

第一部份

6502 微處理器

第1-1章 簡介

6502微處理器可以執行56個不同型態的指令(65C02則可以執行71個不同型態的指令)，而不同的定址模式的組合可提供總計151個的個別指令給使用者(65C02則為212個)。

6502指令集的指令長度，有一位元組、兩位元組、與三位元組等三種。其中第一位元組一定就是代表某運算之**運算碼**(operation code，簡稱OP code)。因此，在指令被拿取後，此一位元組立刻被存入”指令暫存器”，並導引至指令”解碼電路”，產生適當的內部控制信號，送給微處理器內部之各個組件，或甚至是系統中的外部設備。指令的第二與第三位元組(若有的話)則被送入”資料巴士緩衝器”，然後視情況送往**算數邏輯單元**(若它們代表數據)或**程式計數器**(兩位元組代表一項位址)。若第二與第三位元組代表運算元之位址，則表示在指令執行前，微處理器必須再做一次記憶器讀取，以獲得運算元(operand)。

1-1-1 暫存器架構

如圖1-1-1所示，6502微處理器含有六個主要的暫存器：一個16-bit的程式計數器(PC)，一8-bit的累加器(A)，兩個8-bit的索引暫存器(X與Y)，與一8-bit之堆疊指示器(SP)，以及一8-bit的狀態暫存器。

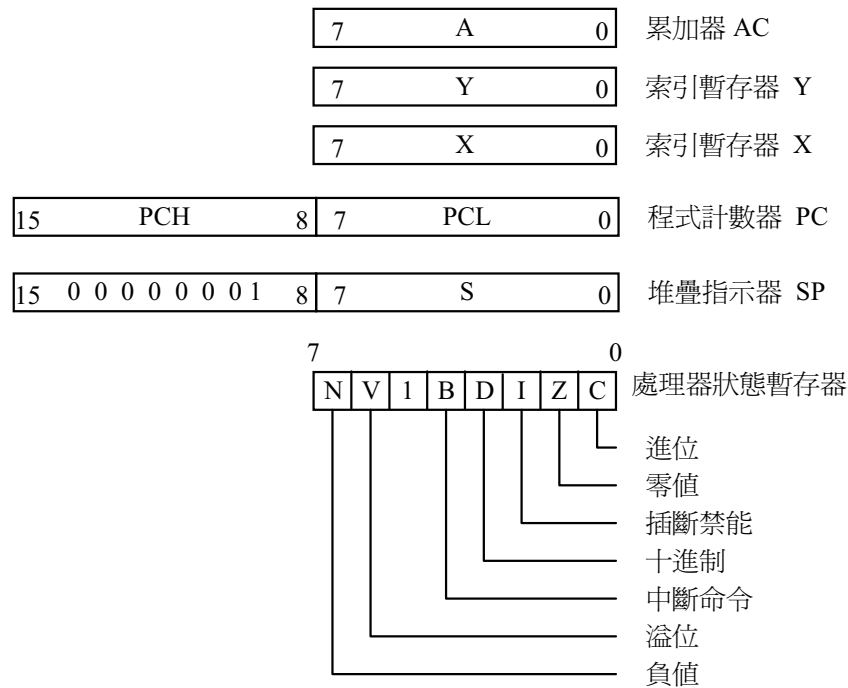


圖1-1-1 6502微處理器之暫存器架構圖

一般用途暫存器

在那6個暫存器中，累加器、X暫存器、與Y暫存器三者可供程式設計者自由運用，用以儲存臨時數據值、或與記憶器溝通、或作為計數器、或其它用途的一般用途暫存器。

累加器是唯一能直接做算術與邏輯運算之暫存器，其用以儲存加、減、AND、OR、以及EOR等運算裡其中的一運算元，以及運算後所得之結果。某些指令亦能將累加器之內含值右移或左移一位。而X與Y暫存器主要作為索引暫存器，以存取記憶器中循序之數據值。而一些指令可將其內含值加一或減一，因此，其亦常被設為一般用途的計數器。

狀態暫存器

第四個要討論的暫存器是”處理器之狀態暫存器”，如圖1-1-2所示，6502微處理器之狀態暫存器含有七個可用之位元。其中五個位元為狀態旗號(status flags)，提供先前指令(大多為前一指令)執行結果之有關訊息，另兩個位元則為控制位元(第2位元— I與第3位元— D)。

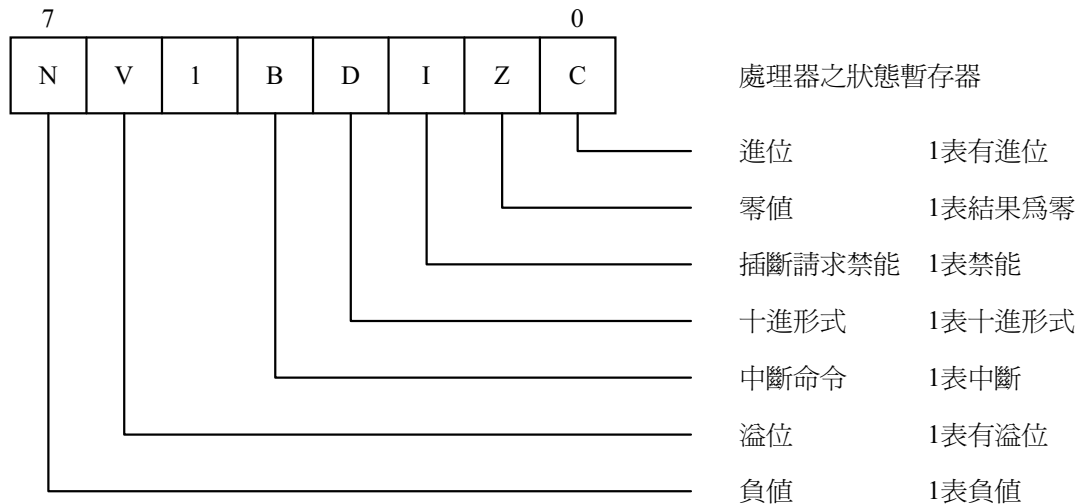


圖1-1-2 6502之狀態暫存器

- 進位旗號(C)**— 用以儲存加法運算所產生之任何進位，減法運算所產生之借位，或移位運算時被移出運算元之位元。進位旗號同時亦反應比較運算之結果。
- 零值旗號(Z)**— 用以顯示運算所得之結果是否為零。若是，則該旗號值置定為1；否則，其值為0。
- 中斷旗號(B)**— 顯示6502微處理器所接收之插斷請求(interrupt request)，究竟是“中斷”指令所引起，抑是外部設備所產生之插斷。
- 溢位旗號(V)**— 僅用於有號數之算數運算。若兩同號數相加或兩異號數相減，產生大於 $+127_{10}$ 或 -128_{10} 之結果，則溢位旗號自動置定為1。
- 負值旗號(N)**— 顯示有號數之算術運算是否產生一負數結果。若是則該旗號之值為1；否則其值為0。事實上，其亦可用以顯示累加器最高位元的內含值。

6502具有數個可檢測上述狀態旗號的指令。根據這些檢測結果，6502用以決定執行兩個可能指令系列中之一項。所有的檢測運算，均不會影響旗號原先的內含值。因此並非所有6502的指令均影響旗號。

至於狀態暫存器之控制元— **插斷請求禁能位元**(IRQ Disable bit，以I代表)及**十進制控制位元**(Decimal mode bit，以D代表)：

- 插斷請求禁能位元(I)**— 在某些原因微處理器不準備作插斷服務時，該位元可用以拒絕外部設備所提之插斷請求；當重設發生或 6502 正在**服務**前一個插斷時，插斷請求自動失效。當某些指令系列必須在不准被插斷的情形下執行時，程式也可用此位元將外部提出的插斷請求禁能。
- 十進制控制位元(D)**— 該位元用以控制6502之算數邏輯單元(Arithmetic Logic Unit，縮寫為ALU)，使其運作像直接二進加法器，或十進數加法器。於二進形式，ALU將運算元視為8-bit的直接二進數。於十進形式，ALU則將運算元視為一含有兩個BCD (Binary-Coded Decimal) 數字的十進數；換言之算術運算之結果同於平常十進算術之結果。若十進控制位元(D)之值為1，則6502之ALU即動作於十進形式。

堆疊指示器

第五個討論的暫存器為“**堆疊指示器(SP)**”。6502微處理器在執行程式至某一點時，可使其轉移至另一記憶位置上的指令系列繼續執行。在轉移真正發生之前，6502存起目前指令系列之次一欲執行指令的位址。該位址即稱為**回返位址**(return address)。當新的指令系列執行完後，控制會再**回返**至此一先前離開之位置。回返位址即儲存在一稱為**堆疊器(stack)**之特殊記憶區。

堆疊器就是專用以接收那些回返位址之某一部份記憶位置。而為了能清楚堆疊器內各位置使用之情形，就像程式計數器指至記憶器次一欲拿取指令一般，**堆疊指示器(SP)**被用以儲存堆疊器中，次一個可用之記憶位置的位址。在6502內，堆疊器設於記憶器之第一頁(位址0100至01FF之記憶位置)。由於此一限制確定堆疊位址之高次元組恆為01(十六進)，因此6502之堆疊位址暫存器，即堆疊指示器，僅長八個位元。

資料進出6502堆疊器之方式，就如同我們取放堆積桌面的文件一般；最後被放入堆疊器之資料項目，即為第一個被取出者。為此，堆疊器經常被稱為“後進先出”(last in, first out)堆疊器。由於6502之堆疊是朝著位址0之方向成長，因此，推入堆疊器之回返位址，相繼被儲存於讀寫記憶器中位址漸小的記憶位置。每次有一新的八位元位址被**堆入**(push)堆疊器，堆疊指示器之內含值就自動減一，以指到下一緊接之“空”位置。而每當有一八位元位址從堆疊器被**拉取**(pull)，則堆疊指示器之內含值就自動加一。記得，每一十六位元之位址必須以兩個記憶位置加以儲存。必要時，累加器與狀態暫存器的內含值亦可存入堆疊器。

程式計數器

最後要討論的暫存器為“**程式計數器(program counter)**”，用以追蹤指令之執行。程式計數器之內含值永遠等於微處理器下一個要執行之指令所在記憶器的位址。因此，只要根據程式計數器之內容，微處理器即可正確地拿取所要執行之指令。每在微處理器拿取一指令後，程式計數器的內含值即會自動加1，指至下一個指令。由於6502之位址有十六位元寬，是以，其程式計數器的長度亦為十六位元，以便能指到記憶空間內之任一位置。

1-1-2 重設與插斷訊號

有三個輸入信號可供外部設備加以插斷(interrupt)6502微處理器正在執行的程式。這些訊號為一重設信號($\overline{\text{RES}}$)、插斷請求信號($\overline{\text{IRQ}}$)、與不可罩蓋之插斷信號($\overline{\text{NMI}}$)。

- ① **重設($\overline{\text{RES}}$)**— 用以令6502微處理器回復至某一已知狀態，或於電源剛加上時起動6502微處理器。 $\overline{\text{RES}}$ (low)動作時，6502微處理器既不能傳輸訊號亦無法接收資訊。 $\overline{\text{RES}}$ 信號變高準位時，微處理器就會將位址FFFC與FFFD兩記憶位置之內含值，載入程式計數器。該兩記憶位置之內含，稱為**重設向量(reset vector)**，即為微處理器重設後，第一個欲執行之指令位址。
- ② **插斷請求(Interrupt ReQuest) $\overline{\text{IRQ}}$** — 為大多數週邊設備對6502微處理器提出插斷服務請求的輸入源；此地之關鍵字為“請求”。它不像重設信號每次皆能無條件地插斷微處理器，插斷請求信號僅是通知微處理器—系統中有某些設備(如鍵盤、印字機等)正等待著接收或送出資訊。插斷請求唯有當狀態暫存器之插斷禁能位元(I)被清除為0時，才被接受。6502微處理器在接受插斷請求時，會自動將記憶器最頂端的兩記憶位置(十六進位址FFFE與FFFF)之內含值載入程式計數器。此時被存入程式計數器之數值，即稱為**插斷向量(interrupt vector)**。若收到插斷請求時，插斷禁能旗號之值為1，則微處理器會自動忽視該插斷請求，且繼續執行其原來程式。6502並不會“記住”插斷請求，不過，若“I”位元被清除為0且插斷請求信號仍在時，則6502會立即接受插斷。
- ③ **不可罩蓋插斷(Non-Maskable Interrupt) $\overline{\text{NMI}}$** — $\overline{\text{IRQ}}$ 輸入是**可罩蓋的(maskable)**；亦即，視狀態旗號I之值而定，其可被致能或禁能。 $\overline{\text{NMI}}$ 則為不可罩蓋，其不能被禁能。正如 $\overline{\text{RES}}$ 一般， $\overline{\text{NMI}}$ 不僅對微處理器提出插斷請求，而且每次**必能**插斷微處理器。主要設計以在必須引起立即注意之情況(如停電)下，插斷微處理器。在6502不可罩蓋插斷之處理程式的起始位址，固定存於位址FFFA與FFFB(十六進)兩個連續記憶位置。圖1-1-3所示即為6502微處理器之重設與插斷向量的擺放：

| 位址 | 內含 |
|------|---------|
| ⋮ | ⋮ |
| FFFA | NMI - L |
| FFFB | NMI - H |
| FFFC | RES - L |
| FFFD | RES - H |
| FFFE | IRQ - L |
| FFFF | IRQ - H |

圖1-1-3 6502之向量位址

第1-2章 6502指令集

6502微處理器具有56種不同的指令，以及13種定址模式(addressing mode)。這點使它成為被廣泛應用的微處理器之一。本章將分別介紹此13種定址法與56種不同指令，並說明如何用這些指令構成簡單的程式。

1-2-1 指令格式

所有程式均寫成”組譯程式(assembler)”的格式。此種格式將程式的每一行(line)分成四個欄(field)：標題欄，運算碼欄，運算元欄，與註解欄。

| 格式： | 標題欄 | 運算碼欄 | 運算元欄 | 註解欄 |
|-----|--------|------|------|-----------|
| | KEYIN3 | LDY | #00 | ;用以取得元件數目 |

- ① **標題欄(label field)**用以賦予指令所在位置一符號名稱，使程式中的其它指令能參照或指到此一指令。標題欄是選用的(optional)，而事實上，大多數指令都是沒有加標題的(標題欄空白)。不過，若指令欲加標題，則必須遵守標題撰寫的一些規範：(a)標題必須由每一行的最左邊一格開始寫起。(b)每一個標題必須以英文大寫字母(A至Z)開頭。(c)而且長度不能超過六個文(數)字。(d)每一指令的標題必須不同。例如，JMP KEYIN3 指令執行的結果，將使標題 KEYIN3 所對應的記憶位址，無條件地存入程式計數器。使得下一個緊接被執行的指令，即為標題 KEYIN3 的指令。
- ② **運算碼欄(op code field)**，在含有指令的每一行都必須具備，而且運算碼欄必須是6502所含的56個有效助憶符號之一(參見1-2-2節)。除了第一格(column 1)不行外，運算碼欄可由任一格開始寫起。若指令有標題，則助憶符號與標題之間，至少必須空一格。
- ③ **運算元欄(operand field)**用以指明指令所需的數據或運算元位址(有需要時)。運算元與運算碼之間至少亦須空一格。組譯程式能接受五種形式的運算元，各種形式的運算元是以在運算元前加一個字首予以區別，這些字首為：

| 字首 | 運算元形式 |
|----|------------|
| 空白 | 基底十(十進數) |
| \$ | 基底十六(十六進數) |
| @ | 基底八(八進數) |
| % | 基底二(二進數) |
| ' | ASCII |

- ④ **註解欄(comment field)**用以對各指令作一簡要解釋，以方便程式工程師追蹤、或偵錯之用。此欄可以保持空白。由於微處理器不執行註解欄，因此，程式設計者可在該欄加上任何形式的註解。但是記得，註解本文之前必須加一分號(;). 註解亦可獨立自成一行，而不須跟在指令之後。

1-2-2 指令運算碼陣列

我們將6502微處理器所有的指令及其運算碼(Op code)做一摘要，以方便使用者參閱。

| MSD | LSD | | | | | | | | | | | | | | | MSD |
|-----|-----------------|---------------------|-----------------|---|-------------------|-------------------|-------------------|-------------|------|---------------------|---------------|---|-------------------|---------------------|---------------------|-----|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 0 | BRK 7, 1 | ORA ind,X 6, 2 | | | | ORA zpg 3, 2 | ASL zpg 5, 2 | PHP 3, 1 | 3, 1 | ORA imm 2, 2 | ASL A 2, 1 | | | ORA abs 4, 3 | ASL abs 6, 3 | 0 |
| 1 | BPL rel 2, 2 | ORA ind,Y 5, 2 ※ | | | | ORA zpg,X 4, 2 | ASL zpg,X 6, 2 | CLC 2, 1 | 2, 1 | ORA abs,Y 4, 3 ※ | | | | ORA abs,X 4, 3 ※ | ASL abs,X 6, 3 | 1 |
| 2 | JSR abs 6, 3 | AND ind,X 6, 2 | | | BIT zpg 3, 2 | AND zpg 3, 2 | ROL zpg 5, 2 | PLP 4, 1 | 4, 1 | AND imm 2, 2 | ROL A 2, 1 | | BIT abs 4, 3 | AND abs 4, 3 | ROL abs 6, 3 | 2 |
| 3 | BIT rel 2, 2 | AND ind,Y 5, 2 ※ | | | | AND zpg,X 4, 2 | ROL zpg,X 6, 2 | SEC 2, 1 | 2, 1 | AND abs,Y 4, 3 ※ | | | | AND abs,X 4, 3 ※ | ROL abs,X 6, 3 | 3 |
| 4 | RTI 6, 1 | EOR ind,X 6, 2 | | | | EOR zpg 3, 2 | LSR zpg 5, 2 | PHA 3, 1 | 3, 1 | EOR imm 2, 2 | LSR A 2, 1 | | JMP abs 3, 3 | EOR abs 4, 3 | LSR abs 6, 3 | 4 |
| 5 | BVC rel 2, 2 | EOR ind,Y 5, 2 ※ | | | | EOR zpg,X 4, 2 | LSR zpg,X 6, 2 | CLI 2, 1 | 2, 1 | EOR abs,Y 4, 3 ※ | | | | EOR abs,X 4, 3 ※ | LSR abs,X 6, 3 | 5 |
| 6 | RTS 6, 1 | ADC ind,X 6, 2 | | | | ADC zpg 3, 2 | ROR zpg 5, 2 | PLA 4, 1 | 4, 1 | ADC imm 2, 2 | ROR A 2, 1 | | JMP ind 6, 3 | ADC abs 4, 3 | ROR abs 6, 3 | 6 |
| 7 | BVS rel 2, 2 | ADC ind,Y 5, 2 ※ | | | | ADC zpg,X 4, 2 | ROR zpg,X 6, 2 | SEI 2, 1 | 2, 1 | ADC abs,Y 4, 3 ※ | | | | ADC abs,X 4, 3 ※ | ROR abs,X 6, 3 | 7 |
| 8 | | STA ind,X 6, 2 | | | STY zpg 3, 2 | STA zpg 3, 2 | STX zpg 3, 2 | DEY 2, 1 | 2, 1 | | TXA 2, 1 | | STY abs 4, 3 | STA abs 4, 3 | STX abs 4, 3 | 8 |
| 9 | BCC rel 2, 2 | STA ind,Y 6, 2 | | | STY zpg,X 4, 2 | STA zpg,X 4, 2 | STX zpg,Y 4, 2 | TYA 2, 1 | 2, 1 | STA abs,Y 5, 3 | TXS 2, 1 | | | STA abs,X 4, 3 | | 9 |
| A | LDY imm 2, 2 | LDA ind,X 6, 2 | LDX imm 2, 2 | | LDY zpg 3, 2 | LDA zpg 3, 2 | LDX zpg 3, 2 | TAY 2, 1 | 2, 1 | LDA imm 2, 2 | TAX 2, 1 | | LDY abs 4, 3 | LDA abs 4, 3 | LDX abs 4, 3 | A |
| B | BCS rel 2, 2 | LDA ind,Y 5, 2 ※ | | | LDY zpg,X 4, 2 | LDA zpg,X 4, 2 | LDX zpg,Y 4, 2 | CLV 2, 1 | 2, 1 | LDA abs,Y 4, 3 ※ | TSX 2, 1 | | LDY abs,X 4, 3 | LDA abs,X 4, 3 | LDX abs,Y 4, 3 ※ | B |
| C | CPY imm 2, 2 | CMP ind,X 6, 2 | | | CPY zpg 3, 2 | CMP zpg 3, 2 | DEC zpg 5, 2 | INY 2, 1 | 2, 1 | CMP imm 2, 2 | DEX 2, 1 | | CPY abs 4, 3 | CMP abs 4, 3 | DEC abs 6, 3 | C |
| D | BNE rel 2, 2 | CMP ind,Y 5, 2 ※ | | | | CMP zpg,X 4, 2 | DEC zpg,X 6, 2 | CLD 2, 1 | 2, 1 | CMP abs,Y 4, 3 ※ | | | | CMP abs,X 4, 3 ※ | DEC abs,X 6, 3 | D |
| E | CPX imm 2, 2 | SBC ind,X 6, 2 | | | CPX zpg 3, 2 | SBC zpg 3, 2 | INC zpg 5, 2 | INX 2, 1 | 2, 1 | SBC imm 2, 2 | NOP 2, 1 | | CPX abs 4, 3 | SBC abs 4, 3 | INC abs 6, 3 | E |
| F | BEQ rel 2, 2 | SBC ind,Y 5, 2 ※ | | | | SBC zpg,X 4, 2 | INC zpg,X 6, 2 | SED 2, 1 | 2, 1 | SBC abs,Y 4, 3 ※ | | | | SBC abs,X 4, 3 ※ | INC abs,X 6, 3 | F |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

