

## 4장 데이터전송, 주소지정, 산술연산

## 이 장의 내용

- 데이터 전송 명령어
- 덧셈과 뺄셈
- 데이터 관련 연산자와 디렉티브
- 간접 주소지정
- JMP와 LOOP 명령어

### 4.1 데이터 전송 명령어

#### ■ 피연산자(operand)의 유형

- **즉시값(Immediate)** - 상수 정수(식) (8, 16, 32 bits)
  - 값이 instruction에 포함됨
- **Register** - CPU 내의 레지스터 이름
  - 레지스터 이름이 instruction에 부호화되어 포함됨
- **Memory** - 메모리 위치에 대한 참조 정보
  - 주소 또는 주소를 저장한 레지스터 이름이 instruction에 포함됨 (직접 메모리 피연산자 또는 간접 메모리 피연산자)

#### ■ 예

- MOV AX, 100 ; AX ← 100                      즉시값(상수)
- MOV AX, CX ; AX ← CX                            레지스터
- MOV AX, [100] ; AX ← M(DS:100), 16-bit      메모리 - 직접주소
- MOV AX, [SI] ; AX ← M(DS:SI), 16-bit        메모리 - 간접주소

### Instruction Operand 표기 (Intel)

피연산자	설명
reg8	8비트 범용 레지스터: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16비트 범용 레지스터: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32비트 범용 레지스터: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	임의의 범용 레지스터
sreg	16비트 세그먼트 레지스터: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32비트 즉시값(숫자)
imm8	8비트 바이트 즉시값
imm16	16비트 워드 즉시값
imm32	32비트 더블워드 즉시값
reg/mem8	8비트 피연산자로서 8비트 레지스터 또는 메모리 바이트일 수 있음
reg/mem16	16비트 피연산자로서 16비트 레지스터 또는 메모리 워드일 수 있음
reg/mem32	32비트 피연산자로서 32비트 레지스터 또는 메모리 더블워드일 수 있음
mem	8, 16, 32비트 메모리 피연산자

## 직접 메모리 피연산자

### ■ 직접 메모리 피연산자

- instruction에 포함된 메모리 주소에 있는 메모리 피연산자

```
MOV AL, [400]
```

직접 주소지정 방식

### ■ 어셈블리 언어에서는 메모리 주소 대신에 data label을 사용함

- 어셈블러가 data label을 offset 주소로 변환해줌

```
MOV AL, var1   또는  
MOV AL, [var1]
```

→ 이 표기를 선호함

- 예

```
.data  
var1 BYTE 10h           ; 변수(데이터)  
.code  
mov al,var1             ; AL = 10h
```

## MOV 명령어

### ■ MOV dst, src

- 동작:  $dst \leftarrow src$

### ■ operand 사용 규칙

- 두 피연산자는 같은 크기이어야 함
- 두 피연산자가 모두 메모리일 수는 없음
- CS, EIP(또는 IP)는 dst일 수 없음
- immediate값은 segment register로 이동할 수 없음

### ■ 잘못된 사용 예

```
mov ax, bl           (x)   ; 크기가 다름  
mov var1, var2       (x)   ; var1, var2는 data label(변수)  
mov cs, ax           (x)   ; CS가 dst임  
mov ds, 400h         (x)   ; 숫자를 DS로 이동
```

## 사용 예

```
.data  
count BYTE 100  
wVal WORD 2  
.code  
mov bl,count           ; BL ← count(100)  
mov ax,wVal           ; AX ← wVal(2), 16-bit  
mov count,al          ; count ← AL(2), 8-bit  
  
mov al,wVal           ; error(size mismatch)  
mov ax,count          ; error  
mov eax,count         ; error
```

## 잘못된 예

### ■ 잘못된 이유는?

```
.data  
bVal BYTE 100  
bVal2 BYTE ?  
wVal WORD 2  
dVal DWORD 5  
.code  
mov ds,45             immediate move to DS not permitted  
mov esi,wVal          size mismatch  
mov eip,dVal          EIP cannot be the destination  
mov 25,bVal           immediate value cannot be destination  
mov bVal2,bVal        memory-to-memory move not permitted
```

