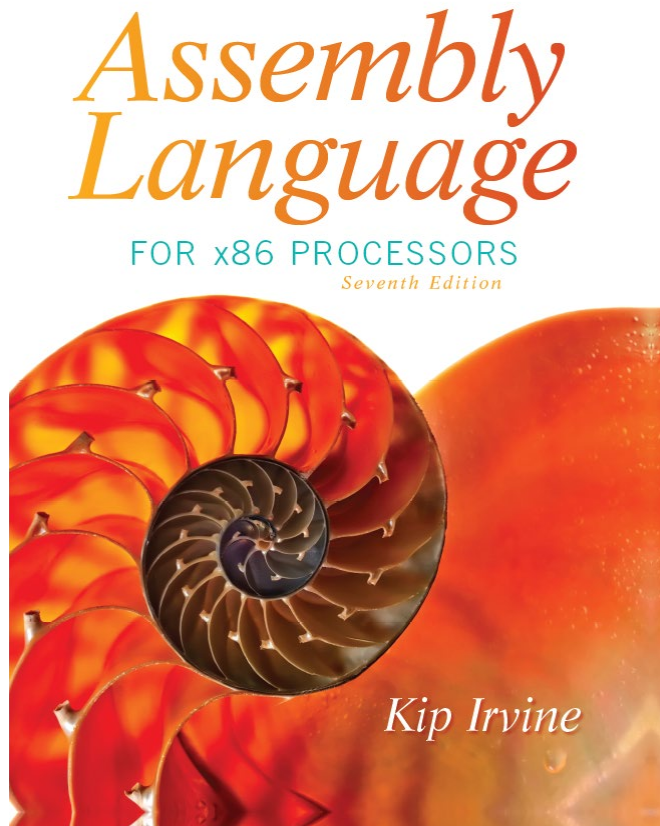


# Assembly Language for x86 Processors

Seventh Edition



## Chapter 4

Data Transfers,  
Addressing, and Arithmetic

# Chapter Overview

- **Data Transfer Instructions**
- Addition and Subtraction
- Data-Related Operators and Directives
- Indirect Addressing
- JMP and LOOP Instructions

# Data Transfer Instructions

- Operand Types
- Instruction Operand Notation
- Direct Memory Operands
- MOV Instruction
- Zero & Sign Extension
- Xchg Instruction
- Direct-Offset Instructions

# Operand Types

- 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

# Instruction Operand Notation

Operand	Description
<i>reg8</i>	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
<i>reg16</i>	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
<i>reg32</i>	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
<i>reg</i>	Any general-purpose register
<i>sreg</i>	16-bit segment register: CS, DS, SS, ES, FS, GS
<i>imm</i>	8-, 16-, or 32-bit immediate value
<i>imm8</i>	8-bit immediate byte value
<i>imm16</i>	16-bit immediate word value
<i>imm32</i>	32-bit immediate doubleword value
<i>reg/mem8</i>	8-bit operand, which can be an 8-bit general register or memory byte
<i>reg/mem16</i>	16-bit operand, which can be a 16-bit general register or memory word
<i>reg/mem32</i>	32-bit operand, which can be a 32-bit general register or memory doubleword
<i>mem</i>	An 8-, 16-, or 32-bit memory operand

# 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

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



alternate format

# Mov Instruction

- Move from source to destination. Syntax:
  - `mov destination,source`
- No more than one memory operand permitted
- CS, EIP, and IP cannot be the destination
- No immediate to segment moves

`.data`

`count BYTE 100`

`Val WORD 2`

`.code`

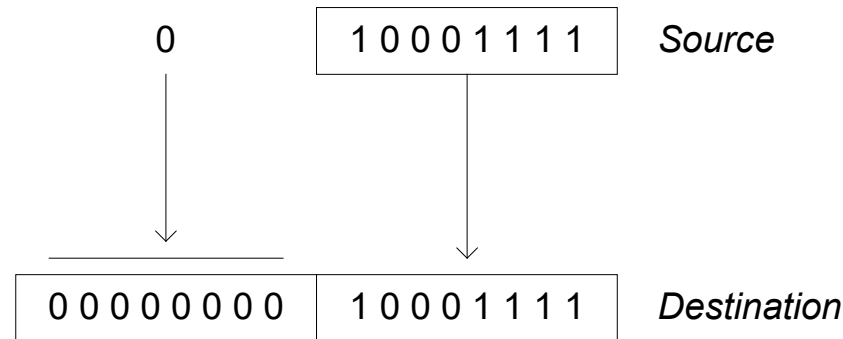
`mov bl, count`

`mov ax, Val`

`mov count, al`

# Zero Extension

When you copy a smaller value into a larger destination, the `movzx` instruction fills (extends) the upper half of the destination with zeros.



```
mov bl,10001111b
```

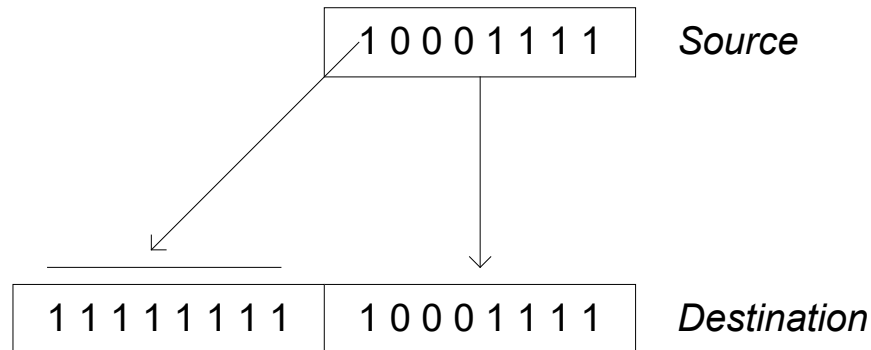
```
movzx ax,bl ; zero-extension
```

The destination must be a register.