BRK imm
7, 1

7, 1 = 執行週期, 位元組
 imm = 立即 zpg = 零頁
 rel = 相對 abs = 絕對 ind = 間接

↑ 若為十進制模式，則加1 到N
 ※若超越頁界限，則加1到N
 ◆若分支跳越發生在同一頁，則加1到N
 若分支跳越發生在不同頁，則加2到N

1-2-3 載入(LOAD)與存出(STORE)指令

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|--------------|--------|--------|---------|
| LDA imm | A9xx | 2/2 | N, Z |
| LDA abs | ADaabb | 4/3 | N, Z |
| LDA zpg | A5aa | 3/2 | N, Z |
| LDA abs,x | BDAabb | 4/3 | N, Z |
| LDA abs,y | B9aabb | 4/3 | N, Z |
| LDA zpg,x | B5aa | 4/2 | N, Z |
| LDA (zpg), y | B1aa | 5/2 | N, Z |
| LDA (zpg,x) | A1aa | 6/2 | N, Z |
| LDX imm | A2xx | 2/2 | N, Z |
| LDX abs | AEaabb | 4/3 | N, Z |
| LDX zpg | A6aa | 3/2 | N, Z |
| LDX abs,y | BEaabb | 4/3 | N, Z |
| LDX zpg, y | B6aa | 4/2 | N, Z |
| LDY imm | A0xx | 2/2 | N, Z |
| LDY abs | ACaabb | 4/3 | N, Z |
| LDY zpg | A4aa | 3/2 | N, Z |
| LDY abs,x | BCaabb | 4/3 | N, Z |
| LDY zpg,x | B4aa | 4/2 | N, Z |

6502的結構設計以記憶體為導向，因此，其最基本的運算即為”將資料移入或移出記憶體”。這些傳遞將涉及三個暫存器：累加器(A暫存器)，X暫存器，與Y暫存器。

將資訊由記憶體傳至微處理器內某一暫存器的動作，稱為**載入(load)**。6502有三個載入指令。

| 指令 | 說明 |
|-----|------------|
| LDA | 由記憶體載入累加器A |
| LDX | 由記憶體載入X暫存器 |
| LDY | 由記憶體載入Y暫存器 |

資料存放之記憶位置的內含值不受載入指令執行之影響。但是，狀態暫存器中有兩個旗號會受到影響，以反映所載入之值的狀況。若載入之值的第七位元(最高次位元)為1，則負數旗號(N)自動置定為1；反之，若第七位元為0，則負數旗號自動清除為0。另外，零值旗號(Z)會自動被置定為1，當載入之值為0時；否則，零值旗號為0。

例： LDA \$12C3

將位址\$12C3之記憶位置內含值，抄入累加器。若該位置原來記憶之內含為\$33，則指令執行完成後，累加器之內含值亦變為\$33。

將微處理器內某一暫存器的內含值，傳送至記憶體某一記憶位置的過程，即稱之為**存出(store)**。就資料被傳遞之方向而言，存出與載入相反。6502微處理器有三個存出指令：

| 指令 | 說明 |
|-----|------------|
| STA | 由累加器存出記憶體 |
| STX | 由X暫存器存出記憶體 |
| STY | 由Y暫存器存出記憶體 |

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|--------------|--------|--------|---------|
| STA abs | 8Daabb | 4/3 | |
| STA zpg | 85aa | 3/2 | |
| STA abs,x | 9Daabb | 5/3 | |
| STA abs,y | 99aabb | 5/3 | |
| STA zpg,x | 95aa | 4/2 | |
| STA (zpg), y | 91aa | 6/2 | |
| STA (zpg,x) | 81aa | 6/2 | |
| STX abs | 8Eaabb | 4/3 | |
| STX zpg | 86aa | 3/2 | |
| STX zpg, y | 96aa | 4/2 | |
| STY abs | 8Caabb | 4/3 | |
| STY zpg | 84aa | 3/2 | |
| STY zpg,x | 94aa | 4/2 | |

存出指令執行的結果，既不影響來源暫存器(A，X，或Y)原先之內含值，亦不影響狀態暫存器。將某一數值存進於記憶體的最簡單方式，就是載入立即值與存出的組合，如

```
LDA    #33
STA    $71
```

上述兩指令執行的結果，就是存出#33之值到零頁記憶位置\$0071去。

1-2-4 算術指令

6502微處理器具有能對直接二進數或二進寫碼十進數(BCD)作加或減運算式的指令。所有加減運算式均牽涉兩個運算元，一個在累加器內，另一個則存於記憶體或為指令之第二位元組(若使用立即定址模式的話)——參閱1-2-12的定址模式。

數目之表示

6502微處理器可用以加減有號數與無號數。雖然這兩種數目於電子電路上之運算對微處理器而言毫無區別，但是程式設計者本身必須了解如何去解釋。

對一**無號數(unsigned number)**而言，資料之每一位元均帶有一權數。權數大小依位置所在而定。就一8-bit資料而言，由右至左，資料位元稱為第0位元(以 b_0 表示)、第一位元(以 b_1 表示)、...、第七位元(以 b_7 表示)等等。各位元位置所具有之權數，與位元位置有直接關係。譬如，第一位元位置之權數則為 2^1 ，而第七位元位置之權數則為 2^7 。

由此可見，一個位元組之資料可代表的無號數，最小為0(每位元均為0時)，而最大為十進數255(每位元均為1時) $\rightarrow (128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255)$ 。

在**有號數(signed number)**，低次七個位元代表數值，權數之大小如同無號數，而最高次位元(b_7)被用以代表數目之符號。若數目為正則 b_7 為0；反之，若數目為負則 b_7 為1。如此，八位元所能代表之正數由十進數0(00000000)至十進數+127(01111111)，而所能代表之負數則由十進數-1(11111111)至十進數-128(10000000)。

| b_7 | b_6 | b_5 | b_4 | b_3 | b_2 | b_1 | b_0 | 位元位置 |
|--|-------|-------|-------|-------|-------|-------|-------|---------------|
| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 資料 |
| 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | 權數 |
| $1*128 + 0*64 + 1*32 + 0*16 + 0*8 + 1*4 + 1*2 + 0*1 = 166$ | | | | | | | | 權數之等效十進值(無號數) |
| $1(\text{負}) \quad 0*64 + 1*32 + 0*16 + 0*8 + 1*4 + 1*2 + 0*1 = -38$ | | | | | | | | 權數之等效十進值(有號數) |

為何-1是以11111111表示，而非以10000001表示？這問題的答案是，這裡的負數是以**2的補數(2's complement)**形式來表示，而非以符號的絕對值形式來表達。使用2的補數之主要原因，是爲了避免有兩個零發生：00000000(正零)與10000000(負零)。

在2的補數形式，零將只有一種表達方式：即00000000。欲獲得一個二進制之**2的補數形式**的負數甚爲容易；只要將該數的正數形式每位元逐次反向(1變0，0變1)，之後再加1即得。下述例子說明了如何以二進制之2的補數形式表達 -16_{10} (2的補數形式)：

| | |
|----------|--------------------|
| 00010000 | $+16_{10}$ |
| ↓ | |
| 11101111 | 取"1補數" |
| + | 加1 |
| 1 | |
| <hr/> | |
| 11110000 | -16_{10} (2補數形式) |

十進算術指令

6502微處理器具有某些特別指令，可以使其內部之算術邏輯單元(ALU)，動作為二進加法器或為十進加法器。在動作為二進加法器時，ALU將兩運算元看成是其值介於00000000至11111111之間的二進數。而在動作成十進加法器時，ALU將兩運算元分別視成為BCD數目，每一個八位元的運算元含有兩個十進數字。

有兩個指令可用以控制ALU的二進加法及十進加法：**SED**(Set Decimal Mode，置為十進形式)與**CLD**(Clear Decimal Mode，去除十進形式)。**SED**指令使ALU動作成十進 (BCD)加法器——將八位元的運算元視成兩位的BCD數目。**SED**指令同時亦使狀態暫存器之“十進制”控制位元(D)置定為1。**CLD**指令，將十進式控制位元D清除為0，使ALU恢復動作成二進加法器。在電源一打開時，十進制控制位元(D)的狀態未定，因此，必須由系統之起始程式(initialization program)加以定義——將之設定為1或清除為0。

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----|-----|--------|---------|
| CLD | D8 | 2/1 | D=0 |
| SED | F8 | 2/1 | D=1 |

加法

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-------------|--------|--------|------------|
| ADC imm | 69xx | 2/2 | N, V, Z, C |
| ADC abs | 6Daabb | 4/3 | N, V, Z, C |
| ADC zpg | 65aa | 3/2 | N, V, Z, C |
| ADC abs,x | 7Daabb | 4/3 | N, V, Z, C |
| ADC abs,y | 79aabb | 4/3 | N, V, Z, C |
| ADC zpg,x | 75aa | 4/2 | N, V, Z, C |
| ADC (zpg),y | 71aa | 5/2 | N, V, Z, C |
| ADC (zpg,x) | 61aa | 6/2 | N, V, Z, C |

大多數微處理器均有兩個加法指令——一個直接將兩運算元相加，另一個尚包括進位。前者通常直接稱之為**加法(ADD)**指令，用於兩個單位元組之數目的相加或多位元組數目之最低次位元組的相加；後者則稱為**進位加法 (Add with Carry, ADC)**指令，用於長度在兩個位元組以上數目之較高次位元組的相加。

應用上，大多數的加法運算均為多位元組數目(multi-byte number)的相加，因此，6502僅具備了進位加法(ADC)的指令。

在符號上，ADC指令的運算通常表示成：

$$A = A + M + C$$

式中，

A代表累加器之內含值

M代表某一記憶位置之內含值(或立即運算元)

C代表進位旗號之值

此即，將累加器之內含值加上指令運算元，再加上進位旗號之值，最後令結果成為累加器的新內含值。因此，若ADC指令正在執行時，進位旗號C之值為1，則上述式子實際變成：

$$A = A + M + 1$$

同樣地，若ADC指令正在執行時，進位旗號之值為0，則上述式子實際變為

$$A = A + M + 0$$

那麼，兩個單位元組之數目，或多位元組數目之最低次位元組如何相加呢？在這兩種情況下，進位旗號須先清除為零，然後再以ADC指令相加。將進位旗號重設為零的指令為CLC (Clear Carry flag)。由於需先清除進位，因此，兩單位元組數目的相加通常以：

CLC

ADC \$71

之形成達成。上述兩個指令，將零頁記憶位置\$0071之內含值加至累加器。

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----|-----|--------|---------|
| SEC | 38 | 2/1 | C=1 |
| CLC | 18 | 2/1 | C=0 |

有一種情況下不須在ADC指令之前，加上一CLC指令。舉例來說，假如欲加入累加器之值是立即運算元(如ADC #33)，而進位旗號又已被前面運算置定為1，則此時只要暗地使用比原來之值小1的立即運算元，即使不用CLC指令，亦可達成相同之目的。例如，若累加器之值欲加33，而事先又已知進位旗號之值必為1，則寫一指令如，

ADC #32

亦可達到相同的效果。不過，說實在的，此種方法並不好。除了使用絕對定址法(不論索引或非索引)外——“參閱1-2-12定址模式”，其它所有的ADC指令均僅佔兩個位元組。

ADC指令執行的結果會影響下面四種旗號：

- ① 進位旗號(C)。若二進位加法之和大於十進數255(\$FF)，或BCD算術之和大於十進數99，則進位旗號設定為1；否則，其值為0。
- ② 零值旗號(Z)。若和為零，則零值旗號自動置定為1；否則，該旗號之值為0。
- ③ 負值旗號(N)。兩有號數相加，若運算結果的第七位元為1，則負值旗號之值為1；否則其值為零。當N旗號之值為1時表示所得結果為負數，為0則表示所得之和為正數。
- ④ 溢位旗號(V)。有號數之兩同號數相加，若結果大於 $+127_{10}$ 或小於 -128_{10} ，使得累加器之第七位元值改變，則溢位旗號自動設定為1；否則，溢位旗號之值為0。

加法運算中，N與V兩旗號只與有號數相加時有關；就加法運算而言，N與V旗號也唯有在有號數相加時，才有意義。多位元組數目的相加，只要在最低次位元組之前將進位清除，即可開始連續執行一系列的LDA(載入)，ADC(相加)，與STA(存出)指令。下面的例子為“兩個雙倍長(16-bit)之加法常式”範例：

```

        ; 該常式將兩個十六位元之數目相加。一個數目儲存在
        ; $30與$31之記憶位置，另一數目則儲存在$32與$33之
        ; 記憶位置。兩數之和再存回$30與$31兩位置。
DPADD   CLC                               ; 清除進位旗號。
        LDA  $30                           ; 先加低次位元組。
        ADC  $32
        STA  $30
        LDA  $31                           ; 再加高次位元組。
        ADC  $33
        STA  $31

```

減法

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|---------------|--------|--------|------------|
| SBC imm | E9xx | 2/2 | N, V, Z, C |
| SBC abs | EDaabb | 4/3 | N, V, Z, C |
| SBC zpg | E5aa | 3/2 | N, V, Z, C |
| SBC abs,x * | FDaabb | 4/3 | N, V, Z, C |
| SBC abs,y * | F9aabb | 4/3 | N, V, Z, C |
| SBC zpg,x | F5aa | 4/2 | N, V, Z, C |
| SBC (zpg),y * | F1aa | 5/2 | N, V, Z, C |
| SBC (zpg,x) | E1aa | 6/2 | N, V, Z, C |

減法指令亦有兩種——一個直接將兩運算元相減，另一個尚包括借位。前者通常直接稱之為**減法(SUB)**指令，用於兩個單一位元組之數目的相減，或多位元組數目之最低次位元組的相減；後者則稱為**借位減法(SuBtract with Carry，以SBC表示)**指令，用以作兩多位元組數目較高次位元組的相減。

應用上，大多數的減法運算均為多位元組數目(multi-byte number)的相減，因此，6502僅具備了借位減法(SBC)的指令。由於 SBC指令恆將借位一併減去，因此必須注意下面的原則：在減法運算之前，進位旗號(C)必須先置定為1，或將之計入減法之內。SEC(SET Carry flag)指令可用以將進位旗號設定為1。

先令進位旗號為1，如果遇到作單位元組之減法時，是否會出現問題？不會，因為指令將進位旗號之補數(而非其真值本身)，當成借位之值，因此，事先將進位旗號置定為1，恰可使借位之值為零——無借位。在符號上，SBC指令的運算通常表示成

$$A = A - M - \bar{C}$$

式中，

A表累加器之內含值

M表某一記憶位置之內含值(或立即運算元)

C表進位旗號之值。

上述式子之意義為，將累加器內含值，減去指令運算元之值，再減去進位旗號的互補值，最後結果再令為累加器新的內含值。

因為進位旗號設定之需求，兩單位長數目的相減，可以SEC與SBC兩連續指令達成。如：

SEC

SBC \$71

即將累加器之內含值減去位址\$0071之記憶位置的內含值，結果(即兩數之差)再存回累加器。若使用零頁或立即定址，則SBC指令佔兩個位元組；但若使用絕對定址，則必須以連續三個記憶位置加以儲存。

SBC指令執行的結果會影響狀態暫存器的四個旗號：

- ① 進位旗號(C)。若相減結果為正數或零，則進位旗號自動設定為1。若相減結果所得為負數，則進位旗號之值為0(顯示借位)
- ② 零值旗號(Z)。若運算所得結果為零，則零值旗號為1；否則，零值旗號為0。注意，若進位與零值旗號同時為1，則結果為零。若進位旗號為1而零值旗號為0，則結果(即兩數之差)為正數。
- ③ 負數旗號(N)。兩有號數相減，若所得結果之第七位元為1，則負數旗號為1；否則其值為0。當N旗號為1表差為負數，N旗號為0表差為正數。
- ④ 溢位旗號(V)。有號數之兩異號(一正一負)數相減，若結果大於 $+127_{10}$ 或小於 -128_{10} ，則溢位旗號之值為1。此種情況使累加器之第七位元於運算中變值。

多倍長數目的相減，僅需在最低位元組開始減之前，將進位旗號置定為1，然後再執行一系列的LDA，SBC，與STA指令即可。每次為一個位元組相減。如下為“兩個雙倍長(16-bit)減法常式”之範例：

```

; 此一常式將儲存於$0030與$0031兩記憶位置的十六位元
; 數目，減去儲存於$0032與$0033兩記憶位置之十六位元
; 數目，並將結果存於$0030與$0031之位置。
; $0030與$0032兩位置分別儲存兩數目之低次位元組。
DPSUB      SEC                      ; 令進位旗號為1。
           LDA   $30                ; 載入被減數之低次位元組。
           SBC   $32                ; 低次位元組相減。
           STA   $30                ; 存出低次位元組之差。
           LDA   $31                ; 載入被減數之高次位元組。
           SBC   $33                ; 高次位元組相減。
           STA   $31                ; 存出高次位元組之差。

```

SBC指令亦可用以求一個數目之負數(2補數形式)。欲達此目的，只要以零減去該數目之正值形式即可。例如，下面的程式即將零頁位置\$0031之內含值變號。

| | | |
|-----|------|---------------|
| SEC | | ；令進位旗號 = 1。 |
| LDA | #00 | ；累加器清除為0。 |
| SBC | \$31 | ；減去\$0031位置之內 |
| STA | \$31 | ；含，並將結果存回。 |

有號數目之算術

就真正電路的運算而言，有號數與無號數並無差異。然而，這並不代表每一加法與減法常式(前述各範例)，均可應用於兩種數目系統中的任何數目，而僅表示這些常式中的加與減技術是普遍性且適用的。

就有號數目之算術而言，其常式可能多包括一些對正數結果與負數結果作不同處理之額外指令；且產生溢位情況之運算，亦可能另作不同的處置。

運算之最後結果的符號與溢位狀態，由狀態暫存器之負數(N)與溢位(V)兩旗號加以反應。有號數目算術之常式，經常利用這兩旗號之最後狀態，決定如何進一步處置運算結果。負值旗號僅反映了所得結果之正負號，而溢位旗號則顯示累加器的內含值是有效的，還是無效的。

溢位旗號(V)唯有在累加器內之結果無效時，才會被置定為1。V旗號唯有當兩同號數相加，或兩異號數相減，所得結果超出 $+127_{10}$ 至 -128_{10} 之範圍時，才會被置定為1。一經置定後，溢位旗號可由一僅長一個位元組的特殊指令將之清除為零—CLV (Clear Overflow(V) flag)。溢位旗號在下一個ADC或SBC指令一開始，亦會被自動清除為零。

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----|-----|--------|---------|
| CLV | B8 | 2/1 | V=0 |

1-2-5 加一與減一指令

6502微處理器具有數個能將X暫存器，Y暫存器，或某一記憶位置之內含值加一或減一的指令。

暫存器之值加一/減一

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----|-----|--------|---------|
| DEX | CA | 2/1 | N, Z |
| DEY | 88 | 2/1 | N, Z |
| INX | E8 | 2/1 | N, Z |
| INY | C8 | 2/1 | N, Z |

我們曾說X與Y暫存器可用作索引暫存器。事實上，這兩個暫存器在不同的應用中，亦可作為一般用途之計數器。

由於6502微處理器具有將一般用途之X與Y暫存器的內含值加一或減一的指令，因此存取連續記憶位置變得十分容易。這些指令是：

| 指令 | 說明 |
|-----|--------------|
| DEX | 將X暫存器之內含值減1。 |
| DEY | 將Y暫存器之內含值減1。 |
| INX | 將X暫存器之內含值加1。 |
| INY | 將Y暫存器之內含值加1。 |

以上四個隱含定址指令(參閱1-2-12定址模式)均各佔一個記憶位置，需兩週期的執行時間。且它們皆影響兩個旗號：

- ① 負值旗號(N)。在加一或減一後，若暫存器所含值之b7=1，則此一旗號自動設定為 1；否則其值為0。而如果您正於一有號數系統運作的話，則此情況表示產生了一個負數結果。
- ② 零值旗號(Z)。若加一或減一運算的結果，使暫存器的內含值變為零，則此一旗號之值為 1；否則其值為0。

下面一範例顯示了一個將八個連續記憶位置(由\$0030起)之內含抄至記憶器另一部份記憶位置(由\$0300起)的常式。在此常式中，X暫存器作為索引暫存器，而Y暫存器則充當位元組計數器。最後一道指令是BNE(以前尚未提出)，不過，其主要效用就是使程式控制繼續跳回NXTBYT處，繼續搬運下一位元組，直到位元組計數器—Y暫存器遞減至零為止。

```

; 此一程式片段將由$0030起之八個記憶位置的內含
; 值，抄至由$0300起之六個記憶位置。
                LDX  #00          ; 索引暫存器之位移 = 0。
                LDY  #08          ; 位元組計數器之值 = 8。
NXTBYT          LDA  $30, X       ; 載入下一位元組。
                STA  $0300, X     ; 存出下一位元組。
                INX                ; 索引暫存器之值加一。
                DEY                ; 位元組計數器減一。
                BEN  NXTBYT       ; 再跳回做下一位元組。

