Assembly Language for Intel-Based Computers, 4th Edition

Kip R. 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

Operand Types

- 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

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

Irvine, Kip R. Assembly Language for Intel-Based Computers, 2003.

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,[00010400]          ; AL = 10h
mov al,var1                ; AL = 10h
mov al,[var1]              ; AL = 10h
```

alternate format

MOV Instruction

- Move from source to destination. Syntax:

  MOV *destination,source*

- Both operands must be the same size.
- 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
```

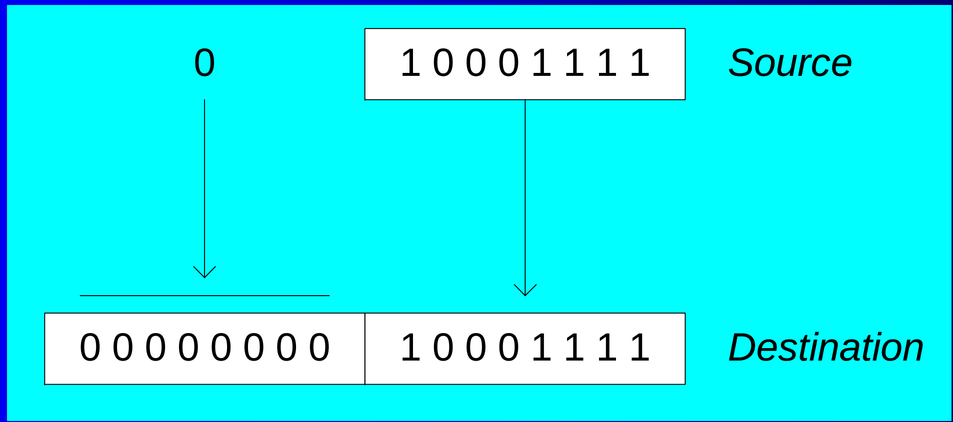
Your turn . . .

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                ; a.
    mov esi,wVal             ; b.
    mov eip,dVal             ; c.
    mov 25,bVal              ; d.
    mov bVal2,bVal           ; e.
```

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.

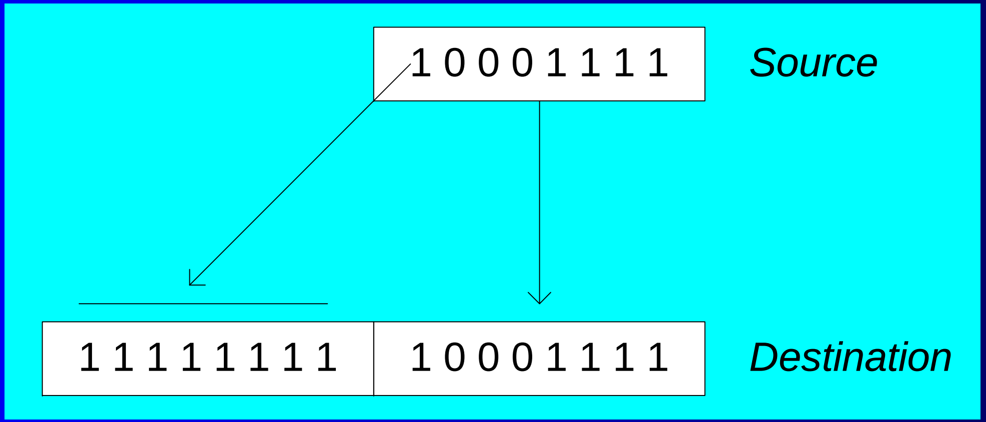



```
mov bl,10001111b
movzx ax,bl                    ; zero-extension
```

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.

LAHF and SAHF Instructions

- Loads/Stores flag values from/to EFLAGS register into/from AH

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

```
mov ah,saveflags            ; load saved flags into AH
sahf                        ; copy into Flags 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 [1/2]

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?

Direct-Offset Operands [2/2]

```
.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 and run?
mov ax,[arrayW-2]          ; ??
mov eax,[arrayD+16]        ; ??
```

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? 8B

```
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]
```

Can not be assembled

If the type of myByetes is WORD, the answer will be 66E6

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.

