。就最簡單之情況而言，這在微處理器內提供了更多之暫存器記憶設施。由於程式存取 CPU 暫存器之資料的速度，遠勝於存取記憶器者，因此，就資料處理速度而言，暫存器記憶優於記憶器記憶。此即所謂的**記憶層次**（memory hierarchy）；各種不同之記憶設備，具有不同之資料存取速度，宛如有“階級”一般。於較複雜之應用上，兩組暫存器之優點為，不用的一組可用以保存微處理器在接收到**插斷**以後的**環境**（environment）（環境即為各種資料現況）。此點將於爾後幾章再加詳論。

正如 8080 一般，Z80 之一般用途暫存器之功能有點特殊。八位元資料可來回搬運於記憶器與任一（七者中）暫存器之間，或來回於任兩暫存器之間。不過，諸如兩運算元之加或 XOR 等算術與邏輯運算，則僅能針對 A（或 A'）暫存器與另一暫存器或記憶位置。運算後之結果則恒存於 A 暫存器。換言之，A 暫存器即為一般所謂之**累加器**（accumulator）。一般而言，目前所選定之 A 暫存器即為 CPU 作算術與邏輯運算之主暫存器，此一情形如圖 2-18 所示。

A 暫存器除外之六個一般用途暫存器，可兩兩組成三個**暫存器對**（register pair）：BC, DE, 與 HL。在 8008, 8080, 與 Z80 之許多運算上，此些暫存器對所儲存的即為一項位址。譬如，如圖 2-19 所示的，H 暫存器即儲存某一項記憶位址之**高**（High）次八位元，而 L 暫存器則儲存該項記憶位址之**低**（Low）次八位元。BC 與 DE 暫存器之原理則同。在 8080, BC 與 DE 兩暫存器對亦可儲存一記憶位址，因此，總共有三個暫存器對能儲存記憶位址**指示器**（pointer）

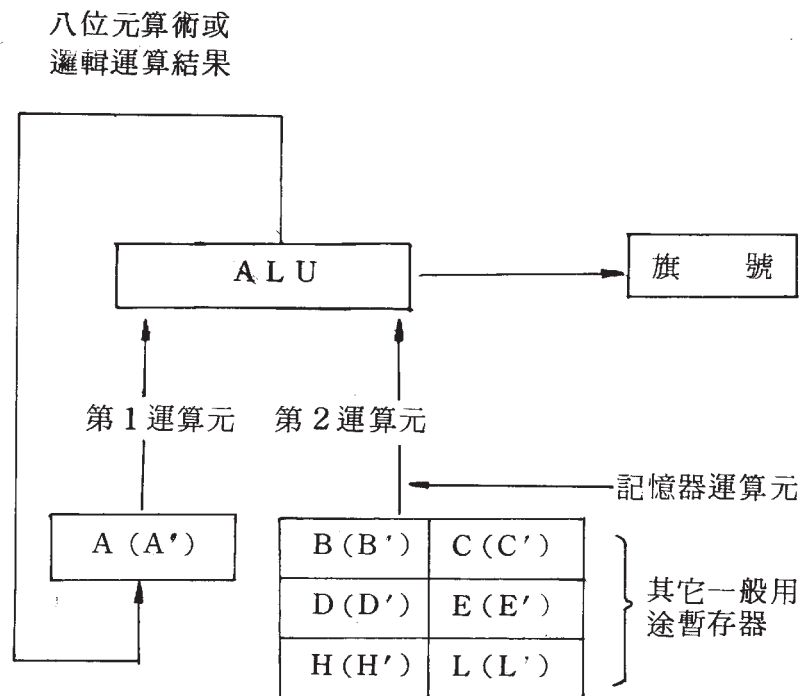


圖 2-18 算術與邏輯運算

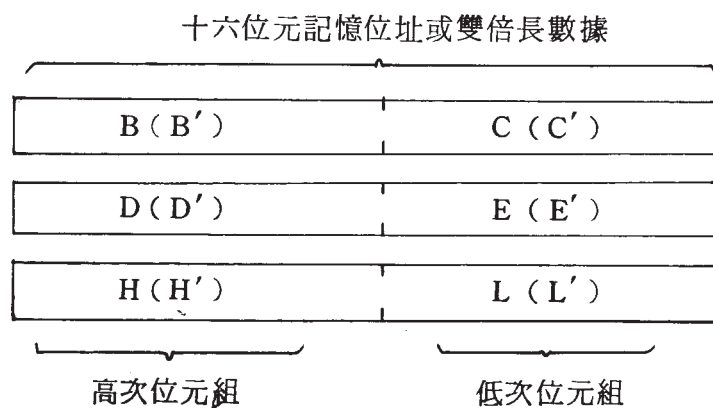


圖 2-19 暫存器對

，指及記憶器中之資料。一般而言，這三個暫存器對將如圖 2-19 般地儲存記憶位址，但是，它們亦可作第二種用途——用於**雙倍長算術** (double-precision arithmetic)，以儲存十六位元之數據。

雙倍長算術涉及十六位元數值之相加、相減、加一、或減一。Z80 之大多數算術與邏輯均以八位元為主，但却容許三個暫存器對、堆疊指示器、與兩索引暫存器間進行有限之雙倍長算術。此一觀點主要在於便利記憶位址之運算。因為，若無十六位元算術，則長十六位元之記憶位址的運作，就必須分兩次進行。圖 2-20 所示即為以暫存器對作雙倍長算術之情形。運算所需之兩十六位元運算元皆來自暫存器對或十六位元暫存器，結果亦存回此些暫存器（對）。運算結果影響旗號。

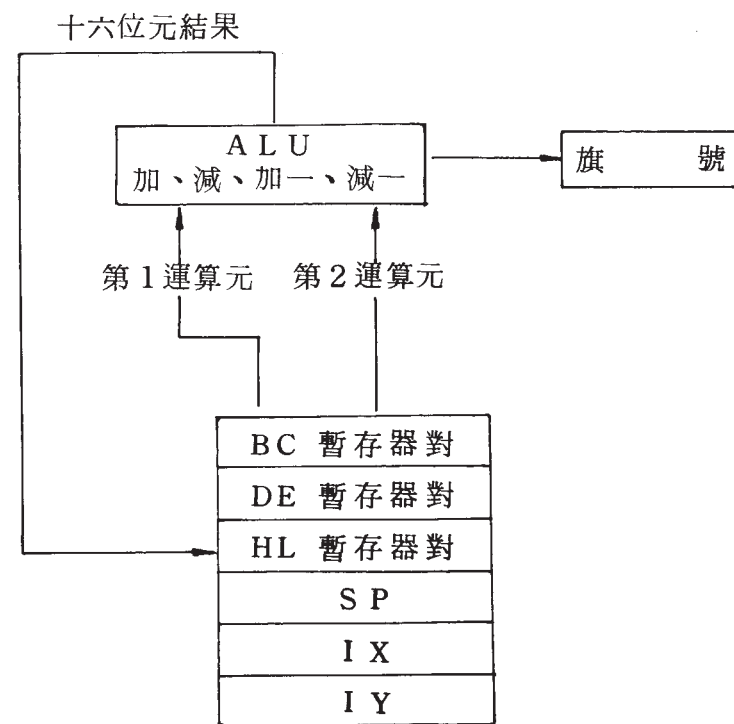


圖 2-20 暫存器對雙倍長運算

2-3-2 旗號暫存器

Z80 微處理器有兩個旗號暫存器 F 與 F'，但每一時刻僅能有一者被使用。F 或 F' 之選擇與 A 或 A' 之選擇一致。任一時刻，若非 A 與 F，則即為 A' 與 F' 被選取。如圖 2-21 所示，Z80 微處理器之旗號暫存器包括六種不同之旗號 (flag) (旗號暫存器有兩個位元沒用)。此些旗號指明了，在算術、邏輯、或其它運算後，微處理器之各種狀態。緊接，我們將此六種旗號，分別介紹於下：

Z80 旗號暫存器之六種旗號中，有四個是可以指令加以測試的。它們是：

1. **進位旗號 (C)**。該旗號值即等於累加器最高次位元所產生之進位值。加法運算時，若累加器之最高次位元產生進位，則進位旗號值置定為 1；否則，其值為 0。減法運算時，若累加器之最高次位元產生借位，則該旗號值亦置定為 1。除此之外，

