Chapter 4

Data Transfers, Addressing, and Arithmetic

4.1 Data Transfer Instructions 79

4.1.1 Introduction 79
4.1.2 Operand Types 80
4.1.3 Direct Memory Operands 80
4.1.4 MOV Instruction 81
4.1.5 Zero/Sign Extension of Integers 82
4.1.6 LAHF and SAHF Instructions 84
4.1.7 XCHG Instruction 84
4.1.8 Direct-Offset Operands 84
4.1.9 Example Program (Moves) 85
4.1.10 Section Review 86

4.2 Addition and Subtraction 87

4.2.1 INC and DEC Instructions 87
4.2.2 ADD Instruction 87
4.2.3 SUB Instruction 88
4.2.4 NEG Instruction 88
4.2.5 Implementing Arithmetic Expressions 89
4.2.6 Flags Affected by Addition and Subtraction 89
4.2.7 Example Program (Addsub3) 92
4.2.8 Section Review 93

4.3 Data-Related Operators and Directives 94

4.3.1 OFFSET Operator 94
4.3.2 ALIGN Directive 95
4.3.3 PTR Operator 95
4.3.4 TYPE Operator 96
4.3.5 LENGTHOF Operator 97
4.3.6 SIZEOF Operator 97
4.3.7 LABEL Directive 97
4.3.8 Section Review 98

4.4 Indirect Addressing 99

4.4.1 Indirect Operands 99
4.4.2 Arrays 100
4.4.3 Indexed Operands 101
4.4.4 Pointers 102
4.4.5 Section Review 103

4.5 JMP and Loop Instructions 104

4.5.1 JMP Instruction 104
4.5.2 LOOP Instruction 105
4.5.3 Summing an Integer Array 106
4.5.4 Coping a String 106
4.5.5 Section Review 107

4.6 Chapter Summary 108

4.7 Programming Exercises 109

Chapter 4

Data Transfers, Addressing, and Arithmetic

4.1 Data Transfer Instructions 79

4.1.1 Introduction 79

- This chapter introduces a great many details, highlighting a fundamental **difference** between assembly language and high-level language.

4.1.2 Operand Types 80

- Three basic types of operands:
  - **Immediate:** a constant integer (8, 16, or 32 bits)
    - value is encoded within the instruction
  - **Register:** the name of a register
    - register name is converted to a number and encoded within the instruction
  - **Memory:** reference to a location in memory
    - memory address is encoded within the instruction, or a register holds the address of a memory location

TABLE 4-1 Instruction Operand Notation.

Operand	Description
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
r/m8	8-bit operand which can be an 8-bit general register or memory byte
r/m16	16-bit operand which can be a 16-bit general register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	an 8-, 16-, or 32-bit memory operand

4.1.3 Direct Memory Operands 80

- Direct Memory Operands
  - A direct memory operand is a named reference to storage in memory
  - The named reference (label) is automatically dereferenced by the assembler. The brackets imply a **deference** operation.

```
.data
var1 BYTE 10h
.code
mov al,var1    ; AL = 10h
mov al,[var1]  ; AL = 10h
```

4.1.4 MOV Instruction 81

- MOV Instruction
  - Move (copy) from source to destination
  - Syntax:

    MOV *destination, source*
    - *destination* operand's contents change
    - *source* operand's contents do not change

  - Both operands must be **the same** size
  - Both operands **cannot** be memory operands
  - CS, EIP, and IP cannot be the destination
  - No immediate to segment moves
  - Here is a list of the general variants of MOV, excluding segment registers:

```
MOV reg, reg
MOV mem, reg
MOV reg, mem
MOV mem, imm
MOV reg, imm
```

- Examples:

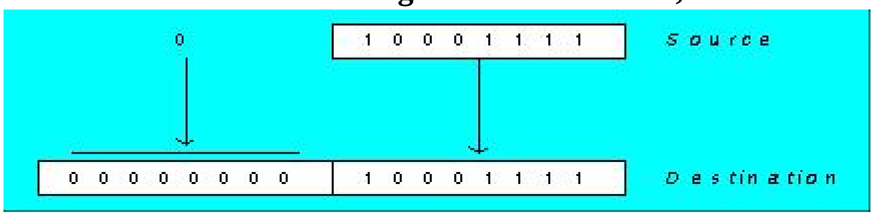
```
.data
count BYTE 100
wVal  WORD 2
.code
mov bl,count
mov ax,wVal
mov count,al
mov al,wVal     ; error, AL is 8 bits
mov ax,count    ; error, AX is 16 bits
mov eax,count   ; error, EAX is 32 bits

.data
bVal  BYTE   100
bVal2 BYTE   ?
.code
```

```
mov bVal2,bVal  ; error, memory-to-memory move not permitted
```

4.1.5 Zero/Sign Extension of Integers 82

- Zero Extension: **MOVZX** instruction
  - When copy a smaller value into a larger destination, the **MOVZX** instruction fills (extends) the upper half of the destination with **zeros**

FIGURE 4-1 Diagram of MOVZX ax, 8Fh.

