

# Assembly Language for x86 Processors

## 7th Edition, Global Edition

Kip Irvine

### 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
- 64-Bit Programming

# 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
wVal  WORD 2
.code
    mov bl,count
    mov ax,wVal
    mov count,al

    mov al,wVal           ; error
    mov ax,count          ; error
    mov eax,count         ; error
```

# Your turn . . .

Explain why each of the following MOV statements are invalid:

`.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**

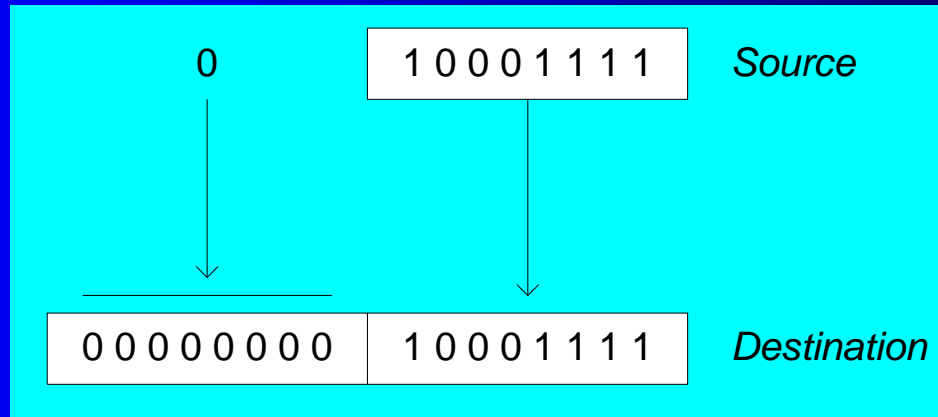
`mov 25,bVal`                      **immediate value cannot be destination**

`mov bVal2,bVal`                      **memory-to-memory move not permitted**



# 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.



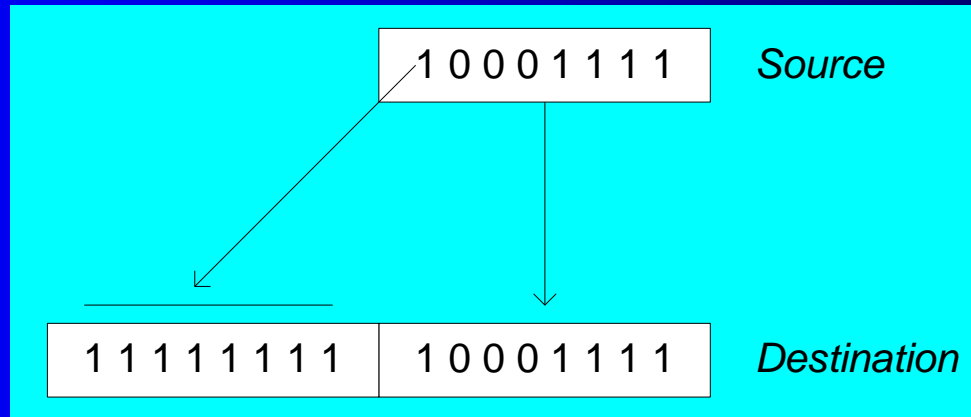
```
mov bl,10001111b
```

```
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

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?

# Direct-Offset Operands (cont)

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

```
; Will the following statements assemble?
mov ax,[arrayW-2]           ; ??
mov eax,[arrayD+16]         ; ??
```

What will happen when they run?

# Your turn. . .

Write a program that rearranges the values of three doubleword values in the following array as: 3, 1, 2.

```
.data
```

```
arrayD DWORD 1,2,3
```

- Step1: copy the first value into EAX and exchange it with the value in the second position.

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

- Step 2: Exchange EAX with the third array value and copy the value in EAX to the first array position.

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

# Evaluate this . . .

- We want to write a program that adds the following three bytes:

```
.data  
myBytes BYTE 80h,66h,0A5h
```

- What is your evaluation of the following code?

```
mov al,myBytes  
add al,[myBytes+1]  
add al,[myBytes+2]
```

- What is your evaluation of the following code?

```
mov ax,myBytes  
add ax,[myBytes+1]  
add ax,[myBytes+2]
```

- Any other possibilities?

# Evaluate this . . . (cont)

```
.data  
myBytes BYTE 80h,66h,0A5h
```

- How about the following code. Is anything missing?

```
movzx ax,myBytes  
mov    bl,[myBytes+1]  
add    ax,bx  
mov    bl,[myBytes+2]  
add    ax,bx                ; AX = sum
```

Yes: Move zero to BX before the MOVZX instruction.



