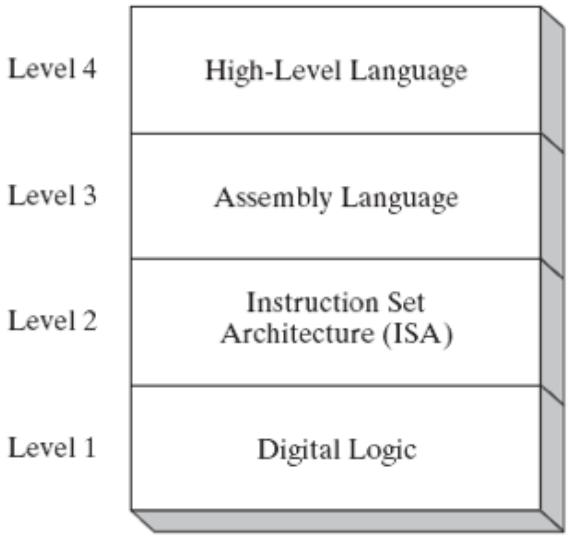
組合語言

tags: 大學必修-筆記

CH1 基本觀念

架構



- Level 4 -> C++、Java...
- Level 3 -> 組語
- Level 2 -> ISA
- Level 1 -> 邏輯層

CH2 x86 程序架構

- IA-32 -> 32 bits Intel Architecture (32位頻寬Intel架構)

General-Purpose Registers (GPA) 通用暫存器

暫存器名稱	中文名稱	用途
AX	累加器暫存器	用在算術運算
BX	基址暫存器	作為一個指向資料的指標
CX	計數器暫存器	用於移位/迴圈指令和迴圈
DX	資料暫存器	用在算術運算和I/O操作
SP	堆疊指標暫存器	用於指向堆疊的**頂部**
BP	棧基址指標暫存器	用於指向堆疊的**底部**
SI	源變址暫存器	在流操作中用作源的一個指標
DI	標的索引暫存器	用作在流操作中指向標的的指標

暫存器		累加器		基址		計數器		資料
32-bit		EAX		EBX		ECX		EDX
16-bit		AX		BX		CX		DX
8-bit	AH	AL	BH	BL	CH	CL	DH	DL

暫存器	堆疊指標	棧基址指標	源變址	標的索引
32-bit	ESP	EBP	ESI	EDI
16-bit	SP	BP	SI	DI

- AH 與 AL 實際上是獨立不互相影響的，但對於 AX 來說 AH 與 AL 又是其一部份

Segment 區段暫存器

暫存器名稱	中文名稱
CS	程式區段暫存器
DS	資料區段暫存器
SS	堆疊區段暫存器
ES	額外區段暫存器
FS	額外區段暫存器
GS	額外區段暫存器

EFLAGS 暫存器

FLAG名稱	中文名稱
CF (Carry)	進位旗標
OF (Overflow)	溢位旗標
ZF (Zero)	零值旗標
AF (Auxiliary)	輔助進位旗標
PF (Parity)	奇偶旗標
DF (Direction)	方向旗標
IF (Interrupt)	中斷旗標
TF (Trap)	單步旗標

Instruction Pointer Register

- 簡稱 IP
- 用來指向下一個 CPU 準備執行的指令所在的 memory address
- 當指令執行後，IP 就會自動指向下一個準備執行的指令所在的 memory address。

CH3 基本組語程式架構

Basic Element

- 數字結尾
  - h → 16進位
  - d → 10進位
  - b → 2進位
- 指令格式
  - label
  - mnemonic
  - operand
  - comment → 用「;」
  - label: mnemonic [operands] [;comment]

- 指令範本

```
TITLE Program Template           (Template.asm)

; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date:              Modified by:

INCLUDE Irvine32.inc
.data
        ; (這裡插入變數)
.code
main PROC
        ; (這裡插入可執行的 instructions)
        exit
main ENDP
        ; (這裡插入額外的程序)
END main
```

Data Types

Unsigned Data Types	功用	Signed Data Types	功用
BYTE	8-bit unsigned integer	SBYTE	8-bit signed integer
WORD	16-bit unsigned integer	SWORD	16-bit signed integer
DWORD	32-bit unsigned integer	SDWORD	32-bit signed integer
QWORD	64-bit integer		
TBYTE	80-bit integer		

Data Types	功用
REAL4	32-bit (4 byte) IEEE短實數
REAL8	64-bit (8 byte) IEEE長實數
REAL10	80-bit (10 byte) IEEE延伸實數

定義

- 定義變數

```
名稱  型態  值

value BYTE 10  ->  定義一個BYTE 名字value 值為10

value BYTE ?   ->  定義一個BYTE 名字value 值未定
```

- 定義矩陣

```
list BYTE 10, 20, 30, 40  -> 定義一個BYTE 大小有四格的矩陣

list BYTE ?, 32, 41h, 00100010b
```

```
利用 DUP()

list BYTE 20 DUP(0)  ->  定義一個大小為 20 byte(格)，內容皆為 0 的矩陣
```

- 定義字串

```
string BYTE "Hallo World!", 0  -> 記得要加上 0，表示結束

string BYTE "Hallo World!", 0dh, 0ah, 0  ->  表示換行
```

directive

- Directive 則是人自己手動加入的，而不屬於 CPU 的指令，作用有兩種：
  - 指示 assembler 要作某些事情
  - 告知 assembler 某些訊息

- 而這個部分，並不會被 assembler 轉為 machine code，一般常見的 directive 用法大概有幾種：
  - 定義常數(constant)
  - 定義儲存資料的記憶體位置
  - 將多段記憶體空間組合為 segment
  - 視情況 include 外部的 source code
  - include 其他檔案

EQU & Symbolic Constants

$ (current location counter)

- 它會一直往後指，不斷更新目前的位置

```
list BYTE 10, 20, 30, 40
ListSize = ($ - list)        ; 得到 list 矩陣的 index 個數

list WORD 1000h, 2000h, 3000h, 4000h
ListSize = ($ - list) / 2    ; 得到 list 矩陣的 index 個數

list DWORD 1, 2, 3, 4
ListSize = ($ - list) / 4    ; 得到 list 矩陣的 index 個數
```

EQU

- 用來定義有名稱的常數
- 定義好之後，值不可再度變更
- 類似 #define

```
PI EQU <3.1416>

move EAX, PI
```

TEXTEQU

- 可以用來定義定串，變成一個變數名稱
- 字串可以是文字、計算式、指令

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">  // 定義為文字

count TEXTEQU %(rowSize * 2)                            // 定義為計算式

setupAL TEXTEQU <mov al, count>                         // 定義為指令
```

= (equal-sign directive)

- 用來定義有名稱的常數
- 定義好之後可以再度變更

```
COUNT = 500
mov al,COUNT
```

矩陣

- 存到記憶體裡的順序
  - 用小頭端 (little-endian)
  - 多位元組數值的最低位位元組首先寫入
  - 高位元去大的位置、低位元去小的位置
  - ex：list DWORD 12345678h

- 矩陣的存取
  - 加上 [] 表示要取值
  - 每移動的距離以 byte 為單位

```
;定義兩個陣列，每個元素的大小分別為 byte 與 word
byteArray   db  5, 4, 3, 2, 1
wordArray   dw  5, 4, 3, 2, 1

;存取 byte 陣列
mov     al, [byteArray]         ;AL = byteArray[0]
mov     al, [byteArray + 1]     ;AL = byteArray[1]
mov     [byteArray + 3], al     ;byteArray[3] = AL

;存取 word 陣列
mov     ax, [wordArray]         ;AX = wordArray[0]
mov     ax, [wordArray + 2]     ;AX = wordArray[1] 注意! 這邊 + 2 bytes 即是 + 1 wo
mov     [wordArray + 6], ax     ;wordArray[3] = AX
mov     ax, [wordArray + 1]     ;這是錯誤的用法，因為元素大小為 word(2 bytes)

