Assembly Language for Intel-Based Computers, 4th Edition

Kip R. Irvine

Chapter 4: Data Transfers, Addressing, and Arithmetic

Lecture 15: ADD, SUB, NEG and how they affect the flags. Lengthof, Sizeof, Type, PTR, Label operators.

Slides prepared by Kip R. Irvine

Revision date: 09/26/2002

Modified by Dr. Nikolay Metodiev Sirakov, March 04,2009

- Chapter corrections (Web) Assembly language sources (Web)

ADD and SUB Instructions

- ADD destination, source
  - Logic: *destination* ← *destination* + source
- SUB destination, source
  - Logic: *destination* ← *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 16
Yval DWORD 30
Zval DWORD 40
.code
    mov eax,Xval
    neg eax                     ; EAX = -16
    mov ebx,Yval
    sub ebx,Zval                ; EBX = -10
    add eax,ebx
    mov Rval,eax                ; -26
```

The example is modified from its original version.

Your turn...

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

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

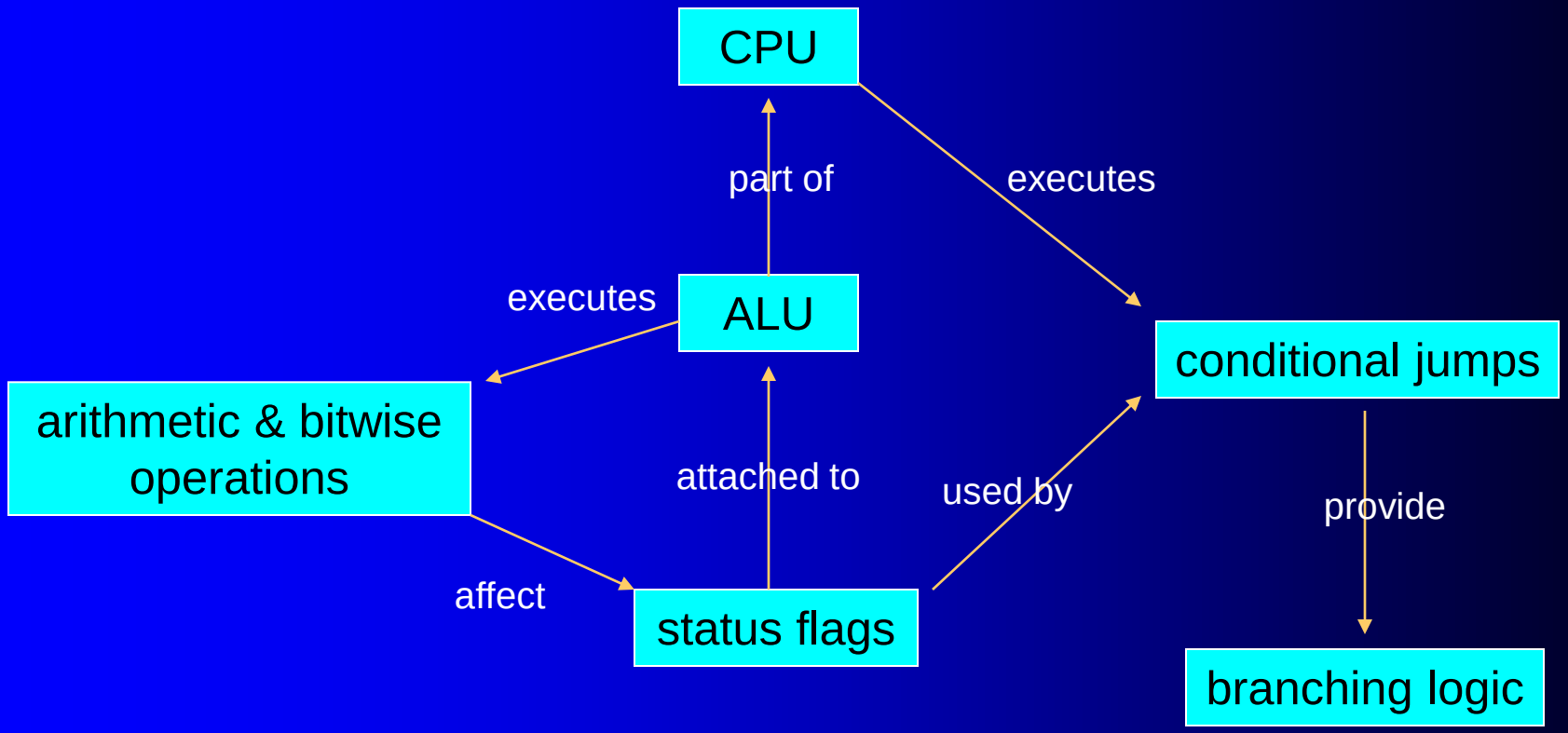
Assume that all values are signed doublewords.