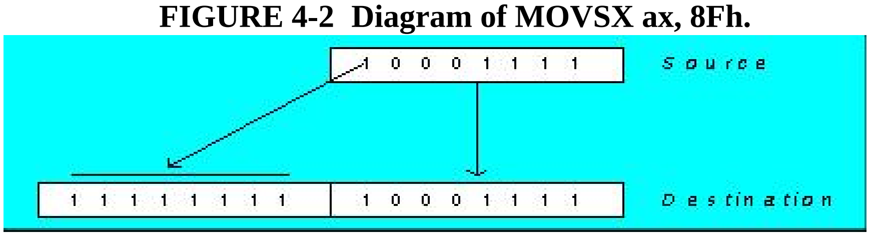


  - The destination must be a **register**

```
mov bl,10001111b
movzx ax,bl       ;  zero-extension - 0000000010001111b
```

- Sign Extension: **MOVSX** instruction
  - The ***MOVSX*** instruction fills the upper half of the destination with a copy of the source operand's sign bit

FIGURE 4-2 Diagram of MOVSX ax, 8Fh.



  - The destination must be a register

```
mov bl,10001111b
movsx ax,bl     ;  sign-extension - 1111111110001111b
```

4.1.6 LAHF and SAHF Instructions 84

- LAHF (**load** status flags into **AH**) instruction copies the low byte of the EFLAGS register into AH. The following flags are copied: Sign, Zero, Auxiliary Carry, Parity, and Carry.

```
.data
saveflags BYTE ?
.code
lahf                    ; load flags into AH
mov saveflags, ah       ; save them in a variable
```

- SAHF (**store** status flags into **AH**) instruction copies the low byte of the EFLAGS register into AH. The following flags are copied: Sign, Zero, Auxiliary Carry, Parity, and Carry.

```
mov ah, saveflags       ; load saved flags into AH
sahf                    ; copy into Flags register
```

4.1.7 XCHG Instruction 84

- XCHG Instruction **exchanges** the values of two operand
  - o At least one operand must be a register
  - o No immediate operands are permitted.

```
.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
```

4.1.8 Direct-Offset Operands 84

- Direct-Offset Operands
  - o A constant offset is added to a data label to produce an **effective address (EA)**
  - o The address is dereferenced to get the value inside its memory location

```
.data
 arrayB BYTE 10h,20h,30h,40h, 50h
.code
 mov al,arrayB+1      ; AL = 20h
 mov al,[arrayB+1]    ; alternative notation
 mov al,arrayB+2      ; AL = 30h
```

4.1.9 Example Program (Moves) 85

- The following program demonstrates most of the data transfer examples form Section 4.1:

```
TITLE Data Transfer Examples       (Moves.asm)

; Chapter 4 example. Demonstration of MOV and
; XCHG with direct and direct-offset operands.
; Last update: 06/01/2006

INCLUDE Irvine32.inc
.data
val1  WORD 1000h
val2  WORD 2000h

arrayB BYTE  10h,20h,30h,40h,50h
arrayW WORD  100h,200h,300h
arrayD DWORD 10000h,20000h

.code
main PROC

;  MOVZX
   mov    bx,0A69Bh
   movzx  eax,bx    ; EAX = 0000A69Bh
   movzx  edx,bl    ; EDX = 0000009Bh
   movzx  cx,bl     ; CX  = 009Bh

;  MOVSX
   mov   bx,0A69Bh
   movsx eax,bx     ; EAX = FFFFA69Bh
   movsx edx,bl     ; EDX = FFFFFF9Bh
   mov   bl,7Bh
   movsx cx,bl      ; CX  = 007Bh

;  Memory-to-memory exchange:
   mov  ax,val1     ; AX = 1000h
   xchg ax,val2     ; AX = 2000h, val2 = 1000h
   mov  val1,ax     ; val1 = 2000h

;  Direct-Offset Addressing (byte array):
   mov al,arrayB      ; AL = 10h
   mov al,[arrayB+1]  ; AL = 20h
   mov al,[arrayB+2]  ; AL = 30h

;  Direct-Offset Addressing (word array):
   mov ax,arrayW      ; AX = 100h
   mov ax,[arrayW+2]  ; AX = 200h

;  Direct-Offset Addressing (doubleword array):
   mov eax,arrayD                ; EAX = 10000h
   mov eax,[arrayD+4]            ; EAX = 20000h
   mov eax,[arrayD+TYPE arrayD]  ; EAX = 20000h

   exit
main ENDP
END main
```