;存取 dword 陣列
mov     ax, [dwordArray]         ;AX = dwordArray[0]
mov     ax, [dwordArray + 4]     ;AX = dwordArray[1] 注意! 這邊 + 4 bytes 即是 + 1
```

CH4 Data Transfers, Addressing, and Arithmetic

Instruction operands 指令運算元

- **register**
  - 指向儲存於 CPU register 中的值。
- **memory**
  - 指向儲存於 memory 中的資料，而 memory address 有可能是直接寫於指令中，或是由 register 的值所組成。
- **immediate**
  - 列於指令中的固定值，儲存於指令本身(在 code segment 中)，而非在 data segment 中。
  - 也就是常數，ex：45
- **implied**
  - 此種 operand 不會明確的顯示。例如：將某個 register 中的值加 1。

Instruction 指令

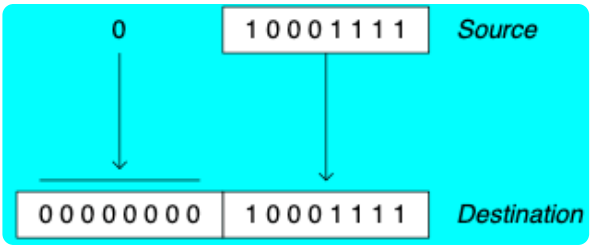
MOV 搬移資料

```
MOV 目地, 來源
```

- 目地及來源，都不能是 immediate (變數名稱)
- 目地及來源，不能一起是 memory (變數名稱)
- 目地及來源，都必須相同大小 (表示 AX 中的值不能 mov 到 BL 中)

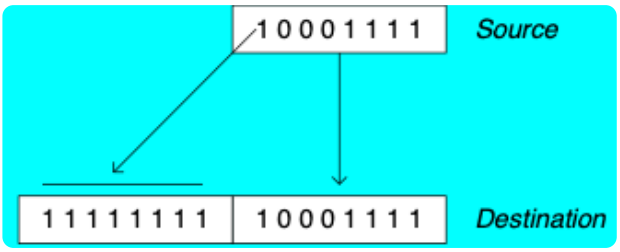
MOVZX 搬移資料(小到大)

- Zero Extension
- 當從 AL 搬到 AX 時 (小搬大)
- AH 的地方會填滿 0



MOVSX 搬移資料(小到大)

- Sign Extension
- 當從 AL 搬到 AX 時 (小搬大)
- AH 的地方會根據你目前的 sign 值(MSB)，來把它填滿



XCHG 交換資料

```
XCHG 要互換的資料, 要互換的資料
```

- 目地及來源，都不能是 immediate (變數名稱)
- 目地及來源，不能一起是 memory (變數名稱)

INC 加一 & DEC 減一

```
INC 變數名稱
DEC 變數名稱
```

```
inc myWord            ; 1001h
dec myWord            ; 1000h
inc myDword           ; 10000001h

mov ax, 00FFh
inc ax                ; AX = 0100h
mov ax, 00FFh
inc al                ; AX = 0000h
```

```
myByte BYTE 0FFh, 0

mov al, myByte        ; AL = FFh
mov ah, [myByte+1]    ; AH = 00h
dec ah                ; AH = FFh
inc al                ; AL = 00h
dec ax                ; AX = FEFF
```

ADD 加 & SUB 減

```
ADD 目標變數, [數字, 變數]
SUB 目標變數, [數字, 變數]
```

```
.data
    var1 DWORD 10000h
    var2 DWORD 20000h

.code
    mov eax, var1        ; 00010000h
    add eax, var2        ; 00030000h
    add ax, 0FFFFh       ; 0003FFFFh
    add eax, 1           ; 00040000h
    sub ax, 1            ; 0004FFFFh
```

NEG 正負號交換

```
NEG [變數名稱, 暫存器]
```

- neg 也就是把那個數字的位元 變成二補數
- 因為 0 的二補數還是 0 (00000000)，所以 neg 0 是 0
- 因為 -128 的二補數還是 -128 (10000000)，所以 neg -128 是 -128

```
.data
    valB BYTE -1
    valW WORD +32767
.code
    mov al,valB       ; AL = -1
    neg al            ; AL = +1
    neg valW          ; valW = -32767
```

Flag

- 當 Flag 值為 1 時 -> set
- 當 Flag 值為 0 時 -> clear

Zero Flag (ZF) 看結果是否為 0

- 如果 ZF = 1 -> 值為 0
- 如果 ZF = 0 -> 值不為 0

```
mov cx, 1
sub cx, 1       ; CX = 0, ZF = 1
mov ax, 0FFFFh
inc ax          ; AX = 0, ZF = 1
inc ax          ; AX = 1, ZF = 0
```

Sign Flag (SF) 看是正是負

- 如果 SF = 0 -> 值為正的

- 如果 SF = 1 -> 值為負的
- SF 的判斷其實就是看最高的位元數字 (MSB)

```
mov al, 0
sub al, 1            ; AL = 11111111b, SF = 1
add al, 2            ; AL = 00000001b, SF = 0
```

Carry Flag (CF) 看 unsigned value 是否超出邊界

- 看無號數 unsigned value
- 當 相加 $\geq 2^n$ 時 or 最高位有 進位 時
- 當相減 由小減大 時 or 最高位有 借位 時
- 如果一個運算的結果超過 [0, 255]

```
mov al, 0FFh
add al, 1       ; CF = 1, AL = 00

mov al, 0
sub al, 1       ; CF = 1, AL = FF
```

- inc 與 dec指令不會影響進位旗標

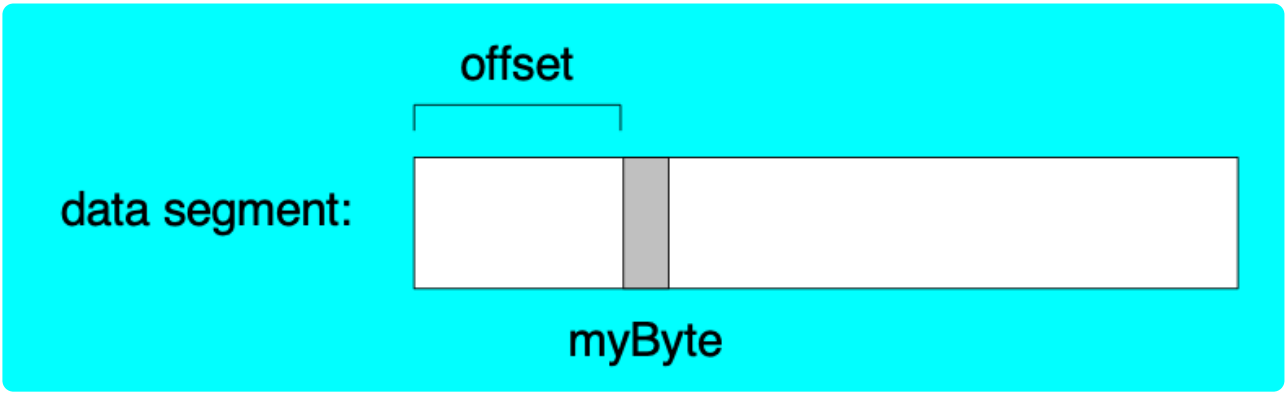
Overflow Flag (OF) 看 signed value 是否超出邊界

- 看有號數 signed value
- 當兩個 正數 相加變成 負數 時
- 當兩個 負數 相加變成 正數 時
- 如果一個運算的結果超過 [-128, 127]

```
mov al, -128
neg al              ; CF = 1    OF = 1

mov ax, 8000h
add ax, 2           ; CF = 0    OF = 0

mov ax, 0
sub ax, 2           ; CF = 1    OF = 0

mov al, -5
sub al, +125        ; OF = 1
```

Data-Related Operators and Directives

OFFSET



`mov esi, OFFSET` 要看起使位置的變數

- 傳回變數從所在區段開始的 偏移 距離
- 也就是傳回變數的起使位置

```
.data
    bVal BYTE ?
    wVal WORD ?
    dVal DWORD ?
    dVal2 DWORD ?
.code            ;  assume bVal's offset : 00404000
    mov esi, OFFSET bVal          ; ESI = 00404000
    mov esi, OFFSET wVal          ; ESI = 00404001 因為 BYTE 1 個 byte
    mov esi, OFFSET dVal          ; ESI = 00404003 因為 WORD 2 個 byte
    mov esi, OFFSET dVal2         ; ESI = 00404007 因為 DWORD 4 個 byte
```

PTR

- 存取的變數大小與當初宣告的變數不同

`型態 PTR 變數名`

- 由大的型態轉成小的型態
- 轉成多大的型態，就從最小位元開始算幾個byte

```
.data
myDouble DWORD 12345678h

mov al, BYTE PTR  myDouble              ; AL = 78h
mov al, BYTE PTR [myDouble+1]           ; AL = 56h
mov al, BYTE PTR [myDouble+2]           ; AL = 34h
mov ax, WORD PTR  myDouble              ; AX = 5678h
mov ax, WORD PTR [myDouble+2]           ; AX = 1234h
```

- 由小的型態轉成大的型態
- CPU 會自動把 byte 顛倒過來

```
.data
    myBytes BYTE 12h, 34h, 56h, 78h

.code
    mov ax, WORD PTR [myBytes]          ; AX = 3412h
    mov ax, WORD PTR [myBytes+2]        ; AX = 7856h
    mov eax, DWORD PTR myBytes          ; EAX = 78563412h
```

TYPE

- 可以回傳以 byte 為單位的大小值

```
.data
    var1 BYTE ?
    var2 WORD ?
    var3 DWORD ?
    var4 QWORD ?