## 여러 가지 MOV 방법

### 메모리에서 메모리로의 전송

- var2 ← var1

```
mov eax, var1 ; AX ← var1
mov var2, eax ; var2 ← AX
```

레지스터를 경유함

### 작은 operand를 큰 operand로 복사

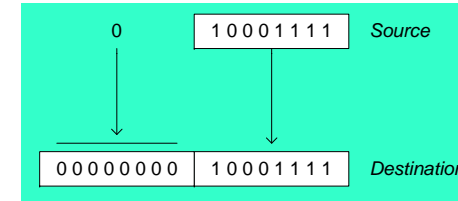
- 작은 operand를 큰 크기로 확장한 후에 mov 명령어 수행
- unsigned number는 zero extension(상위 부분을 0으로 채움) 사용
- signed number는 sign extension(상위 부분을 부호로 채움) 사용
- (예)

4비트	8비트	
	zero확장	sign확장
0101 (5)	0000_0101 (5)	0000_0101 (5)
1011 (11 또는 -5)	0000_1011 (11)	1111_1011 (-5)

## MOVZX 명령어 – zero 확장

### MOVZX reg, r/m

- 동작: reg ← zero-extension(r/m)
- reg는 r/m보다 크기가 큼
  - reg32 ← r/m8 또는 r/m16
  - reg16 ← r/m8

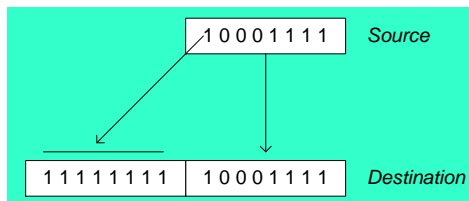


```
mov bl,10001111b
movzx ax,bl ; zero-extension
```

## MOVSX 명령어 – sign 확장

### MOVSX reg, r/m

- 동작: reg ← sign-extension(r/m)
- reg는 r/m보다 크기가 큼



```
mov bl,10001111b
movsx ax,bl ; sign extension
```

## XCHG 명령어 - exchange

### XCHG dst, src

- 동작: dst와 src의 내용을 서로 교환함
- 메모리 간의 교환을 할 수 없음

```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
xchg ax,bx ; exchange 16-bit regs
xchg ah,al ; exchange 8-bit regs
xchg var1,bx ; exchange mem, reg
xchg eax,ebx ; exchange 32-bit regs

xchg var1,var2 ; error: two memory operands
mov ax,var1 ; 메모리 간의 내용 교환은
xchg ax,var2 ; 임시 레지스터를 사용해야 함
mov var1,ax
```

## 직접 오프셋(Direct-Offset) 피연산자

### 직접 오프셋 피연산자(Direct offset operand)

- data\_label + constant 형태로 표현되는 메모리 피연산자
- 어셈블러가 offset 주소로 변환함 (직접주소지정방식과 같음)
- 용도: array 원소 접근

```
.data
arrayB BYTE 10h,20h,30h,40h
.code
mov al,arrayB+1      ; AL = 20h
mov al,[arrayB+1]   ; alternative notation
```

- arrayB의 주소가 100h라고 하면
  - mov al, arrayB → mov al, [100h] (cf) a[0]
  - mov al, arrayB+1 → mov al, [101h] (cf) a[1]



### word array

- 다음 원소의 offset은 2씩 증가

### doubleword array

- 다음 원소의 offset은 4씩 증가

```
.data
arrayW WORD 1000h,2000h,3000h
arrayD DWORD 1,2,3,4
.code
mov ax,arrayW+2      ; AX = 2000h
mov eax,arrayD+4     ; EAX = 00000002h
```

arrayW[1]  
arrayD[1]

### 잘못된 예: 주소가 배열 범위를 벗어남

```
mov ax,arrayW-2      ; ??
mov eax,arrayD+16    ; ??
```

arrayW[-1]  
arrayD[4]