4.2 Addition and Subtraction 87

4.2.1 INC and DEC Instructions 87

- INC and DEC Instructions
  - Add 1, subtract 1 from destination operand, operand may be register or memory
  - INC *destination*

    Logic: *destination* ← *destination* + 1

  - DEC *destination*

    Logic: *destination* ← *destination* – 1

- INC and DEC Examples

```
.data
myWord  WORD 1000h
.code
inc myWord   ; 1001h
move bx, myWord
dec myWord   ; 1000h
```

4.2.2 ADD Instruction 87

- ADD Instruction
  - *ADD destination, source*

    Logic: *destination* ← *destination* + *source*

  - Examples:

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code
mov eax,var1 ; EAX = 00010000h
add eax,var2 ; EAX = 00030000h
```

4.2.3 SUB Instruction 88

- SUB Instructions
  - *SUB destination, source*

    Logic: *destination* ← *destination – source*

  - Examples:

```
.data
var1 DWORD 30000h
var2 DWORD 10000h
.code
mov eax,var1 ; EAX = 00030000h
sub eax,var2 ; EAX = 00020000h
```

4.2.4 NEG Instruction 88

- NEG (negate) Instruction
  - Reverses the sign of an operand
  - Operand can be a register or memory operand

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

- NEG Instruction and the Flags
  - The processor implements ***NEG*** using the following internal operation:

```
SUB 0,operand
```

  - Any **nonzero** operand causes the **Carry flag to be set**
  - Examples

```
.data
valB BYTE 1,0
valC SBYTE -128
.code
neg valB        ; CF = 1, OF = 0
neg [valB + 1]  ; CF = 0, OF = 0
neg valC        ; CF = 1, OF = 1
```

4.2.5 Implementing Arithmetic Expressions 89

- Implementing Arithmetic Expressions
  - Translate mathematical expressions into assembly language
  - Example:

    Rval = -Xval + (Yval – Zval)

```
.data
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
mov eax,Xval
neg eax          ; EAX = -26, EAX = -Xval
mov ebx,Yval     ; EBX =  30, EBX = Yval
sub ebx,Zval     ; EBX = -10, EBX = (Yval – Zval)
add eax,ebx      ; EAX = -36, EAX = -Xval + (Yval – Zval)
mov Rval,eax     ; Rval = -36
```

4.2.6 Flags Affected by Addition and Subtraction 89

- Flags Affected by Arithmetic
  - The ALU has a number of status flags that reflect the outcome of *arithmetic (and bitwise) operations* based on the contents of the destination operand
  - The *MOV instruction* ***never*** *affects the flags*.
- Essential flags:
  - **Zero flag:** Set when destination operand equals **zero**

```
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
```

Note: A flag is *set* when it equals *1*
A flag is *clear* when it equals *0*

  - **Sign flag:** Set when the destination operand is **negative**
    Clear when the destination is positive

```
mov cx,0
sub cx,1        ; CX = -1, SF = 1
add cx,2        ; CX = 1,  SF = 0
```

Note: The sign flag is a copy of the destination's highest bit

  - **Carry flag:** Set when ***unsigned destination*** operand value is out of range

```
mov al,7Fh
add al,1   ; AL = 80, CF = 0
mov al,0FFh
add al,1   ; AL = 00, CF = 1, Too big
mov al,1
sub al,2   ; AL = FF, CF = 1, Below zero
```

  - **Auxiliary Carry:** Set when ***carry out of bit 3*** in the destination operand

```
mov al,0Fh
add al,1   ; AL = 10, AC = 1
```

  - **Parity flag:** Set when the least significant byte of the destination has ***even number of 1 bits***.