.code
    mov eax,TYPE var1   ; 1
    mov eax,TYPE var2   ; 2
    mov eax,TYPE var3   ; 4
    mov eax,TYPE var4   ; 8
```

LENGTHOF

- 計算陣列中的元件數

```
byte1  BYTE 10, 20, 30           ; 3

array1 WORD 30 DUP(?), 0, 0      ; 32
array2 WORD 5 DUP(3 DUP(?))      ; 15

digitStr DWORD "12345678", 0     ; 9

mov ecx, LENGTHOF array1         ; 32
```

SIZEOF

- 傳回的值等於 LENGTHOF 乘上 TYPE 的值

```
byte1  BYTE 10,20,30            ; 3 = 3 * 1

array1 WORD 30 DUP(?),0,0       ; 64 = 32 * 2
array2 WORD 5 DUP(3 DUP(?))     ; 30 = 15 * 2

digitStr DWORD "12345678",0     ; 36 = 9 * 4
```

- 如果目標矩陣為二元矩陣時
- LENGTHOF 及 SIZEOF 只會計算第一列定義的部分

```
.data
array   WORD 10, 20
        WORD 30, 40
        WORD 50, 60

.code
    mov eax, LENGTHOF array     ; 2 -> 只會看 WORD 10,20
    mov ebx, SIZEOF array       ; 4 -> 只會看 WORD 10,20
```

LABEL

- 賦予下一行宣告的資料另一個名字和另一種型態
- LABEL 那一行並不會在記憶體獲取空間
- 取代 PTR 的效果

```
.data
    dwList   LABEL DWORD
    wordList LABEL WORD
    intList  BYTE 00h, 10h, 00h, 20h
.code
    mov eax,dwList          ; 20001000h
    mov cx,wordList         ; 1000h
    mov dl,intList          ; 00h

```

Indirect Addressing 間接定址

- 間接定址 -> 暫存器當作指標
- 利用指標的方式去存取矩陣中的值
- 跟 C 的指標很像

Indirect Operands 間接運算元

- 利用 esi 當作指標，去存取矩陣的元素
- esi 指標加 1 * 矩陣大小(byte) 後為往下一個格子存取

```
.data
    val1 BYTE 10h, 20h, 30h

.code
    mov esi,OFFSET val1
    mov al,[esi]        ; dereference ESI (AL = 10h)

    inc esi
    mov al,[esi]        ; AL = 20h

    inc esi
    mov al,[esi]        ; AL = 30h
```

- 如果 OFFSET 的目標的一個變數而不是矩陣的話

```
.data
    myCount WORD 0

.code
    mov esi,OFFSET myCount
    inc esi                 ; 錯誤
    inc WORD PTR [esi]      ; 要給一個確定的大小才是對的

```

Indexed Operands 索引運算元

- 另一種存取矩陣的方法

常數[暫存器]

[常數 + 暫存器]

```
.data
    arrayW WORD 1000h, 2000h, 3000h

.code
    mov esi, 0
    mov ax, [arrayW + esi]              ; AX = 1000h
    mov ax, arrayW[esi]         ; 另一種寫法

    add esi, 2
    add ax, [arrayW + esi]
```

Pointers

- 直接定義一個 pointer varible 指標變數
- 去存取其它變數的位置
- 指標大小必須為 DWORD

```
.data
    arrayW WORD 1000h, 2000h, 3000h
    ptrW DWORD arrayW              ; 定義一個指標變數
    ptrW DWORD OFFSET arrayW       ; 另一種寫法

.code
    mov esi, ptrW         ; 把指標變數丟給 esi 暫存器
    mov ax, [esi]          ; AX = 1000h
```

JMP and LOOP Instructions 迴圈

JMP 架構

```
JMP 目的標籤
```

```
top:
    ...

    jmp top             ; 跳到 top 標籤
```

LOOP 架構

- 使用 ecx 暫存器，存入要重複的次數
- ecx 會先減 1 再進入迴圈
- JMP 就會重覆執行特定區段的指令
- 迴圈的目的必須在現在位置-128到+127之間

```
mov ax, 0
mov ecx, 5

L1:
    inc ax
    loop L1     ; 迴圈中止後 ax = 5, ecx = 0
```

- 如果 ecx 為 0 時
- 因為 ecx 會先減 1 再進入迴圈

```
mov ecx,0

X2:
    inc ax
    loop X2     ; 迴圈中止後 ax = 4,294,967,296, ecx = 0
```

Nested Loop

```
.data
    count DWORD ?

.code
    mov ecx,100             ; 設定 L1 外迴圈的次數
L1:
    mov count,ecx           ; 把 L1 的次數存起來
    mov ecx,20              ; 設定 L2 外迴圈的次數
L2:
    ...
    loop L2                 ; 重複 L2 迴圈
    mov ecx, count          ; 把 L1 的迴圈次數存回來
    loop L1                 ; 重複 L1 迴圈
```

Ex：把矩陣數字加總

```
intarray WORD 100h, 200h, 300h, 400h

mov edi, OFFSET intarray         ;「 指向 intarray 位址 」
mov ecx, LENGTHOF intarray       ; 「迴圈次數」
mov ax, 0                        ; 歸零

L1:
    add ax, [edi]                ; 加上一整數，加上[]表示取值
    add edi, TYPE intarray       ; 指向下一個整數
    loop L1                      ; 重複直到 ecx = 0
```

Ex：字串複製

```
source BYTE "This is the source string", 0
target BYTE SIZEOF source DUP(0)

mov esi, 0                       ; 索引暫存器
mov ecx, SIZEOF source           ; 迴圈次數

L1:
    mov al, source[esi]          ; 從來源取得字元
    mov target[esi], al          ; 儲存在目標
    inc esi                      ; 移到下一個字元
    loop L1                      ; 重複整個字串
```

CH5 Procedures 程序

使用 Library

- 先 INCLUDE Irvine32
- 作者寫的 Library
- 用 call 去叫 Function

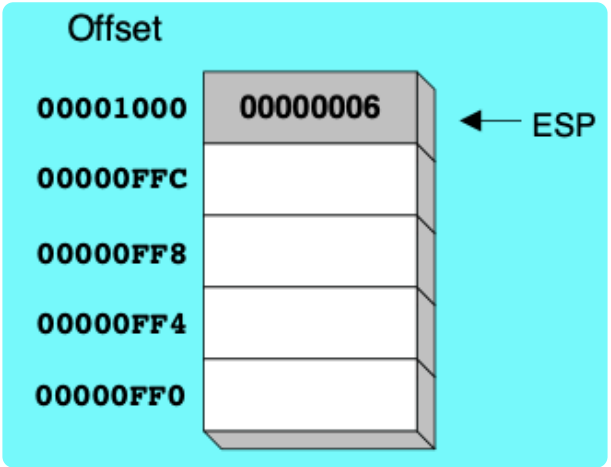
```
INCLUDE Irvine32.inc