```
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 – 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.

Concept Map



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

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 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 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)

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

Added Slide. Gerald Cahill, Antelope Valley College
Lecture15, Arithmetic, 03.10.2005, 3PM-4:15
Irvine, Kip R. Assembly Language for Intel-Based Computers, 2003.

Web site Examples

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:

```
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,   AL = ??

; Example 2
mov al,7Fh                  ; OF = 1,   AL = 80h
add al,1
```

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?

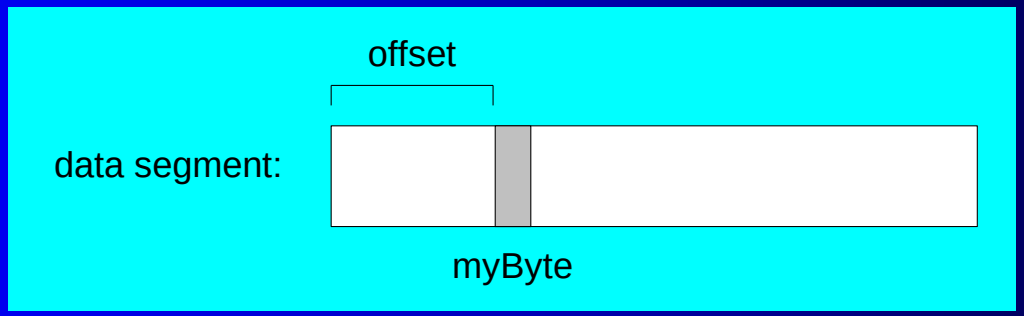
```
mov al,-128
neg al                  ; CF = 1   OF = 1

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
```

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 only have a single segment (we use the 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;
```

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

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 ax,WORD PTR myDouble+2         ; loads 1234h

mov WORD PTR myDouble,9999h        ; saves 9999h
```

To understand how this works, we need to review little endian ordering of data in memory.

Little Endian Order

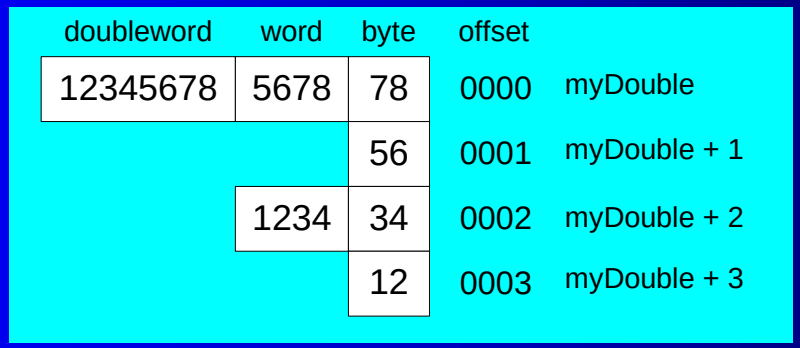
- Little endian order refers to the way Intel stores integers in memory.
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
- For example, the doubleword 12345678h would be stored as:

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
```




```
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.

```
.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
```

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
```