ARITHMETIC

The addition and subtraction operations are called **add** and **sub**. They have the same types of operands as the **mov** instruction:

```
add/sub        reg/mem,   reg/mem/constant
```

And the functionality is

```
add    dest, source    ; dest = dest + source
sub    dest, source    ; dest = dest - source
```

The operands must be of the same size (both doubleword, both word, or both byte), and at least one operand must not be from memory. The **eax, ebx**, **ecx** and **edx** can de used for doubleword operations, the **ax**, **bx**, **cx** and **dx** can be used with operations with words, and the **ah**, **al**, **bh**, … **dl** registers can be used for byte operations.

Example 1: Write assembly code for the statement **A = B – C + 3**, assuming word variables. Usually it is best to do all but the simplest computations in one or more of the CPU registers and then move the result to its final destination:

```
mov    ax, B      ; ax = B
sub    ax, C      ; ax = ax - C i.e. ax = B - C
add    ax, 3      ; ax = ax + 3 i.e. ax = B - C + 3
mov    A, ax      ; A = B - C + 3
```

Example 2: Write assembly code for the statement **A = A + B**:

```
mov    ax, B      ; ax = B
add    A, ax      ; A = A + ax i.e. A = A + B
```

Note that

```
add    A, B       ; ILLEGAL – two memory operands
```

Example 3: Write assembly code for the statement **A = A - 3**:

```
sub    A, 3       ; A = A - 3
```

The statement **A = A + 1** and **A = A -1** occur so frequently that most computers have special instructions for them. (The equivalent **A += 1** and **A -= 1** in high-level programming languages) The 80X86 instructions are:

```
inc / dec      reg / mem
```

inc stands for increment and **dec** for decrement. Their functionality is:

```
inc    dest       ; dest = dest + 1
dec    dest       ; dest = dest - 1
```

The **dest** operand can be a byte, word or doubleword operand.

There is another instruction in the mould of **inc/dec** that is often useful:

```
neg    reg / mem
```

which negates its byte, word or doubleword operand:

```
neg     dest            ; dest = - dest
```

Example 4: Suppose that **Char** is a byte variable containing a lowercase letter. Write assembly language to convert it to the corresponding uppercase letter.

We use the fact that

upper-case-letter + 'a' – 'A' = corresponding lower-case-letter

for example, that **'h' = 'H' + 'a' – 'A'**. Then, the code will be

```
sub     Char,  'a' –  'A'
```

MULTIPLICATION AND DIVISION

Multiplication and division are always much complex in computers than addition and subtraction. It's obvious why they must be: multiplying two **n-digit** numbers will in general produce a **2n-digit** number, and to make division an inverse operation, we should be able to divide a **2n digit** number by an **n-digit** number. Also we want to be able to produce the **remainder** as well as the **quotient** on division.

Also the problem of positive and negative numbers must be dealt with 2's complement doesn't work. For instance, multiplication by -3 and by its 2's complement representation 0FFFDh, considered as an unsigned number (65,533) are two different operations. As a result, it has different operations – **mul** and **div** for unsigned numbers and **imul** and **idiv** for signed numbers.

For multiplication, owing the fact that the product is twice as long as the numbers being multiplied, a special set of registers is used:

Operand Size	Multiplicand	Multiplier	Product
BYTE	AL	REGISTER OR MEMORY	AX
WORD	AX		AX(LOW) and DX(HIGH)
DWORD	EAX		EAX(LOW) and EDX(HIGH)

The instructions are

```
mul     reg/mem         ;unsigned multiply
imul    reg/mem         ;signed (integer) multiply
```

;only the **multiplier** is specified explicitly. The other **operand** and the **product** are specified by the table above using the size of **reg/mem**

If we want to multiply two words and get a word result; as we saw earlier, if a 32-bit signed or unsigned number actually fits in 16 bits, we just have to use the lower-order 16 bits (**ax** here).