.code
    mov eax,1234h
    call WriteHex       ; print 出 hex
    call Crlf           ; 結束一行

```

一些 function

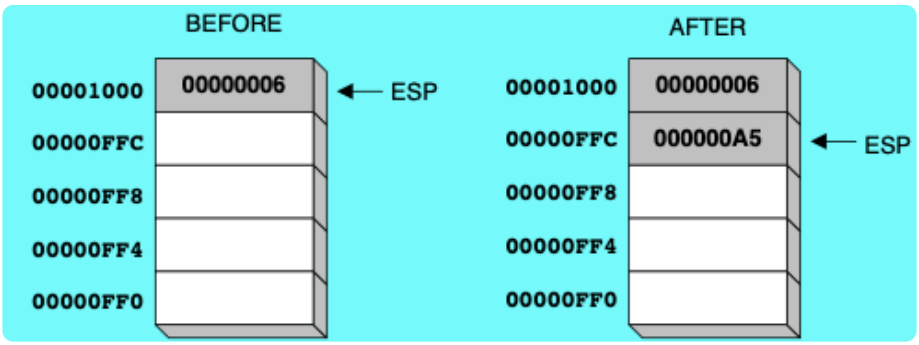
- call Clrscr 清空螢幕
- call DumpRegs 把暫存器和 flag 的值 print 出來
- call Crlf 換行
- call WriteBin 印成二進位
- call WriteDec 印成十進位
- call WriteHex 印成十六進位
- call WriteString 印字串

Stack Operations 堆疊操作

- 堆疊 -> 後進先出 (LIFO)
- 使用 ESP (stack pointer) 來指向堆疊



PUSH



1. 先將堆疊指標的值 減四 (在保護模式下)
2. 再將資料拷貝到堆疊指標所指的位置

POP

1. 複製堆疊中由 esp 所指向的內容到一個16-bit或 32-bit 的目的運算元
2. 並存在變數或暫存器裡
3. 再將 ESP 加 N
4. 如果指到 16-bit → esp+2
5. 如果指到 32-bit → esp+4

其它堆疊指令

- PUSHFD
  - 將 32-bit旗標暫存器PUSH到堆疊
- POPFD
  - 由堆疊中將旗標資料POP到暫存器
- PUSHAD
  - 將一般用途的32-bit暫存器以 eax ecx edx ebx esp ebp esi edi 順序 push 到堆疊
- POPAD
  - 以相反順序POP出來
- PUSHA
  - 將一般用途的16-bit途暫存器以 ax cx dx bx sp bp si di 順序 push 到堆疊
- POPA
  - 以相反順序POP出來

指令操作

- ex：用 Stack 來實作 Nested Loop

```
mov ecx, 100            ; 設定外部迴圈次數

L1:                     ; 外部迴圈開始
    push ecx            ; 把目前外部迴圈的次數存起來

    mov ecx, 20         ; 設定內部迴圈次數
L2:                     ; 內部迴圈開始
    ...
    loop L2             ; 回 L2

    pop ecx             ; 把外部迴圈的次數重新存回 ecx
    loop L1             ; 回 L1
```

Defining & Using Procedures 定義及使用程序

- 大問題可以分成一個個小區塊
- 這樣比較好管理
- 類似 function 的功能

創造一個程序

```
程序名稱 PROC

    ......

    ret
程序名稱 ENDP
```

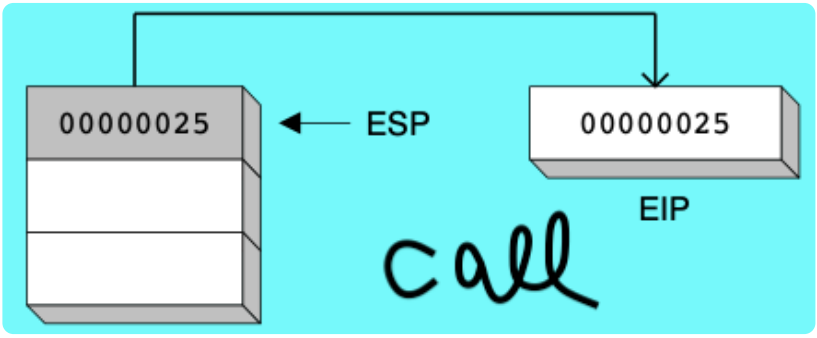
程序例子 call & ret

```
main PROC             // 以下為各指令的 OFFSET (位置)
    call MySub        // 00000020
    mov eax, ebx      // 00000025 -> mov 的 offset
main ENDP

MySub PROC
    mov eax, edx      // 00000040 -> MySub 程序第一行的 offset
    ret
MySub ENDP
```

- call 呼叫程序
- ret (return) 回傳
- call 會 push 00000025 到 stack 上

- 然後把 00000040 存進 EIP(下一個執行指標) 裡
  - 也就是把 call 下一行的 offset 先存進 stack 裡
  - 這樣到時候 call 跑完後才會知道要從 main 的哪裡繼續執行

- ret 會從 stack pop 出 00000025 到 EIP 裡面
  - 把剛剛存的 offset 重新存回 EIP
  - 回到 main 繼續執行



傳遞參數

- 一般都會使用暫存器作為存參數的地方
- 不會有 global local 的問題

```
1  theSum DWORD ?
2      mov eax, 10000h             ; 參數 1
3      mov ebx, 20000h             ; 參數 2
4      mov ecx, 30000h             ; 參數 3
5      call SumOf                  ; eax = ( eax + ebx + ecx)
6      mov theSum, eax             ; 存到 theSum 裡
```

USES 運算子

```
程序名稱 PROC USES [暫存器名稱, 暫存器名稱, ...]
```

- 列出程序中使用到的暫存器，就會被系統保留起來
- 在程序開始之初產生push指令

- 將暫存器值儲存到堆疊中
- 程序結束時產生pop指令回復

```
ArraySum PROC USES esi ecx
    mov eax, 0
 L1:
    add eax, [esi]
    add esi, 4
    loop L1
    ret
ArraySum ENDP
```

- 編碼會被組譯器產生

```
ArraySum PROC
    push esi        // 由組譯器產生
    push ecx        // 由組譯器產生
    mov eax, 0

L1:
    add eax, [esi]
    add esi, 4
    loop L1
    pop ecx         // 由組譯器產生
    pop esi         // 由組譯器產生
    ret
ArraySum ENDP
```

CH6 Conditional Processing 條件處理

布林值運算

AND

- 兩運算元中每對相對應位元間的 and 布林運算，**結果存放目的運算元**
- 總是清除 OF 及 CF，根據目的運算元的值修改 SF、ZF、PF

```
AND 目的地, 來源


00111011
00001111 (and
-------------
00001011
```

- 功能：將字元轉換成大寫字母

```
01100001 = 61h ('a')
11011111 = DFh (and         ; 將字元與 11011111 作 and 運算轉成大寫
---------------------------------
01000001 = 41h ('A')
```

OR

- 兩運算元中每對相對應位元間的 or 布林運算，**結果存放目的運算元**
- 總是清除 OF 及 CF，根據目的運算元的值修改 SF、ZF、PF

`OR 目的地, 來源`

```
00111011
00001111 (or
-------------
00111111
```

- 功能：將0~9整數位元組轉換成 ASCII

```
00000101 = 05h
00110000 = 30h (or       ; 將字元與 00110000 作 or 運算轉成ascii
---------------------------------
00110101 = 35h ('5')
```

XOR

- 兩運算元中每對相對應位元間的 互斥 布林運算，**結果存放目的運算元**
- 總是清除 OF 及 CF，根據目的運算元的值修改 SF、ZF、PF

`XOR 目的地, 來源`

```
00111011
00001111 (xor
-------------
00110100
```

NOT

- 把 0 變 1，1 變 0
- 不影響任何旗標

`NOT 目的地, 來源`

```
00111011 (not
-------------
11000100
```

TEST

`TEST 變數名稱, 變數名稱`

- 其實就是兩變數做 AND
- 但是結果 不會存進目的變數裡面
- 不過結果會觸發 ZF(zero flag)
- 常被用於 發現運算元個別的各個位元是否為 1

- EX：看看第 0、1 位元是否為 1