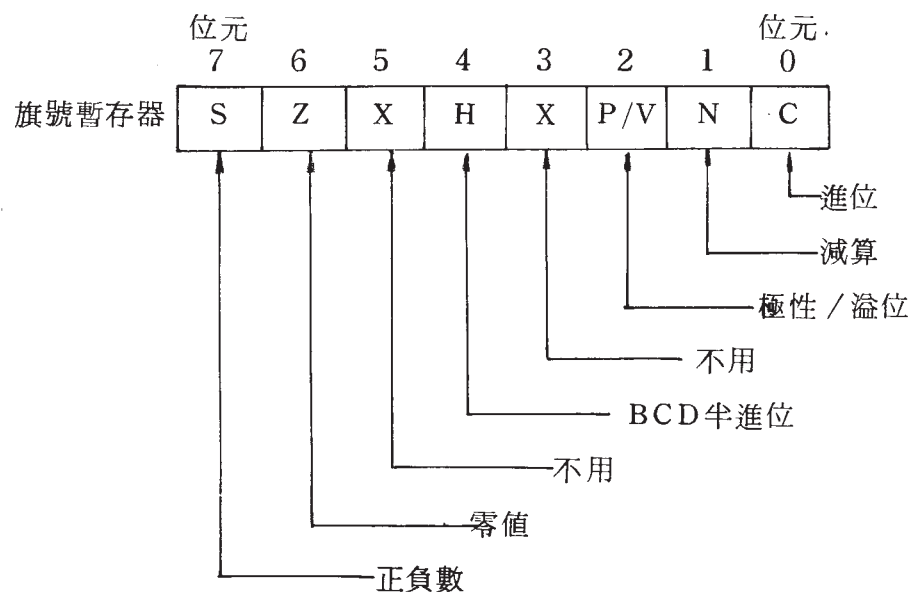


圖 2-21 Z80 微處理器之旗號暫存器

移位與旋轉指令亦會影響進位旗號之值。

2. **零值旗號 (Z)**。若運算結果使累加器之內含為零，則該旗號之值置定為 1；否則，其值為 0。
3. **正負數旗號 (S)**。該旗號主要用於有號數之運算。若運算所得之結果為負數，則該旗號之值置定為 1；否則，旗號之值為 0。由於八位元二進數之第 7 位元 (MSB) 即代表該數之正負數，並且以 1 代表負數，0 表正數。因此，事實上，正負旗號 (S) 之值即等於累加器之第 7 (最高) 位元值。
4. **極性 / 溢位旗號 (P/V)**。該旗號為一雙用途旗號。邏輯運算 (如 AND A, B) 時，其顯示累加器所含結果之極性 (parity)。而有號 2 補數算術時，其代表溢位。若算術運算所得之結果，超過八位元 2 補數形式所能表示之範圍 (-128 至 +127)，則溢位旗號之值為 1。在邏輯運算 (AND, OR, XOR) 時，若所得結果為偶極性，則 P/V 旗號之值為 1。否則，若所得結果為奇極性，則該旗號之值為 0。

Z80 旗號暫存器之旗號中，有兩者是不可測試的，該兩旗號均用於 BCD 算術。

1. **半進位旗號 (H)**。此位元代表 BCD 算術時，低次四位元所產生之進位或借位。Z80 微處理器在執行 DAA (十進制調整) 指令時，該旗號值用以校正先前兩濃縮 (packed) BCD 數之加 (減) 算結果。
2. **減算旗號 (N)**。由於加算與減算後之 BCD 調整方式不同，因此，該旗號用以記錄 Z80 微處理器剛剛所執行的是加法運算或減法運算，以作為 DAA 運算校正之根據。

2-3-3 特殊用途暫存器

所謂特殊用途暫存器，即為在設置時即已指定其專作某一種用途

之暫存器。Z 80 微處理器具有六個特殊用途暫存器；其中，有兩個（程式計數器與堆疊指示器）與 8080 完全相同。茲將此六個暫存器分別描述於後：

1. 程式計數器 (PC)

程式計數器即為儲存微處理器正在拿取之指令，所在位置之記憶位址的十六位元暫存器。Z80 指令有長一位元組、兩位元組、三位元組、與四位元組等四種。舉個例子說，圖 2-22 所示即 Z80 微處理器執行過一八個指令之指令系列時，其程式計數器內含值之變化情形。留意到，程式計數器之內含恒指至次一指令之起始位置，並且，視指令之長度而定，其內含每次自動加一，二，三，或四。對程式設計者而言，程式計數器之內含僅能取入 (loaded) 或存出 (stored)，而不能作任何算術或邏輯運算。

記憶位址		指令執行後 之 PC 內含
0100	第 1 指令 (一位元組)	0101
0101	第 2 指令 (二位元組)	0103
0103	第 3 指令 (三位元組)	0106
0106	第 4 指令 (一位元組)	0107
0107	第 5 指令 (一位元組)	0108
0108	第 6 指令 (一位元組)	0109
0109	第 7 指令 (二位元組)	010B

* 所有位址均為十六進制。

圖 2-22 程式計數器之作業

於 Z 80，每當程式計數器之內含被送出至位址巴士時，其內含值就自動加一。若程式跳越 (program jump) 發生，則新的內含值會自動存入程式計數器，蓋過原先之內含。

2. 堆疊指示器 (SP)

當程式計數器之內含指至外部記憶器中，次一欲執行指令之位置時，堆疊指示器則指至外部記憶堆疊器之堆疊頂端位置。記憶堆疊器之觀念雖非微處理器所僅有，但實際上幾乎每一微處理器均具有堆疊器之設施。所謂**記憶堆疊器** (memory stack)，事實上即為一段特別騰出，以作 CPU 暫存器，旗號暫存器，與程式計數器內含之暫時儲存的記憶區域。Z80 之記憶堆疊器，可座落於外部讀寫記憶器 (RAM) 之任意位置。由於用以儲存“堆疊頂端”之記憶位址，因之，Z 80 之堆疊指示器的長度為十六位元。

記憶堆疊器具有後進先出 (LIFO) 之特性。Z 80 可執行一推入 (PUSH) 指令，將某一 CPU 暫存器之內含推入堆疊器，同時亦可執行一拉取 (POP 或 PULL) 指令，將資料自堆疊器拉回至某一特定之 CPU 暫存器。自堆疊器拉取之資料，恒為最後被推入者。此一特性即如自助餐廳之堆盤一樣，最後被置於盤堆之盤子 (資料)，一定位於盤堆之最頂端 (堆疊頂端)，因此，亦為第一個被取出者。

由於 Z80 之記憶堆疊器是往位址漸小之方向延伸，因此，每有一項資料被堆入堆疊器，堆疊指示器之內含即自動減一。反之，每有一項資料自堆疊器取出，堆疊器指示器之內含值即自動加 1，以指至新的堆疊頂端。圖 2-23 所示即為堆疊指示器之作業情形。

除了平常程式設計者可用之作為資料之暫時寄存所外，堆疊器之設置主要在於便利**插斷** (interrupt) 之處理與**副程式巢串** (subroutine nesting)。此些用法將於爾後幾章再作詳細討論。

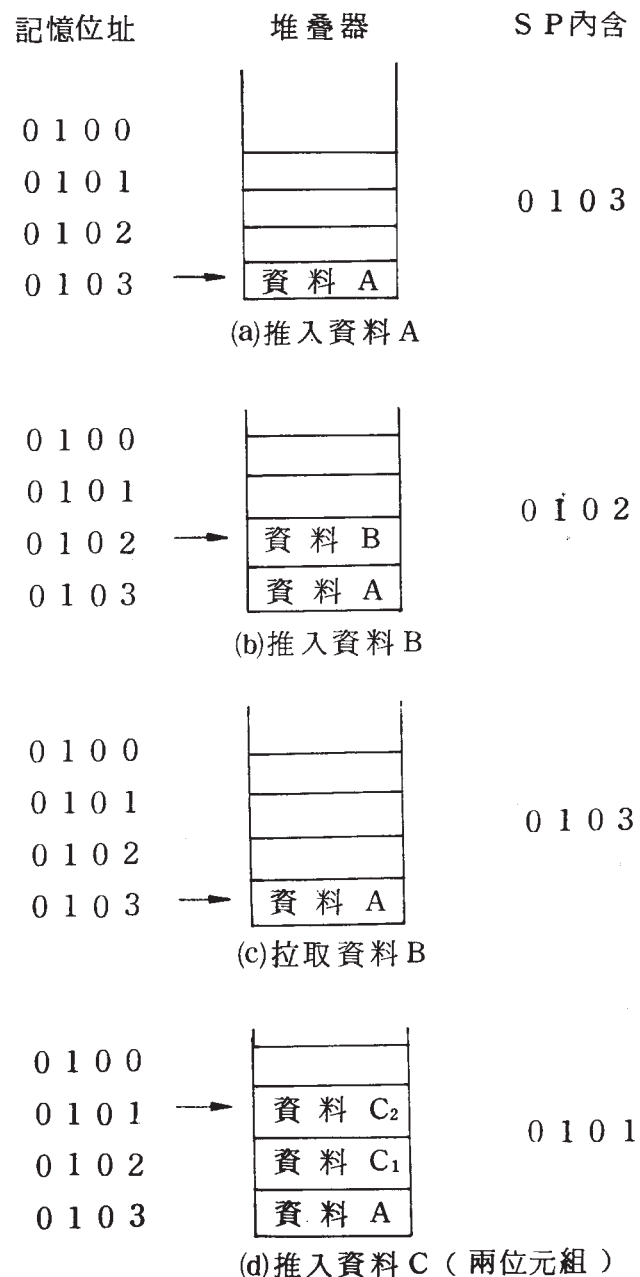


圖 2-23 堆疊指示器 (SP) 之作業

3. 兩索引暫存器 (IX 與 IY)