# Sign Extension

The movsx instruction fills the upper half of the destination with a copy of the source operand's sign bit.



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

The destination must be a register.

# Xchg Instruction

Xchg exchanges the values of two operands. At least one operand must be a register. 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
```

# Direct-Offset Operands (1 of 2)

A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

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

Q: Why doesn't arrayB+1 produce 11h?

# Addition and Subtraction

- Inc and Dec Instructions
- Add and Sub Instructions
- Neg Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
  - Zero
  - Sign
  - Carry
  - Overflow

# INC and DEC Instructions

- Add 1, subtract 1 from destination operand
  - operand may be register or memory
- INC destination
  - Logic:  $\text{destination} \leftarrow \text{destination} + 1$
- DEC destination
  - Logic:  $\text{destination} \leftarrow \text{destination} - 1$

# Inc and Dec Examples

```
.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
```

# Add and Sub Instructions

- ADD destination, source
  - Logic:  $destination \leftarrow destination + source$
- SUB destination, source
  - Logic:  $destination \leftarrow destination - source$
- Same operand rules as for the MOV instruction

# Add and Sub Examples

```
.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 (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
```

# 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
- Essential flags:
  - Zero flag – set when destination equals zero
  - Sign flag – set when destination is negative
  - Carry flag – set when unsigned value is out of range
  - Overflow flag – set when signed value is out of range
- The MOV instruction never affects the flags.

# Zero Flag (ZF)

The Zero flag is set when the result of an operation produces zero in the destination operand.

```
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
```

Remember...

- A flag is set when it equals 1.
- A flag is clear when it equals 0.

# Sign Flag (SF)

The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

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

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

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

# Carry Flag (CF)

The Carry flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

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

```
; Try to go below zero:
```

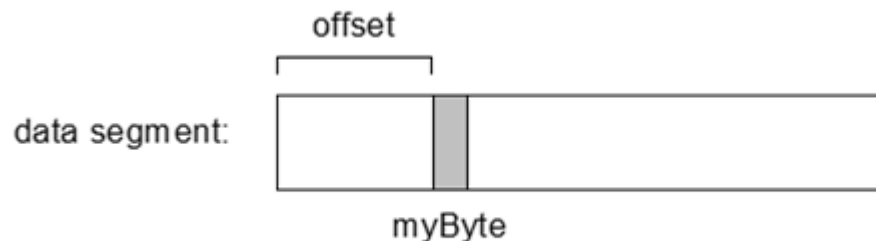
```
mov al,0  
sub al,1      ; CF = 1, AL = FF
```

# Data-Related Operators and Directives

- OFFSET Operator
- PTR Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
- LABEL Directive

# OFFSET Operator

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



- The Protected-mode programs we write use only a single segment (flat memory model).

# OFFSET Examples

Let's 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 language:
```

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

# PTR Operator (1 of 2)

Overrides the default type of a label (variable). Provides the flexibility to access part of a variable.

```
.data
myDouble DWORD 12345678h
.code
mov ax,myDouble                ; error - why?

mov ax,WORD PTR myDouble        ; loads 5678h

mov WORD PTR myDouble,4321h     ; saves 4321h
```

Little endian order is used when storing data in memory

# PTR Operator Examples

```
.data  
myDouble DWORD 12345678h
```

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

```
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
```

# PTR Operator

PTR can also be used to combine elements of a smaller data type and move them into a larger operand. The CPU will automatically reverse the bytes.

```
.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 Operator

The TYPE operator returns the size, in bytes, of a single element of a data declaration.

```
.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 Operator

The LENGTHOF operator counts the number of elements in a single data declaration.

```
.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
```

# SIZEOF Operator

The 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
```

# Pointers

You can declare a pointer variable that contains the offset of another variable.

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

Alternate format:

```
ptrW DWORD OFFSET arrayW
```



# JMP Instruction

- JMP is an unconditional jump to a label that is usually within the same procedure.
- Syntax: **JMP target**
- Logic:  $EIP \leftarrow \mathbf{target}$
- Example:

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

- A jump outside the current procedure must be to a special type of label called a global label

# LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax: LOOP *target*
- Logic:
  - $ECX \leftarrow ECX - 1$
  - if  $ECX \neq 0$ , 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

- The following loop calculates the sum of the integers 5 + 4 + 3 + 2 + 1:

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 FB	loop L1
0000000E		

When LOOP is assembled, the current location = 0000000E (offset of the next instruction). -5 (FBh) is added to the 00000009 : current location, causing a jump to location 00000009  $\leftarrow$  0000000E + FB

# 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
```