```
mov al, 00001101b          ; 輸入值
test al, 00000001b         ; 測試位元 0，結果為 00000001b，ZF = 0
-----------------------------
mov al, 00001101b          ; 輸入值
test al, 00000010b         ; 測試位元 1，結果為 00000000b，ZF = 1
```

CMP

`CMP 目的地, 來源`

- 兩變數作比較，看誰大誰小
- 其實就是把兩個變數 相減 ，目的變數減來源變數
- 結果不會存進目的變數裡面

- 與 flag 的互動

- 無號數
  - 如果目的 > 來源 → ZF = 0, OF = 0
  - 如果目的 = 來源 → ZF = 1, OF = 0
  - 如果目的 < 來源 → ZF = 0, OF = 1

- 有號數

- 如果目的 > 來源 → SF = OF
- 如果目的 = 來源 → ZF = 1
- 如果目的 < 來源 → SF ≠ OF

清除或設定個別 CPU flag

- 設定或清除 zero flag

```
and al, 0          ; zero flag = 1，任何運算元和 0 作 and 運算
or al, 1           ; zero flag = 0，任何運算元和 1 作 or 運算
```

- 設定或清除 sign flag

```
or al, 80h         ; sign flag = 1，任何運算元和 100000000 作 or 運算
and al, 7Fh        ; sign flag = 0，任何運算元和 01111111 作 and 運算
```

- 設定或清除 carry flag

```
stc                ; carry flag = 1
clc                ; carry flag = 0
```

- 設定或清除 overflow flag
  - 設定溢位旗標
    - 將兩個正位元組值相加產生的和為負
  - 清除溢位旗標
    - 將一運元和 0 作 or 運算

```
mov al, 7Fh         ; al = +127
inc al              ; al = 80h(-128) , overflow flag = 1
or eax, 0           ; overflow flag = 0
```

Conditional Jmps 條件跳越

- 條件跳越指令的類型
  - 特殊的 flag 值
  - 兩運算元是否相等
  - Unsigned 數字間的比較
  - Sign 數字間的比較

- 一些幫助記憶的代碼
  - JA 無號數的大於、JB 無號數的小於
  - JG 有號數的大於、JL 有號數的小於
  - E -> equal 等於
  - N -> not 反向

- 特別的 jump
  - JB、JC -> 如果 Carry flag 是 1 會跳到這個 label
  - JE、JZ -> 如果 Zero flag 是 1 會跳到這個 label
  - JS -> 如果 Sign flag 是 1 會跳到這個 label
  - JNE、JNZ -> 如果 Zero flag 是 0 會跳到這個 label
  - JECXZ -> 如果 ECX 是 0 會跳到這個 label

特殊的 flag 值

助憶符	說明	旗標
JZ	若為零則跳	ZF = 1
JNZ	若不為零則跳	ZF = 0
JC	若進位則跳	CF = 1
JNC	若不進位則跳	CF = 0
JO	若溢位則跳	OF = 1
JNO	若不溢位則跳	OF = 0
JS	若負號則跳	SF = 1
JNS	若非負號則跳	SF = 0
JP	同位(偶)則跳	PF = 1
JNP	非同位(偶)則跳	PF = 0

兩運算元是否相等

助憶符	說明
JE	相等則跳
JNE	不相等則跳
JCXZ	若 CX = 0 則跳
JECXZ	若 ECX = 0 則跳

Unsigned 數字間的比較

助憶符	說明
JA	較大則跳
JNBE	不是較小或相等則跳 (=JA)
JAE	較大或相等則跳
JNB	不是較小則跳 (=JAE)
JB	較小則跳
JNAE	不是較大或相等則跳 (=JB)
JBE	較小或相等則跳
JNA	不是較大則跳 (=JBE)

Sign 數字間的比較

助憶符	說明
JG	較大則跳
JNLE	不是較小或相等則跳 (=JG)
JGE	較大或相等則跳
JNL	不是較小則跳 (=JGE)
JL	較小則跳
JNGE	不是較大或相等則跳 (=JL)
JLE	較小或相等則跳
JNG	不是較大則跳 (=JLE)

一些例子

- 任務：如果 **unsigned** EAX 大於 EBX，就跳到標籤

```
cmp eax, ebx
ja  Larger
```

- 任務：如果 **signed** EAX 大於 EBX，就跳到標籤

```
cmp eax, ebx
jg  Greater
```

- 任務：如果 **unsigned** EAX 小於等於 Val1，就跳到標籤

```
cmp EAX, Val1
jbe  L1
```

- 任務：如果**signed** EAX 小於等於 Val1，就跳到標籤

```
cmp EAX, Val1
jle  L1
```

- 任務：比較 EAX EBX，把比較大的那個存進 Large 變數裡

```
    mov Large, bx
    cmp ax, bx
    jna Next
    mov Large, ax
Next:
```

- 任務：如果 ESI 指向的 WORD 記憶體大小等於 0，就跳到標籤

```
cmp WORD PTR [esi], 0
je L1
```

- 任務：如果 ESI 指向的 DWORD 記憶體大小是偶數，就跳到標籤

```
test DWORD PTR [edi], 1        // 與 1 作 and
jz L2
```

- 任務：如果 AL 裡的第0, 1, 3位元都是 1 的話，就跳到標籤

```
and al,00001011b        ; 清空不須要的 bit
cmp al,00001011b        ; 檢查剩下的 bit
je L1
```

位元測試指令

BT (Bit Test)

```
bt bitBase, n
   目標暫存器, 第n個位元
```

- 從一個暫存器中，把第 n 個 bit 複制到 Carry flag 去
- EX：如果 AX 中第 9 個 bit 是 1 就跳到 L1 去

```
bt ax, 9        ; CF = bit 9
jc L1           ; jump if Carry
```

條件迴圈指令

LOOPZ & LOOPE

- LOOPZ 與 LOOPE 相同
- LOOPZ (loop if zero)
- 允許迴圈繼續的條件
  - Zero flag 值為 1
  - ecx > 0
- 運行邏輯
  1. ecx = ecx - 1
  2. 若 ecx > 0 且 ZF = 1，則跳至目的
  3. 否則不進行跳越而繼續下一個指令
- 常用在檢查矩陣中的值是否等於一個給定的值

LOOPNZ & LOOPNE

- LOOPNZ & LOOPNE 相同
- LOOPNZ (loop if not zero)
- 允許迴圈繼續的條件
  - Zero flag 值為 0
  - ecx > 0

- 運行邏輯
  1. ecx = ecx - 1
  2. 若 ecx > 0 且 ZF = 0，則跳至目的
  3. 否則不進行跳越而繼續下一個指令
- 常用在檢查矩陣中的值是否不等於一個給定的值

EX：在矩陣中找第一個正數

```
.data
    array SWORD -3, -6, -1, -10, 10, 30, 40, 4
    sentinel SWORD 0
.code
    mov esi, OFFSET array
    mov ecx, LENGTHOF array
next:
    test WORD PTR [esi], 8000h          ; test sign bit
    pushfd                              ; 將flag放入堆疊
    add esi, TYPE array
    popfd                               ; 將flag從堆疊中拿出
    loopnz next                         ; 迴圈繼續
    jnz quit                            ; none found
    sub esi, TYPE array                 ; ESI 指向該值
quit:

```

條件結構 (Conditional Structures)

If 結構

```
if( op1 == op2 )
  X = 1;
else
  X = 2;
--------------------------------
    mov eax, op1
    cmp eax, op2
    jne L1
    mov X, 1
    jmp L2
L1:
    mov X, 2
L2:
```

```
if( ebx <= ecx )
{
    eax = 5;
    edx = 6;
}
--------------------------------
    cmp ebx, ecx
    ja  next
    mov eax, 5
    mov edx, 6
next:
```

if 中有 and