```
mov al,10001100b
add al,00000010b    ; AL = 10001110, PF = 1
sub al,10000000b    ; AL = 00001110, PF = 0
```

- **Overflow flag**: Set when ***signed destination*** operand value is out of range

```
mov al,7Fh   ; OF = 1,   AL = 80h
add al,1
```

<u>Note:</u> When adding two integers, the Overflow flag is only set when:
- Two positive operands are added and their sum is negative
- Two negative operands are added and their sum is positive

- A hardware viewpoint of signed and unsigned Integers
  - All CPU instructions operate **exactly the same** on signed and unsigned integers
  - The CPU **cannot** distinguish between signed and unsigned integers
  - The programmers are solely responsible for using the correct data type with each instruction
- A hardware viewpoint of Overflow and Carry flags
  - How the **ADD** instruction modifies OF and CF:

    OF = (carry out of the MSB) **XOR** (carry into the MSB)
    CF = (carry out of the MSB)

  - How the **SUB** instruction modifies OF and CF:

    NEG the source and ADD it to the destination
    OF = (carry out of the MSB) **XOR** (carry into the MSB)
    CF = **INVERT** (carry out of the MSB)
    <u>Notation:</u>
    MSB = Most Significant Bit (high-order bit)
    XOR = eXclusive-OR operation
    eXclusive-OR operation only returns a **1** when its two input bits are different
    NEG = Negate (same as SUB 0, operand)

  - Examples:

```
mov al,-128   ; AL = 10000000b
neg al        ; AL = 10000000b, CF = 1, OF = 1

mov al,80h    ; AL = 10000000b
add al,2      ; AL = 10000010b, CF = 0, OF = 0

mov al,1      ; AL = 00000001b
sub al,2      ; AL = 11111111b, CF = 1, OF = 0

mov al,7Fh
add al,2      ; AL = 10000001b, CF = 0, OF = 1
```

4.2.7 Example Program (Addsub3) 92

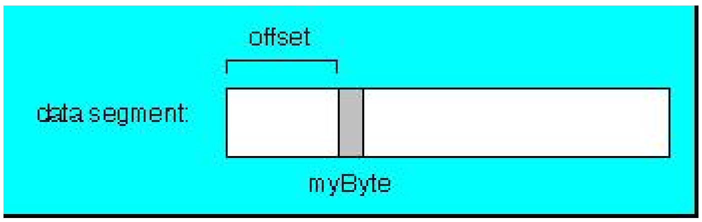
- The following program implements various arithmetic expressions using the **ADD, SUB, INC, DEC, and NEG** instructions, and show how certain status flags are affected:

```
TITLE  Addition and Subtraction        (AddSub3.asm)

; Chapter 4 example. Demonstration of ADD, SUB,
; INC, DEC, and NEG instructions, and how
; they affect the CPU status flags.
; Last update: 06/01/2006

INCLUDE Irvine32.inc

.data
Rval   SDWORD ?
Xval   SDWORD 26
Yval   SDWORD 30
Zval   SDWORD 40

.code
main PROC
  ; INC and DEC
  mov   ax,1000h
  inc   ax          ; 1001h
  dec   ax          ; 1000h

  ; Expression: Rval = -Xval + (Yval - Zval)
  mov   eax,Xval
  neg   eax         ; -26
  mov   ebx,Yval
  sub   ebx,Zval    ; -10
  add   eax,ebx
  mov   Rval,eax    ; -36

  ; Zero flag example:
  mov   cx,1
  sub   cx,1        ; ZF = 1
  mov   ax,0FFFFh
  inc   ax          ; ZF = 1

  ; Sign flag example:
  mov   cx,0
  sub   cx,1        ; SF = 1
  mov   ax,7FFFh
  add   ax,2        ; SF = 1

  ; Carry flag example:
  mov   al,0FFh
  add   al,1        ; CF = 1,  AL = 00

  ; Overflow flag example:
  mov   al,+127
  add   al,1        ; OF = 1
  mov   al,-128
  sub   al,1        ; OF = 1

  exit
main ENDP
END main
```

4.3 Data-Related Operators and Directives 94

4.3.1 OFFSET Operator 94

- OFFSET operator returns the **distance** in bytes, of a label from the beginning of its enclosing segment
  - Protected mode: Offset are 32 bits
  - Real mode: Offset are 16 bits



- OFFSET Example
  - Assume that the data segment begins at **00404000h**

```
.data
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
```

- Relating to C/C++
  - The value returned by OFFSET is a pointer
  - Compare the following code written for both C++ and assembly language

```
// C++ version:
char array[1000];
char * p = array;