此兩十六位元之索引暫存器，用以儲存索引定址法所需之**基底位址** (base address)。於索引定址法，索引暫存器被用作為基底，以指至資料欲存取之記憶區域的起點。指令真正所欲存取之資料的**有效位址** (effective address)，則等於基底位址加上指令上所提供之**位移** (displacement)。位移指明了資料與基底間之“距離”。於 Z80，位移為八位元，並且表示成 2 補數形式。索引定址法大大地簡化了多樣的程式——尤其是處理表列 (list) 或表格資料 (table of data) 之程式。

4. 插斷頁位址暫存器或插斷向量暫存器 (I)

插斷向量暫存器可存入一八位元之記憶位址。當與插斷設備所提供之低次八位元位址組合時，該組合位址即為服務插斷設備之**插斷處理常式** (interrupt handling routine) 的起始位址。

舉個例子來說，假設有一讀紙帶機插斷了 Z80 微處理器。則於認知插斷後，Z80 會通知讀紙帶機之**控制器** (controller) 送出一低次八位元位址。讀紙帶機之控制器然後會送出一低次八位元位址，與插斷向量暫存器所提供之高次八位元位址組合。假設讀紙帶機所提供者為 14H (H 代表基底為 16，14H 即十六進數 14)，而 I 暫存器含 FF，則組合位址即為 FF 14H。如圖 2-24 所示，Z80 微處理器之控制電路然後會至位址 FF 14H 之記憶位置，與其次一位置，拿取讀紙帶機之插斷處理常式的起始位址 (在例中為 E000H)。一般而言，插斷向量暫存器所儲存的，即為一插斷向量表格之高次八位元位址，該向量表格可能含有 128 個插斷設備之插斷向量。

於程式之控制下，Z 80 微處理器可使用三種不同之插斷型態，插斷向量暫存器則用於其中之一種。其餘兩種插斷型態之一，則同於 8080 之插斷作業，容許最高八層之有向插斷 (vectored interrupt)。除了前述三種插斷作業型態外，Z80 亦具有一不可罩蓋 (nonmaskable) 插斷。此四種插斷作業型態，將於後面關專章討論。

5. 記憶復新 (Memory Refresh) 暫存器 (R)

最後一個特殊用途暫存器即為記憶復新暫存器 R。當外部記憶器為動態 (dynamic) 記憶器時，R 暫存器使得此種半導體記憶器能自動作記憶復新，以保持其原有之記憶內含。為了保存其原有記憶內含，動態記憶器每一記憶槽 (memory cell) 之內含，必須反覆地 (2 毫秒一次) 不斷被讀取或復新。每當 Z 80 微處理器拿取 (fetch) 一指令後，記憶復新暫存器之低次七位元值即自動加一。而其第七 (最高次) 位元則保持 LD R, A 指令規劃之結果。當微處理器正在解碼與執行所拿取之指令時，R 暫存器之內含隨同一復新控制信號，被送至位址巴士之低次部位。此時，位址等於 R 暫存器內含之每一記憶槽，即會被復新，而不致發生衝突之現象 (Z 80 CPU 亦同時欲存取該記憶位置)。對程式設計者而言，記憶復新作業是完全看不見的。由於與指令之解碼執行並行作業，因此，記憶復新並不延緩 Z 80 微處理器之作業。

正常情況下，程式設計者並不使用記憶復新暫存器。不過，必要時，程式可抄複 R 暫存器之內含，並加以測試。於記憶復新作業時，插斷向量暫存器之內含被置於位址巴士之高次八位元，以與 R 暫存器所送出之內含，組成十六位元之位址。

2-4 Z80之指令格式

如圖 2-25 所示，Z 80 之指令可為一，二，三，或四個位元組長。每一指令指明了一微處理器欲執行之運算。從簡化之標準看，每一指令包括運算碼與運算元位址或常數兩大部份。運算碼指明了微處

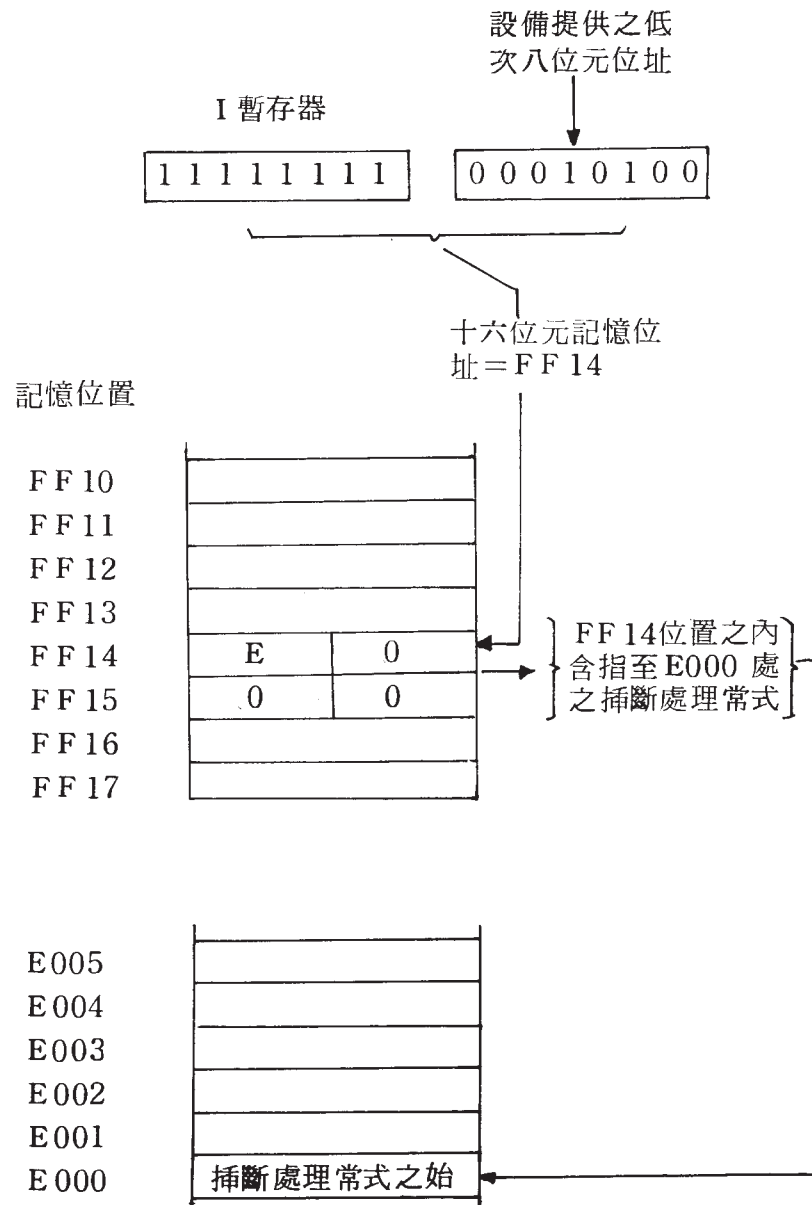


圖 2-24 插斷向量暫存器之作業

理器所必須履行之運算，而運算元位址欄則指明了運算所需之數據（或數據所在之記憶位置的位址）。就嚴格的計算機名詞而言，運算碼僅代表計算機所必須執行之運算，並不包括暫存器碼——指明那一中央處理器內之暫存器參與運算。不過，在微處理器領域，暫存器碼皆包括在指令之運算碼欄內。