```

記憶體之值加一/減一

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----------|--------|--------|---------|
| INC abs | EEaabb | 6/3 | N, Z |
| INC zpg | E6aa | 5/2 | N, Z |
| INC abs,x | FEaabb | 7/3 | N, Z |
| INC zpg,x | F6aa | 6/2 | N, Z |
| DEC abs | CEaabb | 6/3 | N, Z |
| DEC zpg | C6aa | 5/2 | N, Z |
| DEC abs,x | DEaabb | 7/3 | N, Z |
| DEC zpg,x | D6aa | 6/2 | N, Z |

在某些應用上，計數器是設立在記憶體內，而非在X或Y暫存器內，下面兩個指令使程式設計者能將某一記憶位置之內含值加一或減一，而不需使用ADC或SBC指令。它們是：

| 指令 | 說明 |
|-----|--------------|
| DEC | 將記憶位置之內含值減一。 |
| INC | 將記憶位置之內含值加一。 |

上述兩個指令皆可有四種定址法——絕對，零值，絕對索引X，與零頁索引X(參閱1-2-12定址模式)。絕對定址指令佔三個位元組，而零頁定址指令佔兩位元組。此外，這幾個指令皆影響相同之旗號——負數旗號(N)與零值旗號(Z)，但不影響進位旗號令(C)。那如果程式者要使用多位元組的記憶計數器時，該如何處理？

當多位元組的記憶計數器之內含值逐次累增時，則在決定是否需作下一更高位元組之累增時，必須檢測零值旗號，而非進位旗號；若零值旗號為1(表示現行位元組之值為0，並產生進位)，則繼續累增上去(用到下一位元組)。同樣地，若多位元組的記憶計數器之內含值必須每次遞減，則最後決定是否需作次一更高位元組之遞減前，亦需檢驗N旗號；若DEC指令已使N旗號自邏輯0狀態轉至邏輯1狀態(表示借位發生)，則繼續遞減。

1-2-6 邏輯運算指令

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|---------------|--------|--------|---------|
| AND imm | 29xx | 2/2 | N, Z |
| AND abs | 2Daabb | 4/3 | N, Z |
| AND zpg | 25aa | 3/2 | N, Z |
| AND abs,x * | 3Daabb | 4/3 | N, Z |
| AND abs,y * | 39aabb | 4/3 | N, Z |
| AND zpg,x | 35aa | 4/2 | N, Z |
| AND (zpg),y * | 31aa | 5/2 | N, Z |
| AND (zpg,x) | 21aa | 6/2 | N, Z |

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|---------------|--------|--------|---------|
| EOR imm | 49xx | 2/2 | N, Z |
| EOR abs | 4Daabb | 4/3 | N, Z |
| EOR zpg | 45aa | 3/2 | N, Z |
| EOR abs,x * | 5Daabb | 4/3 | N, Z |
| EOR abs,y * | 59aabb | 4/3 | N, Z |
| EOR zpg,x | 55aa | 4/2 | N, Z |
| EOR (zpg),y * | 51aa | 5/2 | N, Z |
| EOR (zpg,x) | 41aa | 6/2 | N, Z |

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|---------------|--------|--------|---------|
| ORA imm | 09xx | 2/2 | N, Z |
| ORA abs | 0Daabb | 4/3 | N, Z |
| ORA zpg | 05aa | 3/2 | N, Z |
| ORA abs,x * | 1Daabb | 4/3 | N, Z |
| ORA abs,y * | 19aabb | 4/3 | N, Z |
| ORA zpg,x | 15aa | 4/2 | N, Z |
| ORA (zpg),y * | 11aa | 5/2 | N, Z |
| ORA (zpg,x) | 01aa | 6/2 | N, Z |

有時我們須檢查某一記憶位置或暫存器的一個或幾個位元(而非全部八個位元)的內含值。

在這種情況下，6502微處理器之邏輯指令是非常有用。6502有三個邏輯運算指令，每一指令皆針對累加器但配合一記憶位置之內含值或一立即運算元加以運算。

| 指令 | 說明 |
|-----|--------------------------------|
| AND | 將運算元與累加器內含值作AND運算 |
| EOR | 將運算元與累加器內含值作互斥(Exclusive-OR)運算 |
| ORA | 將運算元與累加器內含值作OR運算 |

若使用零頁或立即定址，則此三個邏輯指令均佔兩個位元組長，若使用絕對定址，則每一指令會佔三個位元組(參閱1-2-12定址模式)。這三個指令均影響微處理器狀態暫存器的兩個旗號：

- ① 零值旗號(Z)。若運算結果為零，則零值旗號為1；否則其值為0。
- ② 負值旗號(N)。若運算所得結果之第七位元(b7) = 1，則該旗號之值為1；否則其值為0。

AND 指令

AND指令主要設計用以濾掉、罩蓋(mask)，或除去(清除為0)累加器內含值的某些位元，使其餘的位元能再作某種形式之處理。若記憶位置與累加器的內含值之某一位元均為1，則運算後，累加器之該位元位置的結果亦為1；否則，該位元位置均會被清除為0。表1-2-1摘要了此一AND情況。

表1-2-1 邏輯AND運算

| 記憶位元 | 累加器位元 | 累加器之結果 |
|------|-------|--------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

AND指令最適於檢測累加器某些經選擇之位元值是否為1，或遮掉某些不必用之位元。例如，表1-2-2列出了十個十進數字0至9之ASCII碼(假設最高次位元一極性位元，均為0)。

表1-2-2 文數字0至9之ASCII碼

| 文數字 | ASCII | | BCD |
|-----|-------|----------|------|
| | 十六進形式 | 二進形式 | |
| 0 | 30 | 00110000 | 0000 |
| 1 | 31 | 00110001 | 0001 |
| 2 | 32 | 00110010 | 0010 |
| 3 | 33 | 00110011 | 0011 |
| 4 | 34 | 00110100 | 0100 |
| 5 | 35 | 00110101 | 0101 |
| 6 | 36 | 00110110 | 0110 |
| 7 | 37 | 00110111 | 0111 |
| 8 | 38 | 00111000 | 1000 |
| 9 | 39 | 00111001 | 1001 |

若將每一數字ASCII碼之最高次四位元罩蓋掉(清除為0)，則所剩的低次四位元，即為各別十進數字之BCD碼。就以十進數字5為例，

| | |
|-------------------------|------------------------------|
| 5 ₁₀ 的ASCII碼 | 0 0 1 1 0 1 0 1 ₂ |
| | ↓ 罩蓋高次四位元 |
| 5 ₁₀ 的BCD碼 | 0 0 0 0 0 1 0 1 ₂ |

您透過AND邏輯指令，可以非常容易的將高次四位元罩蓋成0，譬如您將5₁₀之ASCII碼先存入累加器，然後再將之與00001111₂做AND運算，即可得到我們所要之結果了。

```
LDA    #$35
AND    #$0F
```

最後累加器之內含為00000101₂或BCD碼05。

OR 指令

ORA指令是將累加器的內含值，一位元對一位元地與指令所提供之運算元做邏輯“或”的運算。只要其中有一個位元為1，則OR運算的結果該對應位元必為1。

ORA指令最常用以將累加器的某些位元設定為1。只要令其與邏輯1作OR運算，我們即可將任意想要的位元設定為1。例如，

```
ORA    #01
```

即將累加器的最低位元設定為1，而其餘位元不變。

表1-2-3 邏輯OR運算

| 記憶位元 | 累加器位元 | 累加器之最後結果 |
|------|-------|----------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

EOR 指令

EOR (Exclusive OR)指令主要用以辨別兩個運算元中，有那些位元是彼此不同的。其亦可用以將某些選定的累加器位元值互補。在記憶位元與累加器位元兩者僅有一者為1的情況，該位元運算結果為1。其它情況，運算結果皆為0。表1-2-4所示即為EOR運算之情形。

表1-2-4 邏輯EOR運算

| 記憶位元 | 累加器位元 | 運算之累加器位元 |
|------|-------|----------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

舉個例，

EOR #\$0F

此指令將累加器的低次四位元反向(互補)，而高次四位元保持不變。EOR指令亦可用以檢測兩個數值是否相等；EOR運算時，零值旗號(Z)唯有當累加器的值與記憶位置的值完全相等時，才會被置定為1。

1-2-7 跳越，分支，比較，與位元檢測指令

前面大部份的程式範例一直是直線形的程式—指令依其在程式中出現的次序先後被執行。程式控制一直未根據執行某一個或某一串指令所得之結果，轉移至程式之其它區段。我們將討論程式控制如何把程式的某一區段(section)傳至另一區段，以及這有何用途。

跳越(JUMP)指令

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----------|--------|--------|---------|
| JMP abs | 4Caabb | 3/3 | |
| JMP (abs) | 6Caabb | 5/3 | |

您曾在閱讀某雜誌時，碰到像“文轉第10頁”的指令嗎？那就是一種跳越運算。6502微處理器之跳越指令，使得次一被執行的指令來自記憶器之其它地方，而非下一個緊接的記憶位置。同樣的，6502之跳越指令是**無條件的(unconditional)**；只要6502 在執行程式時碰上跳越指令，則每次跳越動作必發生。

6502跳越指令之助憶符號是**JMP**。JMP指令之運算元通常為一標題(label)。因此，您在程式中看到JMP指令出現的情形通常像：

```

:
  ADC  $30
  INX
  JMP  THERE
HERE  STA  $33
:
:
THERE SEC
      SBC  $32
:

```

例中，程式跳至標題為THERE之指令，執行SEC，然後執行次一指令SBC \$32，再緊接其他指令...等等。那麼標題為HERE之指令STA \$33有機會被執行嗎？是的，不過唯有在程式其它地方有一指令，將控制轉移至此一位置時，方有可能被執行。

JMP指令用以越過一群在某些條件下，或在程式之其它部份才被執行之指令。此一指令可以使用絕對定址或間接絕對定址(參閱1-2-12定址模式)。由於這兩種方式均為絕對的，因此，跳越指令之跳越範圍可及記憶器之任一位置。

JMP指令在記憶器內佔三個記憶位置(三個位元組)。絕對定址之JMP指令，需要三個週期之執行時間，而間接絕對定址之JMP指令，則需五個週期之執行時間。此外，JMP指令不影響狀態暫存器之任何旗號。

分支(BRANCH)指令

如同跳越指令一般，分支指令使程式控制轉移至某一特定的記憶位置。所不同的是，JMP指令將控制轉移至記憶器之某一**絕對位址**，而分支指令將控制轉移至一**相對位址**——次一指令之前或之後多少位置。

此外，JMP指令與分支指令的另一不同點是，分支指令為**作決策** (decision-making)指令，或**條件跳越**(conditional jump)指令。每一分支指令均檢查狀態暫存器的某一旗號。

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|------------|------|--------|---------|
| BCS rel ** | B0rr | 2/2 | |
| BCC rel ** | 90rr | 2/2 | |
| BEQ rel ** | F0rr | 2/2 | |
| BNE rel ** | D0rr | 2/2 | |
| BMI rel ** | 30rr | 2/2 | |
| BPL rel ** | 10rr | 2/2 | |
| BVS rel ** | 70rr | 2/2 | |
| BVC rel ** | 50rr | 2/2 | |

若現有旗號之狀態滿足分支指令所列之條件，則跳越運算發生；否則，微處理器繼續執行次一緊接之指令。表1-2-5即為6502微處理器所有分支指令以及其所檢查之旗號的一覽表。

表1-2-5 分支指令

| 指令 | 說明 | 跳越發生若 |
|-----|---------------------------------|----------|
| BCC | Branch on Carry Clear | 旗號 C = 0 |
| BCS | Branch on Carry Set | C = 1 |
| BEQ | Branch on result Equal zero | Z = 1 |
| BNE | Branch on result Not Equal zero | Z = 0 |
| BMI | Branch on result Minus | N = 1 |
| BPL | Branch on result Plus | N = 0 |
| BVS | Branch on overflow Set | V = 1 |
| BVC | Branch on overflow Clear | V = 0 |

由表1-2-5中可看出，分支指令是基於狀態暫存器之四種旗號的狀態加以處理——進位旗號(C)，零值旗號(Z)，負值旗號(N)，及溢位旗號(V)。

在知道分支指令是以相對於分支指令所在位置定義出跳越之目的位址後，您可能已猜出，這些指令僅使用一種相對定址法(參閱1-2-12定址模式)。由於此一原故，分支指令跳越所能及之範圍，僅限於分支指令前127個記憶位置，後(往回跳)128個記憶位置。此一所受的真正限制，並不如您想像中的大，因為，只要以一JMP指令與分支指令組合，則又可跳越至記憶器之任何一位置。例如，下面的例子即說明了，如何以一JMP指令與BCC指令之組合，取代一欲跳出前127個位置，後128個位置之範圍的BCS(進位旗號設定時，則跳越)指令。

```

                BCC  CCLEAR      ; C = 0時，執行CCLEAR之指令
                JMP  CSET        ; C = 1時，跳至CSET
CCLEAR  LDA  $20

```

表1-2-6列出了分支指令跳越所及之整個範圍內，每一位置之十六進相對位址。假若您所使用的系統，可以直接輸入助憶符號的型式，則您可不看此一表格，直接輸入其十六位元之絕對位址而非相對位移。否則，此表可供作您的參考，若您想寫一在條件滿足時，能往前跳89個位置之BCC指令，則您可查前跳相對位址表。89的位置是在第9行第5列。由於行數代表低次的十六進數字(LSD)，列數代表高次的十六進數字(MSD)，因此，BCC指令應寫成：

BCC \$59

表1-2-6 分支指令之十六進運算元

前跳相對位址表

| LSD \ MSD | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|-----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 2 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 3 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 4 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 5 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| 6 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
| 7 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 |

回跳相對位址表

| LSD \ MSD | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|-----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 128 | 127 | 126 | 125 | 124 | 123 | 122 | 121 | 120 | 119 | 118 | 117 | 116 | 115 | 114 | 113 |
| 1 | 112 | 111 | 110 | 109 | 108 | 107 | 106 | 105 | 104 | 103 | 102 | 101 | 100 | 99 | 98 | 97 |
| 2 | 96 | 95 | 94 | 93 | 92 | 91 | 90 | 89 | 88 | 87 | 86 | 85 | 84 | 83 | 82 | 81 |
| 3 | 80 | 79 | 78 | 77 | 76 | 75 | 74 | 73 | 72 | 71 | 70 | 69 | 68 | 67 | 66 | 65 |
| 4 | 64 | 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 |
| 5 | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 |
| 6 | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 |
| 7 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

所有分支指令均佔兩個位元組(第一位元組為運算碼，第二位元組為相對位移)。在條件滿足時，這些指令需要三個週期之執行時間(若跳越至另一頁時，則需四個週期)，而當條件不滿足時，則僅需兩個週期的執行時間。

比較(COMPARE)指令

到目前，我們所介紹的分支指令，都是基於對某一旗號狀態的檢查，而此一旗號即反映暫存器中算術結果的狀態。許多情況下，分支檢查必須基於某一暫存器或某一記憶位置之原始未經改變的內容。例如，在某一位置存0時，程式可能必須跳至某一位置。在這些情況，6502微處理器有三個比較(compare)指令可以使用。6502的比較指令影響三個可以分支指令加以檢查之旗號(進位，零值，與負值)，而不改變運算元原有之內含值。此三個比較指令為：

| 指令 | 說明 |
|-----|-----------------|
| CMP | 累加器與記憶位置的内含相比。 |
| CPX | X暫存器與記憶位置的内含相比。 |
| CPY | Y暫存器與記憶位置的内含相比。 |

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|---------------|--------|--------|---------|
| CMP imm | C9xx | 2/2 | N, Z, C |
| CMP abs | Cdaabb | 4/3 | N, Z, C |
| CMP zpg | C5aa | 3/2 | N, Z, C |
| CMP abs,x * | Ddaabb | 4/3 | N, Z, C |
| CMP abs,y * | D9aabb | 4/3 | N, Z, C |
| CMP zpg,x | D5aa | 4/2 | N, Z, C |
| CMP (zpg),y * | D1aa | 5/2 | N, Z, C |
| CMP (zpg,x) | C1aa | 6/2 | N, Z, C |
| CPX imm | E0xx | 2/2 | N, Z, C |
| CPX abs | ECaabb | 4/3 | N, Z, C |
| CPX zpg | E4aa | 3/2 | N, Z, C |
| CPY imm | C0xx | 2/2 | N, Z, C |
| CPY abs | CCaabb | 4/3 | N, Z, C |
| CPY zpg | C4aa | 3/2 | N, Z, C |

CMP指令可以使用八種不同的定址法，與ADC及SBC指令相同。而X或Y暫存器，由於其主要功能在於作索引與計數器，因此，CPX與CPY指令並不需要這些精巧之定址能力，以致僅使用三種定址方法：立即、絕對、與零頁(參閱1-2-12定址模式)。

比較指令之真正運算是，將選定之暫存器的內含值，減去記憶位置的內含值或立即運算元，但並不儲存所得之結果。唯一顯示運算結果的是三個狀態旗號——負值(N)、零值(Z)、與進位(C)。這三個狀態旗號的組合，顯示了暫存器之內含值是小於，等於，或大於指令的運算元資料(立即運算元或記憶位置之內含值)。表1-2-7所列即為比較運算結果的顯示情形。

表1-2-7 比較指令之結果

| | N | Z | C |
|-----------------|----|---|---|
| A, X, 或Y < 記憶內含 | 1* | 0 | 0 |
| A, X, 或Y = 記憶內含 | 0 | 1 | 1 |
| A, X, 或Y > 記憶內含 | 0* | 0 | 1 |

* 唯有以2的補數比較才為有效。

例題1-2-1顯示了一個檢測兩個記憶位置之內含值是否相同，並於記憶器內設定一旗號加以顯示的常式。

例1-2-1 兩記憶位置內含值相等之檢測

；若位址\$71與\$72兩記憶位置之內含值相等，則此常
 ；式將位址\$73之記憶位置的內含值置定為1；否則
 ；，將其值清除為0。

LDX #00 ；旗號先清除為0。
 LDA \$71 ；載入第一個值。
 CMP \$72 ；兩值相比。

```

                                BEN  DONE      ; 不等則跳去DONE。
                                INX          ; 相等，旗號置定為1。
DONE    STX  $73                ; 將旗號存起。

```

例題1-2-2則為另一“記憶位置對於記憶位置”的比較常式。其將兩值中較大者存於較高(位址較大)之記憶位置。例題1-2-3為一暫存器內含值與固定常數之比較常式。此常式使用兩分支指令與一比較指令，以檢測小於，等於，與大於三種情況。

例1-2-2 將兩數按其大小順序排列之常式

；此一常式將位址\$71與\$72兩記憶位置所含的數

；值，依其大小順序排列，值小者在\$71之位置。

```

                                LDA  $72      ; 第二數值載入累加器。
                                CMP  $71      ; 兩數相比。
                                BCS  DONE      ; 若第一數 ≤ 第二數，
                                                ; 則完成。
                                LDX  $71      ; 否則，將兩數對調。
                                STA  $71
                                STX  $72
DONE    :
        :

```

例1-2-3 三途決策常式

；根據累加器之值是小於，等於，或大於7，該常式將

；累加器之值分別存入位址\$71，\$72，或\$73之不同

；記憶位置。

```

                                CMP  #07      ; 累加器之值與7比。
                                BCS  EQGT7     ; 若無借位，則 ≥ 7。
                                STA  $71      ; 否則，累加器之值小於7。
                                JMP  DONE
EQGT7   BNE  GT7
                                STA  $72      ; 累加器之值等於7。
                                JMP  DONE
GT7     STA  $73                ; 累加器之值大於7。
DONE    :
        :

```

例1-2-4為一將連續記憶位置(最多256個)資料搬家的常式。在此常式中，每當搬完一個位元組，CPX指令即被用以檢測“是否所有資料均已搬完？”

例1-2-4 多位元組之搬運(move)常式

；此常式將由位址\$71之記憶位置起的位元組資料(最

；多256個)，抄至記憶器之另一部份：由位址\$0300

；起之記憶位置。位元組的個數存於位址\$10之記憶

；位置。

```

                LDX  #00          ; 索引 = 0。
NXTBYT         LDA  $71, X       ; 載入下一位元組。
                STA  $0300, X     ; 存出下一位元組。
                INX              ; 索引之位移加1。
                CPX   $10         ; 完了嗎？
                BNE  NXTBYT       ; 沒完繼續。
                :

```

位元檢測(BIT)指令

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|---------|--------|--------|---------------|
| BIT abs | 2Caabb | 4/3 | N=M7, V=M6, Z |
| BIT zpg | 24aa | 3/2 | N=M7, V=M6, Z |

比較指令比較的是兩個完整的八位元數值。在某些情況下，我們僅需檢測某一記憶位置內含的某一或某些位元即可，而不必比較兩個完整的八位元資料。前面已經說過，邏輯AND指令可用以遮掉某些不必要之位元，但是，在遮掉不必要位元的過程中，AND指令卻破壞了累加器原有之內含值。當然，被罩蓋的部份可再重新存回，可是，這需要額外的時間與額外的指令。有一種可達到同樣目的，但卻不改變累加器原有內含值之指令，那就是**位元檢測指令(BIT instruction)**。此一指令運算之結果，既不改變累加器之內含值，亦不改變記憶位置的原有內含值。不過，像比較指令一樣，會在狀態暫存器中記錄下狀態的訊息。