; Assembly version
.data
array BYTE 1000 DUP(?)
.code
mov  esi,OFFSET array
```

4.3.2 ALIGN Directive 95

- The ALIGN directive aligns a variable on a byte, word, doubleword, or paragraph boundary.
- ALIGN Example
  - Assume that the data segment begins at **00404000h**

```
.data
bVal   BYTE   ?    ; 00404000
ALIGN  2
wVal   WORD   ?    ; 00404002
bVal2  BYTE   ?    ; 00404004
ALIGN  4
dVal   DWORD  ?    ; 00404008
dVal2  DWORD  ?    ; 0040400C
```

4.3.3 PTR Operator 95

- PTR Operator
  - o Overrides the default type of a label (variable)
  - o Provides the flexibility to access part of a variable
  - o Must be used in combination with one of the standard assembly data type: BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, or TWORD
- Little Endian Order
  - o Little endian order refers to the way Intel stores integers in memory.
  - o Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
  - o For example, the doubleword 12345678h would be stored as:

byte	offset
78	0000
56	0001
34	0002
12	0003

- PTR Operator Examples

doubleword	word	byte	offset	
12345678	5678	78	0000	myDouble
		56	0001	myDouble + 1
	1234	34	0002	myDouble + 2
		12	0003	myDouble + 3

```
.data
myDouble DWORD 12345678h
.code
mov ax,myDouble                 ; error
mov ax,WORD PTR myDouble        ; AX = 5678h
mov ax,WORD PTR [myDouble+2]    ; AX = 1234h
mov al,BYTE PTR  myDouble       ; AL = 78h
mov al,BYTE PTR [myDouble+1]    ; AL = 56h
mov al,BYTE PTR [myDouble+2]    ; AL = 34h
```

- PTR operator can combine elements of a smaller data type and move them into a larger operand

```
.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
```

4.3.4 TYPE Operator 96

- TYPE operator returns the **size**, in bytes, of a single element of a data declaration
- TYPE Example:

```
.data
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?
.code                    ; TYPE
mov eax,TYPE var1        ; 1
mov eax,TYPE var2        ; 2
mov eax,TYPE var3        ; 4
mov eax,TYPE var4        ; 8
```

4.3.5 LENGTHOF Operator 97

- LENGTHOF operator counts **the number of elements** in a single data declaration
- LENGTHOF Example:

```
.data                           ; LENGTHOF
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
.code
mov ecx,LENGTHOF array1         ; 32
```

4.3.6 SIZEOF Operator 97

- SIZEOF Operator returns a value that is equivalent to **multiplying LENGTHOF by TYPE**

```
.data                           ; SIZEOF
byte1  BYTE 10,20,30            ; 3
array1 WORD 30 DUP(?),0,0       ; 64
array2 WORD 5 DUP(3 DUP(?))     ; 30
array3 DWORD 1,2,3,4            ; 16
digitStr BYTE "12345678",0      ; 9
.code
mov ecx,SIZEOF array1           ; 64
```

- A data declaration spans multiple lines if each line (except the last) ends with a comma
- The LENGTHOF and SIZEOF operators include all lines belonging to the declaration

```
.data
array WORD  10,20,
            30,40,
            50,60
.code
mov eax,LENGTHOF array      ; 6
mov ebx,SIZEOF array        ; 12
```

4.3.7 LABEL Directive 97

- LABEL Directive
  - o Assigns an alternate label name and type to an existing storage location
  - o LABEL **does not** allocate any storage of its own
  - o Removes the need for the PTR operator

- LABEL Examples

```
.data
val16   LABEL WORD
val32   DWORD 12345678h
.code
mov ax,val16       ; AX   = 5678h
mov dx,[val16+2]   ; DX   = 1234h

.data
LongValue LABEL DWORD
val1    WORD 5678h
val2    WORD 1234h
.code
mov eax,LongValue  ; EAX  = 12345678h
```

4.4 Indirect Addressing 99

4.4.1 Indirect Operands 99

- Indirect Operands
  - An indirect operand holds the **address** of a variable, usually an array or string
  - It can be dereferenced (just like a pointer).