## 연습

### 자료의 배치를 바꾸기

- 다음 자료를 다음 순서로 바꾸시오: 3, 1, 2
  - .data
  - arrayD DWORD 1, 2, 3

- Step1: arrayD와 arrayD+4의 값을 교환 → 2, 1, 3

```
mov eax,arrayD
xchg eax,arrayD+4
```

- Step 2: arrayD와 arrayD+8의 값을 교환 → 3, 1, 2

```
xchg eax,arrayD+8
mov arrayD,eax
```

## 4.2 덧셈과 뺄셈

### 덧셈과 뺄셈 명령어

형식	동작	설명
INC dst	dst ← dst + 1	increment
DEC dst	dst ← dst - 1	decrement
ADD dst, src	dst ← dst + src	add
SUB dst, src	dst ← dst - src	subtract
NEG dst	dst ← - dst	negate(2의 보수)

- INC, DEC, NEG의 operand는 r/m
- ADD, SUB의 operand는 MOV의 operand와 같은 rule을 적용

## INC과 DEC 예

```
.data
myWord WORD 1000h
myDword DWORD 10000000h
.code
inc myWord          ; 1001h
dec myWord          ; 1000h
inc myDword         ; 10000001h

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

## ADD와 SUB 예

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code
; ---EAX---
mov eax,var1        ; 00010000h
add eax,var2        ; 00030000h
add ax,0FFFFh       ; 0003FFFFh
add eax,1            ; 00040000h
sub ax,1             ; 0004FFFFh
```

## NEG 예

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

## 수식 계산

### ■ 수식의 계산

(ex)  $R = -X + (Y - Z)$

■ 과정: (1)  $-X$  (2)  $Y - Z$  (3)  $(-X) + (Y - Z)$

```
.data
R DWORD ?
X DWORD 26
Y DWORD 30
Z DWORD 40
.code
mov eax,X           ; EAX = X
neg eax             ; EAX = -26 (-X)
mov ebx,Y           ; EBX = Y
sub ebx,Z           ; EBX = -10 (Y-Z)
add eax,ebx         ; EAX = -36 (-X)+(Y-Z)
mov R,eax           ; -36
```

## 연산과 FLAG

### FLAG 레지스터

- 산술/논리 연산의 결과에 따라서 값이 정해짐
- MOV 명령어의 영향을 받지 않음

### 기본적인 FLAG bits

- ZF(Zero Flag) – 결과가 0이면 1
- SF(Sign Flag) – 결과가 음수(MSB=1)이면 1
- CF(Carry Flag) – unsigned value가 표현범위 벗어나면 1  
(unsigned overflow)
- OF(Overflow Flag) – signed value가 표현범위 벗어나면 1  
(signed overflow)

## Zero Flag(ZF)와 Sign Flag(SF)

### ZF

```
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
```

### SF

```
mov al,0
sub al,1           ; AL = 11111111b(-1), SF = 1
add al,2           ; AL = 0000001b, SF = 0
```

- SF는 MSB(부호 bit)값과 같음

## 부호있는 정수와 부호없는 정수

### 부호있는 정수와 부호없는 정수

- signed integer와 unsigned integer 모두 2진수 pattern으로 표현됨
- CPU는 signed와 unsigned integer를 구별할 수 없음
- signed와 unsigned integer의 구분은 프로그램에서 사용하는 instruction에 의해서 이루어짐

(예) 2진수 패턴 1110000b → unsigned integer = 224  
signed integer = -32

### Signed와 Unsigned integer에 대한 산술 연산

- 덧셈과 뺄셈 연산: 구분 없이 같은 명령어를 사용함
- 곱셈과 나눗셈 연산: signed와 unsigned integer에 대해서 구분을 하여 별개의 명령어를 사용

### Singed와 Unsigned 연산과 Flag

- unsigned 연산: Zero, Carry, 보조 Carry 플래그를 사용
- signed 연산: Zero, Sign, Overflow 플래그를 사용

## Carry Flag (CF)와 Overflow Flag (OF)

### CF – unsigned overflow, OF – signed overflow

```
mov al,0FFh       ; 255(unsigned) 또는 -1(signed)
add al,1           ; AL = 00, CF = 1, OF = 0
```