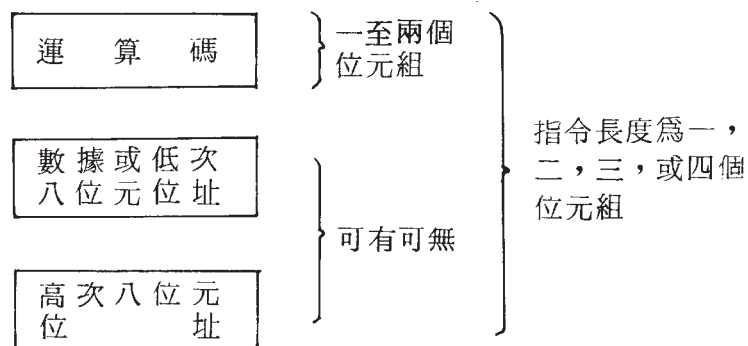


圖 2-25 Z 80 機器指令之格式

於 8080，指令最長為三個位元組。Z 80 由於增加了索引（indexed）指令（指令之各種定址法將於下一章中討論），因此，需要再一個位元組。於 Z 80，除了某些特殊指令之運算碼為兩個位元組外，一般而言，指令之運算碼皆為一個位元組長。

於 Z 80，有些指令在運算碼後需要一個位元組之數據，在此種情況下，指令長兩個位元組（索引指令除外）。有些指令則需指明一十六位元之記憶位址，此時，指令就至少長三個位元組。為了執行指令，中央處理器之控制單元必須拿取整個指令。於八位元微處理器（如 Z 80），微處理器每次僅能自記憶器讀取一個位元組，因此，微處理器在拿取每一指令時，所必須做的“記憶器讀取”次數，即等於指令之長度。由此可見，指令愈長，所必需花費的執行時間也愈長。

一位元組指令

原則上，單位元組指令乃最快速，且最受程式設計者喜愛之指令。Z 80 之一此種指令的典型為：

LD r, r'

該指令意指，“將 r' 暫存器之內容，取入 r 暫存器”。此為一典型“暫存器傳輸”或“暫存器對暫存器”作業。每一微處理器幾乎都有此等指令。這類指令執行完後，r 暫存器的內容同於 r' 暫存器之內含。注意，r 暫存器之內含因運算而改變，而 r' 暫存器之內含並未改變。由於資訊由 r' 暫存器傳送至 r 暫存器，因之，r' 稱為**來源暫存器**（source register），而 r 稱為**目的暫存器**（destination register）。

表 2-1 Z 80 之暫存器碼

暫存器碼	所代表之暫存器
000	B
001	C
010	D
011	E
100	H
101	L
110	(記憶器)
111	A

於計算機內部，每一指令皆表示成二進形式之機器碼。上述之 LD r, r' 僅是組合語言所使用的一種助憶符號，其不能直接存於記憶器。LD r, r' 指令之機器碼為 01DDD SSS₂，僅長一個位元組。其中，DDD 與 SSS 為**暫存器碼**。DDD 指明一 Z 80 暫存器為目的暫存器，而 SSS 指明一暫存器為來源暫存器。Z 80 所使用之暫存器碼如表 2-1 所示。由表中可看出，不論 DDD 或 SSS, 000 恒代表 B 暫存器，001 代表 C 暫存器，……等等。例如，若 LD r, r' 之機器碼為 01001 000₂，則其代表之運算即，將 B 暫存器之

內含抄錄至 C 暫存器。

下面為另一單位元組指令之例子：

ADD A, r

該指令之運算是，將 r 暫存器之內含加至累加器（所謂“加至”意即，結果亦存於累加器）。該指令之運算碼為

1 0 0 0 0 SSS

若 SSS = 010，則上述指令即為

ADD A, D

換言之，被加至累加器者為 D 暫存器之內含。

二位元組指令

將一“定值”（literal）（或稱常數）加至累加器之內含的指令

ADD A, n

即為一兩位元組指令之例子。該指令，運算碼 11000110₂，將緊接於運算碼後之常數 n，加至累加器。例如，若我們欲將累加器之內含加 5，則指令可寫成

ADD A, 5

該指令翻譯成機器碼即

1 1 0 0 0 1 1 0 （運算碼）

0 0 0 0 0 1 0 1 （常數 5）

由於運算元“立即”緊接於指令運算碼之後，因此，此種指令稱為**立即定址**（immediate addressing）指令。

三位元組指令

將位址為 nn 之記憶位置的內含，取入 Z80 微處理器內之累加器 A 的指令

LD A, (nn)

即為一三位元組指令。由於指令指明了運算元之十六位元位址，因此

，指令長三個位元組。該指令之機器碼形式為

0 0 1 1 1 0 1 0
n
n

運算碼八位元

位址之低次八位元

位址之高次八位元

其中，n 代表八位元之位址。

四位元組指令

Z80 有一些長四位元組之指令。將位址 nn 之記憶位置的內含取入 E 暫存器，同時，位址 nn + 1 之記憶位置內含取入 D 暫存器之十六位元取入指令

LD DE, (nn)

即為一四位元組之指令。該指令之機器碼為

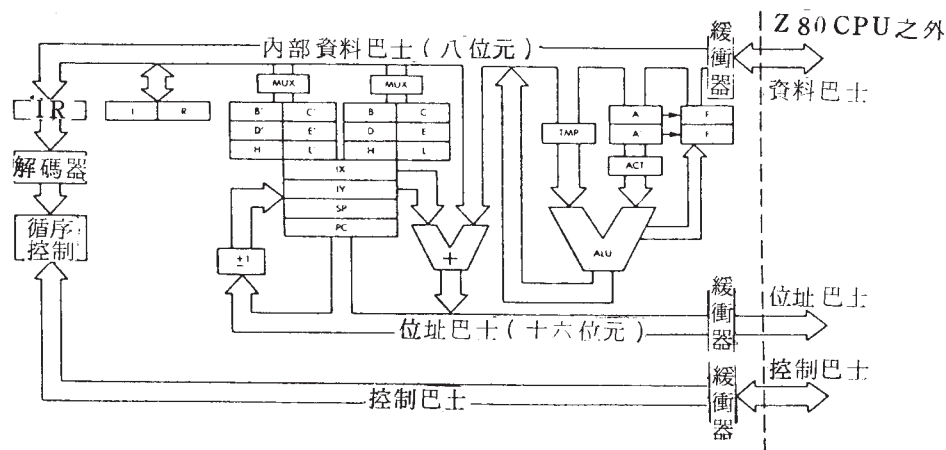
1 1 1 0 1 1 0 1	}	運算碼
0 1 0 1 1 0 1 1		
n	}	運算元位址
n		

前兩位元組為運算碼，後兩位元組為運算元位址。

2-5 Z80之指令執行

此節，我們討論 Z80 微處理器內，程式指令之執行情形。為了配合 2-2 節一般微處理器動作原理之介紹，這兒，我們將 2-3 節所介紹之 Z80 微處理器內部結構，重繪成 2-26 圖。於圖中，右邊“V”字形狀者代表算術／邏輯單元（ALU）。累加器（A 與 A'）位於 ALU 之右輸入，其緩衝暫存器在圖中以 ACT（暫時累加器）表示。ALU 之左輸入亦設有一稱為 TMP 之緩衝器，該緩衝器又稱**臨時暫存器**（

圖 2—26



(temporary register)。

旗號暫存器則位於ALU之右側。其內含主要受ALU運算之影響。

兩組八位元之一般用途暫存器與四個位址暫存器則位於圖形之中央。此些暫存器上連至資料巴士，下接至位址巴士。來自內部資料巴士之資料，經由一多工器(multiplexer，以MUX代表)送至有關之暫存器。由於三個暫存器對可用以儲存記憶位址，因此，此些一般用途暫存器亦與位址暫存器一起連至位址巴士。

插斷頁位址暫存器I與記憶復新暫存器R則個別畫於左邊，並連至內部資料巴士。指令暫存器(IR)，解碼器，與控制電路亦位於最左。

為了簡單起見，於爾後之說明中，我們皆固定採用兩組一般用途暫存器中，非補數的一組(A, B, C, D, E, H, L, 與F)。

前面說過，任何微處理器皆以拿取、解碼、與執行三個階段執行每一指令。此地，我們亦將以同樣步驟介紹Z80執行指令之情形。在正式開始介紹之前，讓我們先簡要地對Z80之時序作一認識。

Z80微處理器每一指令之執行，皆由下列基本作業所組成。

記憶體讀取或寫入

輸入/輸出讀取或寫入

插斷認可

微處理器執行一指令所需之時間，稱為指令週期。而完成上述任一種作業所需之時間，則稱為一機器週期(machine cycle)。於Z80，每一指令週期包括一個或數個機器週期，而每一機器週期又由數個時序週期組成。機器週期又簡稱M週期(M1, M2, ……等等)，而時序週期簡稱T週期(T1, T2, ……等等)。每一指令執行至少包括一機器週期——運算碼拿取之M1週期。圖2-27所示即為一包括三個機器週期之指令週期的例子。