```
.data
val1 BYTE 10h,20h,30h
.code
mov esi,OFFSET val1    ; ESI = the address of Val1
mov al,[esi]    ; AL = 10h, dereference ESI
inc esi
mov al,[esi]    ; AL = 20h
inc esi
mov al,[esi]    ; AL = 30h
```

  - Use PTR to clarify the size attribute of a memory operand

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

4.4.2 Arrays 100

- Array Sum Example
  - Indirect operands are ideal for traversing an array
  - The register in brackets must be incremented by a value that matches the array type

```
.data
arrayW WORD 1000h,2000h,3000h
.code
mov esi,OFFSET arrayW    ; ESI = the address of Val1
mov ax,[esi]       ; AX = 1000h
add esi,2          ; or: add esi,TYPE arrayW
add ax,[esi]       ; AX = 3000h
add esi,2
add ax,[esi]       ; AX = 6000h
```

4.4.3 Indexed Operands 101

- Indexed operands
  - An indexed operand adds a constant to a register to generate an effective address. There are two notational forms:
    [*label* + *reg*] **label**[**reg**]

- Indexed operands Example

```
.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]   ; AX = 2000h
```

- Index Scaling
  - You can scale an indirect or indexed operand to the offset of an array element
  - This is done by multiplying the **index** by the array's **TYPE**

```
.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                         ; ESI = index of array
mov al,arrayB[esi*TYPE arrayB]    ; AL  = 04h
mov bx,arrayW[esi*TYPE arrayW]    ; BX  = 0004h
mov edx,arrayD[esi*TYPE arrayD]   ; EDX = 00000004h
```

4.4.4 Pointers 102

- Pointers
  - Declare a **pointer variable** that contains the **offset of another variable**

```
.data
arrayW WORD 1000h,2000h,3000h
ptrW DWORD arrayW           ; ptrW (pointer variable)
.code
mov esi,ptrW
mov ax,[esi]        ; AX = 1000h
```

  - Alternate format:

```
ptrW DWORD OFFSET arrayW   ; ptrW = Offset (address) of arrayW
```

4.5 JMP and Loop Instructions 104

4.5.1 JMP Instruction 104

- JMP Instruction
  - JMP is an **unconditional** jump to a label that is usually within the same procedure
  - Syntax: JMP target
  - Logic: **EIP ← target**
- JMP Example

```
top:
 .
 .
 jmp top
```

4.5.2 LOOP Instruction 105

- LOOP Instruction
  - The LOOP instruction creates a counting loop
  - Syntax: LOOP target
  - Logic:
    - **First, ECX ← ECX – 1**
    - **Next, if ECX != 0, jump to target**
  - Implementation:
    - The assembler calculates the distance, in bytes, between the offset of the following instruction and the offset of the target label. It is called the relative offset
    - The relative offset is added to EIP.
- LOOP Example
  - Add 1 to AX each time the loop repeats
  - When the loop ends, AX = 5 and ECX = 0

```
    mov ax, 0
    mov ecx,5
L1:
    add ax
    loop L1
```

- Nested Loop
  - If you need to code a loop within a loop, you **must save the outer loop counter's ECX value**
  - In the following example, the outer loop executes 100 times, and the inner loop 20 times

```
.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
```

4.5.3 Summing an Integer Array 106

- Summing an Integer Array

```
TITLE Summing an Array              (SumArray.asm)

; This program sums an array of words.
; Last update: 06/01/2006

INCLUDE Irvine32.inc

.data
intarray WORD 100h,200h,300h,400h

.code
main PROC

  mov  edi,OFFSET intarray        ; address of intarray
  mov  ecx,LENGTHOF intarray      ; loop counter
  mov  ax,0                       ; zero the accumulator
L1:
  add  ax,[edi]                   ; add an integer
  add  edi,TYPE intarray          ; point to next integer
  loop L1                         ; repeat until ECX = 0

  exit
main ENDP
END main
```

4.5.4 Coping a String 106

- The following code copies a string from source to *target:*

```
TITLE Copying a String                  (CopyStr.asm)

; This program copies a string.
; Last update: 06/01/2006

INCLUDE Irvine32.inc

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

.code
main PROC

   mov  esi,0               ; index register
   mov  ecx,SIZEOF source   ; loop counter
L1:
   mov  al,source[esi]      ; get a character from source
   mov  target[esi],al      ; store it in the target
   inc  esi                 ; move to next character
   loop L1                  ; repeat for entire string

   exit
main ENDP
END main
```

4.6 Chapter Summary 108

- Data Transfer
  - MOV – data transfer from source to destination
  - MOVSX, MOVZX, XCHG
- Operand types
  - direct, direct-offset, indirect, indexed
- Arithmetic instructions
  - INC, DEC, ADD, SUB, NEG
- Status Flags
  - Sign, Carry, Auxiliary Carry, Zero, and Overflow flags
- Operators
  - OFFSET, PTR, TYPE, LENGTHOF, SIZEOF
- Loops
  - JMP and LOOP – branching instructions