- ① 負值旗號(N)之值等於被檢測之原始(未經AND)記憶內含的第七位元。
- ② 溢位旗號(V)之值等於被檢測之原始記憶內含的第六位元。
- ③ 若AND運算之結果為零，則零值旗號(Z)為1；否則，零值旗號為0。

BIT指令所作之運算與AND指令相同，但是，其並不將AND運算結果存入累加器。再者，AND指令影響兩個旗號(N與Z)，且兩個旗號均反映AND運算後之狀態，而BIT指令卻影響三個旗號(N, V, 與Z)，但僅有Z旗號反映BIT指令中模擬AND過後的狀態。

BIT指令僅使用絕對與零頁兩種定址法。值得注意的是，唯有零值旗號反映了模擬AND運算之結果。負值與溢位旗號則顯示了最高次兩記憶位元之值。事實上，BIT指令履行了兩種十分不同的檢測功能。舉例說明之：(等候一記憶位元變成0)

```

                LDA  #08          ; 遮掉b3以外之所有位元。
LOOP          BIT  $3210         ; 記憶體位置$3210內含之第三位元b3 = 0？
                BNE  LOOP         ; 繼續檢查，直到是為止。
MEET          :                  ; 然後繼續MEET之指令。
                :

```

請留意，若BNE LOOP 改成BEQ LOOP，則上述範例之指令系列將改為檢測b₃是否為邏輯1？BIT指令可以一次檢測一個以上的位元，不過必須有些限制。當累加器之面罩有兩個或兩個以上之位元為1時，緊接的BIT指令唯有在兩個被檢測之記憶位元均為0時，才會將零值旗號置定為1。否則若兩檢測的記憶位元有一者或兩者皆為1，則零值旗號均會被清除為0。

不過，若所有檢測之位元僅有一者落於第0至第5位元之範圍，而其它之位元落於第6與第7位元(b_6 與 b_7)，則這些多位元的檢測限制即自動消失。因為 b_6 與 b_7 提供有”AND獨立的狀態指示”，所以第6與第7位元有其各自的分支指令(第六位元為BVS與BVC，第七位元為BMI與BPL)加以檢測。舉例說明之一 等待三個記憶位元之任一者變為0。

```

                LDA    #08                ; 選擇第三位元與面罩作檢測。
LOOP          BIT     $3210              ; 檢測記憶位元。
                BEQ     MEET              ; 若 $b_3 = 0$ ，跳出迴路。
                BVC     MEET              ; 若 $b_6 = 0$ ，跳出迴路。
                BMI     LOOP              ; 若 $b_7 = 1$ ，再迴路。
MEET          :                          ; 當 $b_3$ 或 $b_6$ 或 $b_7 = 0$ ，接續此處執行。

```

由於BIT指令僅限於使用絕對與零頁定址，因此，其對表格中元素的檢測，或任何其它索引或間接存取之資料的檢測，不是很好用；不過，對一已知位址上的資料而言，BIT指令經常可取代比較指令。由於第六與第七位元的存取容易(經由N與V旗號)，使得此兩位元頗為理想的被用於儲存狀態訊息。尤其是，兩個位元共可組成四種檢測的狀態：00，01，10，與11。

最廣泛應用N與V旗號者，或許應在6502系列之輸入/輸出結構的**插斷請求作業**(interrupt request scheme)上。

1-2-8 移位(SHIFT)與旋轉(ROTATE)指令

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----------|--------|--------|---------|
| ASL abs | 0Eaabb | 6/3 | N, Z, C |
| ASL zpg | 06aa | 5/2 | N, Z, C |
| ASL acc | 0A | 2/1 | N, Z, C |
| ASL abs,x | 1Eaabb | 7/3 | N, Z, C |
| ASL zpg,x | 16aa | 6/2 | N, Z, C |
| LSR abs | 4Eaabb | 6/3 | N, Z, C |
| LSR zpg | 46aa | 5/2 | N, Z, C |
| LSR acc | 4A | 2/1 | N, Z, C |
| LSR abs,x | 5Eaabb | 7/3 | N, Z, C |
| LSR zpg,x | 56aa | 6/2 | N, Z, C |
| ROL abs | 2Eaabb | 6/3 | N, Z, C |
| ROL zpg | 26aa | 5/2 | N, Z, C |
| ROL acc | 2A | 2/1 | N, Z, C |
| ROL abs,x | 3Eaabb | 7/3 | N, Z, C |
| ROL zpg,x | 36aa | 6/2 | N, Z, C |
| ROR abs | 6Eaabb | 6/3 | N, Z, C |
| ROR zpg | 66aa | 5/2 | N, Z, C |
| ROR acc | 6A | 2/1 | N, Z, C |
| ROR abs,x | 7Eaabb | 7/3 | N, Z, C |
| ROR zpg,x | 76aa | 6/2 | N, Z, C |

6502微處理器有四個指令，可以使一運算元的八個位元，順序向左或向右移動一個位元位置。運算元可為累加器或某一記憶位置的內含值。此四個指令中，有兩個是移位指令，另兩個是旋轉指令。

對所有四個指令而言，微處理器之狀態暫存器的進位旗號，就像運算元展延的“第九位元”一般，因為，該旗號接收了從運算元之一端(若右移則為 b_7 ，若左移則為 b_0)被移出的位元。在**移位**運算，運算元之尾端所空出的位元位置(右移時為 b_7 ，左移時為 b_0)，恒被清除為0。而在**旋轉**運算，運算元之相反一端所空出之位元位置，則為旋轉前進位旗號之值所取代。

| 指令 | 說明 |
|-----|-------------------------------|
| ASL | 累加器左移(Accumulator Shift Left) |
| LSR | 邏輯右移(Logical Shift Right) |
| ROL | 左旋轉(ROtate Left) |
| ROR | 右旋轉(ROtate Right) |

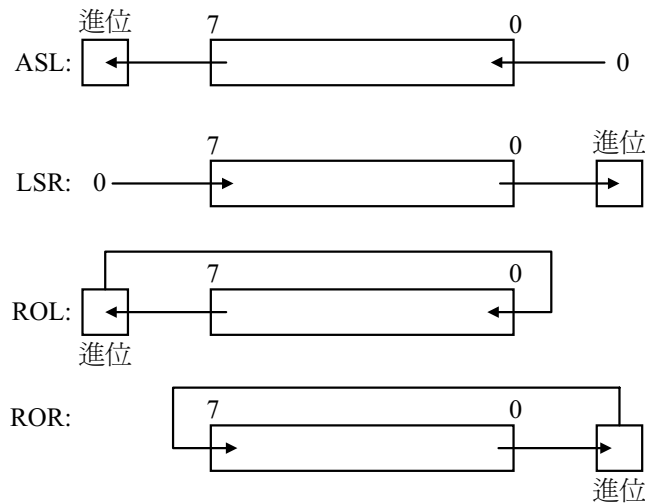


圖1-2-1 移位與旋轉指令之運算情形

若欲對累加器內含運算，則只需在指令的運算元欄上加一“A”即可，如

ASL A (累加器) 佔1個位元組

對記憶位置的運算則可應用四種定址法中的任一種：絕對、零頁、零頁索引|X、與絕對索引|X(參閱1-2-12定址模式)。下面是各種定址的ASL指令：

| | | | |
|-----|-----------|----------|--------|
| ASL | \$1234 | (絕對) | 佔3個位元組 |
| ASL | \$2A | (零頁) | 佔2個位元組 |
| ASL | \$2A, X | (零頁索引 X) | 佔2個位元組 |
| ASL | \$1234, X | (絕對索引 X) | 佔3個位元組 |

除了影響進位旗號(如圖1-2-1所示)外，移位與旋轉指令尚影響其它兩種旗號：

- ① 若移位結果使第七位元為1，則ASL，ROL，與ROR指令使負值旗號(N)置定為1；否則，旗號之值為0。LSR指令則恆使負值旗號為0，因為，其恆將一“0”移入第七位元。
- ② 若經移位之結果為0，則零值旗號(Z)自動置定為1；否則，零值旗號之值為0。

| 進位 旗號 | 位 元 位 置 7 6 5 4 3 2 1 0 | | |
|----------|----------------------------|---------------------------|---------|
| 1 | 0 1 1 0 1 0 0 0 | 運算前 (hex 68, decimal 104) | 原數值 |
| 0 | 1 1 0 1 0 0 0 0 | ASL後 (hex D0 decimal 208) | 原數值乘以 2 |
| 0 | 0 0 1 1 0 1 0 0 | LSR後 (hex 34, decimal 52) | 原數值除以 2 |
| 0 | 1 1 0 1 0 0 0 1 | ROL後 (hex D1 decimal 209) | |
| 0 | 1 0 1 1 0 1 0 0 | ROR後 (hex B4 decimal 180) | |

無號數之移位

單一位元組數目的移位，僅需使用ASL與LSR的位移指令即可。而多位元組數目的移位，則須移位指令與旋轉指令兩者並用。於多位元組之移位運算，進位旗號用以傳遞前一移位之位元組所移出的位元值。

以例爲之，以一儲存於\$71(低次位元組)、\$72、與\$73(高次位元組)之24位元無號數做左移：

| | | |
|-----|------|------------------------|
| ASL | \$71 | ；首先左移低次位元組。 |
| ROL | \$72 | ；次移中間位元組，要考慮進位旗號 用ROL。 |
| ROL | \$73 | ；最後移高次位元組。 |

多位元組數目的右移，則依由左至右之次序，以LSR指令先移高次位元組，再以ROR指令移其餘較低次位元組。下列的右移常式，將\$71位置起的一個三位元組數目，向右移一位元：

| | | |
|-----|------|----------|
| LSR | \$73 | ；移高次位元組。 |
| ROR | \$72 | ；移中間位元組。 |
| ROR | \$71 | ；移低次位元組。 |

有號數之位移

上述所討論的移位與旋轉指令履行了所謂的“邏輯”移位。亦即，其將運算元視爲一個八位元組，而不認符號。結果，若所右移的是一有號數，則符號位元將被往右移一位(像其它位元一樣)，而其原來的位元會被0所取代。若爲左移，則有號數之原符號位元值將移入進位旗號，而其值繼而爲第六位元所取代。很明顯地，程式必須想辦法將異位的符號位元值復原。舉例如下：

例1-2-5 多段有號數之左移常式

；該常式將一儲存於\$71起的記憶位置之多段有號
 ；數，左移一個位元。有號數之長度，以位元組數計，
 ；存於位址\$2F之記憶位置。
 ；

| | | |
|-----|------|-------------|
| LDY | \$2F | ；位元組計數載入Y。 |
| ASL | \$71 | ；移低次位元組。 |
| LDX | #01 | ；位元組索引 = 1。 |
| DEY | | ；位元組計數減一。 |

```

NXTBYT  ROL  $71, X      ; 移位次一位元組。
          INX              ; 更新位元組索引
          DEY              ; 與位元組計數值。
          BNE  NXTBYT     ; 迴路至做完為止。
; 下面之指令將最高次位元組(MSBY)之符號位元
; 復原。
          DEX              ; 將索引指到MSBY。
          LDA  $71, X      ; MSBY載入累加器。
          BCC  MSB0        ; 符號位元 = 1 ?
          ORA  #$80        ; 若是，將1置回符號位元。
          JMP  SOVER
MSB0     AND  #$7F         ; 若非，則符號位元 = 0。
SOVER    STA  $71, X      ; MSBY存回記憶體。

```

就邏輯右移(logical right shift)而言，移位運算使符號位元空出。因此，我們可藉著先將其值記錄在進位旗號內，然後再以ROR指令，將之移入最高次位元組，如此即可保留符號位元。例1-2-6所示為一使用此一技巧的常式。

例1-2-6 多段有號數之右移常式

```

; 該常式將一儲存於位址$71起的記憶位置的多段有號
; 數，右移一個位元。有號數之長度，以位元組計數，
; 存於位址$2F之記憶位置。
          CLC              ; 預備符號位元 = 0之移位
          LDX  $2F         ; 位元組計數載入X。
          DEX              ; 設定MSBY之索引。
          LDA  $71, X      ; 載入最高次位元組。
          BPL  MSB0        ; 符號位元 = 1 ?
          SEC              ; 若是，則預備符號位元 = 1之移位。
MSB0     ROR  A            ; 移位最高次位元組，並存
          STA  $71, X      ; 回記憶體。
          DEX              ; 索引減一，以作次一位元組。
          NXTBYT  ROR  $71, X ; 移位次一最高位元組。
          DEX
          BPL  NXTBYT     ; 迴路至所有位元組均移完為止。

```

1-2-9 暫存器傳輸指令

雖然6502微處理器具有以記憶體為導向之結構，不過，其亦有六個長一個位元之指令，可用以將一暫存器之內含值抄至另一暫存器，而不影響來源暫存器原先之內含值。

| 指令 | 說明 |
|-----|--|
| TAX | 累加器抄至索引暫存器X(Transfer Accumulator to index X) |
| TAY | 累加器抄至索引暫存器Y(Transfer Accumulator to index Y) |

| | |
|-----|--|
| TSX | 堆疊指示器抄至索引暫存器X(Transfer Stack pointer to index X) |
| TXA | 索引暫存器X抄至累加器(Transfer index X to Accumulator) |
| TXS | 索引暫存器X抄至堆疊指示器(Transfer index X to Stack pointer) |
| TYA | 索引暫存器Y抄至累加器(Transfer index Y to Accumulator) |

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----|-----|--------|---------|
| TAX | AA | 2/1 | N, Z |
| TXA | 8A | 2/1 | N, Z |
| TAY | A8 | 2/1 | N, Z |
| TYA | 98 | 2/1 | N, Z |
| TSX | BA | 2/1 | N, Z |
| TXS | 9A | 2/1 | |

對X與Y暫存器之算術運算

由於6502處理器僅能將X與Y暫存器之內含值加一或減一，所以，在更複雜的算術運算裡，傳輸指令提供了一種將X與Y暫存器之內含抄至累加器，並從累加器將運算結果存回X與Y暫存器的簡單方式。試考慮一種欲存取列表之第十、二十、...個元素，而非每一連續緊接元素之狀況。在此種情況下，存取指令必是諸如LDA LIST，Y等的索引指令。其中LIST為表列起始位置之標題。為了存取第十之倍數的元素，每次存取之後，索引暫存器的內含值必須加10。

索引暫存器的內含值如何加10呢？一種方法是連續寫十個加一(INX或INY)指令。另一種更有效率之方法是，將索引暫存器的內含值抄至累加器，加10(立即定址)之後，再存回索引暫存器。

1-2-10 堆疊指令

6502微處理器的堆疊器具有“後進先出”之特性。亦即，最後存入堆疊器的資料項目，最先被取出。相反地，最先被存入者，則最後被取出。**此一特性使得資料存入與取出的次序，正好相反。**堆疊資訊靠一專設的堆疊位址暫存器存取，稱之為**堆疊指示器**(stack pointer，簡稱SP或S)。此一暫存器永遠指在堆疊器內次一緊接可用的空間。每當有一位元組被推入堆疊器後，堆疊指示器的內含值就自動減一。而每當有一位元組從堆疊器被拉出，則堆疊指示器的內含值就自動加一。因此，堆疊是往位址“零”的方向延伸。

6502微處理器的堆疊器是設在第一頁之位址空間內(\$0100至\$01FF之記憶位置)。因此，每當電源一打開，使用者程式則必須將堆疊指示器之內含值歸零(initialize)至\$01FF。由於每一記憶頁僅有256個記憶位置，因此，6502之堆疊器最多僅能儲存256個位元組。堆疊器能儲存什麼樣的資料呢？通常，堆疊器有兩種用途：(1)用以暫存插斷或副程式之回返位址。(2)用以暫時儲存暫存器之內含值。

堆疊指示器指令

電源一打開時，堆疊指示器之內含是未定的，因此，必須由用者程式加以界定。6502微處理器有一特殊指令，可用以定義堆疊指示器之起始值。該指令為：

| 指令 | 說明 |
|-----|------------------|
| TXS | 索引暫存器X內含值抄至堆疊指示器 |

此一指令使用隱含定址，僅佔一個位元組以及需兩個週期之執行時間。由於堆疊由位址\$01FF之記憶位置開始，因此，大多數之系統程式都會將堆疊指示器的起始值界定為\$FF(記得，高次的位址元組\$01是由6502微處理器自動提供的)。

```

;
LDX    #$FF          ; $FF載入X暫存器。
TXS                    ; X之內含值抄至SP。

```

堆疊指示器之內含值，亦可由程式使用另一隱含定址指令將之讀取。

| 指令 | 說明 |
|-----|-----------------|
| TSX | 堆疊指示器內含抄至索引暫存器X |

此一指令甚少使用，因為在大多數情況下，程式設計者並不須“管”堆疊資訊被存於何處。

推入(PUSH)與拉取(PULL)指令

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----|-----|--------|-----------------------|
| PHA | 48 | 3/1 | |
| PLA | 68 | 4/1 | N, Z |
| PHP | 08 | 3/1 | |
| PLP | 28 | 4/1 | N, V, B=1, D, I, Z, C |

6502微處理器具有四個能將累加器與狀態暫存器的內含值存入堆疊器之指令。它們是：

| 指令 | 說明 |
|-----|---------------|
| PHA | 累加器內含值推入堆疊器 |
| PHP | 處理器狀態值推入堆疊器 |
| PLA | 由堆疊器拉取至累加器 |
| PLP | 由堆疊器拉取回至狀態暫存器 |

上述四個指令均為僅佔一個位元組的隱含定址指令。除了所推入的是不同暫存器之內含值外，PHA與PHP指令之動作完全相同。暫存器之內含值皆被推入堆疊器中(堆疊指示器所指的位置)，然後堆疊指示器的內含值減一，指到下一個較低(位址小一)之位置，任一指令均不改變資料來源暫存器之原先內含值。同樣地，PLA與PLP指令先將堆疊指示器之值加一，然後再將指示器所指之記憶位置的內含值載入所要之暫存器(稱為目的暫存器)。

圖1-2-2顯示了PHA指令的實際效用。於圖1-2-2A，堆疊指示器(S)指至堆疊頂端(\$01FF的位置)，而累加器含\$33。在PHA指令執行過後(圖1-2-2B)，堆疊指示器之值已減至\$FE，而累加器的內含值存於\$01FF之位置。若此時微處理器再執行一PLA指令，則堆疊指示器之值將在增至\$01FF，然後\$01FF位置之內含值會被載入累加器。

為何要將暫存器的內含值存於堆疊器呢？那是暫時將暫存器的內含值保留起來，以便暫存器能先作其它之用途(運算)。此尤以副程式(subroutine)應用為然。

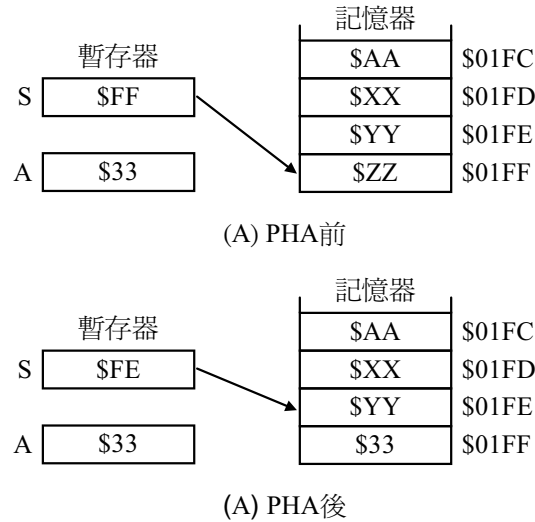


圖1-2-2 累加器內含值推入堆疊器之情形

若僅就前述四個指令看，則似乎僅有累加器與狀態暫存器的內含值能寄存於堆疊器。不錯，就指令本身之功能而言確是如此，但是，若併用X與Y暫存器與累加器之間的暫存器傳輸指令，則我們即可將X與Y暫存器之內含值傳至累加器，之後再推入堆疊器，並可於稍後將之取出並再傳回X與Y暫存器。例1-2-7所示之指令系列，即是將累加器、狀態暫存器，以及X與Y暫存器之內含值存入堆疊器，並於稍後將之取出的情形。注意到，暫存器拉取之次序恰與推入之次序完全相反。

例1-2-7 所有暫存器之值存於堆疊器並取回
；推入暫存器內含值的指令系列。

| | |
|-----|---------------|
| PHP | ；存起狀態暫存器之內含值。 |
| PHA | ；存起累加器之內含值。 |
| TXA | ；存起X暫存器。 |
| PHA | |
| TYA | ；存起Y暫存器。 |
| PHA | |
| : | |
| : | |
| : | 此處為其它指令。 |
| : | |

；拉取暫存器內含值的指令系列。

| | |
|-----|---------------|
| PLA | ；恢復Y暫存器之內含值。 |
| TAY | |
| PLA | ；恢復X暫存器之內含值。 |
| TAX | |
| PLA | ；恢復累加器之內含值。 |
| PLP | ；恢復狀態暫存器之內含值。 |

暫存器內含值存入堆疊器的先後次序須有差別嗎？是的，在大部份情況下確須有差別。累加器之內含值永遠要在X與Y暫存器之前被推入堆疊器，因為推入X與Y暫存器之前的TXA或TYA指令，會破壞累加器內含值。此外若狀態暫存器之值亦欲存起，則其亦必須在X與Y暫存器之前被推入堆疊器，因為TXA與TYA兩指令都會影響狀態暫存器之負值(N)與零值(Z)旗號。