- 255 → 0 (CF=1), -1 → 0 (OF=0)

```
mov al,0           ; 0
sub al,1           ; AL = FF, CF = 1, OF = 0
                  ; (255 또는 -1)
```

- 0 → 255 (CF=1), 0 → 1 (OF=0)

```
mov al,7Fh        ; 127
add al,1           ; AL = 80h, CF = 0, OF = 1,
                  ; (128 또는 -128)
```

- 127 → 128 (CF=0), 127 → -128 (OF=1)

## INC, DEC, NEG 명령어와 CF

- INC와 DEC 명령어는 CF에는 영향을 주지 않음

```
mov al,0FFh      ; 255(unsigned) 또는 -1(signed)
add al,1        ; AL = 00, CF = 1, OF = 0
mov al,0FFh
inc al          ; AL = 00, CF = 0, OF = 0
```

- NEG 명령어는 0이 아닌 피연산자에 대한 연산 시 항상 CF=1이 됨

```
.data
valB BYTE 0,1
valC SBYTE -128      ; 80h
.code
neg valB             ; CF = 0, OF = 0 (00h)
neg valB+1          ; CF = 1, OF = 0 (FFh)
neg valC             ; CF = 1, OF = 1 (80h)
```

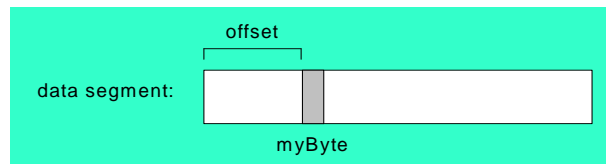
## 4.3 데이터 관련 연산자와 디렉티브

- OFFSET 연산자
- PTR 연산자
- TYPE 연산자
- LENGTHOF 연산자
- SIZEOF 연산자
- LABEL 디렉티브

## OFFSET 연산자

- OFFSET label

- label의 offset 주소 반환
  - Protected mode: 32 bits
  - Real mode: 16 bits



(cf) protected mode 프로그램은 0번지부터 시작하는 단일 segment를 사용한다. (flat memory model)

- 예제

```
.data                                ; start at 00404000
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?

.code
mov esi,OFFSET bVal                 ; ESI = 00404000
mov esi,OFFSET wVal                 ; ESI = 00404001
mov esi,OFFSET dVal                 ; ESI = 00404003
mov esi,OFFSET dVal2                ; ESI = 00404007
```

## C/C++와의 관계

- OFFSET 연산자는 변수의 주소(pointer)를 얻을 때 사용함

```
; C++ version:
char array[1000], *p;
int a, *q;

p = array;
q = &a;
```



```
; assembly language
.data
array BYTE 1000 DUP(?)
a BYTE ?
.code
mov esi,OFFSET array ; ESI is p
mov edi,OFFSET a ; EDI is q
```

## PTR 연산자

- type PTR label

- label이 가리키는 operand의 크기를 type 크기로 재설정

```
.data
myDouble DWORD 12345678h ; little endian
; 78 56 34 12 순서로 저장
.code
mov ax, myDouble ; error - size mismatch
mov ax, WORD PTR myDouble ; loads 5678h
mov WORD PTR myDouble,4321h ; saves 4321h
mov bl, BYTE PTR myDouble ; loads 78h
mov bl, BYTE PTR [myDouble+1] ; loads 56h
```

```
.data
myBytes BYTE 12h,34h,56h,78h
.code
mov ax, WORD PTR myBytes ; AX = 3412h
mov eax, DWORD PTR myBytes ; EAX = 78563412h
```

## TYPE 연산자

- TYPE label

- label이 가리키는 data의 크기를 반환 (단위 byte)

```
.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
```

(cf) C언어: sizeof(a)