```
if (al > bl) && (bl > cl)
    X = 1;
--------------------------------
// 用正面的邏輯去實作
// code 比較長
cmp al, bl        ; 第一次比較
    ja L1
    jmp next
L1:
    cmp bl, cl    ; 第二次比較
    ja L2
    jmp next
L2:               ; 都是對的時
    mov X, 1      ; 把 X 設成 1
next:
--------------------------------
// 用反面的邏輯去實作
// code 比較短
cmp al, bl        ; 第一次比較
    jbe next      ; 離開 if false
    cmp bl, cl    ; 第二次比較
    jbe next      ; 離開 if false
    mov X, 1      ; 都是對的時
next:
```

if 中有 OR

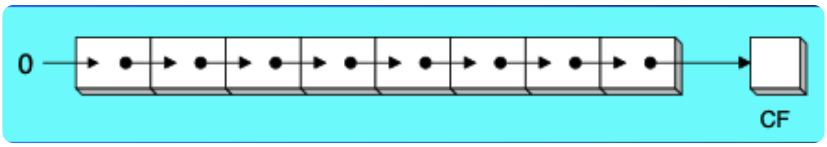
```
if (al > bl) || (bl > cl)
    X = 1;
--------------------------------
cmp al, bl        ; 看條件一是不是對的
    ja L1         ; 對的，跳到 L1
    cmp bl, cl    ; 錯的，看條件二是不是對的
    jbe next      ; 錯的，跳過 L1 的標籤
L1:
    mov X, 1      ; set X to 1
next:
```

while 結構

```
while( eax < ebx)
    eax = eax + 1;
-------------------------------
top:
    cmp eax, ebx    ; 看看條件是不是對的
    jae next        ; 如果錯的話，離開迴圈
    inc eax
    jmp top         ; 重複迴圈
next:
```

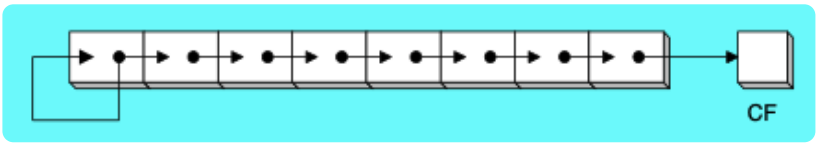
CH7 Integer Arithmetic 整數算數

logical shift 邏輯位移



- 在移動位元後
- 在新建立的位元位置， 填入 0

Arithmetic Shifts 算術位移



- 在移動位元後
- 在新建立的位元位置，填入 該數值原本的正負號位元

SHL (shift left)

- 對目標運算元執行 向左 邏輯移位 ，並且將最低位元填入 0。

```
shl reg, imm8
shl mem, imm8
shl reg, CL
shl mem, CL
```

- SHL 可以執行 2 的次方 數的高速乘法

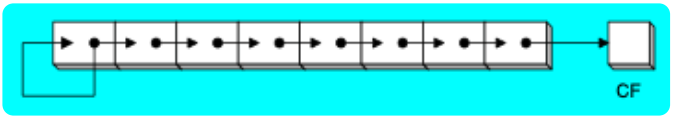
```
EX： 5 * 2 ^ 2 = 20

mov dl, 5
shl dl, 2        ; dl = 20
```

SHR (shift right)

- 對目標運算元執行 向右 邏輯移位 ，並且將最高位元填入 0。
- SHR 可以執行 2 的次方 數的高速除法

```
EX： 20 / 2 ^ 2

mov dl, 5
shr dl, 2        ; dl = 5
```

SAL 和 SAR



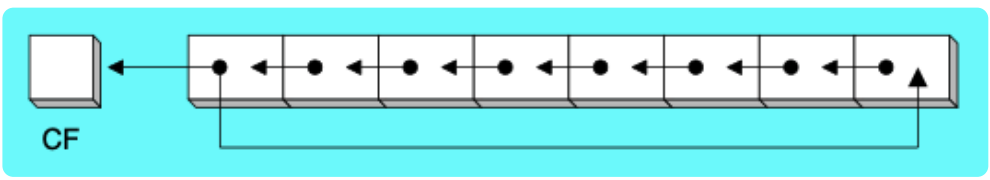
- SAL 和 SHL 功能一樣
- SAR 可以對其目標運算元，執行 向右 算數位移 的運算
- 算術位移保持數字的符號

```
mov dl, -80
sar dl, 1       ; DL = -40
sar dl, 2       ; DL = -10
```

ROL 和 ROR

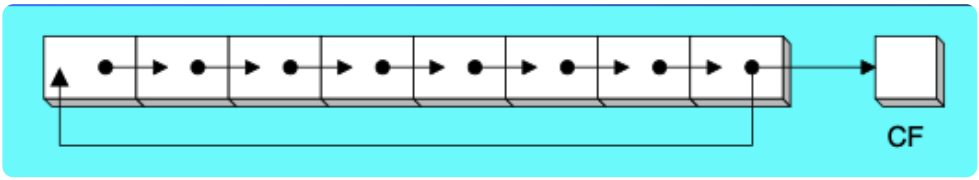
- ROL 將每個位元向左移位
- 最高位元會複製到 CF 以及 最低位元位置
- 沒有位元遺失

```
mov al, 11110000b
rol al, 1        ; AL = 11100001b

mov dl, 3Fh
rol dl, 4        ; DL = F3h
```



- ROR 將每個位元向右移位
- 最低位元會複製到 CF 以及 最高位元位置
- 沒有位元遺失

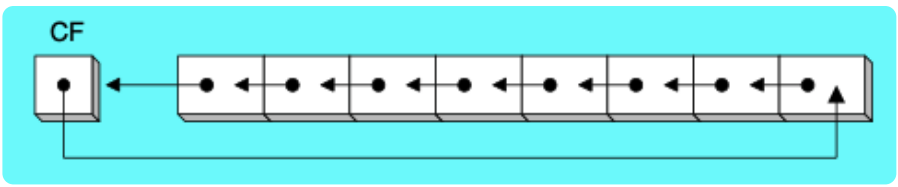
```
1  mov al, 11110000b
2  ror al, 1        ; AL = 01111000b
3
4  mov dl, 3Fh
5  ror dl, 4        ; DL = F3h
```



RCL 和 RCR

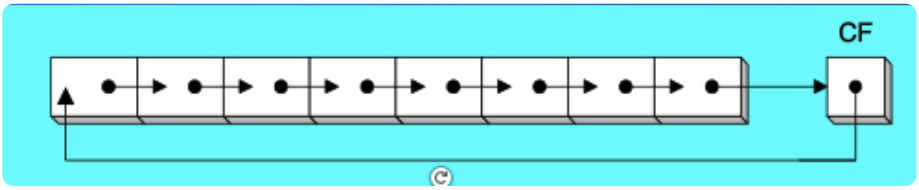
- RCL 將每個位元向左移
- CF 會複製到最小有效位元
- 最大位元複製到 CF 中

```
1  clc                  ; CF = 0
2  mov bl, 88h           ; CF,BL = 0 10001000b
3  rcl bl, 1             ; CF,BL = 1 00010000b
4  rcl bl, 1             ; CF,BL = 0 00100001b
```



- RCR 將每個位元向右移
- CF 會複製到最大有效位元
- 最小位元複製到 CF 中

```
1  stc            ; CF = 1
2  mov ah, 10h     ; CF,AH = 00010000 1
3  rcr ah, 1       ; CF,AH = 10001000 0
```



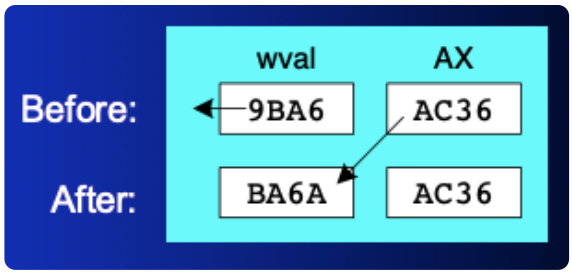
SHLD 和 SHRD

- SHLD 將運算元向左移位指定的位元數
- 移位過程中所產生的未定位元，填入來源運算元 最大 有效位元的位元值
- 來源運算元的值並不會受影響

```
SHLD destination, source, count
```

- EX：
- 將 wval 左移 4 位元
- 將 AX 的最高四個位元，插入 wval 的最低 四個位元

```
.data
    wval WORD 9BA6h
.code
    mov  ax, 0AC36h
    shld wval, ax, 4
```



- SHRD 將運算元向右移位指定的位元數
- 移位過程中所產生的未定位元，填入來源運算元 最小 有效位元的位元值
- 來源運算元的值並不會受影響

```
SHRD destination, source, count
```

- EX：
- 將 wval 右移 4 位元
- 將 AX 的最小四個位元，插入 wval 的最高四個位元