1-2-11 無運算指令

無運算(No Operation，簡記為NOP)指令是一單位元組之隱含定址指令，其通常用於程式開發之際。無運算指令即什麼事都不做的指令，該指令不影響任何暫存器，狀態暫存器，或記憶位置之內含值。但其確達成了保留記憶空間的有用功能。

在設計程式時，程式設計者可在將來預加上指令的地方，先填上NOP指令。由於每一NOP指令僅佔一個位元組之記憶空間，因此，每一欲保留記憶空間之處，至少須填入兩個NOP指令。

除了可用以預留記憶空間外，NOP指令亦可用以填補去掉指令的位置。使得某一指令之去除，不致牽一髮而動全身，造成有關之分支與跳越指令的目的位址，皆需重新更改。

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----|-----|--------|---------|
| NOP | EA | 2/1 | |

1-2-12 6502定址法

6502微處理器所以會普遍被採用的原因之一，就是它在定址(addressing)上提供了類似迷您電腦之彈性。6502共有十三種定址法；下面，我們將分別介紹此十三種定址法。

立即定址法

於立即定址法，指令的第二位元組即為運算元(數據)。在組合語言規則上，立即運算元通常於運算元之前加一#字首表示。例如

LDA #\$33

即是將十六進數33(等於十進數51)存入累加器的指令。運算元33在指令上是放在緊接運算碼之後。所有立即定址法的指令長度均為兩個位元組；運算碼佔一位元組，運算元亦佔一位元組。

絕對定址法

絕對定址法就是將運算元位址的明確值，直接放在指令上。絕對定址法能直接選取6502位址空間內，65,536個記憶位置中的任一記憶位置。所有使用絕對定址法的指令都須有三個連續記憶位置加以儲存。第一位元組為指令之運算碼，第二與第三位元組則分別為運算元位址的低次與高次位元組。例如，

LDA \$12C3

就是一將位址12C3記憶位置的內含存入累加器的絕對定址指令。指令上的12C3，即為真正運算元的絕對位址。若微處理器正在執行此一指令時，位址\$12C3記憶位置儲存3C，則指令執行完後，累加器的內含值亦將變為3C。上述指令，經翻譯成機器碼後，真正儲存在記憶器內的情形為：

| 位址 | 內含 | 說明 |
|--------|------|-------------|
| nnnn | \$AD | 絕對定址LDA之運算碼 |
| nnnn+1 | \$C3 | 運算位元址之低次元組 |
| nnnn+2 | \$12 | 運算位元址之高八元組 |

該指令總共佔據了三個連續記憶位置。換言之，指令之長度為三個位元組。在組合語言規則上，絕對定址通常不以任何記號標示。

零頁定址法

零頁定址法其實是絕對定址法的一種，不過，在此種定址法，微處理器僅選取記憶器中最前端的256個記憶位置——在分頁結構中屬於第零頁。此些位置的十六進位址為0000至00FF。由於較高次的位元組恆為00，因此在零頁定址法的指令上，即加以省略。是以，零頁定址法的指令，僅需兩個記憶位置加以儲存。第一記憶位置儲存指令之運算碼，而緊接第二位置儲存運算元位址之低次位元組。6502微處理器會自動將此一兩個數字(two-digit)的運算元，視成一零頁位址。例如，

LDA \$2A

就是採用零頁定址法，將位址\$002A之位置的內含值存入累加器的指令。

除了JMP (跳越)與JSR (跳至副程式)兩指令外，其它所有能使用絕對定位址的指令，也都可使用零頁定位址。基於零頁定位址能節省記憶空間與執行時間之雙重優點，記憶器之第零頁應隨時儘可能分配作儲存最常用之數據。(以同一指令而言，零頁定址指令比絕對定址指令少佔一個記憶位置，且執行時間亦少一個時序週期)第零頁在臨時數據的儲存上亦甚為有用。

隱含定址法

6502微處理器的指令集中，有很多指令不需要任何運算元，譬如設定或清除狀態暫存器的某一位元、或將某一暫存器的內含值加1或減1、或將某一暫存器的內含值抄至另一暫存器等等。由於這些指令並不需運算元欄，6502只要從運算碼即可獲得足夠的情報，因此，特稱之為隱含定址。下面是一些例子：

| 指令 | 說明 |
|-----|---|
| CLC | 清除進位旗號(Clear Carry flag) |
| DEX | 將X暫存器內含值減1(DEcrement the X register) |
| TAX | 將累加器內含值抄至X暫存器(Transfer Accumulator to X register) |

隱含定址法之指令最為簡短，每一指令僅佔一個記憶位置。

間接絕對定址法

間接絕對定址法僅用於跳越(JMP)指令。JMP指令可使用絕對定址或間接絕對定址兩種方式。JMP指令的實際效應就是將一新位址存入程式計數器，以便微處理器緊接能由那兒拿取指令。於絕對定址，JMP指令的運算元部份即為欲存入程式計數器之目的位址。於間接絕對位址，JMP指令之運算元部份所含，即為儲存十六位元目的位址之兩記憶位置中第一個記憶位置的位址。換言之，真正欲存入程式計數器的十六位元位址(稱為目的位址)尚儲存在記憶器中其它的記憶位置，微處理器必須先根據JMP指令上之位址前去拿取，然後才能將之存入程式計數器。在6502組合語言的記憶符號規則上，間接絕對定址通常以於運算元加一小括弧作識別。例如，

JMP (\$0308)

該指令執行完後，程式計數器的內含值，低次的位元組將變成\$0308記憶位置的內含值，而高次的位元組將變成\$0309記憶位置的內含值。圖1-2-3所示即為此例之情形。在圖中，目的位址為02AB，而跳越指令儲存在0110，0111，與0112三個連續記憶位置。

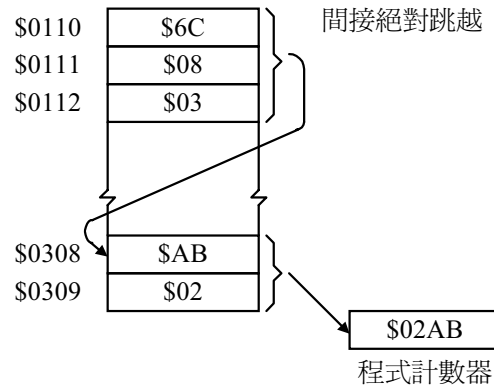


圖1-2-3 間接絕對定址

已知道目的位址為\$02AB，何不就直接使用絕對定址就好？這是因為間接絕對定址具有彈性特質，它允許目的位址“可變”。舉個例子而言，若6502是裝設在一必須服務數個週邊設備的系統，則一間接絕對定址的JMP指令，即可用以服務這些週邊設備。我們可先將服務各設備所需之指令系列寫好，並分別儲存在記憶體許多不同的地方，相對於同一設備的指令系列在一起。然後再選定固定的連續兩記憶位置，如\$0308與\$0309，以便儲存上述各指令系列之起始位址。如此，不論那一設備，每次當我們欲服務某一設備時，只要令微處理器執行：

JMP (\$0308)

即可。因為，在此之前，我們可先利用其它指令，依據請求服務之設備的不同，將其所對應不同的起始位址先填入\$0308與\$0309的位置。如此，雖然微處理器每次均執行JMP(\$0308)指令，但因\$0308與\$0309位置之內含已有改變，因此，事實上微處理器可跳至許多不同的位置。不像絕對定址，微處理器永遠只能跳至指令上所指明之唯一位置。

在資料處理應用上，一個單獨的6502微處理器可能用以接收來自數個鍵盤的資料。在此種情況下，跳越指令之目的位址將視當時正在接收的是那一按鍵而定。來自鍵盤1的輸入必須存在一個地方，而來自按鍵2的輸入又必須存在另一個地方。如此類推。間接絕對定址亦容許程式儲存在ROM或PROM，而目的位址卻在RAM中的應用。

絕對索引定址法

在絕對索引定址模式中，運算元之有效位址(即真正位址)等於X或Y暫存器之內含值，加上指令上之絕對位址。亦即

$$\text{有效位址} = \text{絕對位址} + X$$

或

$$\text{有效位址} = \text{絕對位址} + Y$$

所有絕對索引定址指令均佔滿三個連續記憶位置。第一位置為運算碼，第二、三位置為一絕對位址，低次位元組在前，高次位元組在後。在組合語言符號上，絕對索引定位址以在運算元後加上有關索引暫存器之名稱(即X或Y)表示。例如

LDA \$12C3, X

即是一絕對索引定址之LDA指令。若索引暫存器X之內含值為\$05，則該指令即將位址\$12C8(=\$12C3 + \$05)位置之內含值存入累加器。

絕對索引定址對存取某一表列(list)之資料最為有用。在此種應用，表列之起始位址可作為指令上之絕對位址，而索引暫存器則用以儲存位移(displacement)—指明欲存取的是表列中之第幾個元素。譬如，若索引暫存器之內含值為\$05，即表示將存取表列之第6項元素等。再者，若能建立一索引暫存器之內含值每次累增之迴路，即可存取整個表列一系列連續的記憶位置。

零頁索引定址法

零頁索引定址與零頁定址之關係，就如同絕對索引定址與絕對定址之關係。就零頁索引定址而言，運算元的有效位址等於索引暫存器(X或Y)之內含值，加上指令所含之零頁基底位址。例如，

LDA \$33, X

即為一零頁索引定址指令。若X暫存器之內含值為\$05，則上述指令將把\$0038(\$0033+\$05)之位置的內含載入累加器。

如絕對索引定址一般，零頁索引定址提供了表列應用之潛能。零頁索引定址指令僅需兩個記憶位置加以儲存，而絕對索引定址指令則需三個記憶位置。

有一點必須強調的是，零頁索引定址之有效位址僅限於第零頁範圍內。若零頁基底位址與索引暫存器內含相加之結果超出零頁範圍(亦即和大於\$FF)，則6502將忽略低次位元組所產生之進位。在上述例子中，X之值就被限於\$CC之內。X=CC時，有效位址為\$FF，而當X=CD時，有效位址就又回到\$00。

零頁索引間接定址法

零頁索引間接定址是前面所討論過的零頁索引定址與間接定址兩種方式之組合。記得，零頁索引定址是將索引暫存器的位移加至指令上之基底位址，以獲得運算元之有效位址。

而在間接定址，指令上所含之運算元，事實上為儲存真正數據之位址的兩記憶位置中第一個記憶位置中的位址，而非數據本身，或數據之位址。

這兩個觀念可加以組合。在零頁索引間接定址，X暫存器之位移先加至指令上之零頁位址，產生一新的零頁位址。此一新零頁位址然後再被當成間接位址，以取得運算元。換言之，運算元之有效位址則儲存在新零頁位址所指之位置，以及次一位置。

由於此指令上所含的為零頁位址，因此，所有零頁索引間接位址指令皆僅佔兩個位元組。零頁索引間接定址漂亮之處在於，由於有效位址為十六位元之絕對(最終)位址，因此，只要以一長兩位元組之指令，即可存取6502微處理器的整個記憶空間— 65,536個記憶位置。然而，每一零頁索引間接定址指令的執行，均須耗費六個時序週期，比同一指令的零頁定址多出三週期，也比絕對定址方式多出二個週期。零頁索引間接定址寫成(aa, X)的形式。例如，

LDA (\$3E, X)

即是一零頁索引間接定址的LDA指令。假如X暫存器之內含值為\$33，則6502微處理器會先算出間接位址，\$3E+\$33=\$71，然後再去第零頁之\$71與\$72兩位置上，拿取數據之有效位址。若\$71位置之內含為\$C3，而\$72位置之內含為\$12，則有效位址即為\$12C3，圖1-2-4所示即為此例之詳細情形。

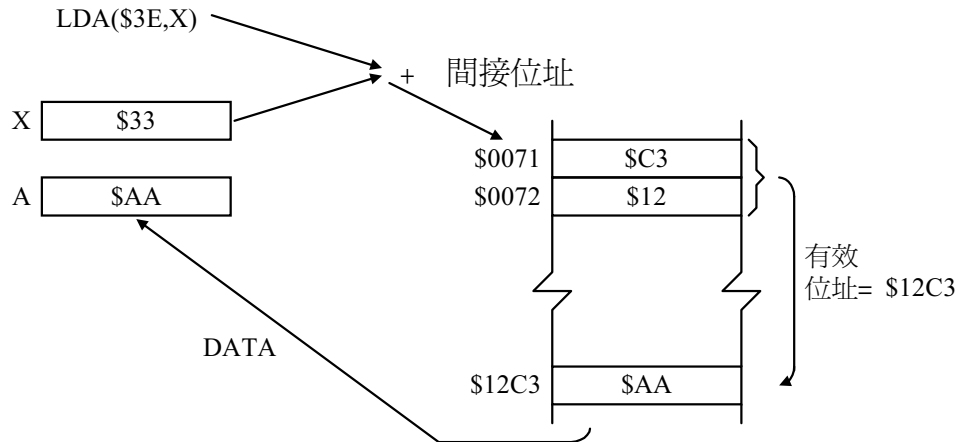


圖1-2-4 零頁索引間接定位

零頁間接索引定址法

就像零頁索引間接定址一樣，零頁間接索引定址亦結合了間接與零頁索引兩種定址，但所應用的方向則相反。零頁索引間接定址模式之技巧稱為**先索引**(Pre-indexing)；而零頁間接索引定址模式，則為間接定址須先實施之後，索引再加至十六位元之記憶位址，因此，此種技巧稱為**後索引**(post-indexing)。零頁間接索引定址在指令上表示成(aa)，Y之形式。例如

`LDA ($3E), Y`

即是一例。若Y暫存器含`$33`，則此一指令將先由零頁位置`$3E`與`$3F`取得基底位址，然後再將基底位址與Y暫存器之內含值相加，獲得有效位址。假使位置`$3E`含`$C3`，`$3F`位置存`$12`，那最後被存入累加器的內含值，將是位址為`$12F6`(=`$12C3` + `$33`)記憶位置的內含值。如圖1-2-5。

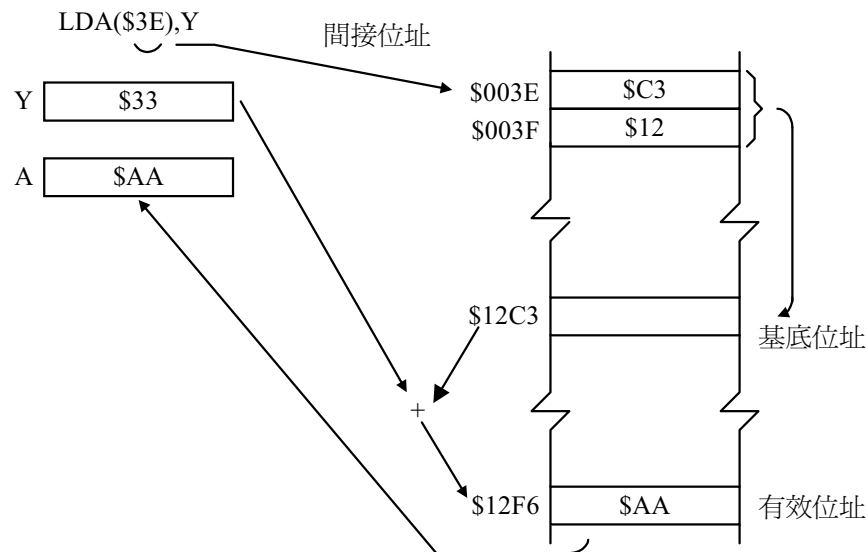


圖1-2-5 零頁間接索引定址

零頁間接索引定址可用以存取許多同樣結構的表格中某一表格內之一已知元素。例如，此一定址法可用於數個使用者共用的一指令系列(副程式)上。於執行間接索引指令之前，用者的叫用程式可先將一獨特基底位址填入零頁運算元所指之位置(上例中之`$3E/$3F`位置)，使得指令能找到正確之數據表格。

相對定址法

相對定址法的有效位址，是相對著下一個欲執行之指令的位址而得來的。亦即，有效位址是將程式計數器之現行值(current value)，加上一正位移或負位移計算而得。加正位移會選到目前指令後面(位址較大的方向)的位置，而加負位移的結果則選取在目前指令之前面(位址更小的方向)的位置。相對定址模式是僅用於八個**分支指令(branch instruction)**上。而分支指令是當某一條件滿足時，使程式控制向前轉移或向後轉移，而條件不滿足時，繼續往下執行下一緊接指令之指令。其亦即一般所謂的**條件跳越指令(conditional jump instruction)**。例如，進位清除時分支(Branch on Carry Clear, BCC)之指令。

```
BCC      NEXT
LDA      #$33
```

在進位旗號為0時，將程式控制轉移至NEXT的位置。否則，若進位旗號為1(不清除)，則微處理器在執行過BCC NEXT指令後，將繼續往下執行緊接的LDA #\$33指令。

所有的分支指令都佔兩個記憶位置。第一位元組為運算碼，第二位元組為位移。由於僅長八位元，因此，位移被限於向前127個位置(正位移)，向後128個位置之範圍(負位移)。

累加器定址法

6502微處理器有四個指令，可將累加器或某一記憶位置的內含左移或右移一個位元位置。若運算元位於累加器內，則該指令即為累加器定址。很明顯的，累加器定址僅是隱含定址的一部份。累加器定址指令均僅佔一個位元組之記憶空間。將累加器之內含值向左順移一位元的指令：ASL A — 即為累加器定址的一個例子。

6502微處理器的十三種定址方式，至此全部介紹完畢。表1-2-8摘列了這些定址法，在表格中，“a”代表一十六進制位址數字，因此，“aaaa”即為四個十六進制位址的一般形式。例如，aaaa 可能代表12F6。



表1-2-8 6502定址法運算元形式

| 定址法 | 運算元形式 |
|--------|---------------|
| 立即 | #aa |
| 絕對 | aaaa |
| 零頁 | aa |
| 隱含 | |
| 間接絕對 | (aaaa) |
| 絕對索引，X | aaaa，X或aaaa X |
| 絕對索引，Y | aaaa，Y或aaaa Y |
| 零頁索引，X | aa，X或aa X |
| 零頁索引，Y | aa，Y或aa Y |
| 零頁索引間接 | (aa，X)或(aa X) |
| 零頁間接索引 | (aa),Y或(aa) Y |
| 相對 | aa或aaaa |
| 累加器 | A |

1-2-13 6502定址法摘要

立即定址法-# —運算元位於指令的第二個位元組。

絕對定址法-a —指令的第二、三個位元組組成一16位元位址。

| | | | | |
|--------|--------|-------|---|-------|
| 指 令： | OpCode | addrl |  | addrh |
| 運 算 元： | | addrh |  | addrl |

零頁定址法-zpg —指令的第二個位元組即為零頁位址。

| | | | |
|--------|--------|-----|------|
| 指 令： | OpCode | zpg | |
| | | + | zpg |
| 運算元位址： | | | 有效位址 |

累加器定址法-A —僅為單一位元組的指令，運算元為累加器(Accumulator)。

隱含定址法-i —僅為單一位元組的指令，運算元被指令隱含定義。

零頁間接索引定址法-(zpg),y —這個定址法常引用Y當做間接位址，指令的第二位元是零頁位址，而該零頁位置的內含加上Y索引暫存器形成有效位址。

| | | | |
|--------|--------|-----------|-------|
| 指 令： | OpCode | zpg | |
| | | (zpg + 1) | (zpg) |
| | | + | Y 暫存器 |
| 運算元位址： | | | 有效位址 |

零頁索引間接定址法-(zpg,x) —這個定址法常引用X當做間接位址，指令的第二位元是零頁位址，而該零頁位址加上X索引暫存器內含的結果指向16位元的有效位址。

| | | | |
|--------|--------|-----------|-------|
| 指 令： | OpCode | zpg | |
| | | | zpg |
| | | + | X 暫存器 |
| | | (Sum + 1) | (Sum) |
| 運算元位址： | | | 有效位址 |

零頁索引，X-zpg,x —指令的第二位元是零頁位址，零頁位址加上X索引暫存器形成有效位址。

| | | | |
|--------|--------|-----|-------|
| 指 令： | OpCode | zpg | |
| | | | zpg |
| | | + | X 暫存器 |
| 運算元位址： | | | 有效位址 |

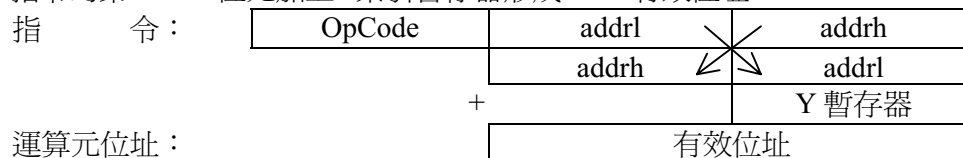
零頁索引，Y-zpg,y —指令的第二位元是零頁位址，零頁位址加上Y索引暫存器形成有效位址。

| | | | |
|--------|--------|-----|-------|
| 指 令： | OpCode | zpg | |
| | | | zpg |
| | | + | Y 暫存器 |
| 運算元位址： | | | 有效位址 |