## LENGTHOF 연산자

- LENGTHOF label

- label과 같은 줄에 선언된 원소의 개수를 반환
- comma로 구분된 경우에는 다음 줄도 포함 (예: mword)

```
.data
byte1 BYTE 10,20,30 ; 3
array1 WORD 30 DUP(?),0,0 ; 32
array2 WORD 5 DUP(3 DUP(?)) ; 15
array3 DWORD 1,2,3,4 ; 4
digitStr BYTE "12345678",0 ; 9
mword WORD 10,20,30, ; 6
40,50,60
mword2 WORD 10,20,30 ; 3
WORD 40,50,60
.code
mov ecx,LENGTHOF array1 ; 32
```



## SIZEOF 연산자

### ■ SIZEOF label

- LENGTHOF와 TYPE의 곱을 반환
- 원소들이 차지하는 크기 (단위 byte)

```

.data                                     SIZEOF
byte1  BYTE 10,20,30                    ; 3
array1 WORD 30 DUP(?),0,0               ; 64=32*2
array2 WORD 5 DUP(3 DUP(?))             ; 30=15*2
array3 DWORD 1,2,3,4                    ; 16=4*4
digitStr BYTE "12345678",0              ; 9

.code
mov ecx,SIZEOF array1                    ; 64
    
```

(cf) C언어: sizeof(array)

## LABEL 디렉티브

### ■ label LABEL type

- 기존 label의 위치에 다른 type의 label 이름을 부여함
- PTR operator를 사용할 필요가 없도록 함
- 기억장소의 추가적인 할당은 없음

```

.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
    
```

- dwList, wordList, intList는 모두 같은 주소의 label이지만 참조하는 자료형의 크기가 다름

## ALIGN 디렉티브

### ■ ALIGN bound

- bound: 1, 2, 4, 또는 16
- 변수를 bound 크기의 배수 위치에 배치

```

.data                                     ; start at 00404000
bVal  BYTE ?                             ; 00404000
ALIGN 2
wVal  WORD ?                              ; 00404002
wVal2 WORD ?                              ; 00404004
ALIGN 4
dVal  DWORD ?                             ; 00404008
dVal2 DWORD ?                             ; 0040400C
    
```

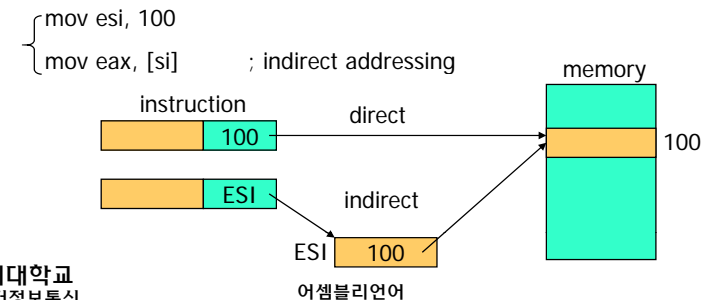
### ■ ALIGN을 사용하는 이유

- 16비트 데이터와 32비트 데이터는 각각 2의 배수와 4의 배수의 주소에 배치되어야 데이터를 빠르게 접근할 수 있음
- real mode 세그먼트는 16의 배수 주소에서 시작함

## 4.4 간접 주소지정

### ■ 간접 주소지정

- 명령어에 operand의 주소를 직접 포함하는 것 대신에 operand의 주소를 저장한 위치를 포함
  - register indirect addressing
  - memory indirect addressing – IA32에서 지원하지 않음
- C/C++의 pointer와 관련됨
  - mov eax, [100] ; direct addressing



## 간접 피연산자

- 피연산자의 유효주소(EA: effective address) = reg
- 간접 피연산자
  - operand 주소를 갖고 있는 register
  - 사용하기 전에 유효한 주소로 초기화해야 함
- 간접 피연산자로 사용하는 레지스터
  - 32-bit mode: 범용 레지스터(EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP)
  - 16-bit mode: SI, DI, BX, BP (cf) DS:SI/DI/DX, SS:BP

```
.data
val1 BYTE 10h,20h,30h
.code
mov esi,OFFSET val1
mov al,[esi]           ; AL = M[DS:SI] (10h)
inc esi
mov al,[esi]           ; AL = 20h
inc esi
mov al,[esi]           ; AL = 30h
```