Thus we could code the statement **A = B * C** (**A**, **B** and **C** are signed word variables) by writing

```
mov     ax, B
imul    C               ;ax is not written, it is assumed
mov     A, ax           ;dx ignored, better be sure it can be!
```

If the product is small enough, the high order bits in **dx** will be all sign bits, or in the case of unsigned multiplication, all zeroes.
Notice that a **constant multiplier is not allowed**. This can be handled by moving the constant to a register and then multiplying. For instance:

```
mov     bx, constant
imul    bx              ;multiplicand is ax, Result is dx/ax
```

To avoid this inconvenience, the 80386 and up has added a workaround:

```
imul    reg1, re2/mem, const    ;reg1=reg2/mem * constant
```

and

```
imul    reg1, reg2/mem/const    ;reg1 = reg1 * reg2/mem/constant
```

The **first two operands must be of the same size** – word or double word – and the **product** is the same size as the **operands**. **There is no version for unsigned multiplication by constant, and there is no version for any kind of division by constants.** To make division the inverse of multiplication, the register (pair) containing the number to be divided is twice as long as the divisor ant the result. Also, **division** has **two** results, the **quotient** and the **remainder**.

Operand Size	Dividend	Divisor	Quotient	Remainder
BYTE	AX	REGISTER OR MEMORY	AL	AH
WORD	AX and DX		AX	DX
DWORD	EAX and EDX		EAX	EDX

The instructions are

```
div     reg/mem     ;unsigned multiply
idiv    reg/mem     ;signed (integer) multiply
```

only the **divisor** is specified explicitly. The **dividend** and the result are specified by the table above using the size of **reg/mem**

We have a problem if we want to divide a word by a word. We need some way of extending the word to double word in **dx** and **ax**. Similar problems arise when dividing by bytes and double words. The 80X86 comes to the rescue with instructions to do the extension for signed numbers: In the word case, we would normally have a 16-bit number that we wish to divide by another 16-bit number. In the computer, we need to convert the dividend (=numerator) into a 32-bit number. That's easy enough in the case of unsigned numbers- just add 16 zero bits to the left. In case of signed numbers, we need to add on 16 zero bits if the number is positive and 16 one bit if the number is negative.

Sign extension comes up so often that the 80X86 have special instructions to accomplish it:

Sign Extensions

```
cbw     ;convert the signed byte in al to a word in ax
cwd     ;convert the signed word in ax to a double word in dx, ax (high ;order in dx); ax
        unchanged
cdq     ;convert the signed double word in eax to a quad word in edx, ;eax (high order in
        edx); eax unchanged
cwde    ;convert the signed word in the signed word in ax to a double ;word in eax
```

In all cases, the conversion is done by extending the sign bit.

Code for the statement A = B/C for signed word variables is:

```
mov     ax, B
cwd
idiv    C
mov     A, ax     ;the remainder is in dx, if needed
```

Code for the statement X = Y % 5 (remainder) for signed byte variables is

```
mov     al, Y
cbw
mov     bl, 5
idiv    bl        ;idiv 5 won't work
mov     X, ah     ;The quotient is in al
```

When you want to do division of unsigned numbers, all you have to do to extend the dividend. For instance, to do A = B/C assuming the variables are unsigned words, do:

```
mov     ax, B
mov     dx, 0     ;extend unsigned B to 32 bits
div     C
mov     A, ax     ;the remainder is in dx, if needed
```

(**sub dx, dx** is a better way to set dx to 0)

In these examples below: assume that all variables are signed words

Example 1

Code A = (B+C) / (X +1)

```
mov     ax, B
add     ax, C       ;ax = B + C
mov     bx, X
inc     bx          ;bx = bx + 1
cwd                 ;dx, ax = B + C
idiv    bx
mov     A, ax
```

Example 2

*Code A = 3 * C – 14 *B*

```
mov     ax, 3
imul    C
mov     bx, ax      ;save 3 * C temporarily in bx
inc     bx          ;bx = bx + 1
cwd                 ;dx, ax = B + C
idiv    bx
mov     A, ax
```

Example 3

*Code A = (B/C) * (D+1).*
*First compute B * (D+1), which will be the product in* **dx** *and* **ax** *and then divide by C.*

```
mov     ax, D
inc     ax
imul    B
idiv    C
mov     A, ax
```