絕對索引，X-a,x—指令的第二、三位元加上X索引暫存器形成16-bit有效位址。



絕對索引，Y-a,y—指令的第二、三位元加上Y索引暫存器形成16-bit有效位址。



相對位址定址法-r—這是一個程式計數器相對位址定址法，僅用於分支跳越指令。假使檢測條件吻合，則指令的第二位元組(位移)會被加上程式計數器(計數器已指向下一指令位址)，位移大小限定於-128~+127。

間接絕對-(a)—僅用於跳越指令。指令的第二、三位元形成一”位址指引”，該指令完成後，程式計數器的低次元組內含值將載入該”位址指引”的第一個位元組，而高次元組則將載入該”位址指引”的第二個位元組。

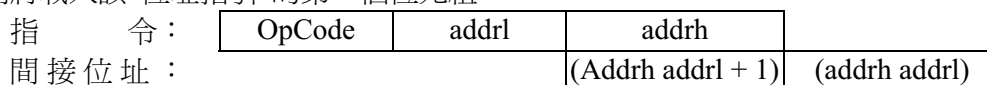


表1-2-9 定址法摘要

| 定址模式 | 指令週期 | 佔用記憶位元組 |
|-----------------|----------|---------|
| 1. 立即 | 2 | 2 |
| 2. 絕對 | 4 (3) | 3 |
| 3. 零頁 | 3 (3) | 2 |
| 4. 累加器 | 2 | 1 |
| 5. 隱含 | 2 | 1 |
| 6. 零頁間接索引 (d),y | 5 (1) | 2 |
| 7. 零頁索引間接 (d,x) | 6 | 2 |
| 8. 零頁，X | 4 (3) | 2 |
| 9. 零頁，Y | 4 | 2 |
| 10. 絕對，X | 4 (1, 3) | 3 |
| 11. 絕對，Y | 4 (1) | 3 |
| 12. 相對 | 2 (2) | 2 |
| 13. 間接絕對(Jump) | 5 | 3 |

標注：

- (1) 頁界限，當形成位址時該位址會超越頁界限，加1個週期。
- (2) 分支跳越，當分支跳越發生加，加1個週期。
- (3) 讀-修改-寫，加2個週期。

1-2-14 指令集摘要

表1-2-10所示則為一更詳細的6502指令集摘要表。此一表格不管在爾後幾章之閱讀，或日後從事程式設計時，都甚具有參考價值。

表 1-2-10 6502指令集摘要

| 指令 | 立即 | 絕對 | 零頁 | 累加器 | 隱含 | (間, X) | (間, Y) | 零頁, X | 絕對, X | 絕對, Y | 相對 | 間接 | 零頁, Y | 符號 |
|--|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|-------------------|
| 運算 | OP n # | OP n # | OP n # | OP n # | OP n # | OP n # | OP n # | OP n # | OP n # | OP n # | OP n # | OP n # | OP n # | N V B D I Z C |
| ADC A←M←A | 69 2 2 | 4D 4 3 | 65 3 2 | | | 61 6 2 | 71 5 2 | 75 4 2 | 7D 4 3 | 79 4 3 | | | | N V Z C |
| AND A←M←A | 29 2 2 | 2D 4 3 | 55 3 2 | | | 21 6 2 | 31 5 2 | 55 4 2 | 3D 4 3 | 39 4 3 | | | | N Z |
| ASL C←b ₇ b ₀ ←0 | | 0E 6 3 | 06 5 2 | 0A 2 1 | | | | 16 6 2 | 1E 7 3 | | 9D 2 2 | | | N Z C |
| BCC C←1時接越 | | | | | | | | | | | B0 2 2 | | | N |
| BCS C←1時接越 | | | | | | | | | | | | | | N |
| BEQ Z←1時接越 | | | | | | | | | | | F0 2 2 | | | N |
| BIT A←M | | 2C 4 3 | 24 3 2 | | | | | | | | | | | N Z |
| BNE Z←1時接越 | | | | | | | | | | | 30 2 2 | | | N |
| BPL N←0時接越 | | | | | | | | | | | 10 2 2 | | | N |
| BRL N←0時接越 | | | | | | | | | | | | | | N |
| BRK 中斷 | | | | | 00 7 1 | | | | | | | | | N |
| BVC V←0時接越 | | | | | | | | | | | | | | N |
| BVS V←1時接越 | | | | | 18 2 1 | | | | | | 50 2 2 | | | N |
| CLC 0←C | | | | | D8 2 1 | | | | | | 70 2 2 | | | N 0 |
| CLD 0←D | | | | | | | | | | | | | | N 0 |
| CLI 0←I | | | | | S8 2 1 | | | | | | | | | N 0 |
| CLV 0←V | | | | | B8 2 1 | | | | | | | | | N 0 |
| CMP A←M | | | | | | C1 6 2 | D1 5 2 | D5 4 2 | DD 4 3 | D9 4 3 | | | | N Z C |
| CPX X←M | | CD 4 3 | CS 3 2 | | | | | | | | | | | N Z C |
| CPY Y←M | | EC 4 3 | E4 3 2 | | | | | | | | | | | N Z C |
| CPY Y←M | | CC 4 3 | C4 3 2 | | | | | | | | | | | N Z C |
| DEC M←1→M | | CE 6 3 | C6 5 2 | | CA 2 1 | | | D6 6 2 | DE 7 3 | | | | | N Z |
| DEX X←1→X | | | | | 88 2 1 | | | | | | | | | N Z |
| DEY Y←1→Y | | | | | | 41 6 2 | 51 5 2 | 55 4 2 | 5D 4 3 | 59 4 3 | | | | N Z |
| EOR A←M←A | | 4D 4 3 | 45 3 2 | | | | | | | | | | | N Z |
| INC M←1→M | | EE 6 3 | E6 5 2 | | | | | | | | | | | N Z |
| INX X←1→X | | | | | EA 2 1 | | | | | | | | | N Z |
| INY Y←1→Y | | | | | | | | | | | | | | N Z |
| JMP 接越至新位置 | | 4C 3 3 | | | ES 2 1 | | | | | | | 6C 5 3 | | N Z |
| JSR 接越至新位置 | | 20 6 3 | | | CS 2 1 | | | | | | | | | N Z |
| LDA M←A | | AD 4 3 | A5 3 2 | | | A1 6 2 | B1 5 2 | B5 4 2 | BD 4 3 | B9 4 3 | | | | N Z |
| LDX M←X | | | | | | | | | | | | | | N Z |
| LDY M←Y | | AE 4 3 | A6 3 2 | | | | | | | | | | | N Z |
| LSR 0→b ₇ b ₀ ←C | | AC 4 3 | A4 3 2 | 4A 2 1 | | | | B4 4 2 | BC 4 3 | BE 4 3 | | | B6 4 2 | N Z |
| NOP 無運算 | | 4E 6 3 | A6 5 2 | | | | | 56 6 2 | 5E 7 3 | | | | | 0 Z C |
| ORA A←M←A | | 0D 4 3 | 05 3 2 | | | 01 6 2 | 11 5 3 | 15 4 2 | 1D 4 3 | 19 4 3 | | | | N Z |
| PHA A→M←S | | | | | 48 3 1 | | | | | | | | | N |
| PHP P→M←S | | | | | 08 3 1 | | | | | | | | | N |
| PLA S←1→S | | | | | 68 4 1 | | | | | | | | | N |
| PLP S←1→S | | | | | 28 4 1 | | | | | | | | | N |
| ROL 接越至新位置 | | 2E 6 3 | 26 5 2 | 2A 2 1 | | | | 26 6 2 | 3E 7 3 | | | | | N Z C |
| ROR 接越至新位置 | | | | | | | | | | | | | | N Z C |
| RTI 斷程式返回 | | 6E 6 3 | 66 5 2 | 6A 2 1 | | | | 76 6 2 | 7E 7 3 | | | | | N Z C |
| RTS 斷程式返回 | | | | | 40 6 1 | | | | | | | | | N Z C |
| SBC A←M←A | | ED 4 3 | E5 3 2 | | 60 6 1 | | | | | | | | | N Z C |
| SEC 1←C | | | | | 38 2 1 | | | F5 4 2 | FD 4 3 | F9 4 3 | | | | N Z C |
| SED 1←D | | | | | F8 2 1 | | | | | | | | | N 1 |
| SEI 1←I | | | | | | | | | | | | | | N 1 |
| STA A←M | | 8D 4 3 | 85 3 2 | | 78 2 1 | | | 95 4 2 | 9D 5 3 | 99 5 3 | | | | N |
| STX X←M | | | | | | 81 6 2 | 91 6 2 | | | | | | 96 4 2 | N |
| STY Y←M | | 8E 4 3 | 86 3 2 | | | | | | | | | | | N |
| TAX A←X | | 8C 4 3 | 84 3 2 | | AA 2 1 | | | 94 4 2 | | | | | | N |
| TAY A←Y | | | | | | | | | | | | | | N |
| TSX S←X | | | | | AS 2 1 | | | | | | | | | N Z |
| TXA X←A | | | | | BA 2 1 | | | | | | | | | N Z |
| TXS X←S | | | | | 8A 2 1 | | | | | | | | | N Z |
| TXA X←S | | | | | 9A 2 1 | | | | | | | | | N Z |
| TYA Y←A | | | | | 98 2 1 | | | | | | | | | N Z |

M₇ 記憶位號第七位元
M₆ 記憶位號第六位元
n 位元組數

+ 加
- 減
AND 與
OR 或
(+) EOR 異或

X 索引
Y 索引
A 累加器
M 暫存器
M₆ 暫存器位號
M₆ 暫存器位號

第1-3章 插斷與重設

插斷(interrupt)有兩種型式：**可罩蓋(maskable)**的插斷—可被暫時忽略之插斷，與**不可罩蓋(nonmaskable)**的插斷—必須被立即處理之插斷。

重設(reset)是一將系統**重新設定(initialize)**於某種已知狀態的作業。所有微處理器一定都被設計成電源一打開時就重設。另外大多數微處理器亦允許其內部暫存器在其它時候，能由一外部信號加以重新設定—此一動作稱為**重始**。

插斷與重設在功能上類似，因為它們都利用**向量指示器(vector pointer)**，來決定下一個欲拿取之指令的記憶位址。**6502**微處理器是以位址\$FFFA到\$FFFF的記憶位置來儲存向量指示器。其中\$FFFA與\$FFFB兩位置儲存不可罩蓋插斷的向量指示器，\$FFFC與\$FFFD兩位置儲存重設的向量指示器，而\$FFFE 與\$FFFF兩位置則儲存了可罩蓋插斷請求之向量指示器。

1-3-1 插斷

插斷是一外部產生之信號欲暫時延緩微處理器正在執行的程式，且令程式控制轉移至某一專為**服務**發出插斷之設備而設計的副程式(下一章節將介紹)。週邊設備以插斷“通知”微處理器，它有資料要輸入微處理器，或必須由微處理器輸出資料給它。此一技巧可免去微處理器浪費寶貴時間不停地**取樣(polling)**各週邊設備的狀態以查驗有無服務的需求。

插斷同時亦可用以告知微處理器某些緊急之狀況—如停電。週邊設備可藉著使 $\overline{\text{IRQ}}$ 動作，而對**6502**微處理器提出插斷服務**請求(request)**。而諸如停電之緊急情況則可透過 $\overline{\text{NMI}}$ 的起動，**強迫****6502**微處理器作立即之處理。 $\overline{\text{IRQ}}$ 與 $\overline{\text{NMI}}$ 上面皆加一橫短線，表示此兩信號皆為低電位(low level)動作。

1-3-2 插斷請求(IRQ)

如何得知系統中的週邊設備是否需要服務呢？第一種方式就是設計一主要程式，使得微處理器在每隔一段時間，就停止資料之處理，而**取樣**系統中每一設備之狀態，看其是否需要服務。這種方法非常浪費微處理器的時間。另一種辦法，就是利用插斷請求(IRQ)的設計，只要任何外部設備需要微處理器的服務，它就可以利用此一輸入通知**6502**。如此，**6502**微處理器就不必浪費寶貴的時間，隨時回頭查看是否有週邊設備需要服務了！而其實 $\overline{\text{IRQ}}$ 的功能很像電話機的鈴聲一樣，顯示了系統正“叫著”需要服務。當然，就像忙碌時可先將電話鈴聲擱在一旁一樣， $\overline{\text{IRQ}}$ 信號亦可先被忽視，直到**6502**微處理器有時間對它做出反應為止！

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----|-----|--------|---------|
| CLI | 58 | 2/1 | I=0 |
| SEI | 78 | 2/1 | I=1 |

插斷控制指令

6502內的狀態暫存器之插斷禁能位元(I)，決定**6502**微處理器是否對 $\overline{\text{IRQ}}$ 線上的插斷請求做出反應。兩個控制此一位元的指令為：

| 指令 | 說明 |
|-----|----------|
| CLI | 清除插斷禁能位元 |
| SEI | 置定插斷禁能位元 |

指令CLI將插斷禁能位元(I)清除為零；在此種狀態下，6502微處理器只要感應到插斷請求信號，便會對外部設備提供插斷服務。而指令SEI可將插斷禁能位元設定為1，此種情況下，對緊接而來所有的 $\overline{\text{IRQ}}$ 插斷請求，都將為6502微處理器所忽略。每當電源一打開，6502微處理器即自動將插斷禁能位元設定為1。

6502微處理器對 $\overline{\text{IRQ}}$ 的反應

若插斷禁能位元為0，而且某一外部設備使 $\overline{\text{IRQ}}$ 動作，則6502微處理器在執行完目前的指令後，即會自動開始一個八週期的插斷系列。在此一系列中，6502微處理器將三個位元組的“回返”資訊推入堆疊器(程式計數器之高、低位元組以及狀態暫存器之內含)。然後將IRQ向量指示器(\$FFFE與\$FFFF兩記憶位置之內含，前者含低次位元組)載入程式計數器內。同時，6502微處理器亦將插斷禁能位元設定為1，以暫時將爾後的插斷請求擋掉！

利用堆疊指示器(S)、程式計數器(PC)、狀態暫存器(P)、與堆疊器某四個位置(\$01C0至\$01C3)等等的內含值於反應前與反應後的變化情形，說明6502微處理器對 $\overline{\text{IRQ}}$ 信號的反應(見圖1-3-1)。於1-3-1A圖，堆疊指示器指至次一可用堆疊位置(在此例假設為\$01C3)，程式計數器含主程式之次一指令的位址(高次位址PCH，低次位址PCL)，而狀態暫存器之I位元清除為零。於圖1-3-1B， $\overline{\text{IRQ}}$ 反應後，原來程式計數器與狀態暫存器的內含值已存入堆疊器，因此，堆疊指示器指至位址\$01C0的位置。再者，程式計數器之低次位元組與高次位元組，現在分別含\$FFFE與\$FFFF兩記憶位置的內含值。同時，處理器狀態暫存器之I位元被置定為1。I位元是狀態暫存器中，唯一受此次反應影響的位元。

此時，程式計數器已載入專設計以服務產生插斷請求之設備的程式之起始位址。此一專為服務提出插斷請求之設備而設計的程式，稱之為**插斷服務常式**(Interrupt Service Routine, ISR)或**插斷處理程式**(interrupt handler)。插斷服務常式必須履行兩種功能：其必須辨認產生插斷請求之設備(若系統含一個以上的此等設備的話)，同時，其必須履行該設備所需求之作業。

在ISR中，程式首先必須讀取系統中每一個設備的狀態暫存器，找出那一設備之插斷請求位元被設定為1，即可確認該設備就是提出插斷請求者；一旦插斷設備找到，插斷服務常式就必須拿取一新的位址——專屬插斷設備之插斷服務常式的起始位址。這裡提到了兩個插斷服務常式，第一個插斷服務常式的起始位置為\$FFFE與\$FFFF兩記憶位置的內含，此常式先進行辨認並找出插斷的設備，一旦找到即跳往此常式為各個週邊設備所定義的各自專屬插斷服務常式去(後者所提及之插斷服務常式)。6502系列的週邊設備都將其插斷請求位元，設在某一暫存器之第6或第7位元。因此確認取樣系列中可以由一系列的BIT與分支指令來組成。

例1-3-1即為一含有四種設備之系統的取樣系列。設備1、2、4三者皆僅能產生一插斷請求，且請求狀態顯示在各自狀態暫存器(SDEV1、SDEV2、SDEV4等位置)的第7位元。設備3則可產生兩個個別之插斷請求，並將請求狀態顯示於其狀態暫存器(SDEV3位置)的第6與第7位元。

系統中設備1，2，4的插斷請求狀態以BIT指令加以詢問，該指令將設備狀態暫存器之第6與第7位元，分別載入**微處理器狀態暫存器**之溢位(V)與負值(N)旗號。對設備1與設備2，一BIT指令決定設備是否處於提出插斷請求狀態。但設備3則需兩個分支指令(BMI與BVS)加以檢測，因為，此一設備能產生兩個不同插斷請求。設備4則不需再問了，因為假若設備1、2、3皆沒產生插斷請求，那一定就是設備4了！

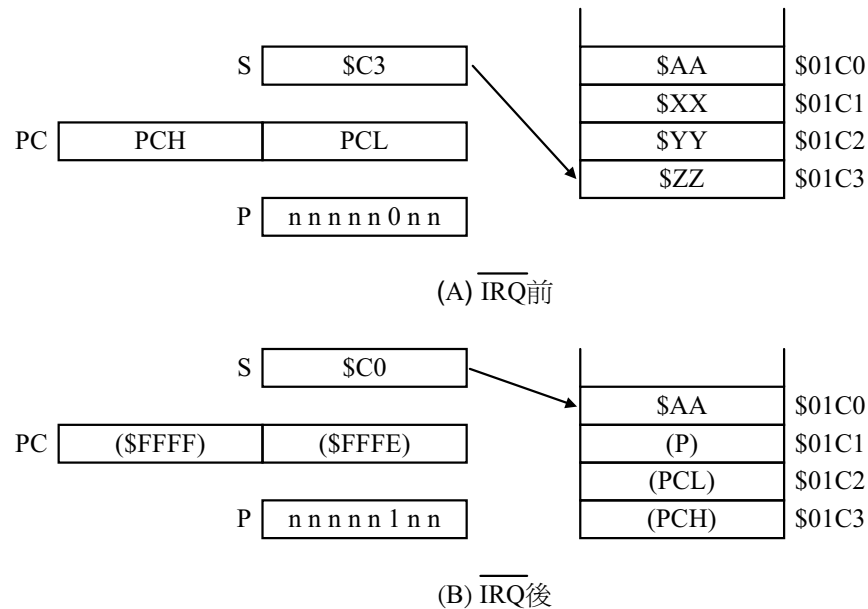


圖1-3-1 6502微處理器對IRQ之反應

您會發現，例1-3-1的取樣系列，將系統中的設備指定了優先次序(priority)；設備1擁有最高優先，而設備4為最低優先。雖然看起來似乎微處理器在抵達設備4的插斷服務常式前，必須費上好一段功夫，因為，在服務設備4之插斷請求前，微處理器必須執行許多的指令。但是，事實上設備1與2之取樣僅需六個週期時間，且設備3之取樣僅需八個週期時間。因此在服務設備4前；6502微處理器僅需20週期(~5微秒)的時間。

例1-3-1 插斷取樣系列

```

:
BIT   SDEV1      ; 設備1提出插斷請求嗎？
BMI   JISR1      ; 若是，則跳至JISR1。
BIT   SDEV2      ; 設備2提出插斷請求嗎？
BMI   JISR2      ; 若是，則跳至JISR2。
BIT   SDEV3      ; 插斷請求來自設備3嗎？
BMI   JISR3A     ; 若是，則跳至JISR3A
BVS   JISR3B     ; 或JISR3B。
JMP   ISR4       ; 若非，那必為設備4了。
JISR1  JMP   ISR1 ;
JISR2  JMP   ISR2 ;
JISR3A JMP   ISR3A ;
JISR3B JMP   ISR3B ;
:

```

插斷服務常式內之指令

除了實際傳遞資訊於插斷週邊設備與6502微處理器間的指令外，插斷服務常式還需要其它什麼指令呢？大多數插斷服務常式之開頭皆為一把將會受服務常式改變之暫存器的現有內含值存入堆疊器之指令。

當然，副程式末了必須有相對的指令，將這些暫存器內含一一由堆疊器提取。此外，大多數插斷服務常式亦包括一CLI(清除插斷禁能位元)指令，以允許具有更高優先次序的設備再提出插斷服務請求。CLI指令於插斷服務常式中之**位置**，將視目前設備的優先次序而定。在最不優先設備之插斷服務常式裡，CLI可能緊接於將暫存器值存入堆疊器之指令後；亦即，位於副程式之開頭。反之，具有最高優先次序之設備的插斷服務常式，甚至就不含CLI指令，因為，根本沒有其它任何設備“**夠資格**”在該插斷服務常式執行過程中，再提出插斷請求並接受微處理器的服務。

除了暫存器內含的存取，以及CLI指令外，每一插斷服務常式一定還有一**插斷回返指令**：**RTI(ReTurn from Interrupt)**。此一指令必定是插斷服務常式最後一個被執行之指令，使其程式控制由插斷服務常式回返至叫用程式。下面我們將立即討論此一指令。

插斷回返(RTI)指令

每一插斷服務常式最後一個被執行之指令一定是：

| 指令 | 說明 |
|-----|-----------------------------|
| RTI | 插斷回返(ReTurn from Interrupt) |

RTI指令使6502微處理器的程式計數器與狀態暫存器，恢復其在插斷前之原有值。執行RTI指令亦自動使IRQ插斷請求致能，記得嗎？因為狀態暫存器在被推入堆疊器前，其插斷禁能位元(I)之值為零。