# What's Next

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

# 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:  $destination \leftarrow destination + 1$
- DEC *destination*
  - Logic:  $destination \leftarrow destination - 1$

# 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
```

# Your turn...

Show the value of the destination operand after each of the following instructions executes:

```
.data
myByte BYTE 0FFh, 0
.code
    mov al,myByte           ; AL = FFh
    mov ah,[myByte+1]       ; AH = 00h
    dec ah                  ; AH = FFh
    inc al                  ; AL = 00h
    dec ax                  ; AX = FEFF
```

# 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
```

Suppose AX contains -32,768 and we apply NEG to it. Will the result be valid?



# 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.

```
.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
```

# Implementing Arithmetic Expressions

HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

$$Rval = -Xval + (Yval - Zval)$$

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

# Your turn...

Translate the following expression into assembly language.  
Do not permit Xval, Yval, or Zval to be modified:

$$Rval = Xval - (-Yval + Zval)$$

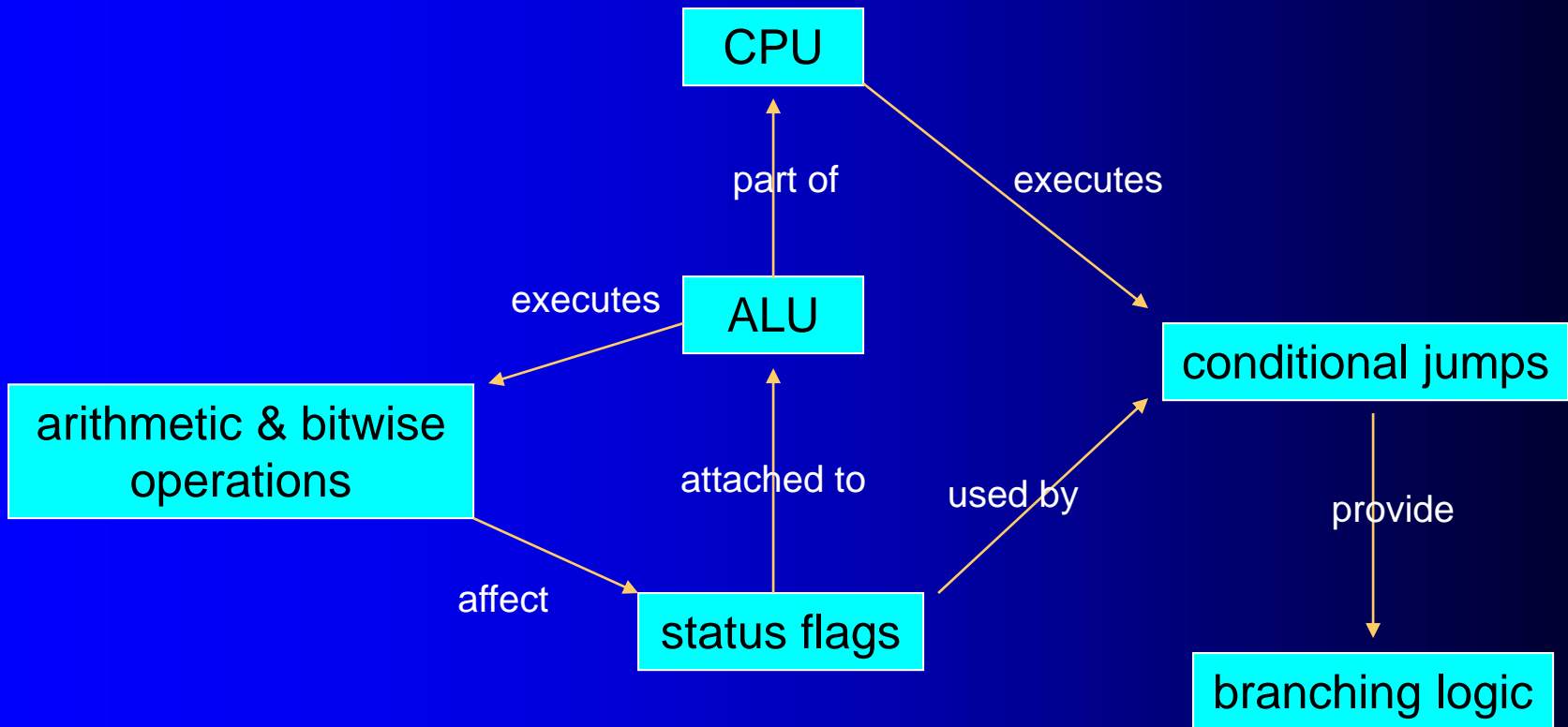
Assume that all values are signed doublewords.

```
mov ebx,Yval
neg ebx
add ebx,Zval
mov eax,Xval
sub eax,ebx
mov Rval,eax
```

# 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.

# Concept Map



You can use diagrams such as these to express the relationships between assembly language concepts.