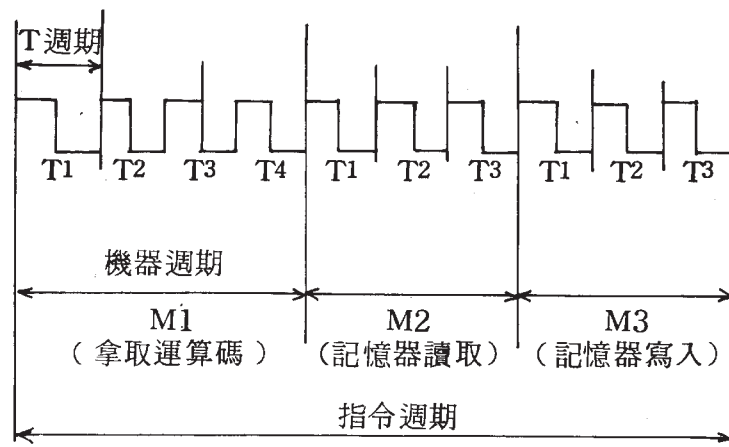


圖 2-27 Z80 之基本時序

2-5-1 拿取週期

於 Z80，每一 M1 週期之前三個 T 週期（分別稱為 T1, T2, 與 T3），即為指令之拿取階段。由於微處理器在執行任何指令之前皆必須先拿取，因此，對所有指令而言，這三個 T 週期都是相同的。指令拿取之過程如下：

T1：送出程式計數器（PC）之內含。

第一步是將欲拿取指令之位址（即程式計數器之內含），經由位址巴士送至記憶體，如圖 2-28 所示。記憶體會收到此一位址，並且位址解碼電路會將之解碼。幾百 ns (10^{-9} 秒) 後，被選取之記憶位置的內含即會出現於記憶體之輸出端，此些接腳即連接於資料巴士上。幾乎任何計算機皆利用記憶體讀取的時間，將程式計數器之內含更新：

T2：程式計數器之內含值加一。

於 T2 週期末了，記憶內含即可由資料巴士取入微處理器。

T3：指令取入指令暫存器。

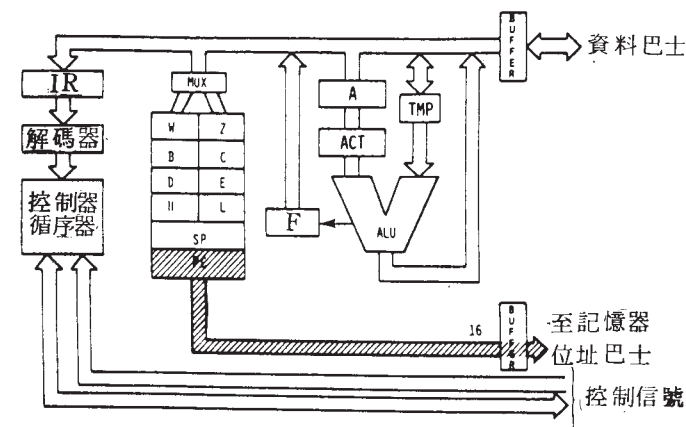


圖 2-28 指令拿取——PC 值送給記憶體

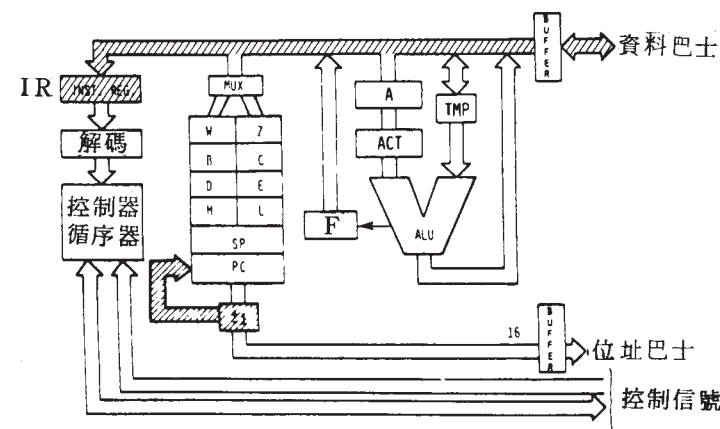


圖 2-29 PC 值加一，拿取指令存入指令暫存器

2-5-2 解碼與執行

T 3 時，讀出記憶體之指令被置於資料巴士，並送入 Z80 內之指令暫存器，開始解碼。

注意，M1 之 T4 週期一定需要。指令一旦於 T3 存入指令暫存器後，就必須開始解碼與執行。這至少需一個 T 週期，T4。

有少數指令之 M1 週期還需有第 5 時序週期（即 T5）。不過，在大多數指令，此一 T5 週期自動省略。凡是指令執行需要一個機器週期以上，則時序在經 M1 之 T4 以後，自動進入 M2 之 T1，開始第 2 個機器週期。