```
mov  ax, 234Bh
mov  dx, 7654h
shrd ax, dx, 4
```

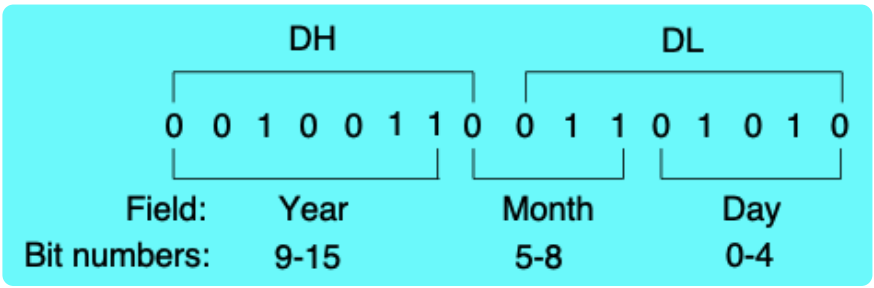
一些例子

算算數

- AX 乘以 26

```
mov ax, 2           ; test value

mov dx, ax
shl dx, 4           ; AX * 16
push dx             ; save for later
mov dx, ax
shl dx, 3           ; AX * 8
shl ax, 1           ; AX * 2
add ax, dx          ; AX * 10
pop dx              ; recall AX * 16
add ax, dx          ; AX * 26
```

取日期



- 把月份取出來

```
mov ax, dx          ; make a copy of DX
shr ax, 5           ; shift right 5 bits
and al, 00001111b   ; clear bits 4-7
mov month, al       ; save in month variable
```

CH8 Advanced Procedures 程序

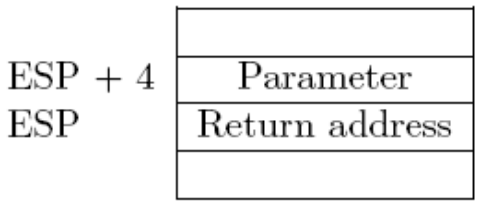
函式呼叫與 stack

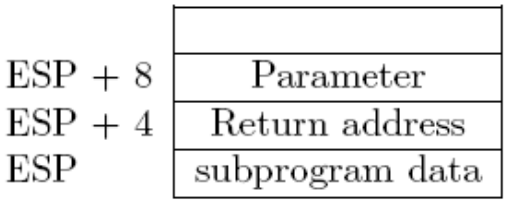
- 使用 stack 來存取

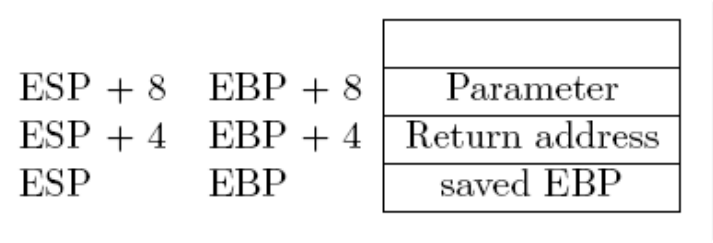
- return 的位址
- 傳遞的參數
- 函式內的區域變數

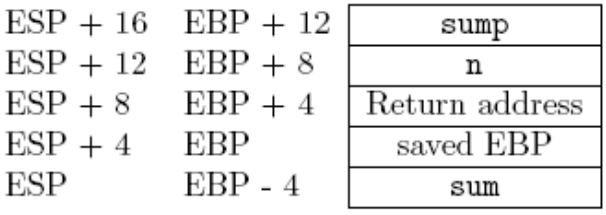
- 什麼是 EBP
  - EBP -> base pointer
  - EBP 中存著在 stack 中的 return 位址
  - EBP 的指標位址並不會改變

- 為什麼要 EBP
  - 因為 ESP 的值會隨著 stack 的增加而變動
  - 指到記憶體的位址會有所不同
  - 這樣去存取變數很麻煩
  - 所以設計一個 EBP 使它等於一開始的 ESP









- stack 注意事項
  - PUSH、POP 兩個指令

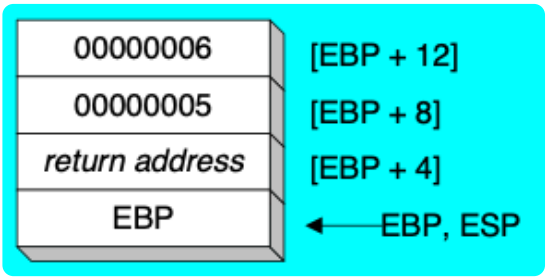
- ESP (stack pointer) EBP register 的搭配
- PUSH 會將 ESP 中的位址 - 4

- 如何實作一個 stack

1. 呼叫一個函式
2. 把參數 push 到 stack 裡
3. 把 return 的位址 push 到 stack 裡
4. 在函式裡把 EBP push 到 stack裡
5. 把 EBP 等於 ESP
6. 如果有需要使用到區域變數，要先將 ESP 往下移出一些空間出來

```
push ebp                ; 將 EBP 的內容先存到 stack 中
mov ebp, esp            ; EBP = ESP

    ; 函式的實作內容置於此處

pop ebp                 ; 還原 EBP 的值
ret
```

```
.data
    sum DWORD ?
.code
    push 6                  ; 第二個參數
    push 5                  ; 第一個參數
    call AddTwo             ; EAX = sum
    mov sum, eax            ; save the sum

AddTwo PROC
    push ebp                ; stack 起手式
    mov ebp, esp
    mov eax, [ebp + 12]     ; 存取第二個參數 (6) 使用位址來存取參數
    add eax, [ebp + 8]      ; 存取第一個參數 (5) 使用位址來存取參數
    pop ebp
    ret 8                   ; stack 起手式結束
AddTwo ENDP                 ; EAX contains the sum
```



區域變數

如何定義一個區域變數

- 使用 LOCAL 自訂變數名稱
- 使用 stack 中的 EBP 直接指向記憶體
- EBP 減掉所有區域變數的大小

```
push ebp
mov ebp, esp
sub esp, LOCAL_BYTES        ; 在此處根據需求保留給 local 變數的空間

       ; subprogram 的實作內容置於此處

mov esp, ebp
pop ebp
ret
```

LOCAL

- 定義一個區域變數
- 直接接在 PROC 後面
- 需要給定大小

```
MySub PROC LOCAL var1:BYTE, var2:WORD, var3:SDWORD
```

使用 EBP

ESP + 16	EBP + 12	sump
ESP + 12	EBP + 8	n
ESP + 8	EBP + 4	Return address
ESP + 4	EBP	saved EBP
ESP	EBP - 4	sum

- 第一個 參數 -> EBP + 8
- 第二個 參數 -> EBP + 12
- 第一個 區域變數 -> EBP - 4
- 第二個 區域變數 -> EBP - 8

  .

  .

  .
- 函式的例子

```
; (C 語言程式碼)

; void calc_sum(int n, int *sump) {
;     int i, sum = 0;
;
;     for(i = 1 ; i <= n ; i++)
;         sum += i;
;
;     *sump = sum;
; }
```

```
calc_sum:
    push ebp
    mov ebp, esp
    sub esp, 4      ;保留儲存 local 變數的 memory address 的空間

    mov dword   [ebp - 4], 0    ; sum = 0;
    mov ebx, 1                  ; 假設 EBX = i

for_loop:
    cmp ebx, [ebp + 8]          ; i <= n
    jnle end_for                ; jump if not(less & equal)

    add [ebp - 4], ebx          ; sum += i
    inc ebx
    jmp for_loop

end_for:
    mov ebx, [ebp + 12]         ; ebx = sump
    mov eax, [ebp - 4]          ; eax = sum
    mov [ebx], eax              ; *sump = sum

    mov esp, ebp
    pop ebp
    ret
```

CH9 Strings and Arrays

- 處理 array 的指令

- 使用 ESI、EDI 做為索引之用
  - DF 設定好後，ESI EDI 會自動加減

- Direction flag (DF) 來判斷索引為遞增或遞減
  - CLD：將 DF 設定為 0，為 遞增
  - STD：將 DF 設定為 1，為 遞減

- Repeat Prefix 可以用來重複陣列操作指令，方向由 DF 控制
  - 用法：REP [指令]
  - 重複條件：ECX > 0
  - 比較用延伸用法(CMP)：REPZ、REPNZ

LOASx 與 STOSx

LOASx (Load)

- 把 ESI 指向的值存入 AL/AX/EAX
- EX：把一個陣列內的值印出來

```
.data
    array BYTE 1, 2, 3, 4, 5, 6, 7, 8, 9
.code
        mov esi, OFFSET array
        mov ecx, LENGTHOF array
        cld

L1:     lodsb                   ; 從 AL 中讀取值
        or al, 30h              ; convert to ASCII
        call WriteChar          ; display it
        loop L1
```

STOSx (Store)

- 把在 AL/AX/EAX 中存入的值，放進 EDI 所指向的位置
- EX：把一個陣列放滿 0FFh

```
.data
    Count = 100
    string1 BYTE Count DUP(?)
.code
    mov al, 0FFh               ; value to be stored
    mov edi, OFFSET string1    ; 用 EDI 指向存入目標
    mov ecx, Count             ; character count
    cld                        ; 把 DL 設成往上加
    rep stosb                  ; fill with contents of AL
```

MOVSx

- 將記憶體中 ESI 指向的位置的值複製到 EDI 指向的位置
- EX：把一個陣列的值複製到另陣列中

```
.data
    source DWORD 0FFFFFFFFh
    target DWORD ?