# 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
```

# Signed and Unsigned Integers

## A Hardware Viewpoint

- All CPU instructions operate exactly the same on signed and unsigned integers
- The CPU cannot distinguish between signed and unsigned integers
- YOU, the programmer, are solely responsible for using the correct data type with each instruction



# Overflow and Carry Flags

## A Hardware Viewpoint

- How the **ADD** instruction affects OF and CF:
  - $CF = (\text{carry out of the MSB})$
  - $OF = CF \text{ XOR MSB}$
- How the **SUB** instruction affects OF and CF:
  - $CF = \text{INVERT}(\text{carry out of the MSB})$
  - negate the source and add it to the destination
  - $OF = CF \text{ XOR MSB}$

MSB = Most Significant Bit (high-order bit)  
XOR = eXclusive-OR operation  
NEG = Negate (same as SUB 0,operand )

# 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
```

# Your turn . . .

For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

<code>mov ax,00FFh</code>				
<code>add ax,1</code>	<code>; AX=0100h</code>	<code>SF=0</code>	<code>ZF=0</code>	<code>CF=0</code>
<code>sub ax,1</code>	<code>; AX=00FFh</code>	<code>SF=0</code>	<code>ZF=0</code>	<code>CF=0</code>
<code>add al,1</code>	<code>; AL=00h</code>	<code>SF=0</code>	<code>ZF=1</code>	<code>CF=1</code>
<code>mov bh,6Ch</code>				
<code>add bh,95h</code>	<code>; BH=01h</code>	<code>SF=0</code>	<code>ZF=0</code>	<code>CF=1</code>
<code>mov al,2</code>				
<code>sub al,3</code>	<code>; AL=FFh</code>	<code>SF=1</code>	<code>ZF=0</code>	<code>CF=1</code>

# Overflow Flag (OF)

The Overflow flag is set when the signed result of an operation is invalid or out of range.

**; Example 1**

```
mov al,+127
```

```
add al,1
```

```
; OF = 1,    AL = ??
```

**; Example 2**

```
mov al,7Fh
```

```
add al,1
```

```
; OF = 1,    AL = 80h
```

The two examples are identical at the binary level because 7Fh equals +127. To determine the value of the destination operand, it is often easier to calculate in hexadecimal.

# A Rule of Thumb

- When adding two integers, remember that 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

What will be the values of the Overflow flag?

```
mov al,80h
add al,92h                ; OF = 1
```

```
mov al,-2
add al,+127               ; OF = 0
```

# Your turn . . .

What will be the values of the given flags after each operation?

```
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
```

# What's Next

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

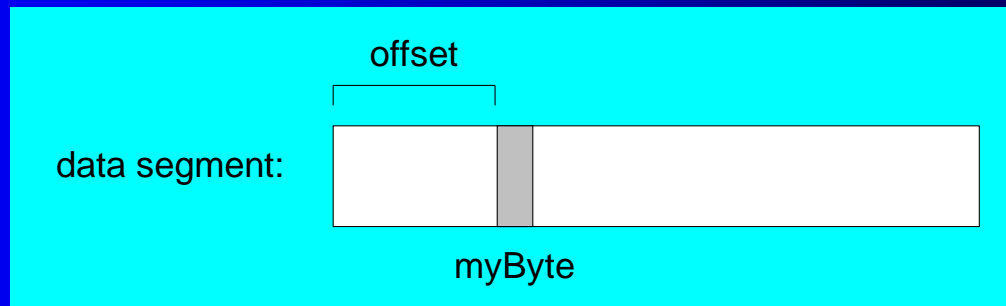
# Data-Related Operators and Directives

- 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

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 (see Section 3.4.9).

# Little Endian Order

- Little endian order refers to the way Intel stores integers in memory.
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
- For example, the doubleword 12345678h would be stored as:

byte	offset
78	0000
56	0001
34	0002
12	0003

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions.