表 2-2 所示即為每一指令執行時，微處理器所發生的變化。由於 Z80 的表尚未問世，因此，這兒只有以 8080 之表格代替。

下面，我們看一例子。

LD D, C

該指令對應於 8080 之 MOV r1, r2 指令，請參照表 2-2 之第一行。

LD D, C 指令所履行之運算，即將 C 暫存器之內含，抄至 D 暫存器，此一運算之情形如圖 2-30 所示。

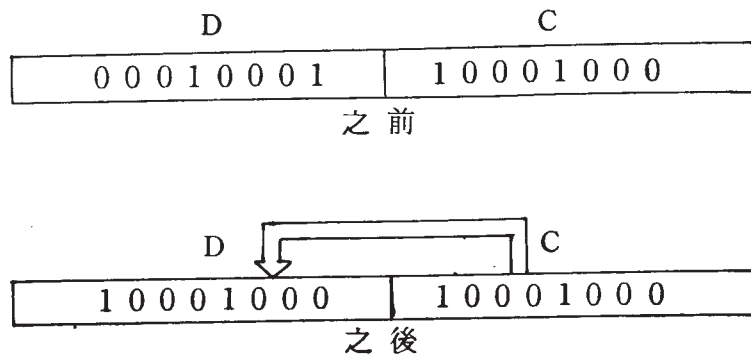


圖 2-30 C 暫存器內含抄入 D

M1 週期之前三個 T 週期用以由記憶體拿取指令。如圖 2-31 所示，於 T3 末了，指令已存於指令暫存器內，並開始解碼。

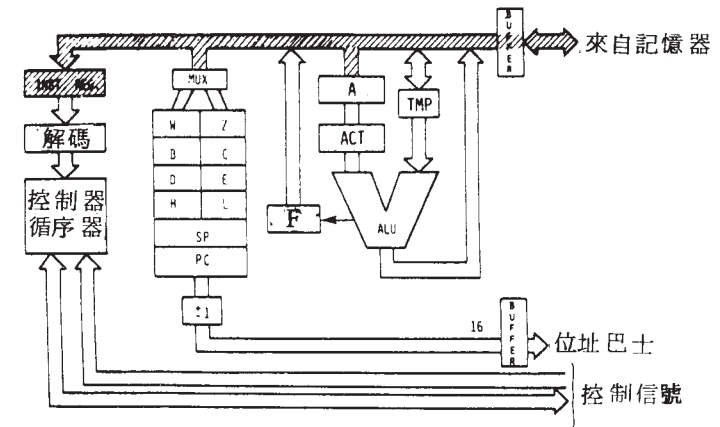


圖 2-31 指令自記憶體抵達指令暫存器

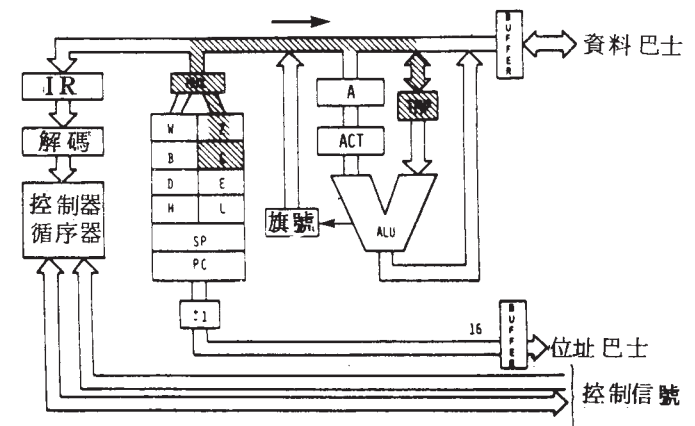


圖 2-32 來源暫存器之內含存入 TMP

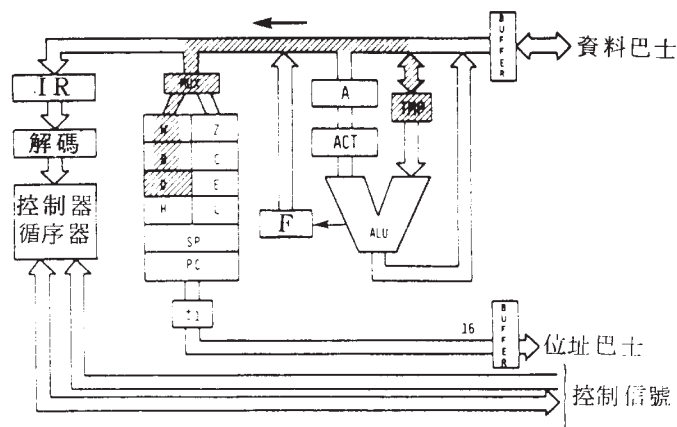


圖 2-33 TMP 之內含存入 D 暫存器

T4 期間，C 暫存器之內含取入臨時暫存器 TMP，如圖 2-32 所示。

T4：(SSS) → TMP

T5 期間，TMP 之內含存入 D 暫存器，如圖 2-33 所示。

T5：(TMP) → DDD

LD D, C 指令之執行至此結束。C 暫存器之內含已存入目的暫存器 D。因此，該指令之執行僅需一個機器週期 M1 就夠了。

指令之執行時間可輕易算出。標準 Z80 之時序週期是 500 ns。因此，LD D, C 指令之執行時間為 $5 \times 500 \text{ ns} = 2.5 \mu\text{s}$ 。

有人或許要問，為何將 C 暫存器之內含傳至 D 暫存器需要兩個時序週期 (T4 與 T5)，而非一個？為何 C 暫存器之內含必須先送至 TMP，再由 TMP 送至 D 暫存器？難道不能簡單地，於一個 T 週期內，將 C 暫存器之內含直接送至 D 暫存器嗎？

此問題的答案是，由於內部暫存器製作之原故，因此不可能。事

實上，所有的內部暫存器都屬於微處理器晶片上之讀寫記憶體的一部份。於 RAM 內，每次僅能有一個字組被選出。因此，欲同時讀寫 RAM 之兩個不同記憶位置，根本不可能。此乃為何 C 暫存器之內含必須先讀取，並存入一臨時暫存器 TMP，然後再轉寫入目的暫存器 D 之原故。這是幾乎所有單晶微處理器所共同具有之限制。雙口式 (dual-port) 的 RAM 可用以克服此一限制。

2-5-3 重要練習

為了使讀者能真正徹底了解 Z80 微處理器指令執行之原理，下面，我們緊接再舉幾個例子。

ADD A, r

該指令意指：“將 r 暫存器 (二進碼 SSS 所指明者) 之內含，加至累加器 (A)，並將結果存於累加器”。換言之，運算前，累加器含加法運算之其中一運算元 (因之為來源暫存器)；運算後，累加器含加法運算之結果 (因之為目的暫存器)。由於指令僅顯明地指出 r 暫存器，而隱含累加器 A 為來源兼目的暫存器，因而，此種指令稱為隱含 (implicit) 指令。隱含指令之優點為指令簡短，執行迅速。

現在，我們開始看 ADD A, r 指令之執行情形。該指令長一個位元組，需要兩個機器週期 (M1 與 M2) 之執行時間。正如尋常一樣，M1 之前三個 T 週期期間，指令由記憶體被取至指令暫存器 (IR)。T4 一開始，指令被解碼與執行。假設被加至累加器 A 者為 B 暫存器，則指令之運算碼為 10000000_2 (B 暫存器之代號為 000)。

T4：(SSS) → TMP, (A) → ACT

有兩個傳輸同時進行。如圖 2-34 所示，首先，來源暫存器 (此例為 B) 之內含傳至 TMP，亦即，ALU 之右輸入。同時，累加器之內含亦傳輸至臨時累加器 (ACT)。檢閱 2-34 圖，您就會相信此兩項傳輸能同時進行。因為，它們各自使用不同之路徑。B 至 TMP

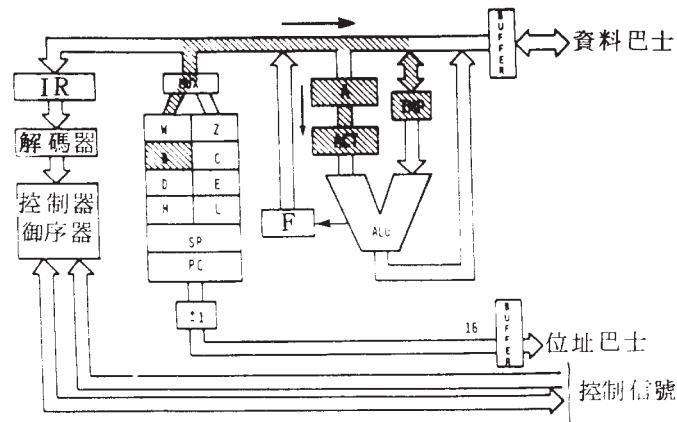


圖 2—34 兩項傳輸同時發生

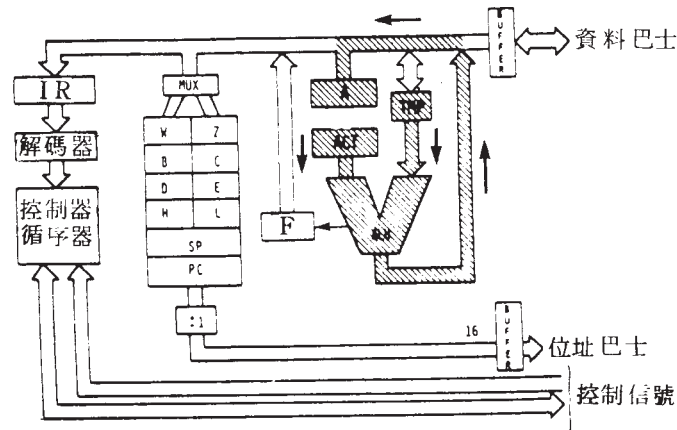


圖 2—35 加算發生

之傳輸使用內部資料巴士，而累加器至ACT之傳輸則利用一與資料巴士無關之簡短路徑。爲了爭取時間，因此兩項傳輸同時進行。此刻，ALU之左輸入與右輸入均已正確定值。左輸入爲累加器內含值，而右輸入爲B暫存器之內含。因此，加算可以開始。正常情況下，我們希望加算於M1之T5期間進行。不過，事實上並沒有。微處理器引入了第2個機器週期M2。在M2之T1期間，什麼事都沒做！M2之T2週期加法運算發生。如圖2-35所示，ACT之內含與TMP之內含相加，最後結果存入累加器。

問：爲何加算欲延遲至M2機器週期之T2期間才完成。而不在M1之T5期間就完成呢？（這實在是一十分難答的問題。因爲，其涉及CPU之設計技巧。）

答：這是大多數CPU所使用的一種標準設計“技巧”。其稱之爲**拿取／執行重疊**（fetch/execute overlap）。基本的觀念是這樣的：由2-35圖可看出，真正加算執行僅用及ALU與資料巴士，並不用及暫存器RAM。而我們（或控制單元）知道，任何指令執行完後的緊接三個狀態，即爲次一指令之M1週期的T1，T2，與T3。此些狀態的執行僅需用及程式計數器與位址巴士。存取程式計數器涉及存取暫存器RAM（這說明了此同一技巧爲何不能用於LD r, r'指令）。因此，爲了爭取時間，我們寧可讓加算延緩至M2週期完成，而將加算與次一指令之拿取重疊進行（即2-28圖與2-35圖合併進行），如圖2-36所示。

M2之T1期間，資料巴士必須用以送出狀態訊息，致其無法用於加算。爲此，真正之加算必須延緩至T2期間方能實現。至此，可算完整地回答了上述的問題。

上述拿取／執行重疊技巧之優點已十分明顯。若加算於M1之T5期間完成，則次一指令之拿取必須等T5完成後才能開始。而於拿取／執行重疊技巧，次一指令之拿取只要M1之T4結束後，即可開始。因此，可以節省一個時序週期之時間。當然，拿取／執行重疊技巧