.code
    mov esi, OFFSET source
    mov edi, OFFSET target
    movsd
```

CMPSx

- 將記憶體中比較 ESI 位置與 EDI 位置的值
- EX：如果 source > target，跳至 label L1，不然跳至 label L2

```
.data
    source DWORD 1234h
    target DWORD 5678h

.code
    mov esi, OFFSET source
    mov edi, OFFSET target
    cmpsd                   ; 比較 ESI EDI
    ja L1                   ; jump if source > target
    jmp L2                  ; jump if source <= target
```

SCASx

- 將記憶體中比較 AL/AX/EAX 中 存入的值 與 EDI 位置的值
- EX：在一個陣列中找 "F" 這個字元

```
.data
    alpha BYTE "ABCDEFGH", 0

.code
    mov edi, OFFSET alpha
    mov al, 'F'                ; 把 'F' 放入 AL 中代表我們要找它
    mov ecx, LENGTHOF alpha
    cld
    repne scasb                ; 重複直到不一樣
    jnz quit
    dec edi                    ; EDI points to 'F'
```

二維陣列

- 之前有提到過二維陣列可以用下面這個方法定義

```
table  BYTE  10h,  20h,  30h,  40h,  50h
       BYTE  60h,  70h,  80h,  90h, 0A0h
       BYTE 0B0h, 0C0h, 0D0h, 0E0h, 0F0h
NumCols = 5
```

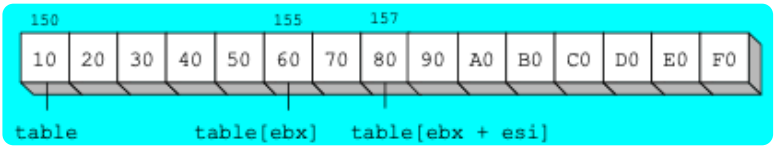
- 現在也可以用這個方法定義

```
table  BYTE  10h, 20h, 30h, 40h, 50h, 60h, 70h, 80h, 90h, 0A0h, 0B0h, 0C0h, 0D0h,
NumCols = 5
```

- 只不過要用下面的方法來取值
  - $data(n, m) = n * M + m$
  - n = 列
  - m = 行
  - M = 原定義的陣列中行數為多少

- EX：在上面 5x3 的陣列中取 table[1][2]

```
RowNumber = 1
ColumnNumber = 2

mov ebx, NumCols * RowNumber                ; NumCols = 5
mov esi, ColumnNumber
mov al, table[ebx + esi]
```



CH10 Structure and Macro

Structure 結構

- 給予邏輯相關變數的模板或模式。

- 巢狀結構

- SIZE 與 LENGTH 皆為其所有變數的總和
  - 宣告成結構陣列要特別小心，每次換到下一個陣列值的地址差距

- 定義方式

```
name STRUCT

        STRUCT 裡面放的各式變數

name ENDS
```

- 宣告、使用方式

```
.data
name StructName < 變數1, 變數2.....>

.code
MOV 對應暫存器, name.變數1
```

- EX：

```
; 定義 STRUCT

Employee STRUCT

    IdNum BYTE "000000000"
    LastName BYTE 30 DUP(0)
    Years WORD 0
    SalaryHistory DWORD 0,0,0,0

Employee ENDS
---------------------------------
; 取 STRUCT 中的值

mov dx, worker.Years
mov esi, OFFSET worker
mov ax, (Employee PTR [esi]).Years

mov ax, [esi].Years             ; 無效，指令不明確
```

Union 聯集

- 與結構相似，但是內部所有變數共享一個記憶體位置
- SIZE 與 LENGTH 由變數中最大者決定
- 宣告方式和使用方式與Struture類似

```
name UNION

        UNION 裡面放的各式變數

name ENDS
```

MACRO 巨集指令

- MACRO 類似於指令函數，定義完後可以重複使用，組譯時會將所有MACRO全部取代成定義好的指令集
- 定義方式

```
name MACRO [ 參數1, 參數2...]    //參數可選

        MACRO 裡面放的指令集

name ENDM
```

Conditional-Assembly Directive 條件式組譯指引

- 組譯時就會運行這些指引，可以完成自我報錯的功能

IFB

- 若參數為空(呼叫時沒給)，進入IFB'
- EXITM 後面可以加 <-1>(ture) 或 <0>(false)

```
IFB <參數>              //若參數為空 = true
  EXITM             // 退出巨集
ENDIF
```

IF, ELSE, ENDIF

```
IF boolean-expression
        statements
[ELSE                    //可選
        statements]
ENDIF
```

- Boolean Expression
  - LT、GT、EQ、NE、LE、GE

IFIDN, IFIDNI

- 比較兩個給予的參數名稱，若相同 = true
- IFIDN 有大小寫分別、IFIDNI 沒有

```
IFIDNI <參數1>, <參數二>    //此處參數可為任何內容、主要是比名稱
    statements
ENDIF
```

Special Operators 特殊運算子

- Substitution (&) 用於代替參數成被MACRO定義的對象

```
ShowRegister MACRO regName    //宣告MACRO，將regName改成參數

.data
tempStr BYTE " &regName=",0    //本宣告內容會變成 " EDX="

.code
ShowRegister EDX                  //呼叫巨集，使此參數改名為EDX
```

- Expansion (%) 用於將參數從常數改成文本提供MACRO輸入
  - mGotoXY %(5*10), %(3+4) → mGotoXY 50, 7

- Literal-Text (<>) 用於將多個文本參數組合成單一文本

- Literal-Character (!) 用於將上述特殊運算子改成正常文本

Repeat Directive 迴圈指引

WHILE

```
WHILE constExpression        //類似於boolean-expression
        statements
ENDM                        //迴圈直到constExpression = false
```

REPEAT

```
REPEAT constExpression        //參數為重複的次數
        statements
ENDM                        //迴圈重複的次數
```

FOR

- 迴圈會重複將引數代入參數後執行，然後重複至所有引數都被代入過

```
FOR 參數,< 引數1, 引數2, 引數3,...>
        statements
ENDM
```

FORC

- 類似於FOR，只不過帶入的改為字串中的每一個字元

```
FOR 參數,<字串>
        statements
ENDM
```

Reference

- 小信豬的原始部落
  - http://godleon.blogspot.com/2008/01/machine-language-cpu-machine-language.html (http://godleon.blogspot.com/2008/01/machine-language-cpu-machine-language.html)
- 組語維基教科書
  - https://zh.m.wikibooks.org/wiki/X86組合語言 (https://zh.m.wikibooks.org/wiki/X86%E7%B5%84%E5%90%88%E8%AA%9E%E8%A8%80)
- Xuite 筆記
  - https://blog.xuite.net/asd.wang/alog/269336-[Masm]+Assembly+筆記+-+Ch3+組合語言基礎 (https://blog.xuite.net/asd.wang/alog/269336-%5BMasm%5D+Assembly+%E7%AD%86%E8%A8%98+-+Ch3+%E7%B5%84%E5%90%88%E8%AA%9E%E8%A8%80%E5%9F%BA%E7%A4%8E)