RTI指令十分類似於RTS指令。兩個指令都是由副程式回返至叫用程式(叫用程式可為另一個副程式)。用RTI的副程式，其被叫用是自動的，由一外部產生的插斷請求信號引起。而用就RTS之副程式，該副程式的叫用是由主程式控制的，由JSR指令引起。RTI與RTS兩者亦皆為隱含定址，長一個位元組，以及需六個週期執行時間的指令。純粹從動作觀點而言，RTS與RTI兩者之唯一區別是，RTS由堆疊器拉取兩位元組的資訊(程式計數器之高、低位元組)，而RTI則由堆疊器拉取三個位元組的資訊(程式計數器之高、低位元組、及狀態暫存器之內含)。事實上，若某一副程式在開始作業前，需先將狀態暫存器之值存起。則您亦可以PHP(狀態暫存器之值推入堆疊器)指令作為該副程式之第一指令，而以RTI(而非RTS)作為該副程式之回返指令，其結果亦同。

1-3-3 不可罩蓋插斷(NMI)

另一插斷—不可罩蓋插斷 $\overline{\text{NMI}}$ ，為那些在插斷請求被禁能期間無法等待的高優先設備或事件(諸如停電)提供另一條門徑。與 $\overline{\text{IRQ}}$ 不同的是， $\overline{\text{NMI}}$ 所產生的插斷不可被漠視(即被罩蓋或禁能)。一感應到 $\overline{\text{NMI}}$ 為低電位時，6502只要一完成正在執行的指令，就會立即開始處理不可罩蓋插斷。

不可罩蓋插斷($\overline{\text{NMI}}$)恒比可罩蓋插斷($\overline{\text{IRQ}}$)具有較高優先。倘若有一插斷請求與一不可罩蓋插斷同時發生，則6502微處理器將毫無疑問地處理不可罩蓋插斷。除此之外， $\overline{\text{NMI}}$ 的動作與 $\overline{\text{IRQ}}$ 幾已無差別！在接受 $\overline{\text{NMI}}$ 插斷時，6502微處理器同樣執行一個八週期的插斷系列，只不過，這一回向量指示器是來自\$FFFA(低次位址位元組)與\$FFFB(含位址之高次位元組)兩記憶位置，而非\$FFFE與\$FFFF。

事實上，圖1-3-1亦可用以說明6502微處理器如何對 $\overline{\text{NMI}}$ 反應。除了(B)圖程式計數器的內含應改成(\$FFFB)與(\$FFFA)之外，其餘完成相同。

由於 $\overline{\text{NMI}}$ 是服務優先事件的專線，因此，我們有必要知道，6502微處理器能多快對此一插斷採取反應。假設我們考慮“最壞的情況”。最壞的情況發生於 $\overline{\text{NMI}}$ 動作之時，6502微處理器正要拿取一最長指令的運算碼，如絕對索引定址之INC(記憶位置值加一)或DEC(記憶位置內含值減一)指令，此等指令需要七個週期的執行時間。此時，至少必須再經過六個週期的時間，6502微處理器方能接受插斷，並開始一個八週期的插斷系列。在系列中的第八個週期，6502微處理器將拿取插斷服務常式第一個指令之運算碼。此一指令可能是將資料來回傳遞於週邊設備之輸入(LAD)或輸出(STA)指令。此種情況至少還需四個週期。因此將這些執行時間加起來可得知，以NMI插斷而言，資料在17個週期內(~4.25微秒)，即可傳至或傳出週邊設備。

1-3-4 中斷

| 指令 | 指令碼 | 周期/字節 | 影響的狀態位元 |
|-----|-----|-------|----------|
| BRK | 00 | 7/1 | B=1, I=1 |

中斷(BRK)指令常被用於軟體研發過程中檢查”程式做得怎麼樣了？它可令6502微處理器執行一插斷系列。在執行中斷指令時，6502微處理器會把狀態暫存器的BRK命令旗號(B)設定為1、程式計數器加一，然後將三個位元組的資料推入堆疊器內(程式計數器的低、高位元組，與狀態暫存器)。最後，將IRQ向量(\$FFFE與\$FFFF兩位置的內含)載入程式計數器。

由於BRK指令所取用的插斷向量與 $\overline{\text{IRQ}}$ 所取用者同，因此該插斷服務常式應包括能辨別插斷是由 $\overline{\text{IRQ}}$ 抑或BRK所引起的指令。此種辨別最簡單的方式就是，查看已存入堆疊器的狀態暫存器中BRK命令位元(B)的狀態。若B=1，那麼服務常式是由BRK指令所引起，否則，該服務常式勢必就是由 $\overline{\text{IRQ}}$ 所引起的了。

由於BRK指令是一種研發過程中臨時性的指令，因此BRK置入於程式時，其必然需重疊於現有指令，這對單一(因為BRK是單一位元組指令)或兩位元組(因為已存於堆疊器的返回位址將指在BRK後第二個位元組上)的現有指令不會造成問題。但若BRK所欲重疊的是一個三位元組的指令，則由插斷服務常式返回時，6502微處理器將回到該指令的第三位元組，並且將之當成運算碼(opcode)而造成錯誤。這種情況，應該是以BRK指令填補該指令的運算碼並利用NOP指令填補指令的第三位元組。

1-3-5 重設

程式計數器的內含值即為6502微處理器次一欲拿取指令所在的記憶位置。在系統電源加上之初，如何設定程式計數器之內含值？最初設定程式計數器之值，也因而起始6502微處理器作業，乃一**重設**信號—RES。此一信號且用於停電之後或任何想要的時刻，重設或重始微處理器。當RES輸入線為低電位時，6502微處理器將處於“失能”狀態；資訊無法寫入微處理器，亦無法由微處理器讀取。同時，微處理器內部所有暫存器的值亦處於未定狀態。

$\overline{\text{RES}}$ 線上之電壓一上升到高位階(high level)後，6502微處理器會立即開始一為時六週期的**起動系列**(start sequence)。

正當處於此一系列時，6502微處理器將IRQ禁能位元設定為1，使微處理器正在起動時，暫先不接受任何插斷。同時將\$FFFC與\$FFFD兩記憶位置之內含，分別載入程式計數器的低次與高次位元組。

\$FFFC與\$FFFD兩個位置所含，必須為6502在重設後第一個欲執行之指令的位址。事實上，這就是監督程式內某一段起動程式之起始位址。(由於電源停掉時，此兩位置之內含仍須保存，因此必須是ROM的位置。實際上起動程式亦駐位於ROM內)。

例1-3-2 6502微處理器之重設程式

；該程式在系統電源一打開或重設時，置定一6502

；組成之微電腦系統的起始狀態。

```
RESET      LDX  # $FF          ; 堆疊指示器值=$01FF。
            TXS
            :
            :      輸入/輸出之起始指令
            :
            LDA  #00          ; A, X, 及Y暫存器之
            TAX              ; 值清除為零。
            TAY
            CLD              ; 清除十進作業形式。
            CLC              ; 清除進位旗號。
            CLI              ; 插斷致能。
            JMP  USERPG      ; 開始執行用者程式。
```

例1-3-2說明了一重設程式的典型指令系列：

- ① 程式一開始即先置定堆疊指示器之起始值。通常堆疊指示器一開始皆會指定到 \$01FF 的位置。因為那是堆疊器第一個位置。但並無理由說此堆疊指示器不能指至其它的記憶位置。
 (1-1)為何堆疊指示器之起始值要最優先設定呢？因為，若緊接馬上有停電等事故發生時，則此類不可罩蓋之插斷便能立刻被處理。
 (1-2)在重始(initialize)堆疊指示器後，程式接著應開始設定系統中各週邊設備之暫存器的起始位置。因I/O起始指令與系統結構有關，所以例1-3-2中並未將之列出。
- ② 接著的三個指令(LDA #00、TAX、TAY)，分別將微處理器內的A、X、與Y暫存器內含清除為零。有些系統的重設程式並無這幾個指令。此地將之列出的主要原因，乃為再次強調，電源一打開時，6502微處理器內的暫存器內含是未定的。特別注意，於六個週期的起動系列內，6502微處理器並未將這些暫存器清除為零。
- ③ 緊接著的兩個指令，CLD與CLC，則是任選的；其它的重設程式可能將其中之一或兩者皆設定為1，亦可能將其中一位元置定為1，而另一位元清除為零。
- ④ 在堆疊器與所有暫存器的起始值都設定好後，程式將插斷請求致能(插斷於6502起始系列時被禁能)。例1-3-2的最後指令JMP USERPG，使微處理器緊接著跳至一標題為USERPG的使用者程式上。

除電源一打開之外，有時我們亦需將一微電腦系統重新設定成某一已知狀態。例如，當程式控制進入一無窮迴路，或系統中某一零件壞掉而致使整個系統中止時。由於將電源關掉令系統重新開始的辦法，會使讀/寫記憶器內所儲存之資訊完全消失。因此在此種情況下，大部份微電腦系統皆採用了RES信號(用按鍵或其它方式)，使系統重設或重始。

第1-4章 副程式

假若某一運算(含一串指令)在程式中必須做一次以上，爲了不必每次將該運算所對應的指令系列一再重寫，就可使用副程式的設計。**副程式(subroutine)**，即爲僅寫一次但程式需要時可反複加以執行的一系列指令。將控制由程式主要流程轉移至副程式的過程，稱爲副程式**叫用(calling)**。一旦副程式被叫用，6502微處理器即轉而執行副程式所含的指令，並於執行完後，再回至叫用的程式，繼續往下執行未完之指令。

1-4-1 副程式指令

叫用副程式的指令必須具備三種功能：

- ① 這些指令必須能存取程式計數器的內含值。一旦副程式被執行完後，此一位址必須被用以回返至叫用副程式的程式處。該位址因而稱爲**回返位址(return address)**。
- ② 這些指令必須使微處理器開始執行副程式。
- ③ 這些指令必須利用程式計數器被存起的內含值，令微處理器回返至原來程式離開之處，並由那兒繼續往下執行程式的指令。

上述之三個功能，可由兩個指令加以達成：**跳越至副程式 (Jump to SubRoutine,以JSR代表)**，以及**副程式回返(ReTurn from Subroutine，以RTS代表)**。

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|---------|--------|--------|---------|
| JSR abs | 20aabb | 6/3 | |

| 指令 | 指令碼 | 週期/位元組 | 影響的狀態位元 |
|-----|-----|--------|-----------------------|
| RTS | 60 | 6/1 | |
| RTI | 40 | 6/1 | N, V, B=1, D, I, Z, C |

跳越至副程式(JSR)指令

JSR指令達成了儲存回返位址以及令微處理器執行副程式的兩種功能。被存起來的位址是JSR指令第三個位元組所在之記憶位置的位址。(很明顯地，這並非真正的回返位址。在回返時，此一位址會再加一，以便指在真正之回返位置。此點將併入RTS指令討論)。回返位址儲存在哪兒呢？在**堆疊器**裏。這表示JSR指令的動作就像PHA或PHP指令。不過並不盡相同。PHA與PHP僅將一個位元組的資料存入堆疊器，而JSR指令則將兩個位元組的資料存入堆疊器。在存畢程式計數器的內含值後，JSR指令將指令上第二及第三位元組之絕對位址，存入程式計數器，此舉令控制轉移至副程式的起始位址。

JSR指令長三個位元組——第一位元組爲運算碼，二與三位元組爲副程式的起始位址。副程式起始位址可爲一絕對記憶位址，亦可爲一代表絕對位址的標題(在組合語言原始程式時)。由於涉及堆疊運算，因此，JSR指令需要六個週期的執行時間。

現在，讓我們考慮一典型跳越至副程式指令，JSR \$0503的執行情形，假設該指令儲存在\$0201、\$0202、與\$0203的位置。

| 記憶位置 | 指令 |
|--------|--------------------|
| \$0201 | JSR (實際是存運算碼— 20H) |
| \$0202 | \$03 |
| \$0203 | \$05 |
| : | : |
| : | : |
| : | : |
| \$0503 | (副程式的第一個指令) |

此範例以圖解說明堆疊指示器(S)、程式計數器(PC)、與堆疊器的 內含值變化如下：

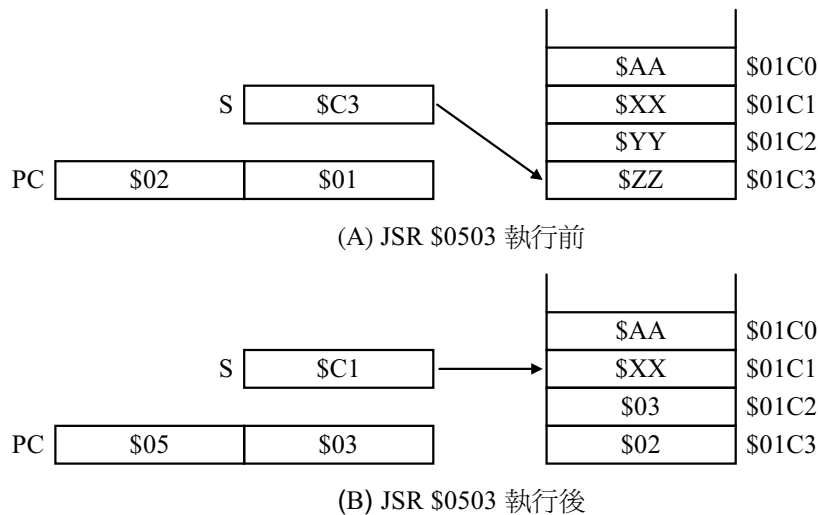


圖1-4-1 JSR指令執行之情形

在1-4-1A圖，程式計數器指在JSR指令的第一個位元組(\$0201之位置)，而堆疊指示器指在堆疊器中下一個可用的位置(此例為\$01C3)。圖1-4-1B，JSR \$0503指令執行過後，①程式計數器存入副程式的起始位址。②而堆疊指示器則指至新的緊接次一個可用的位置(\$01C1)。③回返位址— JSR指令第三位元組所在位置之位址，已存入堆疊器。前面說過，事實上，**6502微處理器必須回返至緊接JSR指令後的指令**。在此例中，6502必須回至\$0204之位置(起)的指令。

副程式回返(RTS)指令

RTS指令使6502微處理器由副程式回返至叫用程式(含JSR指令的程式)。副程式中最後一個被執行的指令恒為RTS指令。RTS指令長一個位元組，需要六個週期的執行時間，而且不影響任何旗號。

JSR指令將兩個位元組的位址推入堆疊器— 該位址即為JSR指令的最後位元組所在之位置的位址，在例子中即為\$0203。為了拿取該位址，RTS指令將堆疊指示器之值加一(依前面的例子而言，即成為\$C2)，取出堆疊指示器所指之位置(\$01C2)的內含(回返位址之低次位元組)，存入程式計數器的下半部位(低次八位元)，然後，堆疊指示器之值再加一，再取出回返位址之高次位元組，存入程式計數器的上半部。最後程式計數器之值加一，指至程式中JSR指令之下一個指令。

當然，RTS指令執行時有一隱含的條件，堆疊指示器必須指至JSR指令所用過(建立)之堆疊位置；即回返位址所存的堆疊位置。因此，副程式中只要有任何推入動作發生，就必須同時有同等數目的拉取指令被執行。

JSR與RTS如何併用

在個別介過JSR與RTS兩個副程式指令後，讓我們舉一個例子說明此兩指令如何併用。例1-4-1顯示了圖1-4-1所示的JSR指令，以及位於\$0503位置起的副程式，該副程式的功能為將Y暫存器的內含值雙倍。JSR指令起於\$0201的位置。副程式則始於\$0503的位置，止於\$0508之位置上的RTS指令。

6502微處理器必須回返至那一記憶位置呢？毫無疑問的，是JSR \$0503指令之後緊接的第一個位置。由於JSR指令佔三個位元組，所以該是\$0204之位置。圖1-4-2所示即為RTS指令執行前後，堆疊指示器(S)、程式計數器(PC)、與堆疊器之內含值變化的情形。

例1-4-1 副程式叫用與回返

| 記憶位置 | 指令 | 運算元 | 註解 |
|--------|-------|--------|-----------|
| \$0201 | JSR | \$0503 | ；副程式叫用指令。 |
| \$0204 | : | : | ；副程式回返至此。 |
| | : | : | |
| | (副程式) | | |
| \$0503 | PHA | | ；累加器之值存起。 |
| | TYA | | ；Y暫存器之值抄至 |
| | ASL | A | ；累加器，雙倍後 |
| | TAY | | ；，再存回Y。 |
| | PLA | | ；取回累加器之值。 |
| \$0508 | RTS | | ；回返。 |

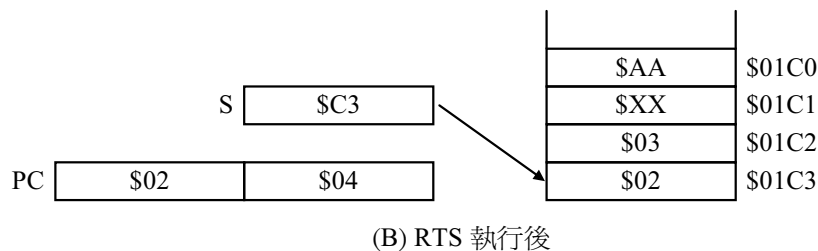
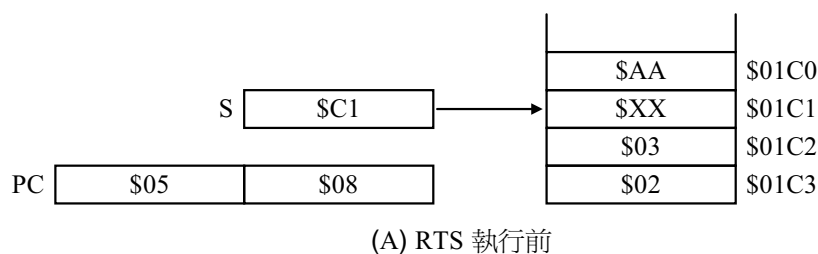


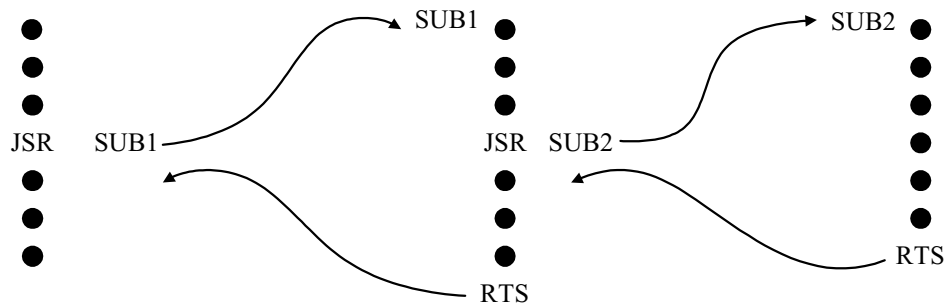
圖1-4-2 RTS指令之執行情形

1-4-2 副程式巢串

在一個副程式中，亦可包括一個或一個以上叫用其它副程式的JSR指令。副程式再叫用副程式的過程，即稱為**副程式巢串**(subroutine nesting)。例1-4-2所示即為副程式SUB1再叫用副程式SUB2的副程式巢串情形。

巢串通常以**層次(levels)**加以描述。在範例1-4-2中，副程式SUB1叫用副程式SUB2，而副程式SUB2未再進一步叫用任何其它副程式，因此該副程式巢串稱為**一層的巢串**(one level of nesting)。理論上，副程式的巢串可達無限多層；副程式SUB2可再叫用副程式SUB3，SUB3 再叫用SUB4，....等等。不過，由於每一JSR指令均將兩個位元組的位址推入堆疊器，因此實際上堆疊器的容量限制了巢串的最高層次。由於6502的堆疊器有256個記憶位置，所以6502的副程式巢串最高僅可達127層。然而，事實上，用到此一最高極限的機會實在太少了。

例1-4-2 副程式巢串



第1-5章 2500AD 6502 Assembler/Linker簡易使用說明

本文件目的在於提供對於2500AD 6502 assembler/linker感到陌生的人，一個快速入門的方案：對於熟悉使用此assembler/linker的人，本文件則用以做為參考。若您想要一窺2500AD assembler/linker全貌，或想進一步了解此功能強大的軟體，請您閱讀該軟體提供的使用手冊。

1-5-1 Assembler

x6502.exe操作說明

2500AD 6502 assembler提供兩種操作模式：prompt mode和command line mode，請從下面的實例中，直接學習這兩種操作模式。

1.1 提示模式(Prompt Mode)

請在DOS prompt(不管是純MS-DOS或Windows 95的DOS模式)下，依照assembler存放的路徑執行x6502，如下例：

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\X6502 (enter)
```

則出現下面的訊息...

```
6502 Macro Assembler Copyright (C) 1990 by 2500AD Software Inc. Version 5.02a      ①
Listing Destination (N, T, D, E, L, P, <CR> = N): (enter)                          ②
Generate Cross Reference? (Y/N <CR> = No): (enter)                                ③

Input Filename: (enter)                                                            ④
Output Filename: (enter)                                                            ⑤
```

說明：

- ① 本文件中以5.02a版的assembler做為說明對象。
- ② 要求指定list的目的地。
- ③ 是否產生交互參考資料，回答Y的話，交互參考資料會列在整個list的最後面。
- ④ 要求輸入欲組譯的原始程式檔名。
- ⑤ 要求輸入組譯完成後，輸出的OBJ檔檔名(如未指定，則其將直接套用輸入檔名)。

[範例] (產生list file)

```
6502 Macro Assembler Copyright (C) 1990 by 2500AD Software Inc. Version 5.02a
Listing Destination (N, T, D, E, L, P, <CR> = N): D (enter)
Generate Cross Reference? (Y/N <CR> = No): (enter)

Input Filename: DEMO1 (enter)
Output Filename: (enter)
```

結果出現如下訊息...