也並非隨時都可使用的。使用此一技巧，必須確信巴士與各資源不致造成衝突時，方能實施。

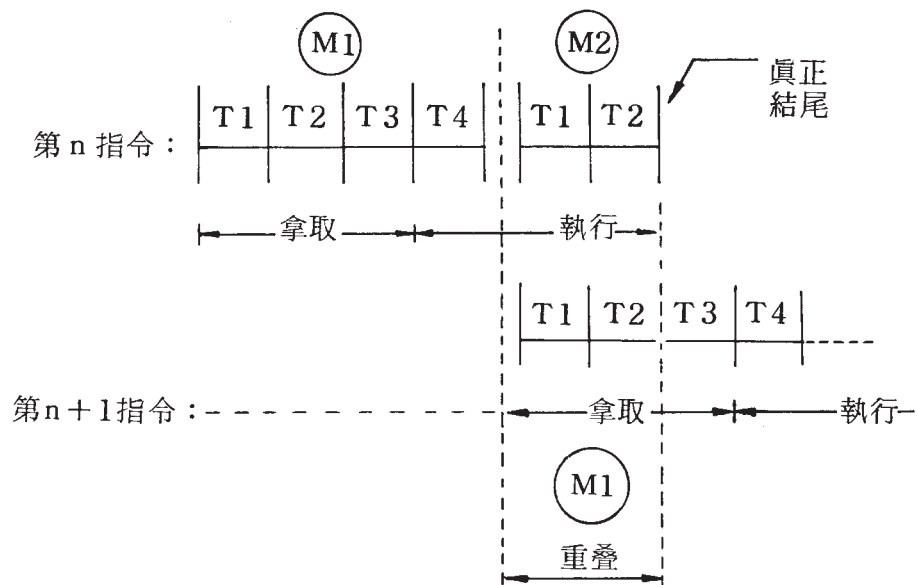


圖 2-36 拿取 / 執行互相重疊之技巧

問：我們可更進一步利用此一技巧，使用M2 之T3 執行一更長之指令嗎？

答：此一問題可由表 2-2 找得答案。該表詳細列出了 Intel 8080 之每一指令執行時，系統（包括CPU）內所發生之一切變化情形。緊接，我們再看一個更複雜之指令：

ADD A, (HL)

該指令之運算碼為 10000110_2 ，意指“將HL 暫存器對內含所指之記憶位置的內含，加至累加器，結果存於累加器。”換言之，除了累加器外，另一運算元含於記憶器之某一記憶位置，而該記憶位置的位址即為HL 暫存器對的內含。

該指令之由來事實上還有一段歷史。為了維持與8008 之吻合性

，因此，8080 保留了此一指令。早期之8008 並無直接定址（direct addressing）能力，記憶位置的選取都靠以H與L兩暫存器作為指示器。必須強調的是，8080 與Z80 並無此種限制。它們都有直接定址能力。以H與L作指示器之暫存器間接定址（register indirect addressing）變成了一種額外之優點，而非缺點。

現在，我們開始看此一指令之執行（該指令對應於8080 之ADD M指令，表2-2 之第16 個）。M1 之T1, T2，與T3 狀態如常用以拿取指令。T4 期間，累加器之內含傳至緩衝暫存器ACT，定義了ALU 左輸入之內含。

為了取得欲加至累加器之第二項數據，微處理器必須再存取記憶器。該位元組資料之位址含於H與L 暫存器。因此，H與L 暫存器之內含必須送出於位址巴士上。

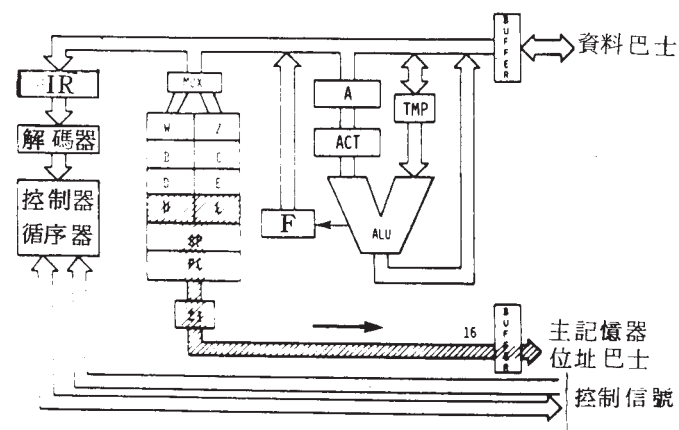


圖 2-37 HL 暫存器之內含送至位址巴士

如圖 2-37 所示，M2 機器週期，HL 暫存器內含被讀出，置於位址巴士，如同程式計數器之內含被置於位址巴士一般。前面曾說過，T1 期間，狀態資訊輸出於資料巴士，但此處並未如此。簡言之，一共僅需兩個 T 週期：第一 T 週期，記憶器讀取資料，第二 T 週期，資料備好於資料巴士，並被送至 ALU 之右輸入——TMP。

ALU 之兩邊輸入現已皆固定。此時之情況就宛如先前之加法指令，只需將兩數相加即可。為了應用拿取 / 執行重疊技巧，加算並未在 M2 之 T4 期間執行，而一直延緩至 M3 之 T2 才進行。如圖 2-35 所示，T2 期間，ACT 與 TMP 之內含相加，結果存於累加器：

$ACT + TMP \rightarrow A$ 。

問：對程式設計者而言，上述指令之執行時間為何呢？

7.5us ? 或 4.5us ?

下面，我們再看一個更複雜之指令。該指令使用直接定址法（指令本身含運算元之記憶位址），並使用兩個看不見之暫存器 W 與 Z。

LD A, (nn)

LD A, (nn) 指令翻成機器碼儲存於記憶器中之情形如圖 2-38 所示。該指令長三個位元組，運算碼為八位元：00111010₂。緊接運算碼後即為運算元之十六位元位址（低次八位元在前），指令之效用即將該位址之記憶位置的內含，取入累加器內。

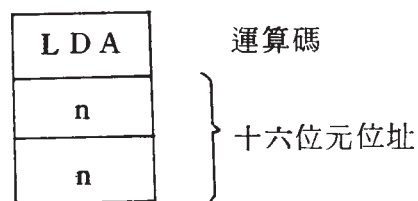


圖 2-38 LD A, (nn) 為一三位元組指令

M1 之 T1, T2, 與 T3 期間，同樣為指令運算碼拿取。T4 期間，控制單元將指令解碼，發現必須再拿取指令之次兩緊接位元組，以獲得運算元之位址。

T4 週期乃是必須的，因為，指令必須解碼。由於解碼不必用到整個週期，因此，有部份時間還是浪費了。不過，這是**時序同步邏輯**（clock-synchronous logic）之想法。由於微處理器內部以**微指令**（microinstruction）作解碼與執行，因此，這是**微程式化控制**（

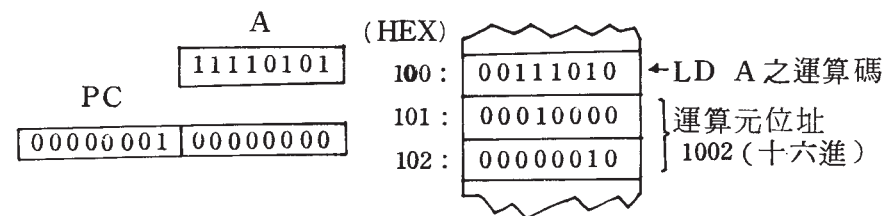


圖 2-39 LD A 執行前

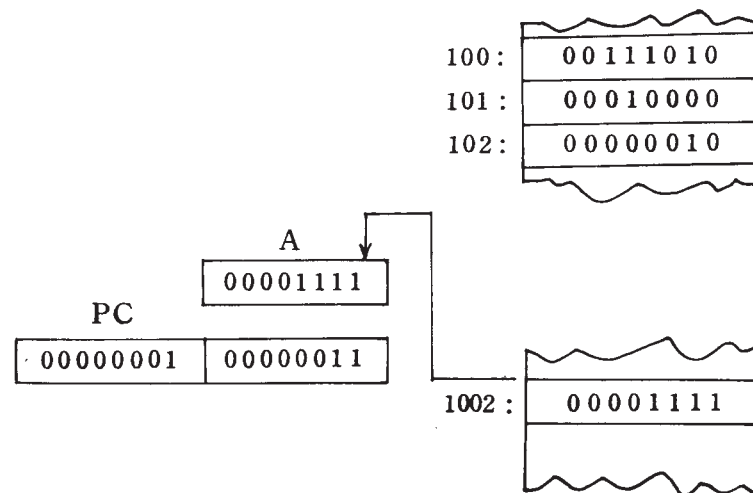


圖 2-40 LD A 執行後

(microprogrammed control) 所必須付出之代價。

如圖 2-39 所示，微處理器緊接拿取指令之次兩個位元組。指令之實際效應如圖 2-40 所示。

於 Z80，有兩個特別的暫存器是程式設計者所不知曉的。如圖 2-37 所示，此兩個暫存器即 W 與 Z。

第 2 機器週期 M2：如常地，前兩狀態 (T1 與 T2) 用以拿取程式計數器所指之記憶位置的內含。程式計數器內含於 T2 期間加 1。有時，T2 結束之前，資料即已由記憶器送至資料巴士。T3 結束之前，該字組必須被取入微處理器內之臨時暫存器 Z。圖 2-41 所示即為該指令之第二位元組 (B2) 取入 Z 暫存器的情形。