## 간접 피연산자와 PTR 연산자

- operand의 크기 지정
  - 간접 피연산자가 지시하는 피연산자 크기가 명확하지 않을 수 있음
  - 피연산자의 크기를 명확하게 하기 위해서 PTR operator를 사용

```
.data
myCount WORD 0
.code
mov esi,OFFSET myCount
inc [esi]               ; error: ambiguous size
inc WORD PTR [esi]     ; ok

add ax,[esi]            ; ok
add [esi],20            ; error: ambiguous size
add BYTE PTR [esi],20  ; ok
```

## 간접 피연산자와 배열

- 배열 원소를 다루는 데에 indirect operand가 유용함
  - 특히 순차적으로 다루는 데 편리함
- 예: 배열의 합

```
.data
arrayW WORD 1000h,2000h,3000h
.code
mov esi,OFFSET arrayW
mov ax,[esi]
add esi,2                ; 2대신에 TYPE arrayW 사용가능
add ax,[esi]
add esi,2
add ax,[esi]             ; AX = sum of the array
```

- 다음 원소를 사용할 때에 주소를 원소의 크기만큼 증가시킴

## 인덱스 피연산자

- 피연산자 유효주소(EA)= reg + const
  - 형식: [reg + const] 또는 const[reg]
  - [const + reg]

```
.data
arrayW WORD 1000h,2000h,3000h
.code
mov esi,0
mov ax,[arrayW + esi]   ; AX = 1000h
mov ax,arrayW[esi]     ; alternate format
add esi,2
add ax,[arrayW + esi]
```

- 인덱스 피연산자로 사용하는 register
  - 32-bit mode: general purpose registers (cf) SS:EBP
  - 16-bit mode: SI, DI, BX, BP (cf) SS:BP

## 인덱스 배율(Index Scaling)

- 피연산자 유효주소(EA) =  $\text{const} + \text{reg} * \text{scale}$ 
  - 형식: `const[reg*scale]` ; scale: 1, 2, 4
  - scale은 TYPE operator를 사용하여 얻을 수도 있음
  - 워드, 더블워드 배열을 다루는 데 유용함

```
.data
arrayB BYTE 0,1,2,3,4,5
arrayW WORD 0,1,2,3,4,5
arrayD DWORD 0,1,2,3,4,5

.code
mov esi,4
mov al,arrayB[esi*TYPE arrayB] ; 04
mov bx,arrayW[esi*TYPE arrayW] ; 0004
mov edx,arrayD[esi*TYPE arrayD] ; 00000004
```

## 포인터(Pointer)

- 포인터 변수
  - 다른 변수의 주소를 가지고 있는 변수

```
.data
arrayW WORD 1000h,2000h,3000h
ptrW DWORD arrayW ; pointer variable
ptrW2 DWORD OFFSET arrayW ; 위와 같은 값

.code
mov esi,ptrW
mov ax,[esi] ; AX = 1000h

mov ptrW, OFFSET arrayW ; OK
mov ptrW, arrayW ; Error (data)
```

- 변수 선언에서 초기값으로 사용되는 변수 이름은 변수의 주소를 나타냄 (OFFSET varname과 같음)
- 명령어에서 사용되는 변수 이름은 변수의 값을 나타냄

## NEAR and FAR pointers

- Pointer 유형

	NEAR pointer	FAR pointer
의미	같은 세그먼트 내의 주소	다른 세그먼트에 속한 주소
표현	offset	segment:offset
16-bit mode	16-bit	32-bit (16 + 16)
32-bit mode	32-bit	48-bit (16 + 32)

- flat model을 사용하는 protected mode 프로그램은 near pointer만 사용함.

## TYPDEF 연산자

- TYPDEF 연산자
  - 사용자 정의 자료형을 만든다.
  - 포인터 자료형을 명확히 나타내는 데 유용함.