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

- INC *destination*
  - INC *reg/mem*
  - Logic: *destination* $\leftarrow$ *destination* + 1
- DEC *destination*
  - DEC *reg/mem*
  - Logic: *destination* $\leftarrow$ *destination* – 1

```
.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
```

Irvine, Kip R. Assembly Language for Intel-Based Computers, 2003.

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 = FEFFh
```

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 SBYTE -1
valW SWORD +32767
.code
    mov al,valB                ; AL = -1
    neg al                     ; AL = +1
    neg valW                   ; valW = -32767
```

Suppose AX contains –32768 and we apply NEG to it. Will the result be valid?

Storage Type	Range (low–high)	Powers of 2
Signed byte	–128 to +127	-2^7 to $(2^7 - 1)$
Signed word	–32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	–2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Implementing Arithmetic Expressions

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

```
Rval = -Xval + (Yval – Zval)
```

```
Rval SDWORD ?
Xval SDWORD 26
Yval SDWORD 30
Zval SDWORD 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 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 – destination equals zero
  - Sign flag – destination is negative
  - Carry flag – unsigned value out of range
  - Overflow flag – signed value out of range
- The MOV instruction never affects the flags.

Zero Flag (ZF)

Whenever the destination operand equals Zero, the Zero flag is set.

```
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
```

A flag is set when it equals 0.

A flag is clear when it equals 1.

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
-----------------------------------------------
mov ax,00FFh
add ax,1                        ; CF = 0, AX = 0100h
```

In the second example, we tried to generate a negative value. Unsigned values cannot be negative, so the Carry flag signaled an error condition.

Your turn . . .

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

```
mov ax,00FFh
add ax,1              ; AX=0100h  SF=0 ZF=0 CF=0
sub ax,1              ; AX=00FFh  SF=0 ZF=0 CF=0
add al,1              ; AL=00h    SF=0 ZF=1 CF=1
mov bh,6Ch
add bh,95h            ; BH=01h    SF=0 ZF=0 CF=1

mov al,2
sub al,3              ; AL=FFh    SF=1 ZF=0 CF=1
```

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

; Example 2
mov al,-128
sub al,1                       ; OF = 1
```

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 Carry and Overflow flags after each operation?

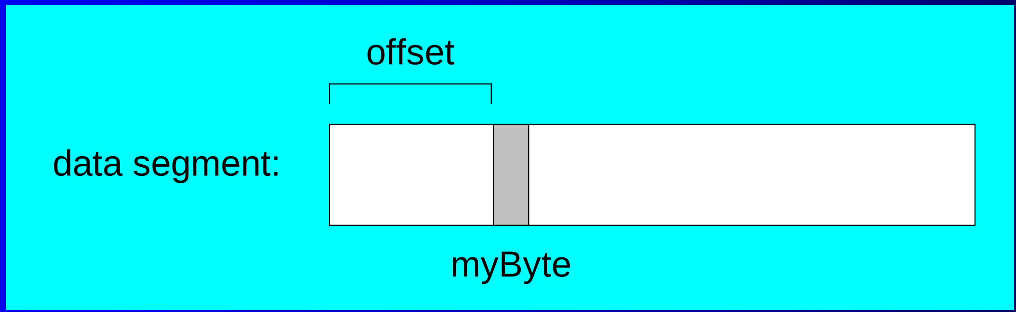
```
mov al,-128
neg al                  ; CF = 0   OF = 1

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

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

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, from the beginning of its enclosing segment
  - Protected mode: 32 bits
  - Real mode: 16 bits



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;
```

```
.data
myArray BYTE 1000 DUP(?)
.code
mov  esi,OFFSET myArray        ; ESI is p
```

ALIGN Directive

- To align a variable on a byte, word, doubleword boundary
- CPU can process data stored at even-numbered addresses more quickly than those at odd-numbered addresses.

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

PTR Operator

To override 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

- 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+1]        ; AX = 5634h
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.

```
.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

.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
```

Spanning Multiple Lines [1/2]

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 [2/2]

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
```