# 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 (cont)

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
```

# Your turn . . .

Write down the value of each destination operand:

```
.data
```

```
varB BYTE 65h,31h,02h,05h
```

```
varW WORD 6543h,1202h
```

```
varD DWORD 12345678h
```

```
.code
```

```
mov ax,WORD PTR [varB+2]           ; a. 0502h
mov bl,BYTE PTR varD               ; b. 78h
mov bl,BYTE PTR [varW+2]           ; c. 02h
mov ax,WORD PTR [varD+2]           ; d. 1234h
mov eax,DWORD PTR varW             ; e. 12026543h
```



# 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.

	LENGTHOF
<code>.data</code>	
<code>byte1 BYTE 10,20,30</code>	<code>; 3</code>
<code>array1 WORD 30 DUP(?),0,0</code>	<code>; 32</code>
<code>array2 WORD 5 DUP(3 DUP(?))</code>	<code>; 15</code>
<code>array3 DWORD 1,2,3,4</code>	<code>; 4</code>
<code>digitStr BYTE "12345678",0</code>	<code>; 9</code>
 <code>.code</code>	
<code>mov ecx,LENGTHOF array1</code>	<code>; 32</code>

# SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.

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

# Spanning Multiple Lines (1 of 2)

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
```

# Spanning Multiple Lines (2 of 2)

In the following example, `array` identifies only the first `WORD` declaration. Compare the values returned by `LENGTHOF` and `SIZEOF` here to those in the previous slide:

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

.code
mov eax,LENGTHOF array           ; 2
mov ebx,SIZEOF array             ; 4
```

# LABEL Directive

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

```
.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
```

# What's Next

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

# Indirect Addressing

- Indirect Operands
- Array Sum Example
- Indexed Operands
- Pointers



# Indirect Operands (1 of 2)

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
mov al,[esi]                ; dereference ESI (AL = 10h)

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

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

# Indirect Operands (2 of 2)

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
```

Should PTR be used here?

```
add [esi],20
```

yes, because [esi] could point to a byte, word, or doubleword

# Array Sum Example

Indirect operands are ideal for traversing an array. Note that 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
    mov ax,[esi]
    add esi,2                ; or: add esi,TYPE arrayW
    add ax,[esi]
    add esi,2
    add ax,[esi]             ; AX = sum of the array
```

ToDo: Modify this example for an array of doublewords.

# 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]`

```
.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]
    etc.
```

ToDo: Modify this example for an array of doublewords.

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

# 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
```

# What's Next

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

# JMP and LOOP Instructions

- JMP Instruction
- LOOP Instruction
- LOOP Example
- Summing an Integer Array
- Copying a String



# JMP Instruction

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

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

A jump outside the current procedure must be to a special type of label called a **global label** (see Section 5.5.2.3 for details).

# 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 the current location, causing a jump to location 00000009:

$$00000009 \leftarrow 0000000E + FB$$

# Your turn . . .

If the relative offset is encoded in a single signed byte,

(a) what is the largest possible backward jump?

(b) what is the largest possible forward jump?

(a) -128

(b) +127

# Your turn . . .

What will be the final value of AX?

10

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

How many times will the loop execute?

4,294,967,296

```
mov ecx,0  
x2:  
inc ax  
loop x2
```

# 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
```

# Summing an Integer Array

The following code calculates the sum of an array of 16-bit integers.

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

# Your turn . . .

What changes would you make to the program on the previous slide if you were summing a doubleword array?



# Copying a String

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

## good use of sizeof

**L1:**

# Your turn . . .

Rewrite the program shown in the previous slide, using indirect addressing rather than indexed addressing.

# What's Next

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

# 64-Bit Programming

- MOV instruction in 64-bit mode accepts operands of 8, 16, 32, or 64 bits
- When you move a 8, 16, or 32-bit constant to a 64-bit register, the upper bits of the destination are cleared.
- When you move a memory operand into a 64-bit register, the results vary:
  - 32-bit move clears high bits in destination
  - 8-bit or 16-bit move does not affect high bits in destination

# More 64-Bit Programming

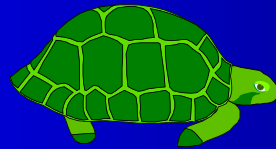
- MOVSXD sign extends a 32-bit value into a 64-bit destination register
- The OFFSET operator generates a 64-bit address
- LOOP uses the 64-bit RCX register as a counter
- RSI and RDI are the most common 64-bit index registers for accessing arrays.

# Other 64-Bit Notes

- ADD and SUB affect the flags in the same way as in 32-bit mode
- You can use scale factors with indexed operands.

# Summary

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



*46 69 6E 61 6C*