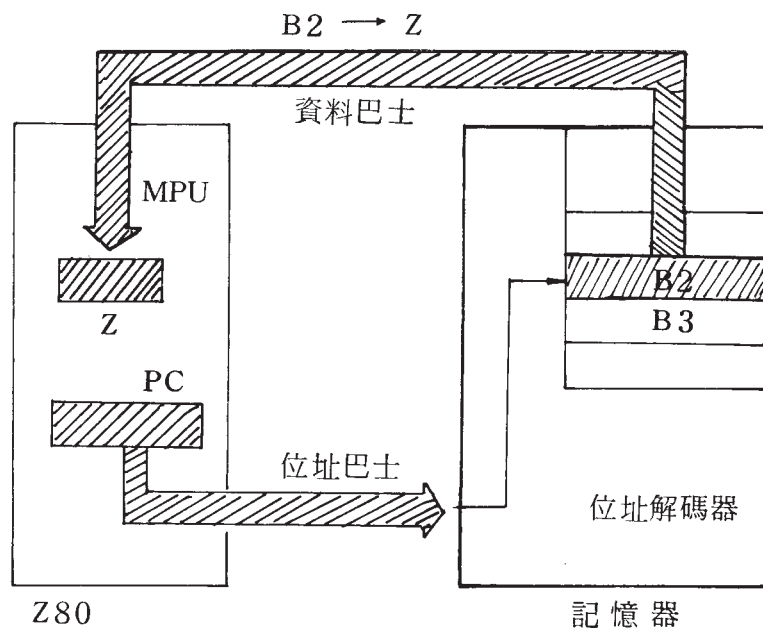


圖 2-41 指令之第二位元組取入 Z 暫存器

第 3 機器週期 M3：同樣地，程式計數器之內含再被置於位址巴士，加一，然後，指令之第三位元組 (B3) 由記憶器被讀入微處理器內之 W 暫存器。此時，亦即 M3 之 T3 末了，W 與 Z 暫存器含 B2 與 B3 —— 運算元之十六位元位址。該位址原來存於記憶器中，緊接運算碼之後的兩個記憶位置。第三機器週期至此結束。(W 與 Z 含運算元位址。為了拿取真正運算元，該位址必須再送給記憶器。(W 與 Z 暫存器請參閱 2-37 圖)。

第 4 機器週期 M4：此時，W 與 Z 之內含由位址巴士輸出，送給記憶器。T2 末了，被選取之記憶位置的內含出現於資料巴士。T3 末了，其被取入累加器 A。指令執行完全結束。

由上述敘述讀者可看出，為了拿取直接定址指令，微處理器必須作三次“記憶器讀取”。然後，拿取真正運算元又需再作一次記憶器讀取。由於記憶器存取需要時間，因此，此種作法在時間上甚為不經濟。此亦即為何一般微處理器設計時，其內部欲多設暫存器，以及使用諸如隱含、零頁等定址法之原故。

問：該指令可使用其它暫存器而不使用 W 與 Z 暫存器嗎？

答：不。若指令使用其它暫存器，如 H 與 L，則此些暫存器之原先內含就遭破壞了。寫程式時特別注意到，將任何新值存入某一暫存器，其原先之內含值即遭破壞。因此，切勿因疏忽而造成無意破壞其它暫存器內含之舉。W 與 Z 暫存器之必要性亦在此。

問：那不用 W、Z，用 PC 總行吧？

答：更不行。PC 之內含必須再被用以拿取下一指令，因此，不能被破壞。

還有一種型態之指令必須研究，那就是改變程式指令執行之順序的分支 (branch) 或跳越 (jump) 指令。截至目前為止，我們一直假設指令是依其儲存在記憶器中之順序被執行。事實上，有些指令可改變此一順序。此些指令被執行的結果，將使微處理器跳至另一個新的記憶位置上之指令去執行，而非次一緊接指令。下面即為一例：

J P n n

該指令執行的結果，程式控制跳越至位址 n n 之記憶位置上的指令，微處理器從那兒起再開始執行指令。該指令即為表 2-2 上之第 18 行，“JMP addr”指令。此地的敘述將根據表上的此一行。

該指令為三位元組之指令，第一位元組為運算碼 11000011₂。緊接兩位元組為跳越運算之目的位置的位址。觀念上，跳越指令之實際效應為以跳越指令上之目的位址，取代程式計數器之原先內含。不過，實際上，為效率原故，作法上稍有點不同。

同前地，M1 之前三個 T 週期為指令運算碼拿取。在 T4 期間，指令被解碼，其它未有任何事件發生（X）。次兩機器週期，微處理器拿取指令之第二與第三位元組。M2 時，第二位元組被拿取，並存入 W 暫存器。如同加算例子，緊接兩步驟與次一指令拿取互相重疊。

緊接的兩步驟是：WZ 暫存器之內含送出，以及 (WZ)+1 → PC。換言之，次一指令拿取是使用 WZ 暫存器之內含，而非程式計數器之內含。WZ 之內含然後加 1，存入程式計數器。以備次次指令之拿取。控制單元會記錄跳越指令之執行，並以不同之方式執行次一緊接指令。

此兩額外步驟之效應如下：

W 與 Z 暫存器的內含被置於資料巴士。因此，次一拿取的指令是 W 與 Z 暫存器所指之記憶位置，而非 PC 所指。此即為**跳越**（jump）。W 與 Z 暫存器的內含然後被加一，並存入程式計數器。因此，跳越運算後，微處理器能再正確地拿取次一指令。

問：能夠不用 W 與 Z，而僅用 Z 暫存器嗎？

答：是。不過，速度較慢。我們可將第二位元組取入 Z，然後拿取第三位元組，並將之置於程式計數器之高次位元組（PCH）。不過，在使用程式計數器內含前，Z 暫存器之內含必須再取入程式計數器之低次位元組（PCL）。這就減緩了作業速度。因此，為了節省時間

，我們同時使用了 W 與 Z 暫存器，並且，WZ 暫存器之內含並不存入程式計數器然後再送出，而是直接送至位址巴士，以拿取次一新的指令。

Z80 系統硬體結構之介紹至此先告一段落。緊接，我們將開始介紹 Z80 微處理器之定址法，指令集，並教讀者如何以之設計各式各樣的程式。

文件名稱： Z80 微電腦軟體硬體第 1、2 章

文件分類	I
文件編號	00028
文件批號	01

製作群	原稿掃描	文稿編輯
	原稿圖文分離	文稿整合
	原稿辨識	文稿校對
	文稿成品輸出	特別感謝名單

文件完成日期	初版	2007-02-08	其他
	再版		加註

文件出處	原圖書書名	Z80 微電腦軟體硬體
	原圖書作者	陳金迫
	原圖書出版者	儒林圖書有限公司
	原圖書出版日	民國 70 年 8 月

DDSC 文件 版權宣告

本文件版權屬原輸出公司、出版社、圖書公司或原著作人所有，作商業用途者請自行洽上述公司，本文件僅可在非商業上流傳或供私人收集資料用。另由於資料老舊 **DDSC** 不對原書內的內容負責，且除了更正原書內的錯字、漏字之外一切照原書內容所用的文字顯示。

檔名格式說明：

DDSC — 文件分類 — 文件編號 — 文件批號 — 文件名.PDF

以 DDSC 為起頭，加上 1 個字母為分類代碼，再加上以 5 位數由 00001 起的編號，加上 2 位數由 01 起的編號，加上完整的文件名稱而成的。

其中分類代碼詳見下面列表。文件批號指該文件為非合訂版的，可能因書的內容過多而分批完成的，此項可有可無。

文件分類代碼說明	
代 碼	說 明
A	小說／文學類文章類
B	娛樂類
C	天文類
D	科學類
E	古文明事物類
F	自然界類
G	古怪事物類
H	動／植物類
I	電子類
J	電腦類
K	教育教學類

Documents Digitize Service Center 製作
1998-2007