`PBYTE TYPDEF PTR BYTE` ; BYTE에 대한 포인터 자료형 정의

```
PWORD TYPDEF PTR WORD
PDWORD TYPDEF PTR DWORD

.data
arrayW WORD 1,2,3
arrayD DWORD 4,5,6
ptr1 PWORD arrayW
ptr2 PDWORD arrayD

.code
mov esi, ptr1
mov ax,[esi]
mov esi, ptr2
mov eax, [esi]
```

## 4.5 JMP와 LOOP 명령어

- 무조건 이동
  - (ex) JMP
- 조건부 이동
  - (ex) LOOP, Jcc ...
  - 조건은 플래그 레지스터와 ECX의 내용에 의하여 정해짐

## JMP 명령어

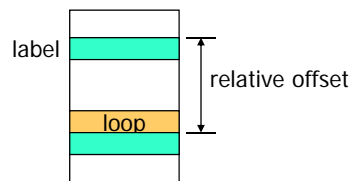
- JMP label
  - 동작: label 위치의 명령어로 jump함 (EIP ← label 주소)
- 예

```

top:                                ; code label
...
...
    jmp top
    
```

## LOOP 명령어

- LOOP label
  - 동작: 특정한 횟수를 반복 수행함 (ECX: 루프 카운터, 반복횟수 계수)
    - $ECX \leftarrow ECX - 1$
    - if (ECX != 0) jump to label
  - assembler는 label을 relative offset으로 바꾸어서 기계어로 변환
  - real mode에서는 CX가 루프 카운터로 사용됨
- LOOPD와 LOOPW 명령어
  - LOOPD는 ECX를 루프 카운터로 사용
  - LOOPW는 CX를 루프 카운터로 사용
- $relative\ offset = label\ 주소 - 다음명령어주소$



## LOOP 예

- 합 5+4+3+2+1을 계산

offset	machine code	source code
00000000	66 B8 0000	mov ax, 0
00000004	B9 00000005	mov ecx, 5
00000009	66 03 C1	L1: add ax, cx
0000000C	E2 <b>FB</b>	loop L1
0000000E		

relative offset = 9 - E(14) = -5 (FBh)

- relative offset의 범위: -128 ~ +127 (8-bit signed 정수)
  - relative offset이 지정된 범위를 벗어나면 어셈블할 때에 오류 발생

## 연습

### ■ AX의 결과?

```
mov ax,6
mov ecx,4
L1:
inc ax
loop L1
```

10

### ■ 반복 횟수 ?

```
mov ecx,0
X2:
inc ax
loop X2
```

$2^{32} = 4,294,967,296$

## 중첩된 Loop

### ■ 사용법

- 바깥 loop의 loop count값 ECX를 저장하고 ECX를 안쪽 loop의 loop count값으로 초기화해야 함.

```
.data
count DWORD ?
.code
mov ecx,100 ; set outer loop count
L1:
mov count,ecx ; save outer loop count
mov ecx,20 ; set inner loop count
L2: .
.
loop L2 ; repeat the inner loop
mov ecx,count ; restore outer loop count
loop L1 ; repeat the outer loop
```

## 정수 배열의 합

### ■ 16비트 정수 배열의 합

```
.data
intarray WORD 100h,200h,300h,400h
.code
mov edi,OFFSET intarray ; address of intarray
mov ecx,LENGTHOF intarray ; loop counter(4)
mov ax,0 ; zero the accumulator
L1:
add ax,[edi] ; add an integer
add edi,TYPE intarray ; point to next integer(+2)
loop L1 ; repeat until ECX = 0
```

### ■ 연습

- doubleword array의 합 계산

## 문자열 복사

### ■ source 문자열을 target으로 복사

```
.data
source BYTE "This is the source string",0
target BYTE SIZEOF source DUP(0)
.code
mov esi,0 ; index register
mov ecx,SIZEOF source ; loop counter
L1:
mov al,source[esi] ; get char from source
mov target[esi],al ; store it in the target
inc esi ; move to next character
loop L1 ; repeat for entire string
```

good use  
of SIZEOF

### ■ 연습

- indexed addressing대신에 indirect addressing을 사용하여 수정