```

2500 A.D. 6502 Macro Assembler - Version 5.02a
-----
Input Filename:      DEMO1.ASM
Output Filename:     DEMO1.OBJ

Lines Assembled: 51      Errors: 0

```

並產生兩個檔案：DEMO1.OBJ、DEMO1.LST。下面是DEMO1.LST檔案內容片段...

```

13          P_ROM  .SECTION      OFFSET $C000, RANGE $C000 $FFF9
14 0000      POWER_ON:
15 0000  A2 FF          LDX  #$FF
16 0002  9A            TXS
17 0003  A0 00          LDY  #0
18 0005  A9 27          LDA  #<DATA_DB
19 0007  85 00          STA  DATA_ADDR_L
20 0009  A9 00          LDA  #>DATA_DB

```

行數編號

原始程式部分

此行指令所組譯出來的machine code

此行指令在所處section中的offset，link完後，此offset值會調整為絕對位址

此外，程式若有語法錯誤，也會顯示在list file中，如下例...

```

13          P_ROM  .SECTION      OFFSET $C000, RANGE $C000 $FFF9
14 0000      POWER_ON:
15 0000  A2 FF          LDX  #$FF
16 0002  9A            TXS
17 0003  A0 FF          LDY  #$FFF
    ***** demo1.asm : Line 17 *****
    ***** # TOO LARGE *****
18 0005  A9 27          LDA  #<DATA_DB
19 0007  85 00          STA  DATA_ADDR_L

```

1.2 命令線區模式(Command Line Mode)

沿用上一個範例，您若想要用command line mode去組譯一個原始程式，可以鍵入如下命令...

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\X6502 DEMO1 -D
```

1.3 檔案類型(Filename Extension)

以下是當您使用2500AD 6502 assembler/link時，常用的檔案類型...

| Assembler | |
|-----------|---------------------------------------|
| .asm | 組合語言原始程式 |
| .obj | 原始程式經組譯後所產生的目的檔 |
| .lst | 程式列表檔 |
| Linker | |
| .obj | 指定給linker，用來做程式連結的輸入檔，與上面所列.obj是同一個檔案 |
| .tsk | 可執行的binary machine code檔 |
| .hex | Intel Hex format的linker輸出檔 |
| .dcf | 2500AD高階除錯控制檔 |
| .sym | 程式符號檔，內含程式所定義的symbol、label及其對應的位址 |
| .map | 程式載入資訊檔，內含程式節區範圍、及其對應的載入位址 |

1.4 組合語言語法(Assembly Language Syntax)

1.4.a Number Base Designations

2500AD assembler可接受五種形式的運算元(operand)，這些形式由加冠於數(文字)之字首或字尾一特定符號而定義。

| Prefix | Operand Form | Suffix |
|--------|-----------------------|--------|
| None | Base 10 (Decimal) | D |
| \$ | Base 16 (Hexadecimal) | H |
| @ | Base 8 (Octal) | O or Q |
| % | Base 2 (Binary) | B |
| “ or ‘ | ASCII | ” or ’ |

[例] 16_{10} 能被表示成：

| 寫法 | 進制 |
|---------------------|------|
| 16或16D | 十進制 |
| \$10或10H | 十六進制 |
| @20或20O或20Q | 八進制 |
| %00010110或00010110B | 二進制 |

1.4.b Program Comments

對每一行程式而言，在分號「；」後面的部分，**assembler**一律視為註解，請參考下面範例：

```
LDX  #$FF      ; STACK POINTER = $01FF
TXS              ; HERE IS PROGRAM COMMENTS
```

1.4.c Program Counter

在程式中，可以用獨立的金錢符號「\$」，來代表目前**program counter**的值，所以下列的範例中，**JMP**指令會形成「自己跳到自己」的無窮迴圈。

```
STORE_END:
      JMP  $
```

1.4.d Global Labels

對**2500AD assembler**而言，**label**必須以英文字母或底線符號做為開頭，而且是大小寫有別的。**label**長度不能超過32個字元，而且**label**名稱後面必須加上分號(如下面範例中之①、③)，但名稱若以該行的第一個字元做為起始位置，名稱後面可以不加分號「：」(如下面範例中之②)。

```
POWER_ON:                                ①
      LDX  #$FF      ; STACK POINTER = $01FF
SET_SP                                     ②
      TXS              ; HERE IS PROGRAM COMMENTS
MOVE_DATA:                               ③
      LDY  #0
      LDA  #<DATA_DB
      STA  DATA_ADDR_L
      LDA  #>DATA_DB
      STA  DATA_ADDR_H
```

1.4.e Local Labels

Local label 與**Global label**非常相似，但 **local label**的定義只有在其”**local區域**”(在兩個**global label**之間)有效；也因此，當程式從一個**local區域**到另一個**local區域**後，您可以重用某些在其它**local區域**用過的**label**名稱。**Local labels**具有以下幾點特性：

- 必須以問號「？」做為名稱的開頭或結尾
- 可使用任何字元做名稱，而長度有效到32個字元
- **local label**的有效範圍僅及於兩個**global label**之間

[範例]：

```

DELAY1:
        LDX  #0
?LOOP1:
        DEX
        BNE  ?LOOP1                                ①
?LOOP2:
        LDY  #10
        DEY
        BNE  ?LOOP2
DELAY2:
        LDY  #0
?LOOP1:
        DEY
        BNE  ?LOOP1                                ②
        JMP  ?LOOP2                                ③

```

說明：

- ① 這道BNE指令，會跳到DELAY1下面的那個?LOOP1
- ② 這道BNE指令，會跳到DELAY2下面的那個?LOOP1
- ③ 這道JMP指令，會被assembler視為語法錯誤，因?LOOP2的有效範圍僅止於DELAY1和DELAY2之間。

善用local label，相信您以後寫程式時，就不必爲了替label取名字而大傷腦筋。

1.4.f High / Low byte

```

18 C005  A9 27          LDA  #<DATA_DB          ①
19 C007  85 00          STA  DATA_ADDR_L
20 C009  A9 C0          LDA  #>DATA_DB          ②
21 C00B  85 01          STA  DATA_ADDR_H
:
:
35 C027          DATA_DB:                        ③
36 C027  00 01 02 03 04 DB  $00,$01,$02,$03,$04,$05,$06,$07,$08,$09,$FF
      05 06 07 08 09
      FF

```

由上面的list file片段中，您可以看到DATA_DB的位址在\$C027(③)，由①處可看出，用小於符號「<」即可取得DATA_DB位址的low byte，由②處可以看出，用大於符號「>」即可取得DATA_DB位址的high byte。

1.5 組譯指引(Assembler Directives)

Directive如欲自第一行(column)開始，則必須冠以十進制的小數點(“.”)否則2500AD Assembler將會誤以為是label。您也可以不管它位於哪一行，一律冠上十進制的小數點以明確區分其即為directive。下面我們就以範例程式來說明常用的assembler directive的用法及作用。

DEMO1.ASM

| | | | |
|--------------|--|------------------------------------|---|
| | .CHIP | 6502 | ① |
| | .SYNTAX | 6502 | ② |
| | .SYMBOLS | | ③ |
| | .OPTIONS | DCH | ④ |
| | .LINKLIST | | ⑤ |
| W_RAM | .SECTION | PAGE0, RANGE \$0 \$7F, REF_ONLY | ⑥ |
| DATA_ADDR_L: | DS | 1 | ⑦ |
| DATA_ADDR_H: | DS | 1 | |
| ALABEL_DS: | DS | 10 | |
| P_ROM | .SECTION | OFFSET \$C000, RANGE \$C000 \$FFF9 | ⑧ |
| POWER_ON: | LDX | #\$FF | |
| | TXS | | |
| | LDY | #0 | |
| | LDA | #<DATA_DB | |
| | STA | DATA_ADDR_L | |
| | LDA | #>DATA_DB | |
| | STA | DATA_ADDR_H | |
| NEXT_BYTE: | LDA | (DATA_ADDR_L),Y | |
| | CMP | #\$FF | |
| | BEQ | STORE_END | |
| | STA | ALABEL_DS,Y | |
| | INY | | |
| | JMP | NEXT_BYTE | |
| STORE_END: | JMP | \$ | |
| LABEL_BLK: | | | |
| BLKB | 10,\$A5 | | ⑨ |
| DATA_DB: | | | |
| DB | \$00,\$01,\$02,\$03,\$04,\$05,\$06,\$07,\$08,\$09,\$FF | | ⑩ |
| DATA_DW: | | | |
| DW | \$0000,\$0001,\$0002,\$0003,\$0004,\$FFFF | | ❶ |
| IRQ_ISR: | RTI | | |
| NMI_ISR: | RTI | | |
| VECTAB | .SECTION | OFFSET \$FFFA, RANGE \$FFFA \$FFFF | ❷ |
| | DW | IRQ_ISR | |
| | DW | POWER_ON | |
| | DW | NMI_ISR | |
| | END | | ❸ |

說明：

- ① 告訴assembler，請它以6502 CPU的指令碼來組譯下面這段程式。
- ② 告訴assembler，以下的語法是標準6502定義的語法。
- ③ 告訴assembler/linker，請它產生symbol file以便於debug。
- ④ 告訴assembler/linker，請它產生map file(D)、高階除錯控制檔(C)，輸出檔格式為Intel Hex format(H)。
- ⑤ 告訴assembler/linker，當link完成後，要將list file中的address值調整成最終的正確位址。請參照下面的比較：

沒使用LinkList directive，list file中的address欄位都還只是在該section中的offset值：

| | | | |
|---------|-----------|----------|------------------------------------|
| 13 0000 | P_ROM | .SECTION | OFFSET \$C000, RANGE \$C000 \$FFF9 |
| 14 0000 | POWER_ON: | | |
| 15 0000 | A2 FF | LDX | #\$FF |
| 16 0002 | 9A | TXS | |

使用了LinkList directive，list file中的address欄位已經調整成絕對位址值：

| | | | |
|---------|-----------|----------|------------------------------------|
| 13 | P_ROM | .SECTION | OFFSET \$C000, RANGE \$C000 \$FFF9 |
| 14 C000 | POWER_ON: | | |
| 15 C000 | A2 FF | LDX | #\$FF |
| 16 C002 | 9A | TXS | |

- ⑥ 宣告以下開始為一個名為W_RAM的section，此section位於zero-page(page0)。此section範圍從\$00~\$7F，若使用的空間超過此範圍，在link時會出現錯誤訊息。最後宣告此section為reference only，並不產生指令碼佔有實際的空間。一般而言，只有屬於RAM範圍的section才會宣告成reference only。此外，請注意到section的名稱後面不可以加冒號(label名稱後面則要加冒號)。
- ⑦ 此DS 1 (Define Storage) directive宣告保留1 byte的空間給變數用，此directive適用於RAM範疇。所以下面的DATA_ADDR_H的位址將因前一個保留1 byte之directive而成為\$0001。請參考下面的list file片段：

| | | | |
|---------|--------------|----------|---------------------------------|
| 8 | W_RAM | .SECTION | PAGE0, RANGE \$0 \$7F, REF_ONLY |
| 9 0000 | DATA_ADDR_L: | DS | 1 |
| 10 0001 | DATA_ADDR_H: | DS | 1 |

- ⑧ 宣告以下開始為一個名為P_ROM的section，此section的起始位址為\$C000(offset \$C000)，範圍為\$C000~\$FFF9(RANGE \$C000 \$FFF9)。
- ⑨ 宣告從此位址開始，定義10個byte，其值為\$A5，請參照下面的list file片段：

| | | | |
|---------|----------|------------|---------|
| 30 C01A | 4C 1A C0 | JMP | \$ |
| 31 | | | |
| 32 C01D | | LABEL_BLK: | |
| 33 C01D | | BLKB | 10,\$A5 |
| 34 | | | |
| 35 C027 | | DATA_DB: | |

再參考下面的Intel Hex format output file片段：

```
:10 C010 00 FF F0 07 99 02 00 C8 4C 0D C0 4C 1A C0 A5 A5 A5 99
:10 C020 00 A5 A5 A5 A5 A5 A5 00 01 02 03 04 05 06 07 08 69
```

\$C01A處是JMP \$指令

從\$C01D開始，連續10個byte的\$A5

- ⑩ DB(Define Byte)directive宣告後面的數值以byte的大小來存放，請參照下面的listfile段：

```
35 C027          DATA DB:
36 C027 00 01 02 03 04 DB $00,$01,$02,$03,$04,$05,$06,$07,$08,$09,$FF
          05 06 07 08 09
          FF
```

- ⑪ DW(Define Word)directive宣告後面的數值以Word大小來存放，參照下面list file片段：

```
38 C032          DATA DW:
39 C032 0000 0100 0200 DW $0000,$0001,$0002,$0003,$0004,$FFFF
          0300 0400 FFFF
```

- ⑫ 宣告以下開始一個名為VECTAB的section，其起始位址為\$FFFA。這個section的宣告，用意就是要組成6502 CPU的中斷向量表。
- ⑬ 告訴assembler，程式到此結束，以下不必再組譯(即使有寫指令也一樣不再組譯)。

DEMO2.ASM

```
FIVE_TIME
    .CHIP      6502
    .SYNTAX    6502
    .SYMBOLS
    .OPTIONS   DCH
    .LINKLIST
    .INCLUDE    MAC.INC                                ①

ADD_COUNT EQU 6                                        ②
IFDEF FIVE_TIME                                       ③
SUB_COUNT = 5                                         ④
ELSE
SUB_COUNT = 4
ENDIF

W_RAM      .SECTION  PAGE0, RANGE $0 $7F, REF_ONLY
TEMP:      DS      1
```

```
P_ROM      .SECTION      OFFSET $C000, RANGE $C000 $FFF9
```

```
POWER_ON:
```

```
    LDX    #$FF
    TXS
    LDA    #0
    STA    TEMP
    LDX    #ADD_COUNT
```

```
?MORE_ADD:
```

```
    ADD    TEMP,TEMP,#3
    DEX
    BNE    ?MORE_ADD
    LDX    #SUB_COUNT
```

```
?MORE_SUB:
```

```
    SUB    TEMP,TEMP,#2
    DEX
    BNE    ?MORE_SUB
    JMP    $
```

```
IRQ_ISR:
```

```
    RTI
```

```
NMI_ISR:
```

```
    RTI
```

```
VECTAB     .SECTION      OFFSET $FFFA, RANGE $FFFA $FFFF
```

```
    DW    IRQ_ISR
    DW    POWER_ON
    DW    NMI_ISR
    END
```

⑤

MAC.INC

```
ADD:       .MACRO        RESULT,OP1,OP2
```

⑥

```
    CLC
    LDA    OP1
    ADC    OP2
    STA    RESULT
    .ENDM
```

⑦

```
SUB:       .MACRO        RESULT,OP1,OP2
```

```
    SEC
    LDA    OP1
    SBC    OP2
    STA    RESULT
    .ENDM
```

說明：

- ① 用INCLUDE directive將MAC.INC含括進來，置於INCLUDE處成為原始程式的一部份。
- ② 告訴assembler，往後碰到ADD_COUNT字串時，將之代換成6這個字(串)。
- ③ 條件式組譯，當條件成立時(程式中定義了FIVE_TIME這個symbol)，則組譯介於IFDEF到ELSE之間的程式，若條件不成立，則組譯ELSE到ENDIF之間的程式。
- ④ EQU directive可以簡寫成等號「=」。
- ⑤ 使用macro (這兒用了ADD macro)來提高程式的閱讀性。
- ⑥ 定義ADD這個macro，注意macro名稱後面要加分號，並以.MACRO關鍵字表明這是一個macro。
- ⑦ 最後以.ENDM表明此macro到此結束。

DEMO3.ASM

| | | | |
|--------------------|------------------|--|---|
| | .CHIP | 6502 | |
| | .SYNTAX | 6502 | |
| | .SYMBOLS | | |
| | .OPTIONS | DCH | |
| | .LINKLIST | | |
| BANK0 | .SECTION BLKB | OFFSET \$8000, RANGE \$8000 \$A000, INDIRECT \$A000-\$8000,\$00 | ① |
| BANK1 | .SECTION BLKB | OFFSET \$8000, RANGE \$8000 \$A000, INDIRECT \$A000-\$8000,\$11 | ② |
| BANK2 | .SECTION BLKB | OFFSET \$8000, RANGE \$8000 \$A000, INDIRECT \$A000-\$8000,\$22 | |
| BANK3 | .SECTION BLKB | OFFSET \$8000, RANGE \$8000 \$A000, INDIRECT \$A000-\$8000,\$33 | |
| W_RAM TEMP: | .SECTION DS 1 | OFFSET 0, RANGE \$0 \$7F, PAGE0, REF_ONLY | ③ |
| P_ROM POWER_ON: | .SECTION | OFFSET \$C000, RANGE \$C000 \$FFF9 | |
| | LDX | #\$FF | |
| | TXS | | |
| | LDA | #\$55 | |
| | STA | TEMP | |
| | JMP | \$ | |

```

IRQ_ISR:
    RTI

NMI_ISR:
    RTI

VECTAB    .SECTION    OFFSET $FFFA, RANGE $FFFA $FFFF
          DW    IRQ_ISR
          DW    POWER_ON
          DW    NMI_ISR
          END

```

說明：

- ① 宣告了一個叫做BANK0的section，並指定它為indirect section。在此，定義indirect section的目的是為了做multi-ROM-bank，因6502的addressing range只有64K byte，在需要大容量資料的場合就顯得不足，所以一般based on 6502 CPU的單晶片都會把bank-switching的機制做進去IC內。而對不同的bank而言，其mapping到6502的實際位址是相同的，所以我們需要利用indirect section的特性，來讓assembler/link允許我們將個別的資料定義在6502相同的位址。
- ② 宣告了一個叫做BANK1的section，並指定它為indirect section。您可以看到，它和BANK0這個section所佔用的位址都相同。
- ③ 當使用了indirect section之後，zero-page section的宣告會和之前有點不同(之前的範例中，宣告的型態是比較省略的方式)，否則會出現組譯錯誤的訊息。

1-5-2 Linker

link.exe操作說明

2500AD 6502 linker提供四種操作模式：prompt mode、command line mode、data file mode和enhanced data file mode，不過在本文件中，我們只說明前三種模式，請從下面的實例中，直接學習這三種操作模式。

2.1 提示模式(Prompt Mode)

請在DOS prompt(不管是純MS-DOS或Windows 95的DOS模式)下，依照linker存放的路徑執行link，如下例：

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\LINK
```

則出現下列訊息...

| | |
|--|---|
| 2500 A.D. Linker Copyright (C) 1990 by 2500AD Software Inc. Version 5.03d | ① |
| Input Filename: DEMO1 (enter) | ② |
| Input Filename: (enter) | ③ |
| Output Filename: (enter) | ④ |
| Library Filename: (enter) | ⑤ |
| Options (D,G,P A,R,S[U] C,M,N,SM,Z E,H,T,X,1,2,3 <CR> = Default) : (enter) | ⑥ |

說明：

- ① 本文件中以5.03d版的linker做為說明對象，您使用的若非此版本，某些地方可能會有差異。
- ② 輸入要link的.OBJ檔檔名(此處以DEMO1檔為例)。
- ③ 如果還有另外一個檔案也要link進來，則在此輸入它的.OBJ檔檔名，若沒有，則直接按enter。
- ④ 在此可以鍵入輸出檔的檔名(如未指定，將以輸入之檔名為之)。
- ⑤ 若有使用library的話則輸入library檔檔名，沒有則按enter。
- ⑥ link option，與原始程式中OPTIONS directive作用相同，直接按enter就會以原始程式指定的option來link。若想要產生binary executable file，則將H參數改為X即可。但對indirect section而言，使用X參數會出現錯誤訊息。

2.2 命令線區模式(Command Line Mode)

請在DOS prompt下，依照linker存放的路徑執行link，並加上參數「-C」，如下例：

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\LINK -C DEMO1
```

2.3 資料檔案模式(Data File Mode)

請先用文字處理器，完成link file，其內容請參照下例...

DEMO1.LNK

| | |
|-------|---|
| DEMO1 | ① |
| | ② |
| | ③ |
| | ④ |
| DCH | ⑤ |

說明：

- ① 這行所要輸入的字串，相當於在Prompt mode下，第一次要輸入的參數。(Input Filename)
- ② 這行所要輸入的字串，相當於在Prompt mode下，第二次要輸入的參數。(Input Filename)
- ③ 這行所要輸入的字串，相當於在Prompt mode下，第三次要輸入的參數。(Output Filename)
- ④ 這行所要輸入的字串，相當於在Prompt mode下，第四次要輸入的參數。(Library Filename)
- ⑤ 這行所要輸入的字串，相當於在Prompt mode下，第五次要輸入的參數。(Options)

在完成link file後，要執行link動作時，在DOS prompt下輸入：(此即為資料檔案模式)

```
D:\Doc\Doc\tool\guide>C:\SOFT-DOS\X6502\LINK DEMO1
```